Versioned Queries over RDF Archives: All You Need is SPARQL?

Ignacio Cuevas and Aidan Hogan

Department of Computer Science, University of Chile & IMFD Chile

Abstract. We explore solutions for representing archives of versioned RDF data using the SPARQL standard and off-the-shelf engines. We consider six representations of RDF archives based on named graphs, and describe how input queries can be automatically rewritten to return solutions for a particular version, or solutions that change between versions. We evaluate these alternatives over an archive of 8 weekly versions of Wikidata and 146 queries using Virtuoso as the SPARQL engine.

1 Introduction

A key aspect of the Web is its dynamic nature, where documents are frequently updated, deleted and added. Likewise when we speak of the Semantic Web, it is important to consider that sources may be dynamic and RDF datasets are subject to change [11]. It is in this context that various works have looked at versioning in the context of RDF/SPARQL [25,23,7,8,12], with recent works proposing *RDF archives* [5,3,2,6,20] that manage RDF graphs and their historical changes, allowing for querying across different versions of the graph. Within these works, a variety of specialised indexing techniques [3,2,20], query languages [23] and benchmarks [13,6] have been proposed, developed and evaluated. While these represent important advances, many such works propose custom SPARQL extensions, indexes, engines, etc., creating a barrier for adoption.

In fact, versioned queries as proposed in the literature [5] can be supported using off-the-shelf SPARQL engines with years of development, optimisation, and deployment. SPARQL named graphs can, for example, be used to track different versions of individual graphs. However, as Fernandez at al. [6] note, the approach of using pure SPARQL would "*typically render rather inefficient SPARQL queries*". This raises a question: how inefficient will such queries be? If a pure SPARQL solution could be found with reasonable performance, existing SPARQL engines could be used to host and query RDF archives.

In this paper, we present preliminary empirical results addressing this research question. Specifically we look at six representations of RDF archives using named graphs and propose query rewriting mechanisms for them. We then evaluate and compare these representations for an RDF archive of 8 versions of Wikidata [26]. Our experiments compare the sizes of the indexes generated, the time taken for indexing, and the relative costs of query evaluation.

2 Related Work

Various temporal extensions for RDF have been proposed in literature based on annotations [9,19,29]. Proposed temporal extensions for SPARQL include τ-SPARQL [23], T-SPARQL [7], SPARQL-ST [17], SPARQL$^{\mathrm{T}}$ [27], etc. Related to temporality, a number of systems support versioning for RDF, including SemVersion [25], POI [24], x-RDF-3x [14], R43ples [8], Dydra [1] and Ostrich [21].

More recently *RDF archives* (of historical RDF data) have been gaining attention. Fernandez et al. [5] survey the theme, discussing the types of queries that can be run on such archives. Cerdeira-Pena et al. [3], Zaniolo et al. [27] and Taelman et al. [20] propose compressed indexes for RDF archives, while Khurana and Deshpande [12] propose indexes for historical graph data. Bahri et al. [2] use Apache Spark to manage RDF archives in a distributed setting. Benchmarks have also been proposed for RDF archives, including the BEnchmark of RDF ARchives (BEAR) [6], and the Semantic Publishing Benchmark (SPB) [15].

The past years have seen many developments for managing and querying temporal, versioned or historical RDF data. But most of these approaches propose specialised languages, implementations, etc., creating an obstacle for adoption. A number of authors note that one can manage and query RDF archives using vanilla SPARQL, though it may lead to complex or inefficient queries [23,6]. Recently SPARQL has been used to host and query the edit history of Wikidata, but only one representation is explored [22]. This paper describes preliminary experiments to gain insights into the efficiency of off-the-shelf SPARQL engines for hosting RDF archives using different types of representations.

3 Preliminaries

RDF triples are composed of three sets of terms: IRIs ($\mathbf{I}$), literals ($\mathbf{L}$) and blank nodes ($\mathbf{B}$). We do not consider blank nodes in this work as they complicate the detection of changes [28]. An RDF triple $(s,p,o) \in \mathbf{I} \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{L})$ consists of a subject s, predicate p and object o. An RDF graph G is a set of triples. An RDF archive is a tuple of RDF graphs $\mathcal{G} = (G_1, \ldots, G_n)$.

A triple pattern $t \coloneqq (s,p,o) \in (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ is an RDF triple that permits variables from $\mathbf{V}$ to appear in any position. We denote by $\mathrm{vars}(t)$ the variables of t. A solution is a partial mapping $\mu : \mathbf{V} \to \mathbf{I} \cup \mathbf{L}$. We denote by $\mathrm{dom}(\mu)$ the *domain* of μ, i.e., the set of variables for which μ is defined. We say that two solutions μ_1, μ_2 are compatible, denoted $\mu_1 \sim \mu_2$, if and only if $\mu_1(v) = \mu_2(v)$ for all $v \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$. We denote by $\mu(t)$ the image of t under μ, replacing each variable $v \in \mathrm{dom}(\mu) \cap \mathrm{vars}(t)$ by $\mu(v)$ in t. We denote by $t(G) \coloneqq \{\mu \mid \mu(t) \subseteq G \text{ and } \mathrm{dom}(\mu) = \mathrm{vars}(t)\}$ the evaluation of t over G.

SPARQL queries are based on triple patterns and a number of relational operators. Similar to Pérez et al. [16], we define an abstract syntax for a subset of SPARQL of pertinence to this paper as follows. A triple pattern t is a graph pattern. Furthermore, if P and Q are graph patterns, and $V \subset \mathbf{V}$ is a set of

$$\begin{array}{ll}
[P \text{ AND } Q](G) \coloneqq P(G) \bowtie Q(G) & M_1 \bowtie M_2 \coloneqq \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1,\ \mu_2 \in M_2,\ \mu_1 \sim \mu_2\} \\
[P \text{ UNION } Q](G) \coloneqq P(G) \cup Q(G) & M_1 \cup M_2 \coloneqq \{\mu \mid \mu \in M_1 \text{ or } \mu \in M_2\} \\
[P \text{ MINUS } Q](G) \coloneqq P(G) \setminus Q(G) & M_1 \setminus M_2 \coloneqq \{\mu_1 \in M_1 \mid \nexists \mu_2 \in M_2 : \mu_1 \sim \mu_2\}
\end{array}$$

Fig. 1. Semantics for query operators where M, M_1 and M_2 denote sets of solutions

variables, then $[P \text{ AND } Q]$, $[P \text{ UNION } Q]$ and $[P \text{ MINUS } Q]$ are graph patterns. We define the semantics of these graph patterns in Figure 1.[1]

In this paper, we rely on SPARQL datasets to represent RDF archives. A SPARQL dataset $D \coloneqq \{G, (x_1, G_1), \ldots (x_n, G_n)\}$ consists of a default (RDF) graph G and a set of named graphs of the form (x_i, G_i) where $x_i \in \mathbf{I}$, G_i is an RDF graph, and $x_i \neq x_j$ for $1 \leq i \leq m$, $1 \leq j \leq m$, $i \neq j$. We may represent D as a set of quads of the form $(G \times \{*\}) \cup (G_1 \times \{x_1\}) \cup \ldots \cup (G_n \times \{x_n\})$, where $* \notin \mathbf{I} \cup \mathbf{L} \cup \mathbf{V}$ is a special symbol denoting the default graph.[2] Different named graphs can be queried using a `GRAPH` operator, creating quad patterns. A quad pattern $q = (s, p, o, g) \in (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \{*\})$ extends a triple pattern with a fourth element that may be an IRI or $*$. Its evaluation is analogous to that of a triple pattern: $q(G) \coloneqq \{\mu \mid \mu(q) \subseteq G \text{ and } \mathrm{dom}(\mu) = \mathrm{vars}(q)\}$. Following the SPARQL standard [10], we translate a triple pattern (s, p, o) to a quad pattern $(s, p, o, *)$ evaluated only on the default graph. The semantics of the operators defined in Table 1 then remain unchanged simply allowing P and Q to now also represent quad patterns, considered to be (named) graph patterns.

SPARQL provides two operators to initialise a SPARQL dataset: `FROM` and `FROM NAMED`. We combine both for brevity into one operator. A graph pattern P is considered to be a query. Likewise if P is a graph pattern, and M and N are sets of IRIs, then $\text{FROM}_{M,N}P$ is also a query. If $(x_i, G_i) \in D$ let $D(x_i) = G_i$; otherwise if no graph is named x_i in D, let $D(x_i) = \{\}$. The evaluation of $\text{FROM}_{M,N}P$ on a dataset D is defined as $\text{FROM}_{M,N}P(D) \coloneqq P(D_{M,N})$, where the query dataset $D_{M,N} \coloneqq (\cup_{m \in M} D(m) \times \{*\}) \cup (\cup_{n \in N} D(n) \times n)$ is formed from a default graph containing the union[3] of graphs with a name $m \in M$, and all named graphs in D with a name $n \in N$. We will be able to use these operators to define query datasets that capture specific versions in an RDF archive.

4 Versioned Data

Our general approach is to represent an RDF archive as a SPARQL dataset D but there are multiple representations by which this can be achieved, each with its own strength and weaknesses. We propose six different representations falling into three different categories as discussed by various authors (e.g., [24,5]): *Independent Copies* (*IC*), *Change-Based* (*CB*) and *Timestamp-Based* (*TB*).

[1] A basic graph pattern $\{t_1, \ldots t_n\}$ can be represented as $[[t_1 \text{ AND } \ldots] \text{ AND } t_n]$.

[2] We thus assume a *quad store*, and disallow empty named graphs.

[3] Since we do not allow blank nodes, a union or RDF merge is equivalent.

Independent Copies (IC): A natural representation is to store an RDF Archive $\mathcal{G} = (G_1, \ldots, G_n)$ as a SPARQL dataset $D = \{(x_1, G_1), \ldots, (x_n, G_n)\}$, where $x_1, \ldots, x_n$ are IRIs that identify the version. This will result in relatively simple (and thus likely efficient) rewritten queries, but can be expected to occupy a lot of space, particularly where few triples change from version to version.

Change-Based (CB): The core idea of CB representations is to store only triples that change from a given reference version. Along these lines, in the following we denote by $\varDelta_i^j \coloneqq G_i \setminus G_j$ the triples in version i not in version j. We consider four CB representations based on four transformations of $\mathcal{G} = (G_1, \ldots, G_n)$:

$$\begin{array}{rll}
\mathcal{G}_1^n \coloneqq (G_1, \varDelta_1^2, \varDelta_2^1, \ldots, \varDelta_1^n, \varDelta_n^1) & G_i = (G_1 \cup \varDelta_i^1) \setminus \varDelta_1^i & \text{for } 1 < i \leq n \\
\mathcal{G}_{n-1}^n \coloneqq (G_1, \varDelta_1^2, \varDelta_2^1, \ldots, \varDelta_{n-1}^n, \varDelta_n^{n-1}) & G_i = (G_{i-1} \cup \varDelta_i^{i-1}) \setminus \varDelta_{i-1}^i & \text{for } 1 < i \leq n \\
\mathcal{G}_n^1 \coloneqq (G_n, \varDelta_n^{n-1}, \varDelta_{n-1}^n, \ldots, \varDelta_n^1, \varDelta_1^n) & G_i = (G_n \cup \varDelta_i^n) \setminus \varDelta_n^i & \text{for } 1 \leq i < n \\
\mathcal{G}_n^{n-1} \coloneqq (G_n, \varDelta_n^{n-1}, \varDelta_{n-1}^n, \ldots, \varDelta_2^1, \varDelta_1^2) & G_i = (G_{i+1} \cup \varDelta_i^{i+1}) \setminus \varDelta_{i+1}^i & \text{for } 1 \leq i < n
\end{array}$$

As can be seen for the definitions of G_i, these transformations are lossless: we can retrieve any version of the graph from any such transformation. The first two transformations start with the earliest version as a base. The first encodes deltas always with respect to the first version. The second encodes deltas with respect to the previous version. The latter two transformations start with the latest version. The third encodes deltas with respect to the latest version. The forth encodes deltas with respect to the subsequent version. Letting $\mathcal{H} = (H_1, \ldots, H_n)$ such that $H_i = G_{n-i+1}$ $(1 \leq i \leq n)$, i.e., such that $\mathcal{H}$ "reverses" $\mathcal{G}$, we remark that $\mathcal{G}_1^n = \mathcal{H}_n^1$ and $\mathcal{G}_{n-1}^n = \mathcal{H}_n^{n-1}$. Each such transformation can then be represented as a SPARQL dataset with $2n - 1$ named graphs.

In terms of space we expect $\mathcal{G}_{n-1}^n$ and $\mathcal{G}_n^{n-1}$ to be the most efficient as they always encode deltas from a neighbouring version. However, in terms of query rewriting, we expect $\mathcal{G}_1^n$ and $\mathcal{G}_n^1$ to be more efficient as they do not require a recursive construction of all intermediate versions towards the base version. In terms of indexing a new version, we expect $\mathcal{G}_1^n$ followed by $\mathcal{G}_{n-1}^n$ to be most efficient as they require only computing the most recent deltas; we expect $\mathcal{G}_n^1$, followed by $\mathcal{G}_n^{n-1}$, to be much more expensive, requiring a recompute of all deltas. On the other hand, $\mathcal{G}_n^1$ and $\mathcal{G}_n^{n-1}$ should be advantageous for queries over more recent versions, and in particular over the most recent version (a common case).

These four representations are analogous to differential backups, incremental backups, reverse-differential backups, and reverse-incremental backups.

Timestamp-Based (TB): The intuition of the TB representation is to associate each triple with the versions in which it is contained. Along these lines, we denote by $\mathcal{G}(s, p, o) \coloneqq \{i \mid (s, p, o) \in G_i \text{ for } 1 \leq i \leq n\}$ the versions containing (s, p, o).[4] Let $\mathcal{N} \coloneqq \{N \mid \exists (s, p, o) \in G_i : \mathcal{G}(s, p, o) = N, 1 \leq i \leq n\}$ denote the family of sets of versions associated with some triple in (s, p, o). We can then represent the RDF archive with a named graph for each $N \in \mathcal{N}$. However, the number of named graphs can reach 2^n (or the number of unique triples in $\mathcal{G}$). Another

[4] The definition $\mathcal{G} : \mathbf{I} \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{L}) \to 2^{\{1,\ldots,n\}}$ [6] is analogous to $\mathcal{G} = (G_1, \ldots, G_n)$.

option is create a named graph for intervals [5]. More specifically, a triple (s,p,o) is added to an interval $[i,j]$ (for $1 \leq i \leq j \leq n$) if and only if $(s,p,o) \in G_k$ for $i \leq k \leq j$ and either $i = 1$ or $(s,p,o) \notin G_{i-1}$ and $j = n$ or $(s,p,o) \notin G_{j+1}$; in simpler terms, $[i,j]$ is a maximal contiguous interval of versions in which (s,p,o) appears (omitting empty intervals). The upper bound is now $n(n+1)/2$ named graphs, but triples may appear in multiple named graphs for different intervals.

In general, we expect the space to be similar to $\mathcal{G}^n_{n-1}$ and $\mathcal{G}^{n-1}_n$; in other words, quite good. However, as the number of versions n grows, $O(n^2)$ interval graphs need to be unioned in the worst-case to materialise a particular version; CB representations require processing $O(1)$ (in the case of $\mathcal{G}^n_1$ and $\mathcal{G}^1_n$) or $O(n)$ (in the case of $\mathcal{G}^n_{n-1}$ and $\mathcal{G}^{n-1}_n$) named graphs to materialise a particular version.

Notation: We denote IC by $\textsc{i}$; differential, incremental, reverse-differential and reverse-incremental CB by $\textsc{c}^+_d$, $\textsc{c}^+_i$, $\textsc{c}^-_d$ and $\textsc{c}^-_i$, resp.; and interval TB by $\textsc{t}$.

5 Versioned Queries

Given a SPARQL query Q over an RDF graph, we now describe automatic rewritings of Q to generate solutions for different versions. We first focus on rewritings of triple patterns and then generalise. We assume that version parameters are given via HTTP rather than extending the SPARQL syntax. For reasons of space we rather present examples in online material [4].

5.1 Single-Version Queries

A single version query returns $Q(G_v)$ for a specified version v. Our overall strategy is to use `FROM` to construct the graph of the version where possible, as is the case for all versions in $\textsc{i}$ and $\textsc{t}$; for G_1 in $\textsc{c}^+_d$, $\textsc{c}^+_i$; and for G_n in $\textsc{c}^-_d$, $\textsc{c}^-_i$. Otherwise we rewrite each individual triple pattern appearing in Q in order to ensure that it generates the same solutions as it would if evaluated over the graph of the selected version. We now provide more details for each representation

Independent Copies (IC) For $\textsc{i}$ we rewrite Q to $\textsc{from}_{\{\mathsf{v}\},\{\}}Q$, where v is the IRI that names the graph for version v

Change-Based (CB) Recalling the observation that forwards and reverse CB representations are analogous, for brevity we define the rewriting for the forwards direction ($\textsc{c}^+_d$, $\textsc{c}^+_i$) only; the reverse direction ($\textsc{c}^-_d$, $\textsc{c}^-_i$) follows naturally.

For the differential representation, we load the base version and the positive delta into the default graph and, for each triple pattern, we subtract the negative delta which is queried using a quad pattern. Thus for $\textsc{c}^+_d$ we rewrite Q to $\textsc{from}_{\{\mathsf{1},\mathsf{v1}\},\{\mathsf{1v}\}}Q'$, where $\mathsf{1}$, $\mathsf{v1}$, $\mathsf{1v}$ indicate the names of G_1, Δ^1_v and Δ^v_1, respectively; and Q' replaces each triple pattern $(s,p,o) \in Q$ with the named graph pattern $[(s,p,o,*)\,\textsc{minus}\,(s,p,o,\mathsf{1v})]$.

Unfortunately the incremental rewriting is more complex. A first idea would be to take the union of G_1 and all positive deltas $\Delta^1_2, \ldots, \Delta^{v-1}_v$ and then subtract the (union of the) negative deltas $\Delta^2_1, \ldots, \Delta^v_{v-1}$; unfortunately this would overlook triples that were removed from a version $1 < u < v$ but were added back in a later version $u < u' \leq v$ (and were not removed again in a version $u' < u'' \leq v$). Hence a recursive rewriting appears to be necessary. Let $Q_1 \coloneqq \text{FROM}_{\{1\},\{\}} Q$; then $Q_2 \coloneqq \text{FROM}_{\{1\},\{12,21\}} Q'_1$, where Q_1 is the result of replacing each triple pattern (s, p, o) in Q_1 by the named graph pattern $P_2 \coloneqq [[(s, p, o, *) \text{ UNION } (s, p, o, \texttt{21})] \text{ MINUS } (s, p, o, \texttt{12})]$. We can then apply this rewriting recursively: $Q_i \coloneqq \text{FROM}_{\{1\},\{12,\ldots,(\texttt{i-1})\texttt{i},21,\ldots,\texttt{i}(\texttt{i-1})\}} Q'_{i-1}$, where Q'_{i-1} replaces each named graph pattern P_{i-1} appearing in Q_{i-1} with the recursive pattern $P_i \coloneqq [[P_{i-1} \text{ UNION } (s, p, o, \texttt{i(i-1)})] \text{ MINUS } (s, p, o, \texttt{(i-1)i})]$.

This rewriting leads to complex queries. We thus optimise using additional features of SPARQL. To sketch the idea, our goal is then to make sure that each triple pattern only matches triples that appear in a base version and were not removed, or that appear in a positive delta Δ^i_j such that $1 < i < j \leq v$ and do not appear in a (later) negative delta Δ^l_k for $j \leq k < l \leq v$. In practice, for each delta Δ^i_j stored as a named graph `ji`, in offline processing, we index meta-data of the form $(\texttt{ji}, \texttt{ver}, j, \texttt{\$})$, and $(\texttt{ji}, \texttt{type}, \texttt{pos}, \texttt{\$})$ in the case that $i < j$ or $(\texttt{ji}, \texttt{type}, \texttt{neg}, \texttt{\$})$ in the case that $j < i$ ($\texttt{\$} \in \mathbf{I}$ is a reserved name for the meta-data graph). We can then check the aforementioned condition by using aggregation to find the maximum version of a negative delta less than or equals v in which the triple pattern matches, then filtering the base version or any positive delta earlier than this maximum version. While the resulting query is still quite complex, we found it to be more practical than the recursive rewriting.

Timestamp-Based (TB) Let `i:j` denote the name of the graph for the interval of versions $[i, j]$. We rewrite Q to the query $\text{FROM}_{\{I\},\{\}} Q$, where I is the set of IRIs naming intervals in which v is contained; formally: $I \coloneqq \{\texttt{i:j} \mid i \leq v \leq j\}$.

5.2 Delta-Version Queries

Given a query Q, a control version u and a target version v, delta-version queries give solutions in $Q(G_v) \setminus Q(G_u)$. The general strategy for rewriting is to construct a query $[Q_v \text{ MINUS } Q_u]$ where Q_v and Q_u are the respective single-version queries.

5.3 Other SPARQL Features

In the SPARQL algebra, only the evaluation of triple patterns and *property paths* (regular expressions that match arbitrary length paths in the graph) directly accept the graph as input. Hence, given a SPARQL query Q (over a default graph), if we can individually rewrite each triple pattern and (property) path pattern of Q to generate solutions for G_v, then the rewritten query Q' will generate precisely the solutions for G_v. We previously described this process for triple patterns. However, in SPARQL we cannot always express a property path over multiple named graphs in the query dataset. For example, consider a property path `:y+` indicating a path of one or more predicates `:y`, a positive integer $K \geq 1$, and two named graphs $(n_1, \{(c_{2k-2}, \texttt{:y}, c_{2k-1}) \mid 1 \leq k \leq K\})$ and $(n_2, \{(c_{2k-1}, \texttt{:y}, c_{2k}) \mid 1 \leq k \leq K\})$ such that there is path for `:y+` of length

$2K$ (edges) that “alternates” between both named graphs. A `GRAPH` clause with a variable would evaluate this path on each graph separately (we cannot bind the graph variable to two graph names in a single solution). Though we can use `FROM` over n_1 and n_2 to form a default graph for evaluating `:a+`, we can only do this once per query. We can rather use a join of $2K$ `GRAPH` clauses, but K is bounded by the data, not the query (nor the number of versions). Thus while we support property paths for single-version queries on I and T, and delta-version queries on I, we do not know how to support them in the other cases.

6 Experiments

We now perform experiments to address the following three research questions (Q1) Which of the six representations allow for better compression, more efficient indexing, and more efficient updates of a new version? (Q2) How do the query runtimes of compressed representations (*CB*, *TB*) compare with indexing complete versions (*IC*)? (Q3) Which representation works best overall?

Setting We address these questions for a Wikidata archive of 8 weekly truthy versions from 2017-08-09 with 1.506 billion triples, to 2017-09-27 with 1.924 billion triples. The RDF archive consists of 13.477 billion triple–version pairs. Each week 25–93 million triples are added, while 4–6 million triples are removed. We take Wikidata's example queries, defined by users[5], and translate Wikidata-specific features (e.g., the label service) to standard SPARQL. We further filter queries that feature federation to other endpoints, property paths (not supported by all representations), and qualifiers (not in truthy versions). The result is a test set of 146 SPARQL queries. We take Virtuoso as our SPARQL implementation. Runtimes are averaged over three runs. Query timeouts were set to 5 minutes. The machine used has 120GB of RAM and a standard SATA hard-disk.

Indexing We first look at the results of total index sizes for each representation. In Figure 2 we show the index sizes (GB) for each representation, the time taken (min.) to bulk load all versions in the representation, and the time taken (min.) to update a seven-version archive with the eighth version. We see that I has the largest index sizes, followed by C_d^+ and C_d^-, then T, and finally C_i^+ and C_i^-. The bulk load times correlate with index size, with I (thus) being by far the slowest. We see a similar trend in version updates, except that C_d^- is far slower than the other alternatives (even I) as the entire archive must be built from scratch.

Single version queries We apply the rewriting of our 146 SPARQL queries for each of our six representations in order to retrieve results for version 1, 5 and 8. We show the results as box-plots with log y-axis in Figure 3 (with the timeout as the maximum). Median times were generally in the range of 100–1000 ms, though 1–20 queries timed out in each experiment, affecting the mean (shown as a diamond). In terms of mean and median runtimes, I performs the best, with C_d^+, C_d^- and T also performing competitively across the different versions.

[5] https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples

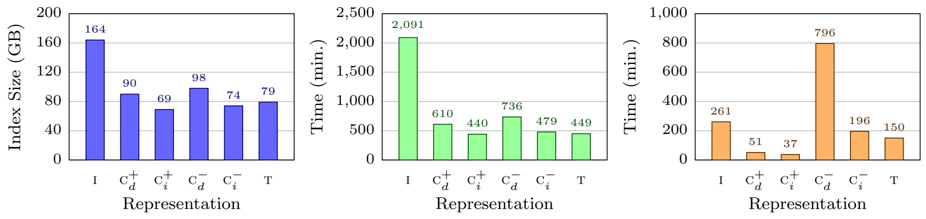

Fig. 2. Indexing details including size (left), bulk load (mid) and version update (right)

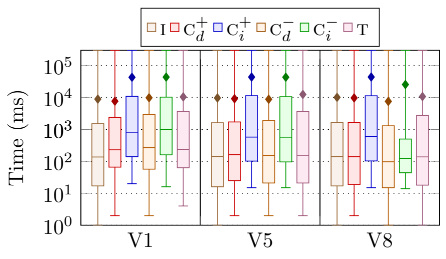

Fig. 3. Single version queries

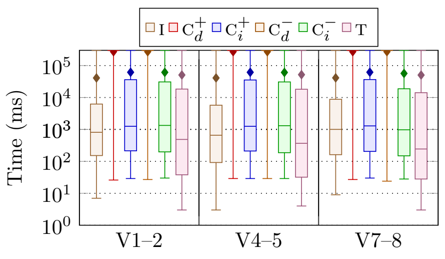

Fig. 4. Delta version queries

Conversely, C_i^+ and C_i^- perform poorly relative to the other options (except C_i^- in the case of version 8). Contrasting query times with index sizes, we note a clear time–space trade-off, where the largest index performs best, the smallest perform worst, and those with intermediate space perform middlingly.

Delta version queries Next we rewrite our 146 SPARQL queries in order to retrieve delta results between versions 1–2, 4–5 and 7–8 from each of our six representations. We again show the results as box-plots with log y-axis in Figure 4 (with the timeout as the maximum). When compared with single version queries, we see an increase in time, where the median runtimes of even the best performing representations approach or exceed 1000 ms more often. This time the best performance is offered by T, followed by I, C_i^+ and C_i^-. Conversely, C_d^+ and C_d^- perform very poorly. Of note is that incremental builds perform better than differential builds; we believe that this is due to the ability to cache smaller graphs, most of which are used to generate results for both versions.

7 Conclusion

We now reflect back on our research questions: (Q1) In terms of indexing space and time, incremental builds with an initial base version (C_d^+) are best. (Q2) In general the uncompressed IC representation (I) offers the best query runtimes, but interval TB (T) is quite competitive, and even outperforms IC for delta-version queries. (Q3) Rather than there being an overall winner, we note a time–space tradeoff, where less compact representations have faster queries and more compact representations have slower queries. Interval TB (T) arguably strikes the best balance for space and time, though this may not hold with more versions, particularly in RDF archives where triples are often added or removed multiple times, as a quadratic number of intervals may need to be accessed.

For future work, it would be of interest to run experiments for other SPARQL engines and other RDF archive benchmarks. Also it would be interesting to run more diverse types of versioned queries, such as delta versions with larger gaps, queries returning versions as solutions, etc.; a related direction would be to consider operators from temporal logics [18]. There are also open questions relating to more optimal/concise query rewritings, and support for paths.

In conclusion: for those who wish to host RDF archives, with different versions of an RDF graph, is SPARQL all you need? Specialised languages and systems can offer more features and consume less time and space. But with some caveats, our results suggest that query rewriting over an off-the-shelf SPARQL engine can be a solid (easy-to-deploy) option for such scenarios.

Online material Please see [4] for code and queries.

Acknowledgements This work was funded by Fondecyt Grant No. 1181896 and ANID Millennium Science Initiative Program ICN17_002.

References

1. J. Anderson and A. Bendiken. Transaction-time queries in dydra. In *Managing the Evolution and Preservation of the Data Web (MEPDaW)*, pages 11–19, 2016.
2. A. Bahri, M. Laajimi, and N. Y. Ayadi. Distributed RDF Archives Querying with Spark. In *European Semantic Web Conference (ESWC)*, pages 451–465, 2018.
3. A. Cerdeira-Pena, A. Fariña, J. D. Fernández, and M. A. Martínez-Prieto. Self-indexing RDF archives. In *Data Compression Conference (DCC)*, pages 526–535, 2016.
4. I. Cuevas. Online Material, 2020. `https://github.com/HunterNacho/sparql-versioning/`.
5. J. D. Fernández, A. Polleres, and J. Umbrich. Towards Efficient Archiving of Dynamic Linked Open Data. In *DIACHRON Managing the Evolution and Preservation of the Data Web*, pages 34–49, 2015.
6. J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth. Evaluating query and storage strategies for RDF archives. *Semantic Web*, 10(2):247–291, 2019.
7. F. Grandi. T-SPARQL: A tsql2-like temporal query language for RDF. In *Local Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems*, pages 21–30, 2010.
8. M. Graube, S. Hensel, and L. Urbas. R43ples: Revisions for Triples - An Approach for Version Control in the Semantic Web. In *Linked Data Quality (LDQ)*, 2014.
9. C. Gutierrez, C. A. Hurtado, and A. A. Vaisman. Introducing time into RDF. *IEEE Trans. Knowl. Data Eng.*, 19(2):207–218, 2007.
10. S. Harris, A. Seaborne, and E. Prud'hommeaux, editors. *SPARQL 1.1 Query Language*. 21 March 2013. Available at `http://www.w3.org/TR/sparql11-query/`.
11. T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan. Observing linked data dynamics. In *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings*, pages 213–227, 2013.

12. U. Khurana and A. Deshpande. Storing and Analyzing Historical Graph Data at Scale. In *International Conference on Extending Database Technology (EDBT)*, pages 65–76, 2016.
13. V. Kotsev, N. Minadakis, V. Papakonstantinou, O. Erling, I. Fundulaki, and A. Kiryakov. Benchmarking RDF Query Engines: The LDBC Semantic Publishing Benchmark. In *Benchmarking Linked Data (BLINK)*, 2016.
14. T. Neumann and G. Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *Proc. VLDB Endow.*, 3(1):256–263, 2010.
15. V. Papakonstantinou, I. Fundulaki, and G. Flouris. Assessing Linked Data Versioning Systems: The Semantic Publishing Versioning Benchmark. In *Scalable Semantic Web Knowledge Base Systems (SSWS)*, pages 45–60, 2018.
16. J. Pérez, M. Arenas, and C. Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.
17. M. Perry, P. Jain, and A. P. Sheth. SPARQL-ST: extending SPARQL to support spatiotemporal queries. In *Geospatial Semantics and the Semantic Web - Foundations, Algorithms, and Applications*, pages 61–86, 2011.
18. A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, pages 46–57. IEEE Computer Society, 1977.
19. A. Pugliese, O. Udrea, and V. S. Subrahmanian. Scaling RDF with time. In *World Wide Web Conference (WWW)*, pages 605–614, 2008.
20. 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.*, 54:4–28, 2019.
21. R. Taelman, M. V. Sande, and R. Verborgh. OSTRICH: Versioned Random-Access Triple Store. In *Comp. of The Web Conference*, pages 127–130, 2018.
22. T. P. Tanon and F. M. Suchanek. Querying the Edit History of Wikidata. In *ESWC Satellite Events*, pages 161–166. Springer, 2019.
23. J. Tappolet and A. Bernstein. Applied temporal RDF: efficient temporal querying of RDF data with SPARQL. In *Extended Semantic Web Conference (ESWC)*, pages 308–322, 2009.
24. Y. Tzitzikas, Y. Theoharis, and D. Andreou. On Storage Policies for Semantic Web Repositories That Support Versioning. In *European Semantic Web Conference (ESWC)*, volume 5021, pages 705–719, 2008.
25. M. Völkel and T. Groza. SemVersion: An RDF-based ontology versioning system. In *IADIS Conference: WWW/Internet*, volume 2006, 2006.
26. D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.
27. C. Zaniolo, S. Gao, M. Atzori, M. Chen, and J. Gu. User-friendly temporal queries on historical knowledge bases. *Inf. Comput.*, 259(3):444–459, 2018.
28. D. Zeginis, Y. Tzitzikas, and V. Christophides. On computing deltas of RDF/S knowledge bases. *TWEB*, 5(3):14:1–14:36, 2011.
29. A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia. A general framework for representing, reasoning and querying with annotated Semantic Web data. *J. Web Sem.*, 11:72–95, 2012.