Scaling Large RDF Archives To Very Long Histories

Olivier Pelgrin, Ruben Taelman, Luis Galárraga, Katja Hose

Scaling Large RDF Archives To Very Long Histories

Olivier Pelgrin
Aalborg University, Denmark
olivier@cs.aau.dk

Ruben Taelman
Ghent University, Belgium
ruben.taelman@ugent.be

Luis Galárraga
Inria, France
luis.galarraga@inria.fr

Katja Hose
Aalborg University, Denmark
khose@cs.aau.dk

***Abstract*—In recent years, research in RDF archiving has gained traction due to the ever-growing nature of semantic data and the emergence of community-maintained knowledge bases. Several solutions have been proposed to manage the history of large RDF graphs, including approaches based on independent copies, time-based indexes, and change-based schemes. In particular, aggregated changesets have been shown to be relatively efficient at handling very large datasets. However, ingestion time can still become prohibitive as the revision history increases. To tackle this challenge, we propose a hybrid storage approach based on aggregated changesets, snapshots, and multiple delta chains. We evaluate different snapshot creation strategies on the BEAR benchmark for RDF archives, and show that our techniques can speed up ingestion time up to two orders of magnitude while keeping competitive performance for version materialization and delta queries. This allows us to support revision histories of lengths that are beyond reach with existing approaches.**

I. Introduction

The ever-growing nature of RDF data and the emergence of large collaborative knowledge graphs have propelled research in efficient techniques for *RDF archiving* [FUPK19], [PGH21], which is the task of keeping track of an RDF graph's change history. RDF archiving is of great value to both maintainers and consumers of RDF data. To the former, archives are the basis of version control [ANR+19], which opens the door not only to novel data processing tasks, e.g., mining of temporal and correction patterns [PTBS19], but also to temporal data analytics [RCS+15], [Bru10]. For data consumers, archives are a way to query the past and study the evolution of a given domain of knowledge [HBS13], [TS19], [AMH21].

From a technical point of view, building and maintaining RDF archives is a very challenging endeavor, primarily due to the massive size of current knowledge graphs. As of April 2022, DBpedia accounts for 220M entities and 1.45B facts[1], and changes from one release to the next one can be in the order of millions [PGH21]. However, large changesets are not the only issue that challenges state-of-the-art RDF archive systems. For instance, DBpedia Live[2] receives continuous updates with changes made by the Wikipedia community. This dynamicity makes DBpedia's revision history extremely long, and exacerbates the challenges of managing an archive for a dataset of that nature. As shown in our experimental section, existing approaches for RDF archiving cannot ingest long histories on large datasets, even when the changes between revisions are small.

We therefore propose an approach to ingest, store, and query long revision histories on very large RDF graphs. Our techniques rely on a combination of dataset snapshots and sequences of aggregated changesets – called *delta chains* [TSH+19] in the literature. We evaluate our approaches on the BEAR benchmark [FUPK19] and show that our techniques can ingest the BEAR-B instant dataset in no more than 2 hours – something that so far has been beyond reach. Moreover, our techniques exhibit competitive runtimes for most types of queries on RDF archives. We implemented our approach on top of OSTRICH [TSH+19], a state-of-the-art engine for archiving large RDF graphs.

The remainder of this paper is structured as follows. Section II elaborates on the background concepts and the state of the art in RDF archiving. Our storage techniques are detailed in Section III, while Section IV explains how to query archives with multiple snapshots and delta chains. Section V provides details of our implementation. The viability of our techniques is evaluated in Section VI. Finally, Section VII concludes the paper and discusses future work.

II. Background and Related Work

A. RDF Graphs and RDF Archives

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 [RS14]. RDF graphs are queried using SPARQL [SH13], whose building blocks are *triple patterns*, i.e., triples that allow variables (prefixed with a '?') in any position, e.g., $\langle$ *?x*, *cityIn*, *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 history of updates. This can be formalized as $\mathcal{A} = \{G_0, \ldots, G_k\}$, with G_i being the graph at version (or revision) $i \in \mathbb{Z}_{\geq 0}$. The transition from G_{i-1} to version G_i is implemented via 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

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

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* $\rangle$.

B. Querying RDF Archives

The literature identifies five types of queries over RDF archives [FUPK19], [TSH+19]. We explain them next and provide examples with a single triple pattern for simplicity.

- **Version Materialization (VM).** These are standard SPARQL queries run against a single version i, e.g., $\langle$ *?s*, *type*, *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*, *type*, *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*, *type*, *Country*, *?v* $\rangle$, which returns pairs $\langle$ *country*, *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 suffices for applications relying on RDF archives.

C. Solutions for RDF Archive Management

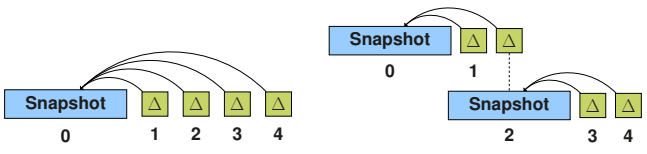
Several solutions have been proposed for managing the history of RDF graphs efficiently. We review the most prominent approaches in this section and refer the reader to [PGH21] for a detailed survey.

In the literature, RDF archive approaches are typically categorized according to their storage architecture. We distinguish three major design paradigms: independent copies (IC), change-based solutions (CB), and timestamp-based systems (TB). IC approaches, such as [VWS+05], 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, they are impractical for today's knowledge graphs due to their prohibitive storage footprint. This fact has shifted the research trend towards CB and TB systems. In a CB solution, some versions are stored as *changesets* (also called deltas) w.r.t. a previous reference version stored as a snapshot. We call a sequence of changesets – representing an arbitrary sequence of versions – and its corresponding reference revision, a *delta chain*. CB approaches require less disk space than IC architectures and are optimal for DM queries – at the expense of efficiency for VM queries. This makes them particularly attractive for version-control systems, e.g., [GHU14], [ANR+19], where changesets are rather small and frequent. TB solutions, on the other hand, optimize for V queries as they store temporal metadata, such as validity intervals or insertion/deletion timestamps [NW10] in specialized indexes.

Recent approaches borrow inspiration from more than one paradigm. QuitStore [ANR+19], 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 [TSH+19] combines the advantages of CB and TB approaches in a *single* delta chain; an initial snapshot stores revision 0, whereas a new revision i is built from a changeset of the form $u_{i-1,i}$ and stored as an aggregated changeset $u_{0,i}$, i.e. the changes between the snapshot to i. This storage architecture is depicted in Figure 1a. OSTRICH supports VM, DM, and V queries on single triple patterns natively. CV, CD, and arbitrary SPARQL queries can be executed by connecting OSTRICH to a query engine. Aggregated changesets have been shown to speed up VM and DM queries significantly w.r.t. a standard CB approach. As shown in [PGH21], [TSH+19], OSTRICH is the only 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 [PGH21], [TMVV22] – even when the original changesets are small. In this paper, we build upon OSTRICH and propose a solution to this problem.

III. Storing Archives with Multiple Delta Chains

As discussed in Section II, 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 1a). 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 1b.



(a) Single delta chain (b) Multiple delta chains

Fig. 1: Delta chain architectures

A. Delta Chains

While creating a fresh snapshot with a new delta chain should presumably reduce ingestion time for subsequent revisions, its impact on query efficiency seems mixed. V queries, for instance, 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 1b. 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 perspectives of storage, ingestion time, and query efficiency next. We then explain how to query archives in a multi-snapshot setting in Section IV.

B. 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* $f : \mathbb{A} \times \mathbb{U} \to \{0, 1\}$ 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 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 I). All strategies start with a snapshot at revision 0. Note that we do not provide an exhaustive list of all possible strategies 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 [TSH+19].

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

Change-ratio. Long delta chains do 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 [FUPK19]:

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

Given two revisions i and j, the change ratio normalizes the number of changes (additions and deletions) between the revisions by their joint size. If we aggregate the change ratios of all the revisions coming after a snapshot revision s, we can estimate the level of redundancy in the current delta chain. A reasonable snapshot strategy would therefore bound $\sum_{i=s+1}^{k} \delta_{s,i}$, put differently: $f(\mathcal{A}, u) \equiv (\gamma \leq \sum_{i=s+1}^{k} \delta_{s,i})$ 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 takes θ times longer than the ingestion 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
$\sum_{i=s+1}^{k} \delta_{s,i}$	-	0.23	0.61	1.08	0.19	0.51
t_k	-	1.00	1.50	2.25	3.38	1.0
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 I: 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.

IV. Querying 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. As is common in other RDF archiving approaches [TSH+19], we focus our algorithms on answering single triple patterns queries, since they constitute the building blocks for more complex query answering, which is outside the scope of this work. All the routines described next are defined w.r.t. to an implicit RDF archive $\mathcal{A}$.

A. 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, $s = 0$; in the presence of multiple delta chains $s = \textit{snapshot}(i)$ corresponds to i's reference snapshot in the archive's history. Our implementation relies on OSTRICH, which can efficiently compute G_i and run queries on top of it.

B. DM Queries

The procedure *queryDM* in Algorithm 1 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. The first one, denoted *deltaDiff*, executes standard DM queries on single triple patterns over a single delta chain as proposed by OSTRICH [TSH+19]. The second routine, called *snapshotDiff*, computes the difference between the results of p on two reference snapshots S_i, S_j. It works by first testing if the delta chains of S_i and S_j are not consecutive (line 2). 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 III-A). We can therefore treat this case efficiently as a standard DM query via *deltaDiff* (line 7).

Algorithm 1 DM query algorithm

```
1: function SNAPSHOTDIFF(S_i, S_j, p)
                              ▷ snapshots S_i, S_j, triple pattern p
2:     d ← distance(i, j)
3:     if d > 1 then
4:         q_i ← query(S_i, p); q_j ← query(S_j, p)
5:         delta ← (q_j \ q_i) ∪ (q_i \ q_j)
6:     else
7:         delta = deltaDiff(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 ← snapshot(i); sid_j ← snapshot(j)
14:     if sid_i = sid_j then  ▷ i and j in the same delta-chain
15:         delta ← deltaDiff(i, j, p)
16:     else ▷ i and j not in the same delta-chain
17:         u_{si,sj} ← snapshotDiff(sid_i, sid_j, p)
18:         u_{si,i}, u_{sj,j} ← ∅
19:         if i ≠ sid_i then  ▷ test if version i is a delta
20:             u_{si,i} ← deltaDiff(sid_i, i, p)
21:         end if
22:         if j ≠ sid_j then  ▷ test if version j is a delta
23:             u_{sj,j} ← deltaDiff(sid_j, j, p)
24:         end if
25:         u_{i,sj} ← mergeBackwards(u_{si,i}, u_{si,sj})
26:         delta ← mergeForward(u_{i,sj}, u_{sj,j})
27:     end if
28:     return delta
29: end function
```

We now have the elements to explain the main DM query procedure (*queryDM*). First, it 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 *deltaDiff* (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 ds.

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, merges 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., $u_{sj,sj}$. 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 revertion operations.

C. V Queries

Algorithm 2 V query algorithm

```
1: function QUERYV(p)  ▷ p a triple pattern
2:     r ← ∅
3:     for c ∈ C do  ▷ C the list of delta chains
4:         v ← singleQueryV(c, p)
5:         r ← merge(r, v)  ▷ merge intermediate results
6:     end for
7:     return r
8: end function
```

Algorithm 2 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 implemented in OSTRICH [TSH+19] via the function *singleQueryV*. 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.

V. Implementation

We implemented the proposed snapshot creation strategies and query algorithms for RDF archives on top of OSTRICH [TSH+19]. We briefly explain the most important aspects of our implementation.

Storage. An RDF archive consists of a snapshot for revision 0 and a single delta chain of aggregated changesets for the upcoming revisions (Fig. 1a). The snapshot is stored as an HDT [FMG+13] 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., ⟨ *?s*, *?p*, *?o* ⟩ or ⟨ *?s*, *cityIn*, *?o* ⟩, which can be used to efficiently compute cardinality estimations – particularly useful for SPARQL engines.

Dictionary. As common in RDF stores [NW10], [WKB08], 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 $s = \textit{snapshot}(k)$ 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 this paper are available in a Zenodo archive[3].

VI. Experiments

To determine the effectiveness of our multi-snapshot approach for RDF archiving, we evaluate the four proposed snapshot creation strategies along three dimensions: ingestion time (Section VI-B), disk usage (Section VI-C), and query runtime for VM, DM, and V queries (Section VI-D).

A. Experimental Setup

We resort to the BEAR benchmark for RDF archives [FUPK19] 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. We omit BEAR-C from our experiments because its query load consists of full SPARQL queries and diverse constructs, which are not supported by our implementation, nor by any other RDF archiving approaches. Table II 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 21046 revisions) after one month of execution. 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 [TSH+19] omitted BEAR-B instant and included only the first 10 versions of BEAR-A. Multi-snapshot solutions, on the other hand, allow us to manage these 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.

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

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

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

We evaluate the different strategies for snapshot creation detailed in Section III-B 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, $d \in \{5, 10\}$ 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.

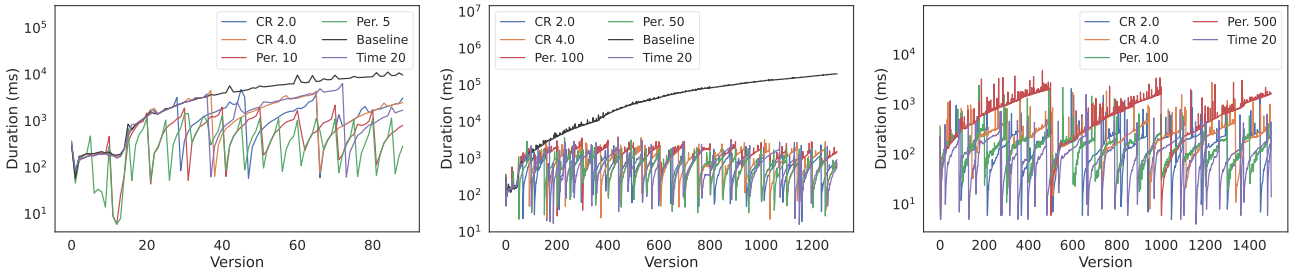
Change-ratio (CR). This strategy depends on a cumulative change-ratio budget threshold γ. We set $\gamma \in \{2.0, 4.0\}$ for all the tested datasets. $\gamma = 2.0$ yields 10 delta chains for BEAR-A, as well as 5, 23, and 151 delta chains for BEAR-B daily, hourly, and instant, respectively. For $\gamma = 4.0$, we obtain instead 6 delta chains for BEAR-A, and 3, 16, and 98 for the BEAR-B alternatives.

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.

We omit the reference systems included with the BEAR benchmark since they are outperformed by OSTRICH [TSH+19].

B. Ingestion Time

Table III 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 III and IV, 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-A and BEAR-B instant in a reasonable amount of time. This explains their absence in Table III. But even when OSTRICH can ingest the entire history (in less than 26 hours), a multi-snapshot strategy still incurs a significant speed-up. This becomes more significant for long histories as observed for BEAR-B hourly, where the speed-up can reach two orders of magnitude. The



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

Fig. 2: 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.

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

TABLE III: Ingestion times in minutes

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

TABLE IV: Disk usage in MB

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. 2, where we also notice that ingestion time reaches a minimum for the revisions following a snapshot.

C. Disk Usage

Unlike ingestion time where shorter delta changes are clearly beneficial, the gains in terms of disk usage need fine-grained tuning because they depend on the dataset as shown in Table IV. 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 the time strategy with $\theta = 20$, despite using more delta chains. This hints that very large aggregated deltas are also inefficient compared to 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 slight worse because it creates too many delta chains.

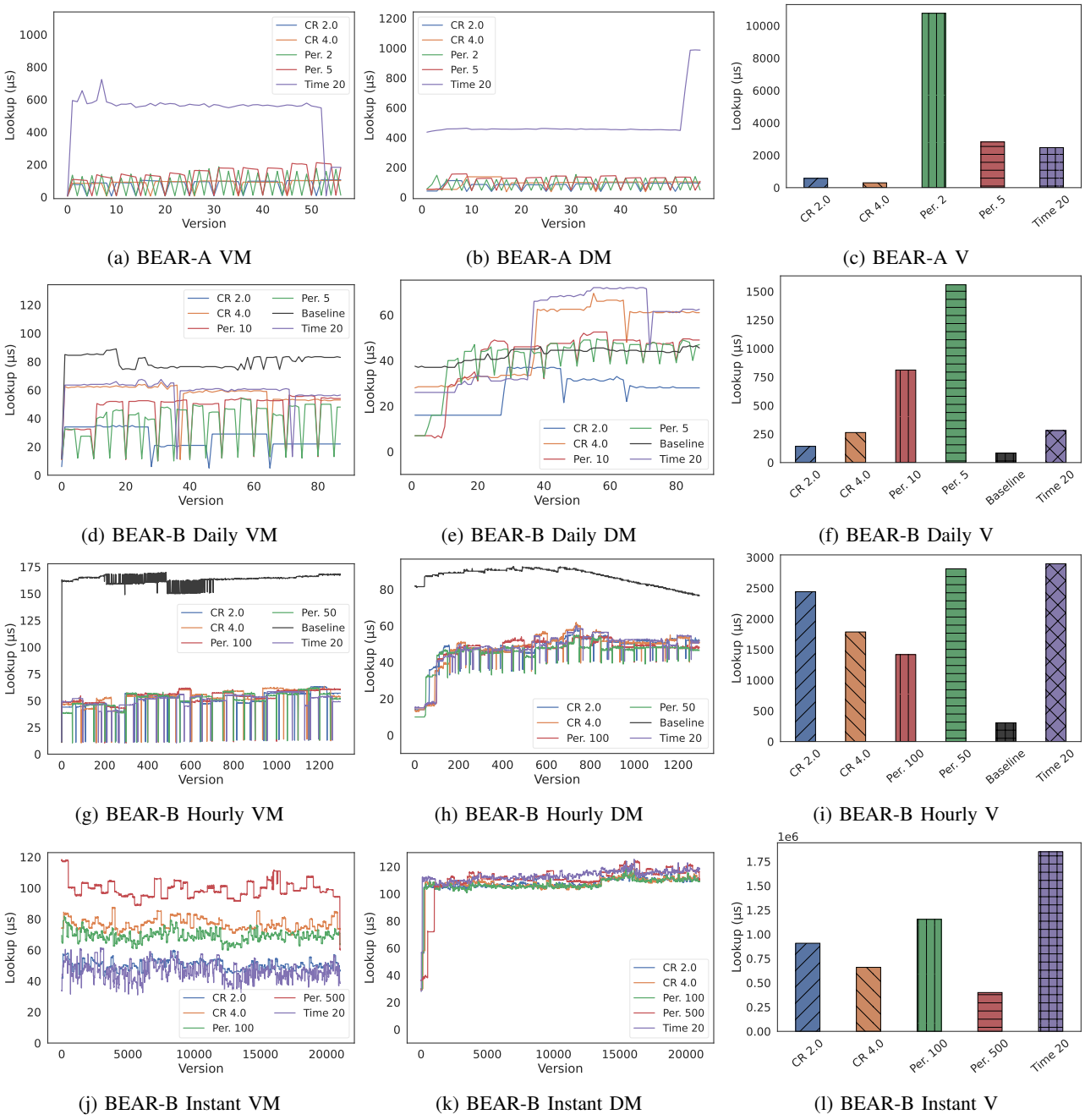
D. Query Runtime

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 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 3a, 3d, 3g, and 3j. 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 was 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 time $\theta = 20$ strategy, which splits the history into two imbalanced delta-chains, where one of them contains the first 53 revisions (out of 58).

2) DM Queries: We report for each revision i in the archive the average runtime of the benchmark DM queries between revisions $\langle\ 0, i\ \rangle$ and $\langle\ 1, i\ \rangle$. 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 3b, 3e, 3h, and 3k. The results shows a rather mixed benefit of multiple delta chains in query runtime: highly positive for the long history of BEAR-B hourly and negligible for BEAR-B daily. Overall, DM queries benefit from short delta chains as illustrated by Figure 3b and to a lesser degree by the periodic strategy with $d = 5$ in Figure 3e. 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 tends to diverge more, which requires the engine to filter fewer results from



(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. 3: Query results for the BEAR benchmark

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.

3) V Queries: Figure 3c, 3f, 3i, and 3l 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 3f and 3i. As described in Algorithm 2, 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. Nonetheless, querying datasets such as BEAR-A and BEAR-B instant is only possible with a multi-snapshot solution.

E. Discussion

We now summarize our findings and draw a few design lessons for RDF archives.

- 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.
- While many short delta chains are detrimental to V queries and often to storage consumption, they are mostly benefitial for VM and DM queries because these query types require us to iterate over changes within delta chains (two in the worst case of DM queries). Moreover, short delta chains reduce ingestion time systematically.
- Disk usage usually benefit from less numerous delta chains, except for long change history and large aggregated deltas.
- 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.

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 [PGH21]) 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.

VII. Conclusion

In this paper we have presented a hybrid storage approach for RDF archiving based on multiple snapshots and chains of aggregated deltas. We studied different snapshot creation strategies and discussed the trade-offs in terms of ingestion time, storage size, and query runtime. Our experimental evaluation shows that our techniques allow us to handle very long revision histories that could not be managed by previous approaches. Moreover, we drew a set of design lessons for RDF archive design that can help users decide the best strategy based on the application scenario. As future work, we plan to develop more complex snapshot strategies, e.g., based on machine learning. Moreover, the development of more efficient encoding and serialization techniques for timestamped deltas is a promising research avenue to further lower storage size. We also plan to study the impact of our techniques on the performance of SPARQL query execution and consider improvements within the landscape of alternative RDF representations and indexing approaches [SLPH22].

Acknowledgments

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B and the Poul Due Jensen Foundation. Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1274521N).

References

[AMH21] Christian Aebeloe, Gabriela Montoya, and Katja Hose. Colchain: Collaborative linked data networks. In *WWW*, pages 1385–1396, 2021.

[ANR+19] Natanael Arndt, Patrick Naumann, Norman Radtke, Michael Martin, and Edgard Marx. Decentralized collaborative knowledge management using git. *J. Web Semant.*, 54:29–47, 2019.

[Bru10] Jörg Brunsmann. Archiving Pushed Inferences from Sensor Data Streams. In *International Workshop on Semantic Sensor Web*, pages 38–46, 2010.

[FMG+13] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange (HDT). *J. Web Semant.*, 19:22–41, 2013.

[FUPK19] Javier D. Fernández, Jürgen Umbrich, Axel Polleres, and Magnus Knuth. Evaluating query and storage strategies for RDF archives. *J. Web Semant.*, 10(2):247–291, 2019.

[GHU14] Markus Graube, Stephan Hensel, and Leon Urbas. R43ples: Revisions for triples - an approach for version control in the semantic web. In *LDQ@SEMANTICS*, 2014.

[HBS13] Thomas Huet, Joanna Biega, and Fabian M. Suchanek. Mining History with Le Monde. In *Workshop on Automated Knowledge Base Construction*, pages 49–54, 2013.

[NW10] Thomas Neumann and Gerhard Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1):256–263, 2010.

[PGH21] Olivier Pelgrin, Luis Galárraga, and Katja Hose. Towards fully-fledged archiving for RDF datasets. *J. Web Semant.*, 12(6):903–925, 2021.

[PTBS19] Thomas Pellissier Tanon, Camille Bourgaux, and Fabian Suchanek. Learning How to Correct a Knowledge Base from the Edit History. In *WWW*, pages 1465–1475, 2019.

[RCS+15] Yannis Roussakis, Ioannis Chrysakis, Kostas Stefanidis, Giorgos Flouris, and Yannis Stavrakas. A flexible framework for understanding the dynamics of evolving RDF datasets. In *ISWC*, volume 9366, pages 495–512, 2015.

[RS14] Yves Raimond and Guus Schreiber. RDF 1.1 primer. W3C recommendation, 2014. http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[SH13] Andy Seaborne and Steven Harris. SPARQL 1.1 query language. W3C recommendation, W3C, 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[SLPH22] Tomer Sagi, Matteo Lissandrini, Torben Bach Pedersen, and Katja Hose. A design space for RDF data representations. *VLDB J.*, 31(2):347–373, 2022.

[TMVV22] Ruben Taelman, Thibault Mahieu, Martin Vanbrabant, and Ruben Verborgh. Optimizing storage of RDF archives using bidirectional delta chains. *J. Web Semant.*, 13:705–734, 2022.

[TS19] Thomas Pellissier Tanon and Fabian M. Suchanek. Querying the edit history of wikidata. In *ESWC*, volume 11762, pages 161–166, 2019.

[TSH+19] Ruben Taelman, Miel Vander Sande, Joachim Van Herwegen, Erik Mannens, and Ruben Verborgh. Triple Storage for Random-access Versioned Querying of RDF Archives. *J. Web Semant.*, 54:4–28, 2019.

[VWS+05] Max Volkel, Wolf Winkler, York Sure, Sebastian Ryszard Kruk, and Marcin Synak. SemVersion: A Versioning System for RDF and Ontologies. *ESWC*, 2005.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.