Accepted version of the article published in: Knowledge and Information Systems, 2024, vol. 66, n. 1. p. 381-417. https://doi.org/10.1007/S10115-023-01967-7

Compressed and Queryable Self-Indexes for RDF Archives *

Ana Cerdeira-Pena[1,2], Guillermo de Bernardo[1,2*], Antonio Fariña[1,2], Javier D. Fernández[3] and Miguel A. Martínez-Prieto[4]

[1*]Department of Computer Science and Information Technology, Universidade da Coruña, Campus de Elviña, A Coruña, 15071, Spain.
[2*]CITIC Research Center, Campus de Elviña, A Coruña, 15071, Spain.
[3]Data Science Acceleration, Hoffmann-La Roche, Grenzacherstrasse 124, Basel, 4058, Basel-Stadt, Switzerland.
[4]Department of Computer Science, University of Valladolid, Plaza de la Universidad 1, Segovia, 40005, Spain.

*Corresponding author(s). E-mail(s): gdebernardo@udc.es; Contributing authors: ana.cerdeira@udc.es; antonio.farina@udc.es; javier_d.fernandez@roche.com; migumar2@infor.uva.es;

Abstract

RDF compression and querying are consolidated topics in the Web of Data, with a plethora of solutions to efficiently store and query static datasets. However, as RDF data changes along time, it becomes necessary to keep different versions of RDF datasets, in what is called an RDF archive. For large RDF datasets, naive techniques to store these versions lead to significant scalability problems. In this paper we present `v-RDF-SI`, one of the first RDF archiving solutions that aims at joining both compression and fast querying. In `v-RDF-SI`, we extend existing RDF representations based on compact data structures to provide efficient support of version-based queries in compressed space. We present two implementations of `v-RDF-SI`,

[*] An early partial version of this article appeared in *Proc DCC'16* [13].

named v-RDFCSA and v-HDT, based respectively on RDFCSA (an RDF self-index) and HDT (a W3C-supported compressed RDF representation). We experimentally evaluate v-RDF-SI over a public benchmark named BEAR, showing that v-RDF-SI drastically reduces space requirements, being up to **40** times smaller than the baselines provided by BEAR, and 4 times smaller than alternatives based on compact data structures, while yielding significantly faster query times in most cases. On average, the fastest variants of v-RDF-SI outperform the alternatives by almost an order of magnitude.

Keywords: RDF, RDF Archiving, RDF compression, self-index

1 Introduction

RDF (*Resource Description Framework*) [55] is a specification for modeling data in the Web that has become increasingly popular. It has been used in many Open Data projects (e.g. *Linked Open Data*) and community efforts (e.g. *schema.org*), becoming the standard to describe and share information in the Web. In 2022, more than 14 million websites published data in RDF (with almost a 3x increase since 2018) [7]. Since RDF only defines a conceptual data model, a number of different solutions have been proposed to physically implement *RDF stores* that are able to store and query RDF data efficiently [2, 42].

In the Web of Data, however, the knowledge is continuously evolving [35], and therefore the facts that describe this knowledge must also change along time. This means that datasets have to be updated to reflect these changing facts. In addition, the historical information about all the different versions of the data must be kept in order to study the evolution of the facts along time, undo changes, or simply keep consistency between datasets as they evolve. The archival of the different versions of a dataset, that are in many cases very similar to their predecessor, is a problem that has been shown to suffer from significant scalability problems when applied to very large volumes of information, as is the case of the Web [27]. Similar challenges have also appeared in the Semantic Web, as the storage of archived versions of RDF datasets, known as RDF archives, needs to handle an increasingly large volume of data. Because of this, the development of solutions that provide efficient storage of archived versions, and provide support for queries involving changes along time, has become an interesting research challenge [24].

Strategies proposed to handle RDF archives typically focus on query performance and disregard the size of the RDF archive, so simple archival techniques can require huge amounts of space [22]. For instance, for a state-of-the-art RDF archive, these strategies may need up to 15 times the space required by a gzip-compressed version of the same data [25]. Trivially, RDF archival solutions provide efficient querying, unlike compressed files, but their space requirements are still far beyond what could be achieved with more refined solutions that aim at exploiting repetitiveness in the RDF data. A natural choice to improve the space utilization is to resort to RDF compression techniques. Many existing solutions take advantage of compact data structures to store the RDF dataset in small space while providing efficient query resolution [3, 9, 21, 52]. These solutions, however, consider static snapshots of the RDF dataset, and have not been designed to be applied to evolving data such as RDF archives.

In this paper we present `v-RDF-SI`, a solution that provides efficient query resolution over a compressed representation of RDF archives. It separates the handling of the different RDF triples in the archive from the management of the versioning information associated to them by splitting those data into two different layers.

The first layer (*rdf-layer*) is in charge of handling the different RDF triples in the archive and is implemented using an RDF representation supporting fast query resolution. In turn, the second layer (*ver-layer*) uses (succinct) bitsequences to encode the versioning information. This provides support for the efficient resolution of version-based queries.

This work is an extension of a preliminary work [13]. The main additional contributions included in this paper are:

- We provide a detailed description of `v-RDF-SI` and show how we can completely uncouple the handling of version-oblivious triples (*rdf-layer*) from the versioning data (*ver-layer*). In the proof-of-concept `v-RDFCSA` devised in [13], we could provide a solution based on `RDFCSA` and two bitsequence representations. However, the separation between both layers heavily depended on the internal arrangement of the triples within `RDFCSA`. The current `v-RDF-SI` solution is more flexible. It is based on giving each version-oblivious triple a unique tripleID to which we can easily relate its versioning information in the *ver-layer*, and providing a way to relate each triple from the *rdf-layer* with its corresponding tripleID. This potentially enables using most existing RDF representations for the *rdf-layer*.
- For the *rdf-layer*, we provide two self-indexed RDF representations that ensure fast query resolution on the compressed data: `RDFCSA` [9] and `HDT` [21]. We show that they yield different space/time tradeoffs, with `HDT` being typically faster than `RDFCSA` but requiring more space. In addition, we leave the door open to other RDF representations.
- For the *ver-layer*, apart from the two bitsequences used in [13] (Plain and RRR), we included four additional bitsequence representations. This allowed us to largely reduce the size of the versioning-layer.
- We included a more comprehensive experimental evaluation that significantly expands the preliminary version. Particularly, we consider the different variants of `v-RDF-SI` that can be built depending on the implementation chosen for *rdf-layer* and *ver-layer*. Moreover, we compare the best `v-RDF-SI` variants both with a baseline solution based on Jena, and

with OSTRICH [56], a recent representation for RDF archives. Finally, while in the preliminary work we only considered queries involving two common triple patterns, in this article we include queries for all the basic triple patterns.

We have compared the different solutions for RDF archiving using BEAR [25], a state-of-the-art benchmark for RDF archives. Our results show that `v-RDF-SI` is able to store the full archive of 325GiB in just 4.7−9.2GiB, and yields query resolution times much faster on average than the reference (Jena based) baseline deployed with BEAR. Among our proposals, `v-RDFCSA` obtains the best compression and `v-HDT` usually yields the best query times. The best `v-RDF-SI` variants typically answer queries in a few milliseconds. We have also run experiments to compare `v-RDF-SI` with OSTRICH. We show that our proposal requires roughly 25% the space of OSTRICH and is significantly faster at query time.

The rest of the paper is organized as follows. Section 2 presents the background on RDF archiving and discusses existing bitsequence representations. Our proposal is presented in Section 3. We first show its two-layered organization and how RDF triples in the *rdf-layer* are synchronized with the versioning information in the *ver-layer*. Then, we also show how typical queries for RDF archives are solved in Section 3.3. The experimental evaluation of `v-RDF-SI` is presented in Section 4. We include the details regarding our implementations `v-RDFCSA` and `v-HDT`, and we also compare them with some existing strategies for RDF archiving in terms of space needs and performance at query time. Finally, in Section 5, we include our conclusions and discuss future research lines.

2 Background

In this section, we briefly introduce some basic concepts relevant to our work. First, in Section 2.1, we describe RDF, SPARQL and the concept of triple pattern. Then, in Section 2.2, we also present some existing solutions for the compression of RDF. Among them, we briefly discuss `HDT` and `RDFCSA`, two queryable RDF representations that will be used in our proposal `v-RDF-SI`. In Section 2.3, we also introduce the concept of RDF archives and discuss the main retrieval functionality they must support. Section 2.4 presents the main state-of-the-art strategies to tackle RDF archiving. Finally, we provide a brief introduction to bitsequence representations, another basic component of `v-RDF-SI`, in Section 2.5.

2.1 RDF and SPARQL

RDF [55] is the standard for the storage of information in the Web of Data. It defines a conceptual representation of the data, that can be seen as a set of triples. Each triple contains a *subject*, a *predicate*, and an *object*, and represents that the subject entity has a property, defined by the predicate, whose value is given in the object. Typically, subjects and predicates are defined using

URIs, whereas objects can be either URIs or literal values.[1] For instance, the triple (`ex:Xavi, ex:playsFor, ex:Barça`) defines that the person identified by `ex:Xavi` plays for the team identified by `ex:Barça`. In practice, an RDF dataset can also be seen as a labeled graph, where the nodes of the graph are the subjects and objects, and each edge corresponds to a property, and is labeled using a predicate. Both conceptual views are equivalent, and provide a simple conceptual model for the representation of information in any area of knowledge.

SPARQL [31] is the standard query language for RDF data. The core of SPARQL queries is based on triple patterns, that are used to define constraints on the triples that are expected as the result of a query. For instance, (`ex:Xavi, ex:playsFor, ?team`) defines a fixed subject and predicate, and contains an unbound object `?team`, identified by the starting `?` symbol; this pattern matches all triples with the given subject and predicate, for any given object value, so the result of the SPARQL query containing this triple pattern would include all such triples. Different triple patterns can be defined depending on which elements of the pattern are unbounded, denoted with `?`: `(s??)` (matches all triples with subject s), `(sp?)`, `(spo)`, `(s?o)`, `(?po)`, `(s?o)`, `(??o)`. In addition to basic triple patterns, SPARQL also supports more complex queries, involving join of multiple triple patterns, unions, filtering and sorting, etc.

2.2 State of the Art of RDF Compression

Since the adoption of RDF as a standard, many solutions to handle RDF data have been proposed. Among them, we could highlight well-known tools such as Virtuoso [19], Blazegraph [57], RDF-3X [48], Tentris [6], BitMat [5], Hexastore [60], and WaterFowl [17]. Despite being well-known strategies, most of them require more space than other existing solutions that are based on compact data structures and are also slower in some scenarios [12]. In this section, we introduce four RDF representations based on compact data structures: `HDT` [21] (and a recent variant), `k2-TRIPLES` [3], `RDFCSA` [9], and *permuted trie indexes* [52]. As indicated above, these solutions usually require much less space than other alternatives, and provide efficient support for at least triple pattern queries, which are the core of the querying capabilities we provide for RDF archives.

`HDT` [21] is one of the first solutions that efficiently tackled RDF compression through the use of compact data structures. It is able to encode the RDF graph using bitsequences with rank/select support (see Section 2.5) and other additional compressed data structures. Thanks to the use of compact data structures, `HDT` is able to achieve much better compression than previous solutions and yield very efficient query times in all the basic triple pattern queries.

`k2-TRIPLES` [3] takes a step forward in RDF compression, by exploiting small-scale regularities in the RDF graph topology to largely outperform `HDT`

[1] For simplicity, we obviate here `bnodes`, a particular type that has only a local scope to the dataset. For our purpose, they can be converted to URIs via skolemization.

in terms of space. It is based on a partition of the RDF dataset by predicate, generating a separate set of (subject, object) pairs for each different predicate in the collection. Each of the resulting subsets is compressed as a (sparse) binary matrix using a data structure called k^2-tree [8]. Triple pattern queries are translated into a collection of simple queries over the collection of k^2-trees. Even though k2-TRIPLES is usually slower than HDT and other alternatives, it achieves the best compression results in most datasets.

RDFCSA [9] addresses the problem of RDF triple compression by adapting compressed suffix-arrays (CSA) [54], widely used for self-indexing natural text, to the representation of RDF triples. RDFCSA stores the collection of triples as a sequence of cyclic strings of length 3, that are sorted in lexicographical order and can be efficiently compressed and searched using the underlying compact self index. Although RDFCSA does not achieve the same compression ratios as k2-TRIPLES, it is significantly smaller than HDT and offers very stable and predictable query times to solve all SPARQL triple patterns.

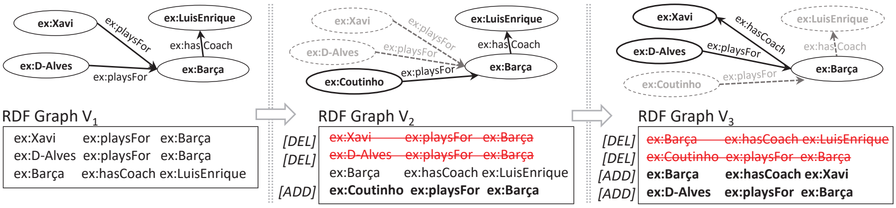
Recently, two new RDF representations were presented. One of them is based on *permuted trie indexes* [52]. The authors create three different permutations of the triples and use compressed trie representations to store each permutation. This allows them to achieve very fast query times for most triple patterns and good compression. A second recent representation is a variant of HDT named iHDT++ [34]. It is based on a previous non-searchable RDF compressor named HDT++ [33] which reorganizes the RDF triples upon HDT to reduce its structural redundancy and improves the space requirements of the original HDT. Basically, iHDT++ includes additional structures on top of HDT++ to efficiently support SPARQL triple patterns. In practice, iHDT++ improves the space/time tradeoff of the original HDT, yet it becomes a much more complex representation.

In addition, diverse compressors have been proposed for RDF streams (see [43] for a review), i.e. where the focus is to dynamically compress a continuous flow of RDF data with minimum latency (at the cost of lesser compression). Examples of such solutions are Streaming HDT [32], based on adapting the aforementioned HDT, RDSZ [26], which uses differential encoding or ERI [23], proposing an interchange format that groups redundancies by RDF predicate. To the best of our knowledge, none of them is able to efficiently resolve SPARQL in compressed space, although recent approaches, such as PatBin [37], mostly based on a dictionary compression, provide limited search functionality.

2.3 RDF archives

Given an RDF dataset with N versions, we define an *RDF archive* $\mathcal{A} = (\mathcal{T}, \mathcal{V})$ storing that dataset as a collection of *version-annotated triples* $\langle T_k, V_i \rangle$, where T_k is a regular RDF triple (s, p, o), and V_i is a label indicating that the triple T_k occurs in the i-th version of the RDF dataset [25].

Figure 1 displays an example of RDF archive, corresponding to three versions of a small RDF dataset. The RDF archive stores in total seven



RDF Graph V_1

ex:Xavi ex:playsFor ex:Barça
ex:D-Alves ex:playsFor ex:Barça
ex:Barça ex:hasCoach ex:LuisEnrique

RDF Graph V_2

[DEL] ~~ex:Xavi ex:playsFor ex:Barça~~
[DEL] ~~ex:D-Alves ex:playsFor ex:Barça~~
ex:Barça ex:hasCoach ex:LuisEnrique
[ADD] **ex:Coutinho ex:playsFor ex:Barça**

RDF Graph V_3

[DEL] ~~ex:Barça ex:hasCoach ex:LuisEnrique~~
[DEL] ~~ex:Coutinho ex:playsFor ex:Barça~~
[ADD] **ex:Barça ex:hasCoach ex:Xavi**
[ADD] **ex:D-Alves ex:playsFor ex:Barça**

Fig. 1 Example of RDF archive.

different facts about players and coaches of a soccer team called *"Barça"*. The leftmost graph corresponds to the first version of the dataset, V_1. It stores three triples: `ex:Xavi` and `ex:D-Alves` play for the team, and its coach is `ex:LuisEnrique`. The middle graph, corresponding to version V_2, reflects a few changes: the two players of V_1 leave the team, so the corresponding triples disappear from the dataset, and a new player `ex:Coutinho` is hired. Finally, in version V_3, `ex:LuisEnrique` is replaced as coach by the former player `ex:Xavi`; also in version V_3, the player `ex:Coutinho` leaves the team but the former player `ex:D-Alves` joins the team again. In this example, the triple `(ex:D-Alves,ex:playsFor,ex:Barça)` holds in V_1 and V_3, so two annotated triples would exist in the RDF archive.

2.3.1 Retrieval Functionality.

When querying RDF archives, SPARQL queries can be extended in order to retrieve information corresponding to specific versions. The three primitive queries that provide the most usual functionalities in an RDF archive are the following ones [24]:

- *Version materialization* queries. Given a query Q, a version materialization query $Mat(Q, V_i)$ returns all the results corresponding to query Q in the version V_i of the archive. For instance, the query $Mat($`(ex:Barça,ex:hasCoach,?x)`$, V_2)$ obtains `ex:LuisEnrique` as the coach in V_2.
- *Delta materialization* queries. The query $Diff(Q, V_i, V_j)$, retrieves all the results of Q that appear in V_i but not in V_j or vice versa; i.e., it detects (deleted) triples that existed in V_i but are not present in V_j, as well as those (added) triples that existed in V_j but not in V_i. For example, $Diff($`(?x,playsFor,ex:Barça)`$, V_1, V_2)$ returns `ex:Xavi` and `ex:D-Alves`, that were present in V_1 but were deleted in V_2. It also returns `ex:Coutinho` as an added triple that appeared in V_2.
- *Version* queries. The query $Ver(Q)$, obtains the results for query Q in all the versions of the archive, and returns all the versions in which each of the results holds. For example, $Ver($`(ex:Barça,hasCoach,?x)`$)$ would return that `ex:LuisEnrique` is present in V_1 and V_2, and `ex:Xavi` is present in V_3.

In the previous examples, the three primitives are applied to basic triple pattern queries, but more complex queries can be built combining the extended search functionalities of SPARQL with one or more of these primitives. For instance, assume that a user wants to find the players of our team that have also been coaches of the team. In a single-version RDF store, this would be answered with a join query between `(?x,playsFor,ex:Barça)` and `(ex:Barça,hasCoach,?x)`, where the join variable `?x` is used to connect both triple patterns. In an RDF archive, we need to apply the version primitives to consider triples that exist in different versions of the dataset. The most straightforward adjustment would be the query $Ver($`(?x, ex:playsFor,ex:Barça)`$) \bowtie Ver($`(ex:Barça,ex:hasCoach,?x)`$)$, combining two version queries that search for all the players and coaches of the team in any version of the archive. If we want to enforce that the player must have coached the team *after* he was a player, we could instead use version materialization queries in all the relevant versions. In this example, for each $i, j \in \mathcal{V}$ such that $i < j$ we could execute $Mat($`(?x,ex:playsFor,ex:Barça`$, V_i) \bowtie Mat($`(ex:Barça, ex:hasCoach,?x)`$, V_j)$.

2.4 State of the Art of RDF Archiving

The problem of RDF archiving has been tackled following different strategies, which are rather similar to archiving solutions in other domains [24]. In this section we describe the three main strategies that have been used: *independent copies* (Ic), *change-based* (Cb), and *timestamp-based* (Tb).

The independent copies strategy (Ic) is the simplest strategy to manage versions. It simply handles each version as an independent dataset, with no relation to the others [36]. The main advantage of this strategy is its simplicity, and the performance it provides for version materialization queries, since queries for a given version are simply executed on the corresponding dataset. Delta materialization queries and version queries, on the other hand, are less efficient since the independent datasets have to be queried and the results joined or subtracted. Additionally, an obvious drawback of this strategy is the space it requires: since it does not take into account the repetitiveness between versions, triples that do never change between versions are stored multiple times, which makes this strategy very inefficient in datasets with a large number of versions even if the number of changes between versions is small. SemVersion [59] and the Quit store [4] are good practical examples of Ic-based RDF archives.

The change-based strategy (Cb) aims specifically at taking advantage of regularities between consecutive versions. Each version V_i stores only the triples that were added or deleted (*deltas*) with respect to the previous one. This leads to potentially huge space savings when compared to Ic strategies. This strategy, followed by R43ples [29], can speed up delta materialization queries, but it yields poor performance in version queries, due to the need to propagate the deltas.

The timestamp-based strategy (Tb) considers the full RDF archive as a single dataset of version-annotated triples. Therefore, solutions following this strategy require a mechanism to annotate each triple with versioning information (i.e. the versions in which the triple is valid). This storage mechanism is well suited for version queries, but practical solutions following this strategy need to balance the poor results obtained in version materialization queries and delta materialization queries by building additional indexes to be able to filter the version information. XRDF-3X [49] (an extension of the RDF-3X store) and our proposed solution [13] are positioned in this strategy.

Finally, we can define a fourth hybrid-based (Hb) strategy, that essentially groups a number of other solutions that combine ideas from the previous ones to improve performance. For instance, practical approaches based on Cb have been improved, like the framework in [18] and TailR [44], by storing fully materialized versions every k deltas, in order to mitigate the performance penalty, hence being assimilated to a hybrid Ic/Cb strategy. On the other hand, Tb/Cb approaches, like R&WBase [58], enhance added or deleted triples with timestamp data. It reduces space requirements, but deltas must be rebuilt, as in Cb, to perform version materialization. More recently, OSTRICH [56] combines Ic, Cb, and Tb into a solution that compresses snapshots using HDT and stores deltas with respect to their corresponding latest snapshot. OSTRICH provides competitive space-time tradeoffs, at the price of limited functionality [51].

All these strategies can be used to efficiently answer some queries on RDF archives, but existing solutions are still far from yielding efficient compression. An experimental evaluation on the BEAR benchmark [25] shows that typical archiving solutions require up to 230GiB of space to handle a dataset whose gzipped size is approximately 23GiB. The goal of our proposal is precisely to take advantage of the capabilities of compressed RDF stores and provide a simple and compact representation of the versioning information, in order to store large RDF archives requiring much less space than the plain dataset, while preserving query capabilities.

2.5 Bitsequence representations

Many succinct data structures are built on top of binary sequences (i.e. *bitsequences*). A plain bitsequence $B[1,n]$ can be seen as a sequence of ones and zeroes where the following operations are typically of interest:

- $access(B,i)$ obtains the value of element $B[i]$.
- $rank_b(B,i)$ gives the number of b bits within $B[1,i]$.
- $select_b(B,i)$ returns the position of the i-th b bit in B.

Even though $access(B,i)$ can be directly obtained in a plain bitsequence, *rank* and *select* operations require additional structures to be efficiently computed. Moreover, depending on the ratio of ones and zeroes, or on their distribution along B, different state-of-the-art bitsequence representations allow us to yield compression. In this paper we will use `plain` bitmaps, where we are not able to efficiently support rank/select operations, as well as five well-known alternatives that do support rank/select. The first variant (`RG`) adds additional structures but keeps B in plain form, whereas the others (`RRR`, SDarray, Delta, and OZ) are compressed representations of B that provide support for the three main operations above.

- RG [28] implements a simple one-level superblock-directory, that stores samples of the cumulative number of ones in B. A $factor$ parameter determines that a superblock counter is kept for each $32 \cdot factor$ bits. Therefore, the space usage of RG is $n(1+1/factor)$ bits. It performs $rank_1$ by checking the closest superblock and then counts ones sequentially between superblocks, thus requiring up to $factor$ contiguous accesses. To solve $select_1$, it first performs a binary search on the superblocks, and a final sequential scanning of the byte containing the i-th bit. A typical configuration uses $factor = 20$, which yields a 5% space overhead.
- RRR [53] is a common representation for either sparse or dense bitsequences (i.e. those with a very high or low percentage of zeroes). In practice, it is useful when the ratio between the number of ones (m) and n is under 0.2 or over 0.8 [47]. RRR compresses B into $nH_0(B)$ bits, but requires also additional $o(n)$ bits for solving $rank/select$ efficiently.

  We used the implementation in [15]. Internally, it splits B into blocks of 15 bits each and stores how many ones exist up to the beginning of each block (additional structures are necessary to support all operations, for details see [15]). A sampling parameter (t_{rrr}) indicates that counters are explicitly kept for every t_{rrr} blocks. In practice, RRR solves $rank_1$ with two random accesses and $3 + 8 \cdot t_{rrr}$ accesses to contiguous memory, whereas solving $select_1$ requires an additional binary search. The overall space of this implementation is around $\log \binom{n}{m} + (\frac{4}{15} + \frac{1}{t_{rrr}})n = nH_0(B) + o(n)$ bits.
- SDarray [50] is a good representation for very sparse bitsequences, particularly it obtains good compression when the ratio between the number of ones (m) and n is under 0.05. SDarray is not parameterizable and requires $nH_0(B) + 2m + o(m) = nH_0(B) + O(m)$ bits. It solves $select_1$ in constant time, and $rank_1$ in time $O(\log(n/m))$.
- Delta [1] is another representation that is very efficient when the number of ones is much smaller than that of zeroes. Delta differentially encodes the m positions of the ones in B using δ-codes. Those gaps ($gaps_i$) are encoded with δ-codes, and samples are included every t_b ones. Those samples include the absolute position of the $(t_b \cdot i)$-th one, and a pointer to the sequence where all the δ-codes for the gaps are kept. $select_1(B, i)$ is solved with a table lookup to the samples if i is a sampled one ($i \mod t_b = 0$), otherwise decoding $i \mod t_b$ δ-codes is required to sum up those gaps. $rank_1(B, i)$ uses binary search over the samples, followed if required by local decoding from the sample preceding the i-th bit. The overall space [46] is $O(\frac{m}{t_b}) \log n + m \log \frac{n}{m} + 2m \log \log \frac{n}{m} + O(m)$.
- OZ [46]. Sometimes the number of zeroes and ones in B are rather similar (we do not have a sparse bitsequence), yet they are clustered forming k runs of ones and $k \pm 1$ runs of zeroes. In [46, pages 89–92], OZ is discussed as a transformation from a bitsequence B with few runs into two sparse bitsequences Z and O that can be handled with any representation for

sparse bitsequences (i.e. `RRR`, `SDarray`, `Delta`, or even `RG`). We consider `OZ` could be interesting in the scope of our versioning data as we do not expect that a triple would appear and disappear many times along versions.

3 Self-Indexing RDF archives (`v-RDF-SI`)

We have designed `v-RDF-SI` following a *lightweight* TB approach, where we encode all the information of the RDF archive in two separate layers. The first layer (*rdf-layer*) represents the set of different triples in the RDF archive; the second layer (*ver-layer*) represents the versioning information. By doing so, `v-RDF-SI`, as other TB-based solutions, only needs to store the set of different triples in the archive. Our *rdf-layer* will only store these *version-oblivious triples*, that are usually a small percentage of the total number of version-annotated triples in the archive. For instance, the BEAR archive [25] contains around 2 billion triples, but removing the versioning information we end up with just around 376 million version-oblivious triples.

The two layers used in our representation divide the archive storage problem into two parts. The *rdf-layer* is in charge of efficiently compressing and querying the version-oblivious triples, that are handled as a regular (compressed) RDF dataset. The *ver-layer* must provide all the versioning information to recover all the version-annotated triples.

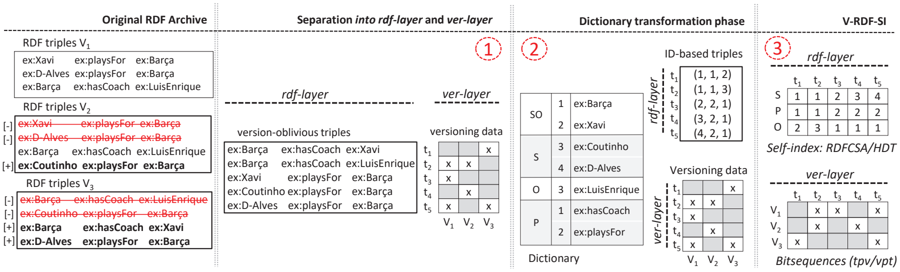

Original RDF Archive
Separation *into rdf-layer* and *ver-layer*
Dictionary transformation phase
V-RDF-SI
RDF triples V1
ex:Xavi ex:playsFor ex:Barça
ex:D-Alves ex:playsFor ex:Barça
ex:Barça ex:hasCoach ex:LuisEnrique
RDF triples V2
[-] ~~ex:Xavi ex:playsFor ex:Barça~~
[-] ~~ex:D-Alves ex:playsFor ex:Barça~~
ex:Barça ex:hasCoach ex:LuisEnrique
[+] **ex:Coutinho ex:playsFor ex:Barça**
RDF triples V3
[-] ~~ex:Barça ex:hasCoach ex:LuisEnrique~~
[-] ~~ex:Coutinho ex:playsFor ex:Barça~~
[+] **ex:Barça ex:hasCoach ex:Xavi**
[+] **ex:D-Alves ex:playsFor ex:Barça**
1
rdf-layer
ver-layer
version-oblivious triples
ex:Barça ex:hasCoach ex:Xavi
ex:Barça ex:hasCoach ex:LuisEnrique
ex:Xavi ex:playsFor ex:Barça
ex:Coutinho ex:playsFor ex:Barça
ex:D-Alves ex:playsFor ex:Barça
versioning data
t1 x
t2 x x
t3 x
t4 x
t5 x x
V1 V2 V3
2
SO 1 ex:Barça
2 ex:Xavi
S 3 ex:Coutinho
4 ex:D-Alves
O 3 ex:LuisEnrique
P 1 ex:hasCoach
2 ex:playsFor
Dictionary
ID-based triples
rdf-layer
t1 (1, 1, 2)
t2 (1, 1, 3)
t3 (2, 2, 1)
t4 (3, 2, 1)
t5 (4, 2, 1)
Versioning data
ver-layer
t1 x
t2 x x
t3 x
t4 x
t5 x x
V1 V2 V3
3
rdf-layer
t1 t2 t3 t4 t5
S 1 1 2 3 4
P 1 1 2 2 2
O 2 3 1 1 1
Self-index: RDFCSA/HDT
ver-layer
t1 t2 t3 t4 t5
V1 x x x
V2 x x
V3 x x
Bitsequences (tpv/vpt)

Fig. 2 Construction of the `v-RDF-SI` representation of our RDF archive.

In Figure 2, we show the process performed to build a `v-RDF-SI` for an RDF archive. The first step (①) generates the set of version-oblivious triples and the versioning information for those triples. These will be stored in our *rdf-layer* and *ver-layer* respectively. In the example, we extract the $n = 5$ different triples $t_1, \ldots, t_5$, and for each of them, we gather the versions in which they occur. In the next step (②), a dictionary is used to replace all the RDF terms in the version-oblivious triples with integer IDs. We assign IDs in the range $[1, n_s]$, $[1, n_p]$, $[1, n_o]$ to subjects, predicates, and objects respectively. In the example, we have $n_s = 4$, $n_p = 2$, $n_o = 3$, and we can see that IDs 1 and 2 (corresponding to *ex:Barça* and *ex:Xavi* terms respectively) are used both for subjects and objects. Therefore, the remaining object *ex:LuisEnrique* is given ID 3, and the subjects *ex:Coutinho* and *ex:D-Alves* receive IDs 3 and

4 respectively. The mappings between RDF terms and IDs are independently indexed using a compressed string dictionary [11, 40].

Finally, both the *rdf-layer* and the *ver-layer* are stored using compact data structures (③). `v-RDF-SI` provides two different representations, `RDFCSA` and `HDT`, to handle the *rdf-layer*. This leads us to variants `v-RDFCSA` and `v-HDT` respectively, that are discussed in Section 3.1. The *ver-layer* is stored using plain or compressed bitsequences. In Section 3.2, we show how versioning data is handled.

Regarding query evaluation, recall that both `RDFCSA` and `HDT` allow solving triple pattern queries efficiently over RDF datasets. When dealing with version-based queries, we rely on the *rdf-layer* to retrieve the version-oblivious RDF triples that match a given triple pattern and then perform bit-based operations in the *ver-layer* to gather versioning information for those triples. To allow this, our *rdf-layer* enriches the solution triples (i.e. the resulting triples provided by the underlying RDF representation when solving a given triple pattern Q') with an identifier for each returned triple t_i. In particular, this identifier corresponds to the position of the triple i within the sequence of ID-based triples. This triple-ID i gives us direct access to its versioning information. For example, if we ask the *rdf-layer* for the triple pattern $Q' \leftarrow$`(1??)` we will recover $\{\langle(1,1,2),\underline{1}\rangle, \langle(1,1,3),\underline{2}\rangle\}$, where the underlined numbers indicate that the triples $(1,1,2)$ and $(1,1,3)$ were respectively at positions 1 and 2 among the ID-based triples in the *rdf-layer*. More details regarding how version-based queries are solved are presented below in Section 3.3.

3.1 *rdf-layer*: RDF triples encoding

We provide two implementations for the *rdf-layer*. They are based on `RDFCSA` and `HDT`, yet other RDF representations could also be used. The only condition required is that the chosen implementation must provide a way to keep the *rdf-layer* aligned with the *ver-layer*, so that when we access an RDF triple from *rdf-layer*, we can use its triple-ID to easily gather its versioning data from *ver-layer*.

A common factor of `RDFCSA` and `HDT` is that they sort the set of ID-based triples by subject, then by predicate, and finally by object. This is the way we sorted triples in Figure 2, and the versioning information associated to those triples is organized accordingly in the *ver-layer*. Therefore, both `RDFCSA` and `HDT` can be directly used for the *rdf-layer*. Considering also that both approaches are relatively stable in query times and provide interesting space-time tradeoffs, we considered them as the best choices. However, we will also outline how other potential solutions could be adjusted to be used for the *rdf-layer* in Section 3.1.3.

3.1.1 `v-RDFCSA`: using `RDFCSA` into the *rdf-layer*

Figure 3 illustrates the steps followed to turn the ID-based triples shown in Figure 2 into an `RDFCSA` self-indexed representation [9]. Assuming the sequence of n ID-based triples $t_1, \ldots, t_n$ is already sorted by subject, predicate, and object, in the first step (①) we place them in a single sequence $S_{id}[1, 3n]$. The first triple is stored in $S_{id}[1, 3]$ (see the shaded boxes in Figure 3), the second in $S_{id}[4, 6]$, and so on. Then (②), the IDs in S_{id} are rewritten into $S^{'}_{id}[1, 3n]$ so that the ID values used for subjects, predicates, and objects do not overlap, as required by `RDFCSA`. To do this, subject IDs are not modified, predicate IDs are increased by n_s, and object IDs are increased by $n_s + n_p$, where n_s and n_p are the number of different subjects and predicates respectively. After this step, subject IDs are in the range $[1, n_s]$, predicates in $[n_s + 1, n_s + n_p]$, and objects in $[n_s + n_p + 1, n_s + n_p + n_o]$. For instance, in the example of Figure 3, the first triple $(1, 1, 2)$ is encoded as $(1, 5, 8)$ in $S^{'}_{id}$.

The purpose of reassigning IDs is to ensure that subject IDs are always smaller than predicate IDs, and that predicate IDs are smaller than object IDs. This is important for step ③, in which we create a suffix array [39] $SA[1, 3n]$ over $S^{'}_{id}$. Thanks to the previous reassignment of IDs, elements in $SA[1, n]$ will always be pointing to subjects within $S^{'}_{id}$, elements in $SA[n + 1, 2n]$ will point to predicates, and elements in $SA[2n + 1, 3n]$ will point to objects. The compressed representation consists of the usual CSA data structures D and ψ [54] (step ④). $D[1, 3n]$ is a bitsequence that is used to mark (with a 1) the position i in the suffix array where the first symbol of the suffix[2] changes (i.e., $S^{'}_{id}[SA[i]] \neq S^{'}_{id}[SA[i-1]]$). Therefore, we can use $S^{'}_{id}[SA[i]] = rank_1(D, i)$ to recover the ID of the source sequence $S^{'}_{id}$ for any given position i of SA. The array $\psi[1, 3n]$ is used to traverse the sequence $S^{'}_{id}$; ψ is built in such a way that $SA[\psi[i]] = SA[i] + 1$. Therefore, if $SA[i] = j$ points to the suffix at position j in $S^{'}_{id}$, then $SA[\psi[i]] = j + 1$ points to the next position in $S^{'}_{id}$, i.e. to the suffix starting at position $j + 1$ ($S^{'}_{id}[j + 1, 3n]$). This means that, using ψ, we can move forward to the next position in $S^{'}_{id}$ and recover its value using D.

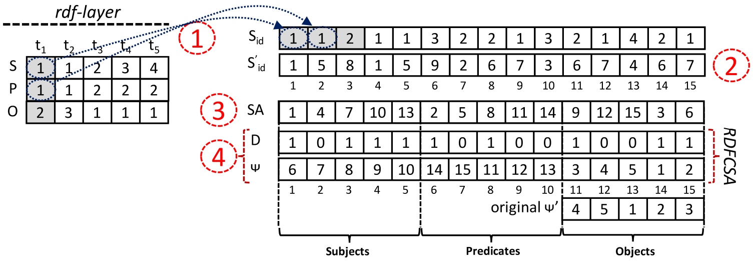


Fig. 3 Construction of the `RDFCSA` representation of an RDF archive.

`RDFCSA` performs a slight modification in $\psi[2n + 1, 3n]$, the region corresponding to the objects of the triples. Computing ψ for any object would return the position of the subject of the next triple in the collection. However,

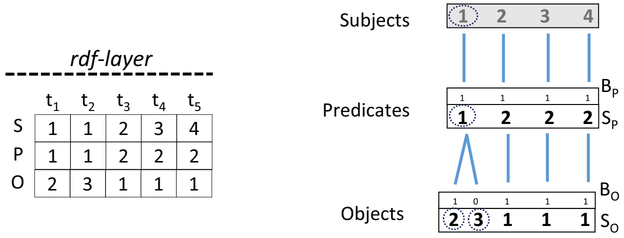
[2] A suffix starting at position j from $S^{'}_{id}$, is defined as the subsequence $S^{'}_{id}[j, 3n]$.

RDFCSA updates ψ such that $\psi[i] \leftarrow \psi[i]-1, \forall i \in [2n+1, 3n]$ (with the particular case that $\psi[i] \leftarrow n$ if $\psi[i]=1$). In this way, the object of each triple now points to the subject of the same triple, which is useful for querying. For example, note that the object of the second triple ($S^{'}_{id}[6]$) in Figure 3 is referenced from $SA[15]=6$. In the original $\Psi^{'}$, we have $\Psi^{'}[15]=3$, and $SA[3]=7$ points to $S^{'}_{id}[7]$ that keeps the subject of the third triple. However, the modified Ψ in RDFCSA contains $\Psi[15]=2$, where $SA[2]=4$ points to $S^{'}_{id}[4]$, which contains the subject of the second triple. Consequently, Ψ becomes cyclical within RDF triples. This allows RDFCSA to search for triple patterns very efficiently [9]. It basically performs an initial binary search using D and ψ to find the range $SA[l,r]$ that matches the query pattern, and then, a traversal $\forall i \in [l,r]$ is performed to recover the matching triples by applying $\Psi[i]$ cyclically up to two times. More details about the query algorithms can be found in [9]. For example, if we want to retrieve all the triples including subject 1, we will use the triple pattern $Q \leftarrow$(1??). The initial binary search on SA gives that $SA[1,2]$ point to $s_1 = S^{'}_{id}[1]$ and $s_2 = S^{'}_{id}[4]$ that are, respectively, the subjects of the first and second triples. Now, if we want to recover the predicate p_1 and object o_1 of the first triple, we compute: $p_1 = rank_1(D, \Psi[1]) = 5$, and $o_1 = rank_1(D, \Psi[\Psi[1]]) = 8$. Hence we have retrieved the first triple $(1,5,8)$ corresponding to $S^{'}_{id}[1,3]$. Finally, we unmap from $S^{'}_{id}$ to the original triple in S_{id} by subtracting $(0,4,4+2)$, obtaining the original triple $(1,1,2)$. Similarly, we can compute the value of p_2 and o_2 , corresponding to the second triple that matched (1??).

A key property of RDFCSA is that any triple can be easily identified by the position in SA where its subject is located. Therefore, keeping the synchronization between RDFCSA and *ver-layer* is straightforward from the fact that $SA[1,n]$ points to the subjects corresponding to all the triples $t_1, \ldots, t_n$. Consequently, we simply have to keep versioning information aligned with $SA[1,n]$. At query time, when we retrieve an RDF-triple, we gather the ID of its subject (s_x), predicate (p_x), and object (o_x), and additionally, we also keep track of the position i of its subject ID in $SA[1,n]$ to finally return $\langle (s_x, p_x, o_x), \underline{i} \rangle$. This gives us direct access to its versioning information. In the example above, the versioning information associated to the triples that matched (1??) is synchronized with $SA[1,2]$. In Figure 2, looking at the versions associated to the first and second triples, we can see that the first matching triple occurs in version V_3 and the second one in versions V_1 and V_2.

3.1.2 v-HDT: using HDT into the *rdf-layer*

Figure 4 displays the structures used in HDT to store the collection of ID-based triples described in Figure 2. We assume that the triples $t_1, \ldots, t_n$ are sorted by subject, predicate, and object. HDT creates a so-named *BitmapTriples* index which exploits the repetitive components of the triples by storing the relevant information in two sequences S_P and S_O. S_P stores the predicate of each different (subject, predicate) pair appearing in the collection. Note that $|S_P| < n$ in most cases, because multiple triples can have the same value of subject and predicate. Sequence S_O stores, in order, all the objects of the triples in the collection, hence $|S_O| = n$. Additionally, two bitsequences B_P and B_O are added to S_P and S_O respectively: B_P marks with a 1 the positions in S_P where the subject of the triple changes; B_O marks with a 1 the positions in S_O where the subject or predicate of the triple changes (i.e. marks the first occurrence for each subject-predicate pair). This information suffices to retrieve the original dataset. Note that the list of subjects which is shaded in Figure 4 is not actually stored, since the subject values can be obtained from the information in B_P.



	t1	t2	t3	t4	t5
S	1	1	2	3	4
P	1	1	2	2	2
O	2	3	1	1	1

Fig. 4 Representation of the triples information in HDT.

In order to answer the query `(1??)`, we first locate the range in S_P corresponding to subject 1. This range can be obtained efficiently using $rank_1$ operations in B_P: $[rank_1(B_P, 1), rank_1(B_P, 2) - 1] = [1, 1]$. Note that in this case, the only predicate related to subject 1 is predicate $1 = S_P[1]$. Then, the process can be repeated in B_O to get the corresponding range $[1, 2]$ in S_O, obtaining the objects $S_O[1, 2] = \{2, 3\}$. Therefore, the result includes triples `(1,1,2)` and `(1,1,3)`. Note that, since we have simple intervals in S_P and S_O, results can be extracted sequentially from S_P and S_O. Similar query algorithms can be used for `(sp?)` and `(spo)` query patterns.

To efficiently answer queries with unbound subject, two small enhancements are applied in practice to the basic index, leading to the representation called HDT-FoQ [41]. Firstly, the sequence S_P is stored using a wavelet tree [30].[3] This provides a faster way to answer `(?p?)` queries, by using $select_1$ to locate all the positions where the query predicate occurs; after identifying the subjects using B_p, the same basic process is used to map the objects in S_O and obtain the query results. To speed up `(?po)` and `(??o)` queries, an adjacency list stores, for each object, the (subject,predicate) pairs related to it (i.e. the ranges in S_P that have to be searched), in predicate order. Therefore, a `(??o)` query is answered by first locating the ranges in S_P that correspond to triples where the object appears, and then extracting the corresponding subject and predicate values from B_P and S_P. For details on these optimized algorithms

[3] A wavelet tree WT over a sequence S built on an alphabet $\Sigma = [1, \sigma]$, represents S implicitly and supports $access(S, i)$, $rank_b(S, i)$, and $select_b(S, j)$, being $b \in \Sigma$, in logarithmic time.

we refer the reader to the original paper [41]. Note that the latest practical implementation of HDT[4] replaces the Wavelet Tree index by an adjacency list.

A key observation in HDT is that triples are sorted in subject, predicate, and object, so their position in the sorted collection can be used to synchronize the *rdf-layer* and the *ver-layer*, exactly like in RDFCSA. In this case, the position in S_O of each result corresponds to the position of the triple in the *ver-layer*. For most query patterns, the position in S_O is obtained during the original search process, since the object values are extracted accessing positions in the sequence S_O. However, for query patterns like (?po) and (??o) patterns, the original query algorithms do not need to access S_O, since the object values are already known, so they extract this information of subject/predicate values from B_P and S_P. For these patterns, we need to add a step to the original algorithms, that extracts the corresponding position in S_O for each result obtained. This is performed by searching for the corresponding object in the range delimited by the (subject,predicate) pair. After this modification, the algorithms for all the query patterns are able to return the result triple IDs coupled with the position i of each result in S_O, and hence in the *ver-layer*. Following the previous example, when searching for (1??), the results returned would be $\langle(1,1,2),\underline{1}\rangle$ and $\langle(1,1,3),\underline{2}\rangle$, that could be located in the *ver-layer* in Figure 2 to determine that the first triple occurs in version V_3 and the second one in versions V_1 and V_2.

This layer could also be deployed using iHDT++, the more recent variant of HDT, but we choose the original because it is widely adopted by the community and we believe that its current space-time tradeoffs are not far from those of iHDT++. Nevertheless, replacing HDT by iHDT++ is straightforward. The only significant change involves rearranging triples to match the (P,S,O) (predicate, subject, object) order required by iHDT++. As in HDT, triples are identified by their object position in iHDT++, which provides the corresponding IDs to access the versioning information in the *ver-layer*.

3.1.3 Other alternatives for the *rdf-layer*

As explained before, both RDFCSA and HDT sort triples by subject, then by predicate, and finally by object, which makes it very easy to access the versioning information: we keep versioning information in the same order, and use the triple IDs to access the corresponding versioning entries. Other solutions based on compact data structures, such as k2-TRIPLES or permuted trie indexes, do not provide such an easy mechanism to map query pattern results to versioning information. In this section, we outline the difficulties for the application of these two techniques in the *rdf-layer*.

When using permuted trie indexes, a different permutation index is used depending on the query pattern, so even if an internal ordering of the triples exists in each index, there is no simple mechanism to globally synchronize results for a given triple pattern with their corresponding versioning information. Trivially, additional data structures to perform the synchronization

[4] Available at https://github.com/rdfhdt/hdt-cpp

can be added to this (or any other) representation, but this would have an important impact on the overall compression and query times would also worsen.

A solution based on `k2-TRIPLES` partitions the triples by predicate, and then stores the triples for each predicate in a separate data structure. In this case, there is a global ordering of the triples: they are sorted first by predicate, and inside each k^2-tree results correspond to 1s in a bitmap L,[5] so they are implicitly sorted. However, there is no direct mechanism to map the results of a query to their position in the global ordering. A straightforward adaptation would be to add $rank_1$ support to the L bitmaps, but this would increase the space requirements of `k2-TRIPLES`, since a specific compact representation is used for the bitmaps and the additional rank structures would potentially void all the compression benefits of `k2-TRIPLES`.

3.2 *ver-layer*: Version information encoding

Let us assume that we have an archive $\mathcal{A}$, that contains N different versions of a dataset, and n *version-oblivious triples* t_k $(1 \leq k \leq n)$. For instance, the archive in Figure 2 has $N = 3$ and $n = 5$. Let us also assume that for each triple t_k we know in which versions it occurs. In this section we propose two encoding strategies for the versioning information.

The first strategy, called `tpv` (*triples per version*) groups the information by version. It stores, in a separate structure per version, the information about all the triples that appear in that version. To do this, it builds separate bitsequences $\mathcal{B}^v_i[1,n]$ $(1 \leq i \leq N)$ for each version of the archive. Each bitsequence is used to mark the triples appearing in the corresponding version, by setting $\mathcal{B}^v_i[k] = 1$ if and only if triple k in the collection occurs in version i. Figure 5 (center) displays an example of the `tpv` strategy for our example archive. The shaded results in $\mathcal{B}^v_2$ correspond to triples 2 and 4, the only two triples that exist in the version 2 of the archive.

The second encoding strategy, called `vpt` (*versions per triple*) groups the information first by triple, and stores for each triple the information of the versions in which the triple is valid. This strategy builds n bitsequences $\mathcal{B}^t_i[1,N]$, each encoding the list of valid versions for the corresponding triple. Therefore, $\mathcal{B}^t_k[i]$ will be set to 1 if triple k is present in version i, and set to 0 otherwise. Figure 5 (right) displays the `vpt` encoding for our example archive. The shaded cells in $\mathcal{B}^t_2$ correspond to versions 1 and 2, marking that triple 2 is present in the versions 1 and 2 of the archive.

Considering a plain representation of the bitsequences, it is easy to see that they use the same space: `tpv` stores N bitsequences of n bits, and `vpt` stores n bitsequences of N bits. However, the bitsequences can be represented using the different encoding techniques presented in Section 2.5, which may lead to significantly different space requirements. Additionally, as we will show in the next section, query algorithms are significantly different in each strategy.

[5] Let us consider a bitmap as a sequence of ones and zeroes with no additional structures to efficiently support *rank* and *select* operations.

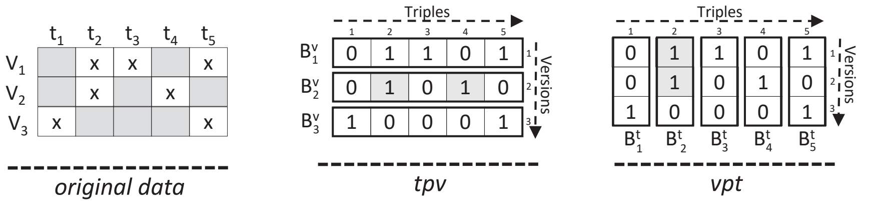


Fig. 5 Examples of the `tpv` and `vpt` strategies to encode the original versioning information from Figure 2.

3.3 Retrieval algorithms

We show below how to deal with the typical retrieval operations for RDF archives presented in Section 2. In our discussion, we do not consider the dictionary translation step that is necessary to map the elements of an RDF triple (triple pattern query) to their corresponding IDs as shown in Figure 2. We also ignore the un-mapping process needed to convert a resulting set of ID-based RDF triples into the final set of RDF triples. Therefore, we assume that queries are composed of the IDs of the subject, predicate, and object that make up each ID-based triple t_i in Figure 2, and results also refer to those triples.

As explained above, all our algorithms start by querying the *rdf-layer* to retrieve all the version-oblivious candidate triples that match a given pattern Q. Recall that those triples are augmented with the triple-ID that is used to find the versioning information within the *ver-layer*. From there on, candidate triples are traversed and filtered using versioning information. For a $Mat(Q, i)$ query, we have to check if each candidate triple occurs in version i; in a $Diff(Q', i, j)$ query, we check if the triple changes from version i to version j; to answer $Ver(Q)$ we simply collect all the different versions in which each candidate triple occurs.

Apart from the general description of the query operations, we will show that some optimizations can also be performed during the traversal of the candidate triples. These optimizations are based on the fact that, for some triple patterns, the candidate triples form a contiguous range $[t_l, t_r]$. In particular, since we sorted the source triples by subject, predicate, and object, in the case of `v-RDFCSA` the subjects of all the candidate triples for operations `(s??)`, `(sp?)`, and `(spo)` make up a contiguous range in the suffix array $SA[l, r]$ [9], and the triple-IDs of those candidate triples are respectively the values $l, \ldots, r$. Therefore, they are aligned with the *ver-layer* and we can use the versioning information for triples $t_l, \ldots, t_r$ to discard some candidate triples without actually traversing them. The same optimizations are also suitable for `v-HDT`, using the position of the object in S_O to track the triples.

Version materialization queries: A query $Mat(Q, i)$ looks for all the triples matching the pattern Q that are active in version i. To answer this query, we start by locating in the *rdf-layer* all the enhanced candidate triples that match our query pattern Q. Then, the versioning information is used to filter the list of candidates. When using the `vpt` strategy, for each candidate triple t_k we check the value of $\mathcal{B}^t_k[i]$. If it is set to 1, the triple is a valid result, otherwise it is discarded. When using the `tpv` strategy, we proceed in a similar way: if $\mathcal{B}^v_i[k]$ is set to 1, the triple is valid, otherwise it is discarded.

The basic query algorithms explained above are valid for any query pattern. However, in some patterns it is possible to use optimized algorithms to speed up queries. Particularly, this optimization is feasible when using the `tpv` versioning strategy and when the query pattern is one of `(s??)`, `(sp?)`, and `(spo)`. In this case, the candidate triples obtained in the *rdf-layer* will always conform a contiguous range $[l, r]$. Since $\mathcal{B}^v_i$ is set to 1 only for triples that are valid in version i, by $c_l \leftarrow rank_1(\mathcal{B}^v_i, l)$ and $c_r \leftarrow rank_1(\mathcal{B}^v_i, r)$ we can determine that the number of valid results in the range is $c = c_r - c_l + 1$. Furthermore, we can locate the actual positions of each valid triple using select operations to locate the positions of the bits set to 1: the valid results will be the the triples at positions t_p, such that $p \leftarrow select_1(\mathcal{B}^v_i, j), \forall j \in [c_l, c_r]$.

Delta materialization queries: A $Diff(Q, i, j)$ query looks for all the triples that match the pattern Q and that have changed their state (either occurring or not) when considering versions i and j (i.e., they occur in version i but not in version j, or vice versa). To answer this query, we start again by locating the enhanced candidate triples, and then filter them using the versioning information.

When using the `vpt` strategy, we need to access two different positions in $\mathcal{B}^t_k$. We set $x \leftarrow \mathcal{B}^t_k[i]$ and $y \leftarrow \mathcal{B}^t_k[j]$. Values x and y just determine if the triple occurred in versions i and j respectively. When using the `tpv` strategy, we perform two similar operations to set $x \leftarrow \mathcal{B}^v_i[k]$ and $y \leftarrow \mathcal{B}^v_j[k]$.

Apart from the way of computing the values of x and y, the process to check each candidate is similar in both strategies. If $x = y$ the triple is discarded. If $x \neq y$ the triple is a valid result. In this case, if $x = 1$ the triple existed in version i but was deleted in version j, whereas if $y = 1$ the triple did not occur in version i and was added in version j.

In the `tpv` strategy, we can again use optimized algorithms to answer queries involving `(s??)`, `(sp?)`, and `(spo)` patterns, where candidate triples are consecutive $(t_l, t_{l+1}, \ldots, t_r)$. For any bitsequence B, let us define the operation $p \leftarrow getNext_1(B, pos)$, that locates the position p $(p \geq pos)$ of the first 1 in B, starting the search from position pos onward. Notice that $getNext_1$ can be easily defined in terms of rank and select operations: it returns pos if $B[pos] = 1$, or $select_1(B, 1 + rank_1(B, pos))$ otherwise. Given the range [l,r], corresponding to the location of all the candidate triples, we initially set $p_1 \leftarrow getNext_1(\mathcal{B}^v_i, l)$ and $p_2 \leftarrow getNext_1(\mathcal{B}^v_j, l)$. Then, we iteratively compare and update positions p_1 and p_2, as long as $r \geq p_1$ or $r \geq p_2$. In each step, we perform the following comparison:

- If $p1 < p2$, the triple at position p_1 is added to the result because it existed in version i but it was removed in version j. We set $p_1 \leftarrow getNext_1(\mathcal{B}^v_i, p_1 + 1)$ to point to the next triple that occurred in version i.

- If $p_2 < p_1$, the triple at position p_2 is added to the result because it did not occur in version i, and it was added in version j. We set $p_2 \leftarrow getNext_1(\mathcal{B}^v_j, p_2 + 1)$ to point to the next triple in version j.
- If $p_1 = p_2$ the triple is discarded because it did not change. We must update both $p_1 \leftarrow getNext_1(\mathcal{B}^v_i, p_1 + 1)$ and $p_2 \leftarrow getNext_1(\mathcal{B}^v_j, p_2 + 1)$.

The search process continues until $r < p_1$ and $r < p_2$.

Version queries: The query $Ver(Q)$ looks for all the triples matching the query pattern in any version of the archive. The results, therefore, include all the versions in which each of the candidate triples was active. In our proposal, this is translated into checking, for each candidate triple and version, whether the candidate triple was active in that specific version. That is, for each candidate triple at position k, in the `vpt` strategy we check if $\mathcal{B}^t_k[i]$ is set to 1, and in the `tpv` strategy, we check if bit $\mathcal{B}^v_i[k]$ is set to 1. We repeat this process for all the versions $i \in [1, N]$.

As in previous queries, optimized algorithms can also be used on the `tpv` strategy for triple patterns `(s??)`, `(sp?)`, and `(spo)`. Knowing that all the candidate triples are in a contiguous range $[l, r]$, we can avoid traversing all the candidate triples. Instead, we can iterate over the N bitsequences $\mathcal{B}^v_i$. In each of them, we can initially set $p = l$, and then compute $p \leftarrow getNext_1(\mathcal{B}^v_i, p + 1)$ to locate the positions of all the ones in the range. This process continues while $p \leq r$, and returns the location of all the triples that are valid for any version.

Note that, in version queries, we can also optimize `vpt` when it obtains the versions in which a given candidate triple k occurs. Instead of sequentially traversing $\mathcal{B}^t_k[i]$ to find the position of the 1s, we can use $getNext_1$ to directly obtain those positions. This optimization would apply for all triple patterns, yet, in practice, we would expect that it could only improve performance in datasets with a large number of versions.

4 Experiments

In this section we describe the experimental evaluation of `v-RDF-SI` that we have conducted. First, in Section 4.1 we outline the datasets and variants used, as well as the main details of the experimental method used. Then, in Section 4.2, we evaluate the performance of the different components of `v-RDF-SI`, in the *rdf-layer* and the *ver-layer*. Finally, in Sections 4.3 and 4.4, we compare our solution with Jena and OSTRICH, two state-of-the-art alternatives.

4.1 Experimental framework

We have implemented different variants of `v-RDF-SI` that use different data structures for the *rdf-layer* and the *ver-layer*. They are experimentally evaluated in this section. First, `v-RDF-SI` can use either `RDFCSA` or `HDT` as the main data structure for the *rdf-layer*, therefore yielding two variants `v-RDFCSA` and `v-HDT`. Second, for each variant of the *rdf-layer*, we can choose the bitsequence organization strategy in the *ver-layer*: `vpt` stores a bitsequence per triple with its versioning information, whereas `tpv` stores a bitsequence per version in the collection. Third, for each of the previous variants, we test different bitsequences to store the versioning information: we test `plain` bitmaps, as well as `RG`, `RRR`, `Delta`, and `OZ` representations. In order to achieve compression in the `vpt` variant, we concatenate all the `vpt` individual bitsequences and build compressed representations for the full bitsequence. Note that we easily retain the ability to access the individual bitsequences because they all have the same number of bits.

The implementations used for `RG`, `RRR`, and `SDarray` are available at libcds[6] library. `Delta` [1] is available at http://pizzachili.dcc.uchile.cl/cst/. For `OZ`, we created our own implementation following the guidelines in [46]. We designed `OZ` in such a way that it chooses the most compact representation for each of the underlying bitsequences Z and O among: *(i)* `RG` with $factor = 20$, *(ii)* `RRR` using sampling rate $t_{rrr} = 32$, *(iii)* `SDarray`, and *(iv)* `Delta` with sampling rate $t_b = 64$. By doing so, we ensure the best possible compression at the cost of probably not using the fastest alternative.

Finally, we also test optimized versions of the query algorithms that take advantage of the rank/select features in most bitsequence representations to speed up some of the queries. As discussed in Section 3.3, these variants (labeled ^{OPT}), can take advantage in some query patterns (`(s??)`, `(sp?)`, and `(spo)`) of the fact that the query results correspond to contiguous ranges in the versioning bitsequences, using rank/select operations to retrieve the versioning information instead of searching results separately. These optimized versions for the `tpv` variant are used in our experiments for `v-RDFCSA` and `v-HDT`. We do not consider the optimization proposed for version queries in the `vpt` variant, since it requires a relatively large number of versions to be useful in practice.

We conduct our experimental evaluation using the BEAR benchmark [25], that provides an RDF archive, a complete collection of query sets covering all the query strategies and triple patterns, and a baseline archiving implementation.

The RDF archive used in BEAR consists of 58 versions of a large dataset, storing weekly crawls from more than 650 domains. This results in an heterogeneous corpus with over 2 billion triples. Disregarding version information, there are only 376 million version-oblivious triples. Among them, there is a small core of 3.5 million triples that do not change at any point in the archive. The dataset grows along time, from 33 million triples in the first version to 66 million triples in the last one. The rate of changes in the dataset is relatively high, with 31% of the triples changing between consecutive versions on average. Table 1 summarizes these statistics of the dataset. We will use two sizes as a reference point for compression: the plain size of the dataset (stored in NTriples[7] format) is `325GiB`, whereas a gzipped version of the NTriples file requires only `23GiB`.

[6] Obtained from https://github.com/fclaude/libcds
[7] https://www.w3.org/2001/sw/RDFCore/ntriples/

$\lvert\mathcal{A}\rvert$	# versions	$\lvert V_0\rvert$	$\lvert V_{57}\rvert$	$\overline{\delta}$	$\mathcal{C}_{\mathcal{A}}$	$\mathcal{O}_{\mathcal{A}}$
2,073M	58	30M	66M	31%	3.5M	376M

Table 1 Corpus statistics; $\lvert\mathcal{A}\rvert$: total triples in the Archive, $\lvert V_0\rvert$,$\lvert V_{57}\rvert$: # triples at version V_0 and V_{57}, $\overline{\delta}$: mean % of triples that change between versions, $\mathcal{C}_{\mathcal{A}}$: static core of the archive (unchanged triples), $\mathcal{O}_{\mathcal{A}}$: number of version-Oblivious triples.

BEAR also contains a varied testbed including *Mat*, *Diff*, and *Ver* queries. It provides queries for all the basic triple patterns: `(s??)`, `(?p?)`, `(??o)`, `(sp?)`, `(s?o)`, `(?po)`, and `(spo)`. For most triple patterns, two query sets are provided, categorized by the number of results of the queries: Q_L (low cardinality: low number of results), and Q_H (high cardinality: high number of results) queries. Each query set consists of 50 different queries. We refer the reader to the original article [25] for additional details on the dataset and query sets, and explanations on a few exceptions. For instance, for `(?p?)` queries a smaller number of queries exists, and for `(spo)` and `(s?o)` queries only low cardinality query sets could be defined.

Finally, BEAR deploys an archiving system that can be used as a baseline. It is based on the Jena TDB store[8] (referred to as Jena hereinafter). The three basic archiving strategies are implemented in this baseline: i) Jena-Ic uses an independent store to represent each version of the dataset; ii) Jena-Cb stores indexes for the triples that are added and deleted in each version; and iii) Jena-Tb annotates each triple with its versioning information and stores the resulting quads in a single store.

`v-RDF-SI` was implemented in C++, whereas Jena was implemented in Java. In the performance comparisons with Jena, we measure elapsed time and execute a warm-up phase in Jena, running each query set twice for each variant and measuring query times during the second execution. This tries to mitigate any effect on the performance of Jena due to disk accesses, and is our best effort to provide the fairest possible comparison between both solutions. Nevertheless, we note that Jena is designed as a disk-based solution, so it may incur by design in other overheads that `v-RDF-SI` does not have, hence a completely fair comparison is not possible due to their different nature.

All our experiments were performed in a system with 2x Intel Xeon E5-2643-v2 @3.50GHz (12 cores, 24 siblings) and 256GiB RAM (DDR3 @1.60GHz). The operating system is Ubuntu 16.04. Our prototype was compiled using gcc 4.8 with full optimizations enabled. The Jena-based solution was compiled and executed using Java version 8.

4.2 Space-time tradeoff of our variants

In order to measure the relative space-time tradeoff of all our implementation variants, we executed all the query patterns using `v-RDFCSA` and `v-HDT`, and the

[8] https://jena.apache.org/documentation/tdb/

variants vpt and tpv with all the bitsequence representations. When possible, we also ran the optimized solutions of some queries.

Note that in this section we ignore the space and time required by the dictionary component of the final solution since the same dictionary can be added to all the solutions and its cost will be identical in all of them.

4.2.1 Choosing a bitsequence implementation

First, we focus on showing how the six different representations for bitsequences lead to different space-time tradeoffs in our v-RDF-SI proposal. Table 2 displays, for each bitsequence strategy, the actual parameters used to tune it in our experiments, and its size (in GiB) when we represent the *ver-layer* of our RDF archive following the tpv and vpt strategies.

Strategy	parameters	Space (GiB)	
		tpv	vpt
Plain	none	2.54	2.54
RG	factor=20	2.67	2.67
SDarray	none	1.69	1.54
RRR	t=64	1.38	1.58
Delta	t_b=64	0.65	0.97
OZ	none	0.34	0.71

Table 2 Detailed space needs for the *ver-layer*.

We can see that the compressed bitsequence representations are more effective for the tpv strategy, probably because those bitsequences have more redundancies than in the vpt counterpart. Also, we can see that compressed bitsequence representations obtain important space savings. Particularly, Delta and OZ, the most effective strategies, require roughly around 15–40% the size of the plain bitsequence.

Figures 6 and 7 show the results obtained by v-RDFCSA for the most relevant query sets. We only display results for the tpv variant, and specifically for subject and object lookup queries with high cardinality (scenarios Q_H^S and Q_H^O in BEAR). We choose this subset of queries because they are the most representative ones, and we opt for the query sets with high cardinality because they better display the differences in performance. Results for the other query patterns, and for query sets with low cardinality, are omitted for brevity, but display roughly the same relative performance among bitsequence implementations. We also omit for brevity similar results for v-HDT, because they follow the same trends. The space-time tradeoff displayed for each variant corresponds to the sampling parameter $t_\psi \in \{256, 64, 16\}$, that affects the query performance of v-RDFCSA [9]. Following the same ideas in [20], we compress Ψ by taking differences between consecutive values of Ψ and then encoding those differences using run-length and Huffman coding. Finally, absolute values from Ψ are retained at regular intervals of size t_ψ to ensure pseudo-random

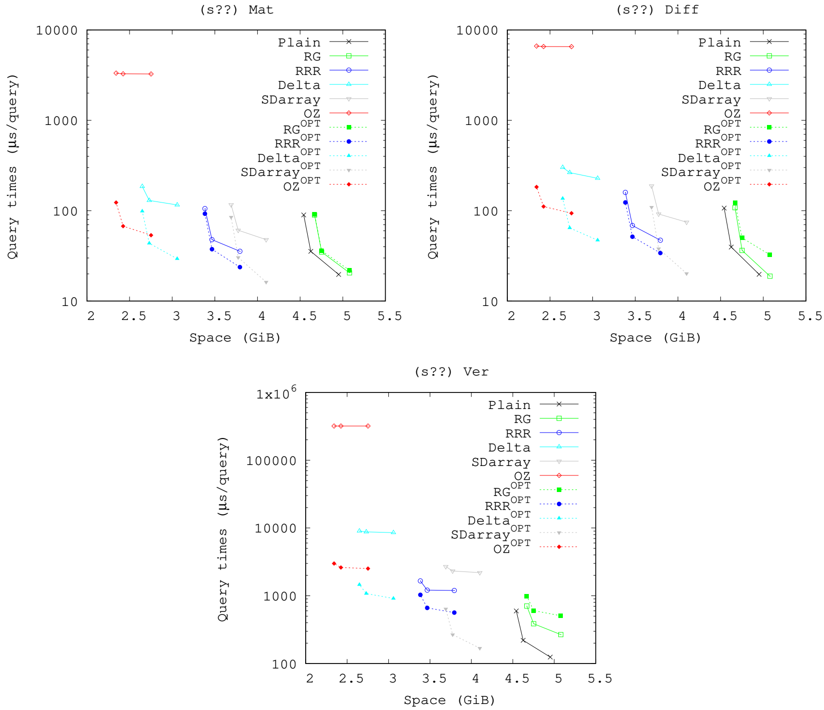


Fig. 6 Query times for subject lookups: Mat, $Diff$, and Ver queries.

access capabilities. Note that, in these figures, we are considering the space required by the triple identifiers and the versioning information combined, and excluding the dictionary.

Our results show that the choice of bitsequence implementation for the versioning information has a very significant impact on the query performance and space requirements. Solutions based on plain bitsequence implementations, and the solutions based on the RG bitsequence (an uncompressed bitsequence with additional support for rank/select queries) require significant space for the overall `v-RDFCSA` representation, whereas the most compact solutions based on `Delta` and `OZ` require much less space. Recall that the space includes both the triple identifiers and the versioning information, but if we considered only the versioning information, plain bitsequence representations would require up to 7 times the space of the most compressed ones.

Figure 6 displays results for subject lookups `(s??)`. For this query we display the results for the basic query algorithms and the optimized ones (labeled with ^{OPT}), that can take advantage of the bitsequence implementations but can only be applied in some query patterns (namely `(s??)`, `(sp?)`, and `(spo)`). Query times are very different between Mat, $Diff$, and Ver queries, but relative comparisons are similar, and we will focus only on relative differences among bitsequences. Results show that `OZ` is the most compact bitsequence, but it is significantly slower when no optimized algorithms are used. The `Delta`

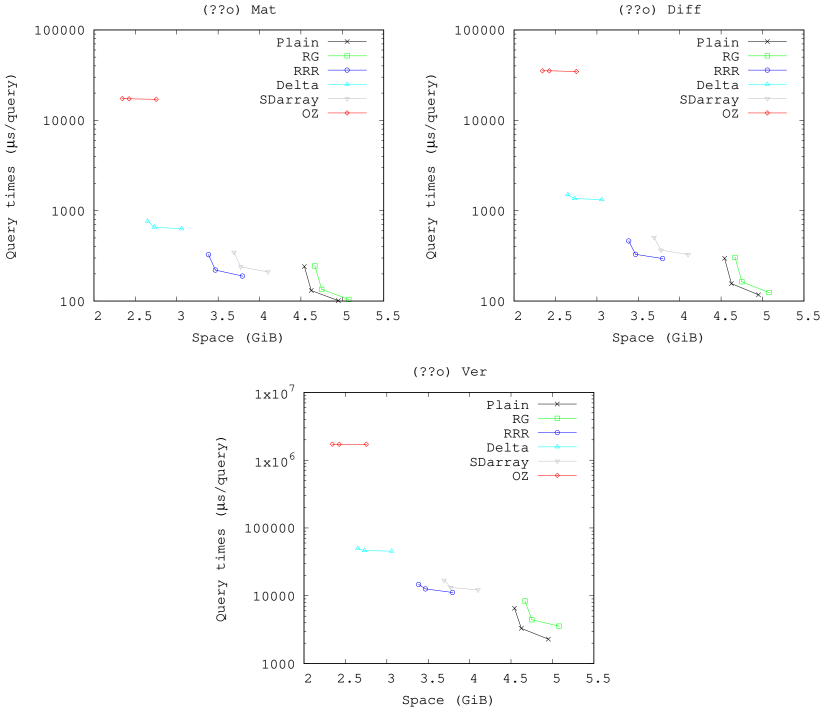


Fig. 7 Query times for object lookups: *Mat*, *Diff*, and *Ver* queries.

bitsequences are also very compact, but they are still much slower when using basic query algorithms; using the optimized query algorithms, they are much more competitive, obtaining query times similar to `RRR` and `SDarray` while requiring significantly less space. The most stable compressed bitsequence is `RRR`, that provides good performance in basic and optimized query algorithms. Finally, both the plain bitsequence representation and the `RG` bitsequence that add rank support are much larger than the alternatives. Additionally, the optimized algorithms seem to have little effect in most cases over uncompressed bitsequences, since the cost of repeated operations is small. Therefore, plain bitsequences with no additional data structures are smaller and usually competitive in query times with `RG`.

Figure 7 displays results for object lookups `(??o)`. Note that in this case no optimized query algorithms exist, so we only display the query times for the regular algorithms. The space-time tradeoff of the different bitsequence implementations is roughly the same as in the previous case. `OZ` is orders of magnitude slower than any other implementation, and `Delta` is also a very good choice for compression but is still 3 times slower than the other proposals. Both `RRR` and `SDarray` offer a reasonable space-time tradeoff, being much faster than `Delta` and much smaller than uncompressed bitsequences, with query times relatively close to those of a plain bitsequence implementation or using `RG`.

The overall comparison results displayed hold for the remaining query patterns: first, a wide space-time tradeoff can be obtained depending on the bitsequence representation chosen to handle the versioning information; second, optimized query algorithms can improve query times significantly in the triple patterns where they can be applied, but they usually provide an advantage only when using compressed bitsequence representations, that are very slow at access operations. In the following sections we will select a subset of the bitsequence configurations that is a good representative of the overall trends and space-time tradeoff. Specifically, for the query patterns that support optimized query algorithms (i.e. (s??), (sp?), (spo), in the tpv versioning strategy) we will display query times only for those optimized algorithms. Additionally, we will display query times for plain bitsequences, RRR, and Delta, as representatives of the space-time tradeoff that can be achieved depending on the bitsequence representation.

4.2.2 Comparing the versioning strategy

Next, we compared the space-time tradeoff of v-RDF-SI depending on both the variant used for the storage of triples and the versioning strategy.

Figures 8 and 9 display the results for v-RDFCSA and v-HDT for subject and object lookups respectively. We display results for the two alternatives to handle the versioning information, vpt and tpv, and considered the bitsequence representations Plain, RRR, and Delta. Again, we only display results for (s??) and (??o) as representative queries, since the trends are similar in the other query patterns.

Let us focus on the comparison between versioning variants vpt and tpv. The two plots in Figure 8 (top) display the performance on subject lookups for version materialization and delta materialization queries. In both cases, and independently of the bitsequence and data structure used, the tpv versioning strategy obtains faster query times, in addition to smaller space usage. The difference is particularly significant in the most space-efficient solutions: the tpv implementation using Delta bitsequences requires less than half the time of the vpt counterpart. Figure 9 (top) displays similar results for object lookups, and again the tpv strategy is faster and smaller.

When considering version queries (bottom plots in Figures 8 and 9), results are slightly different: for subject lookups (s??), displayed in Figure 8 (bottom), the tpv variant is still faster, thanks to the performance of the optimized query algorithms. However, these optimizations are only available for 3 query patterns, namely (s??), (sp?) and (spo). In the remaining query patterns, the vpt strategy becomes faster, as displayed in Figure 9 (bottom) for (??o) patterns.

Considering the overall results, the choice of vpt or tpv versioning strategy is affected mainly by the type of query to be executed: vpt is slower in general in all Mat and $Diff$ queries. However, in Ver queries the decision depends on the triple pattern: when optimized algorithms are available for tpv, this strategy is still faster, but when using basic algorithms vpt becomes the fastest

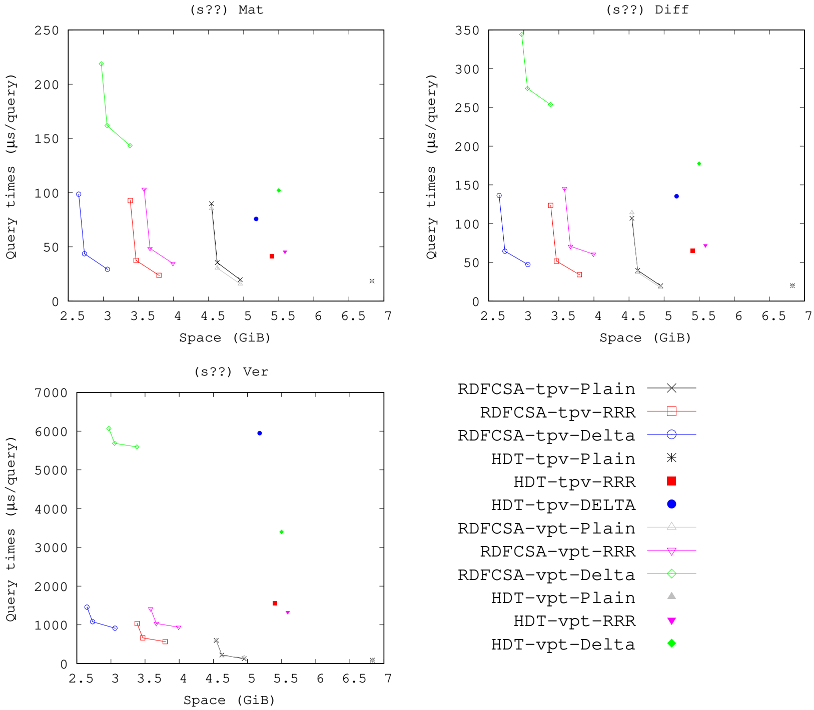


Fig. 8 Subject lookups: *Mat*, *Diff*, and *Ver* queries. Times in μs/query, space in GiB.

alternative. Note that optimized query algorithms can always be used when available, but could also be switched off without changes to the structure, the choice of versioning strategy must be determined on construction time. We consider that `tpv` is the best strategy overall, with `vpt` being only useful when *Ver* queries are expected to be very frequent.

4.3 Comparison with the baseline in BEAR

In this section, we extend our experimental evaluation by comparing our variants with the Jena baseline deployed in BEAR. Throughout this section, we consider not only the query times to perform the query on the versioned triples encoded as IDs, but also the time required to transform the query components to IDs and the reverse transformation of results to the corresponding strings. To do this, we consider a default dictionary, based on *Front-Coding*, to perform the transformations [40]. The dictionary adds 2.3GiB on top of the space requirements of any of our `v-RDF-SI` variants, and requires 1.5–5 μs to process each query result, adding from a few microseconds to a few milliseconds to the query response times of `v-RDFCSA` and `v-HDT`, depending on the query pattern, because the number of results returned can largely differ among them. In this section, we will display results for the seven basic triple pattern queries, in order to better characterize the difference in performance between

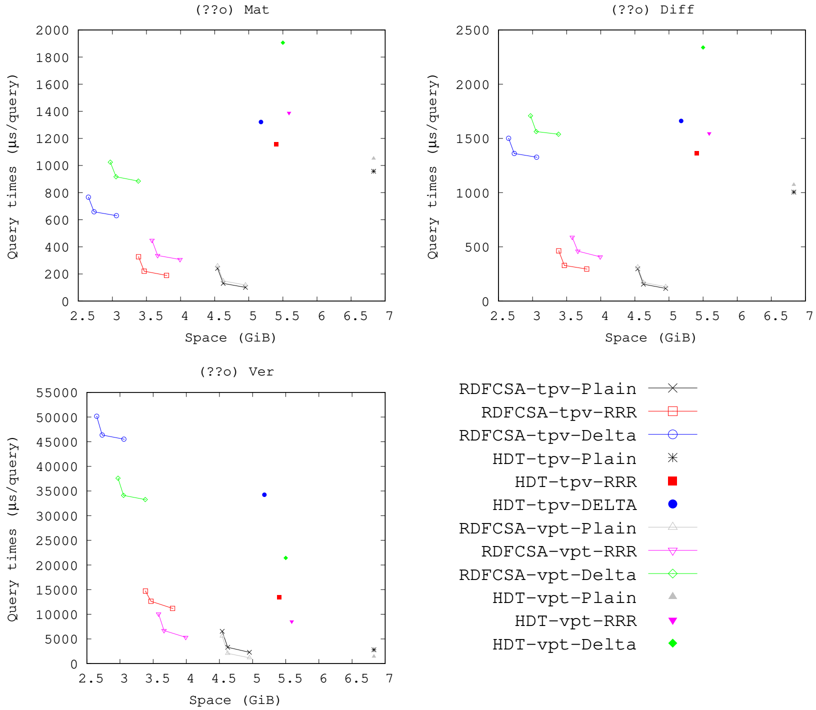


Fig. 9 Object lookups: *Mat*, *Diff*, and *Ver* queries. Times in μs/query, space in GiB.

our proposals v-RDFCSA and v-HDT, and the Jena baselines. For v-RDFCSA we set $t_\Psi = 64$ which provides a balanced space/time tradeoff.

A first important consideration is the space required by each solution. As shown in the previous sections, our variants require roughly 5–9 GiB to store the complete dataset (including the dictionary), depending on the chosen variant. On the other hand, Jena-IC, Jena-CB, and Jena-TB need 230GiB, 138GiB, and 81GiB, respectively. These differences in space are expected, due to the different nature of the solutions. However, note that even the less compact representations based on v-RDFCSA or v-HDT use roughly an order of magnitude less space than Jena. In the rest of this section, we will focus on query performance in order to show that our solutions are not only significantly smaller but also much faster on average than the Jena baseline.

Figures 10 and 11 display the results for all the basic triple patterns on version materialization queries; i.e. the times required to recover all the triples matching the triple pattern in each version. We display results for the 3 main Jena variants, as well as for 4 of our proposals, in order to cover the space-time tradeoff provided: we show results for v-HDT and v-RDFCSA with Plain and Delta bitsequences. All our implementations use the tpv versioning strategy, that is faster than vpt in all cases for this type of queries. Note that the bitsequence implementations selected represent two opposite end points that draw the overall space-time tradeoff provided in both v-RDFCSA and v-HDT.

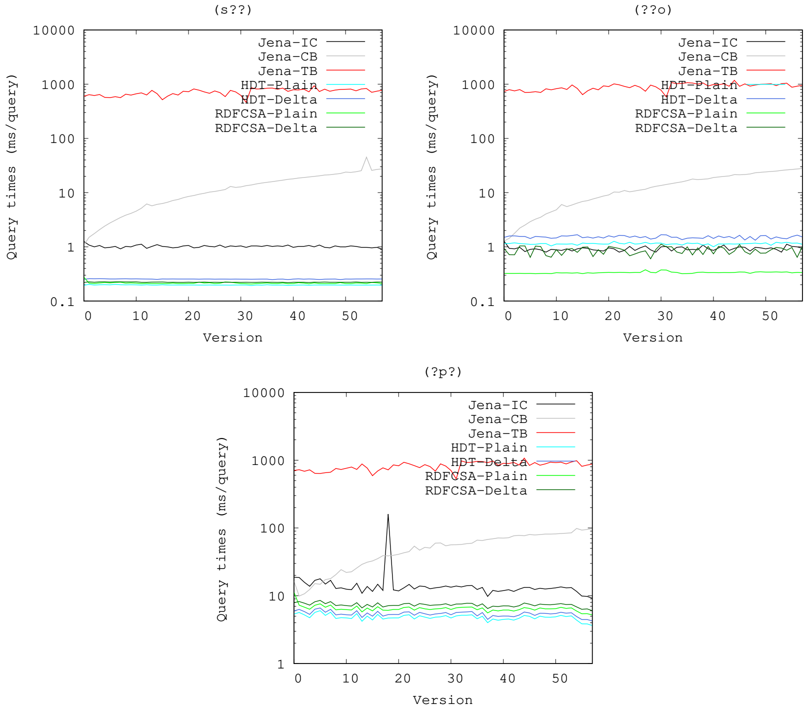


Fig. 10 Version materialization queries for subject `(s??)`, object `(??o)`, and predicate `(?p?)` lookups. Times in ms/query.

Figure 10 displays the results for the most representative query patterns: subject, object, and predicate lookups. Our proposals are faster in general than the Jena baselines. Jena-IC is the most efficient approach from the baseline, as expected, but it is still slower than our fastest solutions in all cases. For `(s??)` queries, Jena-IC requires around 1 ms/query (millisecond per query), whereas `v-RDFCSA` and `v-HDT` use around 0.2 ms/query. In `(??o)` and `(?p?)` queries, Jena-IC is competitive with our solutions in query times, requiring around 1 ms/query in `(??o)` and 10 ms/query in `(?p?)`. Jena-CB is also competitive for queries in the first versions of the datasets, but due to its change-based nature it becomes slower as the version increases, and is on average 5–10 times slower than Jena-IC. Note also that the difference between our implementation alternatives, `v-RDFCSA` and `v-HDT`, as well as the difference between bitsequence implementations, plain and `Delta`, are relatively small: `v-HDT` is faster in `(s??)` and `(?p?)`, whereas `v-RDFCSA` is faster in `(??o)`, but all the solutions are roughly in the same order of magnitude. The effect of the dictionary component is very significant in this scenario, since for queries that return a relatively large number of results, the cost of the dictionary lookups needed to unmap the resulting ID-based triples into the final string-based triples is higher than the cost of querying the *rdf-layer* and the *ver-layer*. This is the case for these three representative patterns.

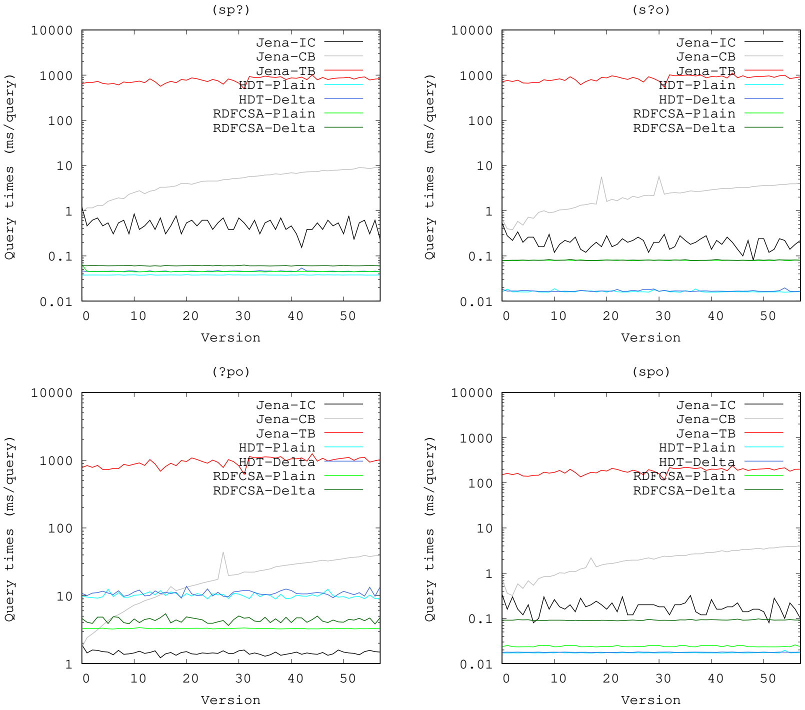


Fig. 11 Version materialization queries for `(sp?)`, `(s?o)`, `(?po)`, and `(spo)` patterns. Times in ms/query.

Figure 11 displays the results for `(sp?)`, `(s?o)`, `(?po)`, and `(spo)` queries. As in the previous triple patterns, our solutions are overall faster than the Jena baseline. Among our variants, `v-HDT` is faster in most cases, and the differences between bitsequence implementations are not very significant in most cases. Only Jena-IC is competitive in query times with our solutions, and is even able to beat them in `(?po)` queries. In the remaining patterns, `v-HDT` and `v-RDFCSA` are faster than Jena-IC, and `v-HDT` is usually an order of magnitude faster. Jena-CB is again competitive only for queries in the first versions of the archive, but around 10 times slower on average.

Figures 12 and 13 display the results for all the basic pattern queries on delta materialization queries. We perform the basic delta materialization queries between version 0 and versions 5, 10, . . . , 55, 57, and display the evolution of the query times as the gap between those two versions increases. Therefore, the plots display the performance of delta materialization queries for increasingly large version gaps.

Results show that, in general, our solutions are faster for delta materialization queries, yet the Jena baselines are competitive in some query patterns. Particularly, in `(??o)` (Figure 12, top-right) and `(?po)` (Figure 13, bottom-left) Jena-IC obtains query times similar to our solutions, but our fastest solutions based on plain bitsequences still yield the best query times. Taking

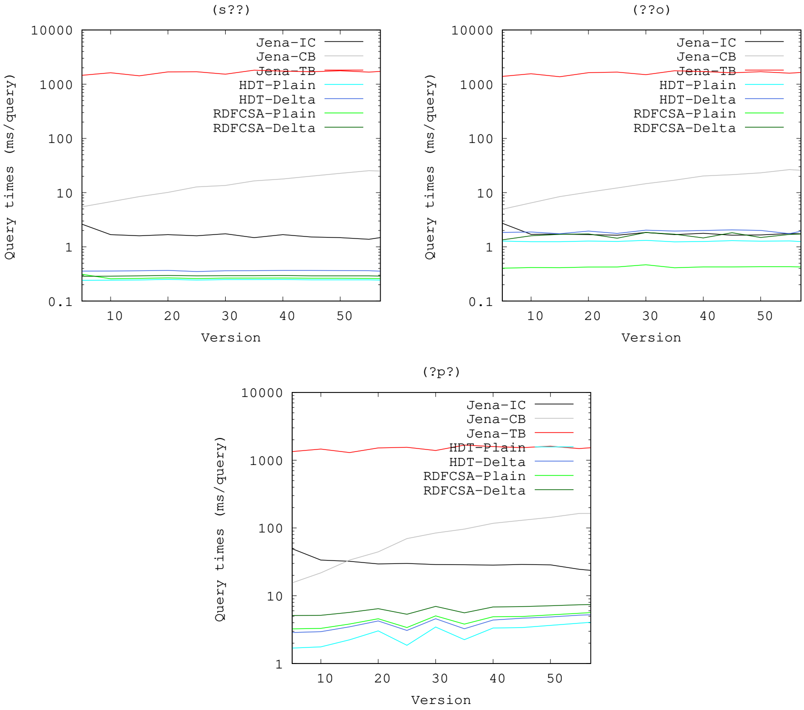


Fig. 12 Delta materialization queries for subject, predicate, and object lookups. Times in ms/query.

into account that Jena-IC requires roughly 20–40 times more space than our representations, we consider our solutions still preferable in this scenario, even if Jena is competitive in query times. For the remaining query patterns, all our variants are faster than Jena, and our best solution is always `v-HDT` with plain bitsequences. It achieves query times that range between 0.01–0.003 ms/-query for the most selective queries displayed in Figure 13 and 2–4 ms/query for `(?p?)` queries, where the cost of the query is dominated by the dictionary lookups. Overall, our solutions provide very stable and efficient query times across all patterns, with worst-case query times below 10 ms/query even for those queries returning many results (up to 1000 triples per query in `(?p?)`).

Figures 14 and 15 display the results for version queries, that retrieve all occurrences matching the query pattern among all versions of the dataset. As in the previous query types, we first display results for the representative subject, object, and predicate lookups in Figure 14 and then for the more selective patterns in Figure 15. In this scenario, we display query times for our approaches using the `vpt` strategy, that showed to be faster than `tpv` in most query patterns.

Both `v-RDFCSA` and `v-HDT` are much faster than the Jena baselines in most query patterns. For `(s??)` and `(??o)` queries, displayed in Figure 14, our fastest variants are 5 times faster than Jena-IC and Jena-CB, which obtain

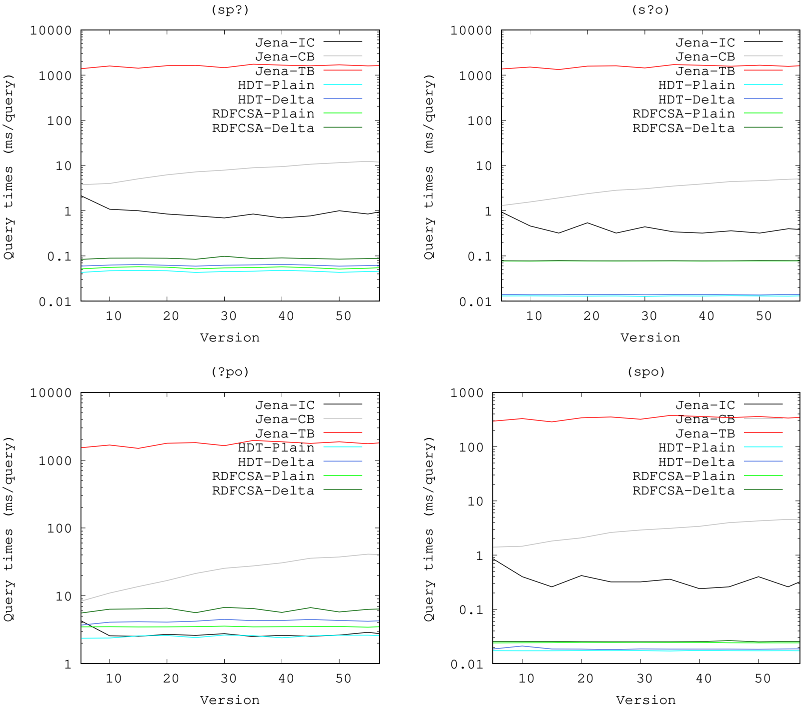


Fig. 13 Delta materialization queries for (sp?), (s?o), (?po), and (spo) patterns. Times in ms/query.

the best results in the baseline. Similar comparison results appear for most of the remaining patterns, including the more selective queries displayed in Figure 15. In most cases, all our variants are faster than the best Jena baseline, in some cases up to 3 orders of magnitude faster. However, Jena-CB is faster in (?p?) queries (Figure 14, bottom), where most of the query time in our representations is spent in dictionary accesses. Jena-CB is also competitive with our variants using Delta compressed bitmaps in (?po) and (??o) queries. Among our solutions, v-HDT is faster than v-RDFCSA in general, and the solutions using plain bitsequences are faster than those using compressed Delta bitsequences, as expected.

Considering the overall results for *Mat*, *Diff*, and *Ver* queries, our variants clearly improve the baseline both in space requirements and query performance. The Jena-IC baseline is competitive in query times with our best solutions, but requires 20–40 times more space. Jena-CB is competitive with our proposals in version queries, but much less efficient in the remaining queries, and still much larger than our proposals. Jena-TB, the most compact baseline, is orders of magnitude slower than any of our solutions.

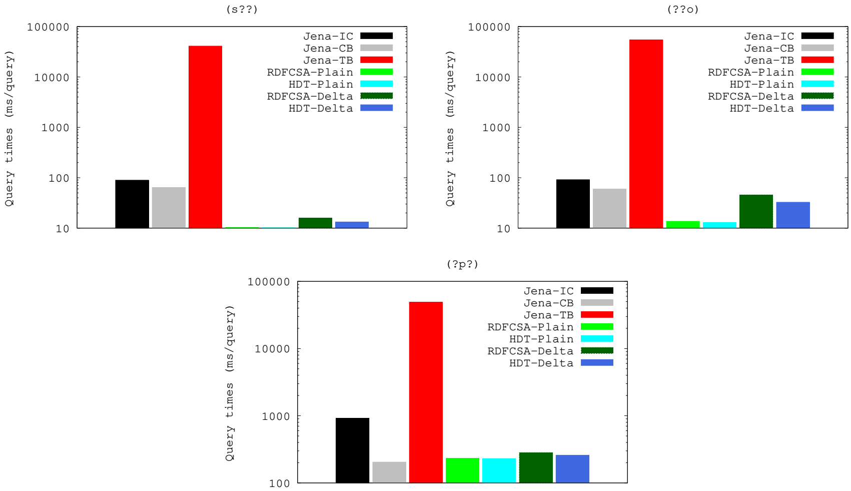


Fig. 14 Version queries for subject, predicate, and object lookups. Times in ms/query.

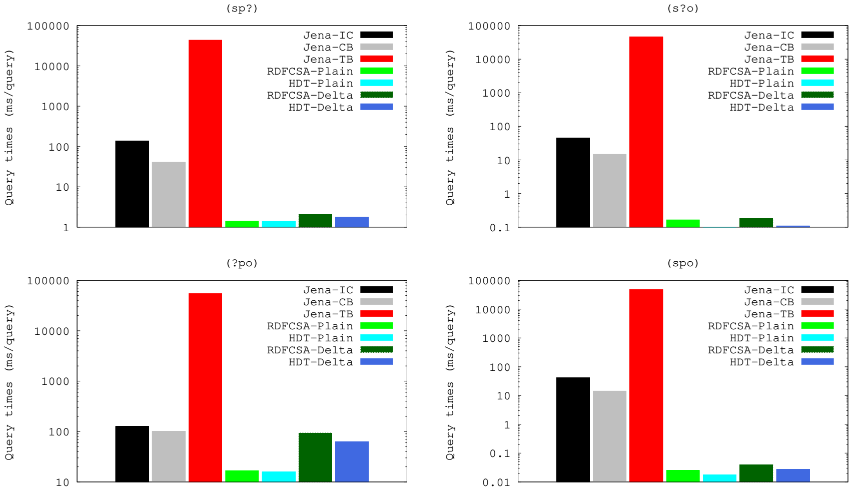


Fig. 15 Version queries for `(sp?)`, `(s?o)`, `(?po)`, and `(spo)` patterns. Times in ms/query.

4.4 Comparison with OSTRICH

We have compared our proposal with OSTRICH [56], a state-of-the-art compact solution offering good space/time tradeoffs [51]. Since we were unable to build the representation of the full BEAR dataset with OSTRICH in reasonable time, and in order to provide an estimation of the relative performance of this tool, we have performed tests using just the first 10 versions of the dataset. OSTRICH requires around 5GiB to store these versions, whereas `v-RDFCSA`, is able to represent them in 1.15 GiB using `Plain` bitmaps (`Delta` bitmaps barely improve space due to the small number of versions). Note that most of the space is taken by the string dictionary; the representation of the integer triples requires less than 0.5 GiB in v-RDFCSA.

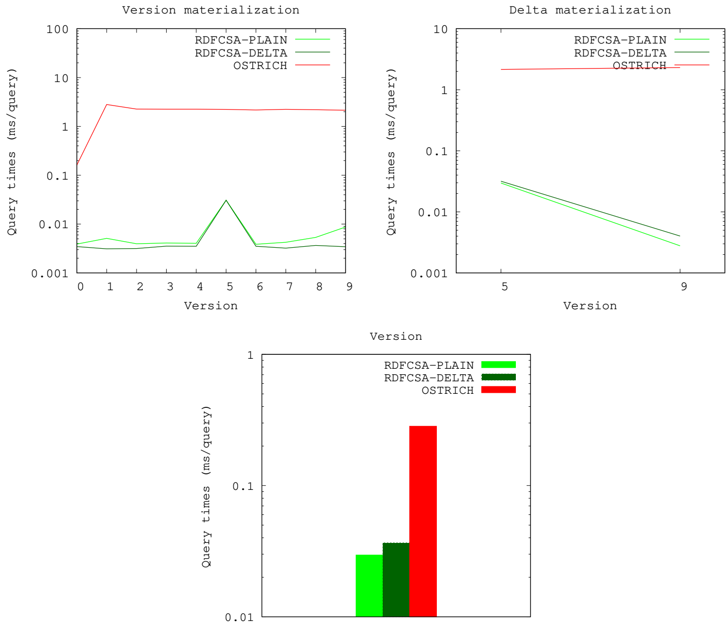


Fig. 16 Times for version materialization queries (top left), delta materialization queries (top right) and version queries (bottom) in OSTRICH and v-RDFCSA. Times in ms/query.

Next, we compare the query performance of v-RDFCSA with OSTRICH. Figure 16 displays a summary of query times obtained running the original BEAR query set on just the first 10 versions. We include results only for the (s??) pattern, but similar trends appear in other patterns, with our proposal being significantly faster than OSTRICH. In version materialization queries (top-left plot), OSTRICH is more competitive when accessing a snapshot (version 0), but is still slower than v-RDFCSA in all cases. In delta materialization and version queries, our proposal is again clearly faster than OSTRICH.

4.5 Summary of experiments

In previous sections, we have covered a variety of variants and query scenarios. In this section, we summarize the overall results and discuss the main trends identified in our experimentation.

The variants of v-RDF-SI tested provide a wide space-time tradeoff: our solution requires roughly 4.5–9GiB to store the full BEAR archive, that requires 325GiB when stored in plain format. The overall size of our representation depends on the choice for the *rdf-layer* and the *ver-layer*. The *rdf-layer*, that stores the version-oblivious triples, requires 1.9–2.3GiB to store the triples when compressed using v-RDFCSA, and 4.3GiB when using v-HDT. Additionally, a Front-coding dictionary is necessary to translate string to ids, requiring an additional 2.3GiB. The *ver-layer* requires 0.3–2.7GiB, depending on the choice of bitsequence and versioning strategy. The bitsequence implementation used is the most important factor: Plain bitsequences require 2.5GiB, whereas compact implementations such as Delta or OZ require less than 1GiB.

The choice of triple representation, the bitsequence implementation and the versioning strategy have a significant effect in query times. In the *rdf-layer*, v-RDFCSA is slower than v-HDT for most query patterns. In the *ver-layer*, variants with Plain bitsequences are one order of magnitude faster than using Delta, ignoring the dictionary query times. The tpv versioning strategy is faster in general than vpt in Mat and $Diff$ queries, especially in queries such as (s??) and (sp?), for which optimized algorithms exist only in the tpv strategy. In Ver queries, the vpt strategy is slightly faster in most queries. Recall, however, that the experiments in Section 4.2 do not consider the dictionary query times. Therefore, differences between variants are less extreme in Section 4.3, once the dictionary query times are incorporated to the measurements.

When compared with the Jena baseline, v-RDF-SI has much lower space requirements and is also clearly faster in almost all queries. Our largest solution is v-HDT with Plain bitsequences, and uses around 9GiB; a much more compact solution based on v-RDFCSA and Delta bitsequences requires around 5GiB. Jena-based solutions, on the other hand, use 80-230GiB, and are slower than our slowest solution in most of the queries. Overall, our fastest variants based on plain bitmaps yield the best query times in almost all cases, even if Jena-IC and Jena-CB are competitive in some queries. Query times of our proposals depend heavily on the query. Our solutions range from less than 0.1 ms/query in (spo) version materialization queries to 300 ms/query for (?p?) version queries. In most queries v-HDT is 2–5 times faster than the equivalent v-RDFCSA, and Delta bitsequences can yield query times up to 5 times slower than Plain bitsequences.

Overall, we consider that v-RDFCSA and the tpv versioning strategy offer the best tradeoff, and that compressed bitsequences are probably the best choice in most scenarios. Note that differences in query times are usually less significant in the most complex queries (see, for instance, (?p?) queries in Figure 14), so the alternatives that achieve better compression are very competitive for most applications. The vpt versioning strategy is still useful for Ver queries, and Plain bitsequences of v-HDT can be used to prioritize performance over memory consumption.

Finally, our experimental comparison with OSTRICH shows that v-RDFCSA clearly overcomes OSTRICH in both space and time. It uses around 30% of its space and yields 1–2 orders of magnitude faster query times.

5 Conclusions and Future Work

In this paper we introduce `v-RDF-SI`, a solution able to efficiently handle large RDF archives in compressed space. Our proposal separates the archive representation into two layers, and provides alternative implementations for each of them: the *rdf-layer* stores the version-oblivious triples in the dataset, and is implemented using `RDFCSA` or `HDT`; the *ver-layer* stores the versioning information, and is implemented following two strategies (`tpv` and `vpt`) that can be coupled with any bitsequence representation. By exploiting the compression capabilities and performance of compact data structures in both layers, `v-RDF-SI` is able to achieve very good compression while providing efficient query support.

Our experimental evaluation using the BEAR benchmark shows that `v-RDF-SI` clearly outperforms the reference Jena baseline, by reducing space requirements up to 40 times while yielding query times much faster on average, and competitive in all queries. In addition, our results show that a space-time tradeoff can be achieved in `v-RDF-SI`: solutions based on `v-RDFCSA` and compressed bitsequences require approximately half the space of solutions based on `v-HDT` and plain bitsequences, but are also 2–5 times slower in most cases. Furthermore, the two versioning strategies that we present yield different performance depending on the queries, being `tpv` the best choice overall.

In addition, we have included a comparison with OSTRICH, a state-of-the-art solution for RDF archiving, where we show that our proposal uses roughly one quarter of the space required by OSTRICH while being significantly faster at query time. This permits us to conclude that our proposals make up a relevant contribution for RDF archiving.

As future work, we plan to integrate our solution in a full SPARQL engine, so it can provide versioning functionalities on all types of SPARQL queries. In addition, we plan to explore other possibilities for the *rdf-layer*. A major pending challenge is the support for changeset ingestion. Dynamic dictionaries or small collections of static dictionaries could be used to handle new versions, but combining these with the rigid ID assignment requirements in the *rdf-layer* is the major challenge. The *rdf-layer* could be implemented using dynamic compact data structures like the dynamic `k2-tree` [10], but this would require to apply some of the changes described in Section 3.1.3 to keep track of triple ordering. Another alternative is the creation of a dynamic `RDFCSA`, using dynamic variants of the CSA [14, 38, 45]. Insertions in the *ver-layer* could be easily implemented in the `tpv` variant with static bitsequences for each new version, but more complex solutions involving dynamic bitsequences [16] would be required for `vpt`.

Statements and declarations

Acknowledgments. The first three co-authors are members of the CITIC, which, as Research Center accredited by the Galician University System, is funded by Consellería de Cultura, Educación e Universidades from Xunta de Galicia, supported in an 80% through ERDF Funds, ERDF Operational Programme Galicia 2014-2020, and the remaining 20% by Secretaría Xeral de Universidades [grant ED431G 2019/01]. The Spanish group is also funded by Xunta de Galicia/FEDER-UE [ED431C 2021/53]; by MICINN [Magist: PID2019-105221RB-C41; FLATCity-POC: PDC2021-121239-C31; SIGTRANS: PDC2021-120917-C21; EXTRA-Compact: PID2020-114635RB-I00; PID2019-105221RB-C41]; by MCIU-AEI/FEDER-UE [BIZDEVOPS: RTI2018-098309-B-C32]; and by Xunta de Galicia / Igape/ IG240.2020.1.185.

Data availability. The BEAR dataset used in our experiments is available at https://aic.ai.wu.ac.at/qadlod/bear.html (named BEAR-A).

References

[1] Abeliuk A, Cánovas R, Navarro G (2013) Practical compressed suffix trees. Algorithms 6(2):319–351. https://doi.org/10.3390/a6020319

[2] Ali W, Saleem M, Yao B, et al (2022) A survey of RDF stores & SPARQL engines for querying knowledge graphs. The VLDB Journal 31(3):1–26. https://doi.org/10.1007/s00778-021-00711-3

[3] Álvarez-García S, Brisaboa N, Fernández J, et al (2015) Compressed Vertical Partitioning for Efficient RDF Management. Knowledge and Information Systems 44(2):439–474. https://doi.org/10.1007/s10115-014-0770-y

[4] Arndt N, Naumann P, Radtke N, et al (2019) Decentralized collaborative knowledge management using git. Journal of Web Semantics 54:29–47. https://doi.org/10.1016/j.websem.2018.08.002

[5] Atre M, Chaoji V, Zaki MJ, et al (2010) Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In: Proceedings of the 19th International Conference on World Wide Web (WWW), pp 41–50, https://doi.org/10.1145/1772690.1772696

[6] Bigerl A, Conrads F, Behning C, et al (2020) Tentris – A Tensor-Based Triple Store. In: Proceedings of the 19th International Semantic Web Conference (ISWC), pp 56–73, https://doi.org/10.1007/978-3-030-62419-4_4

[7] Bizer C, Meusel R, Primpel A, et al (2022) Web data commons—microdata, RDFa, JSON-LD, and microformat data sets. URL https://webdatacommons.org/structureddata/

[8] Brisaboa N, Ladra S, Navarro G (2014) Compact Representation of Web Graphs with Extended Functionality. Information Systems 39(1):152–174. https://doi.org/10.1016/j.is.2013.08.003

[9] Brisaboa N, Cerdeira A, Fariña A, et al (2015) A compact RDF store using suffix arrays. In: Proceedings of the 22nd International Symposium on String Processing and Information Retrieval (SPIRE). Springer, Cham, LNCS 9309, pp 103–115, https://doi.org/10.1007/978-3-319-23826-5_11

[10] Brisaboa NR, Cerdeira-Pena A, de Bernardo G, et al (2017) Compressed representation of dynamic binary relations with applications. Information Systems 69:106–123. https://doi.org/10.1016/j.is.2017.05.003

[11] Brisaboa NR, Cerdeira-Pena A, de Bernardo G, et al (2019) Improved compressed string dictionaries. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM). ACM, pp 29–38, https://doi.org/10.1145/3357384.3357972

[12] Brisaboa NR, Cerdeira-Pena A, de Bernardo G, et al (2022) Space/time-efficient rdf stores based on circular suffix sorting. Journal of Supercomputing 79(5):5643–5683. https://doi.org/10.1007/s11227-022-04890-w

[13] Cerdeira-Pena A, Fariña A, Fernández JD, et al (2016) Self-indexing RDF archives. In: Proceedings of the Data Compression Conference (DCC). IEEE, pp 526–535, https://doi.org/10.1109/DCC.2016.40

[14] Chan HL, Hon WK, Lam TW, et al (2007) Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2):21–es. https://doi.org/10.1145/1240233.1240244

[15] Claude F, Navarro G (2009) Practical rank/select queries over arbitrary sequences. In: Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE). Springer, Berlin, Heidelberg, LNCS 5280, pp 176–187, https://doi.org/10.1007/978-3-540-89097-3_18

[16] Cordova J, Navarro G (2016) Practical dynamic entropy-compressed bitvectors with applications. In: Proceedings of the 15th International Symposium on Experimental Algorithms (SEA), LNCS 9685, pp 105––117, https://doi.org/10.1007/978-3-319-38851-9_8

[17] Curé O, Blin, Guillaume, et al (2014) Waterfowl: A compact, self-indexed and inference-enabled immutable RDF store. In: Proceedings of the 11th Extended Semantic Web Conference (ESWC), LNCS 8465, pp 302–316, https://doi.org/10.1007/978-3-319-07443-6_21

[18] Dong-Hyuk I, Sang-Won L, Hyoung-Joo K (2012) A Version Management Framework for RDF Triple Stores. International Journal of Software Engineering and Knowledge Engineering 22(1):85–106. https://doi.org/10.1142/S0218194012500040

[19] Erling O, Mikhailov I (2009) RDF support in the Virtuoso DBMS. In: Networked Knowledge - Networked Media. Studies in Computational Intelligence, vol 221, Springer, Berlin, Heidelberg, p 7–24, https://doi.org/10.1007/978-3-642-02184-8_2

[20] Fariña A, Brisaboa NR, Navarro G, et al (2012) Word-based Self-Indexes for Natural Language Text. ACM Transactions on Information Systems 30(1):article 1:. https://doi.org/10.1145/2094072.2094073

[21] Fernández J, Martínez-Prieto M, Gutiérrez C, et al (2013) Binary RDF representation for publication and exchange (HDT). Journal of Web Semantics 19:22–41. https://doi.org/10.1016/j.websem.2013.01.002

[22] Fernández JD, Martínez-Prieto MA (2018) RDF Serialization and Archival, Springer, Cham, pp 1–11. https://doi.org/10.1007/978-3-319-63962-8_286-1

[23] Fernández JD, Llaves A, Corcho O (2014) Efficient RDF interchange (ERI) format for RDF data streams. In: Proceedings of the 13th International Semantic Web conference (ISWC). Springer, Berlin, Heidelberg, LNCS 8797, pp 244–259, https://doi.org/10.1007/978-3-319-11915-1_16

[24] Fernández JD, Polleres A, Umbrich J (2015) Towards Efficient Archiving of Dynamic Linked Open Data. In: Proceedings of the First DIACHRON Workshop on Managing the Evolution and Preservation of the Data Web. Co-located with 12th Extended Semantic Web Conference (ESWC), pp 34–49, URL http://ceur-ws.org/Vol-1377/

[25] Fernández JD, Umbrich J, Polleres A, et al (2019) Evaluating query and storage strategies for RDF archives. Semantic Web Journal 10(2):247–291. https://doi.org/10.3233/SW-180309

[26] Fernández N, Arias J, Sánchez L, et al (2014) RDSZ: an approach for lossless RDF stream compression. In: Proceedings of the 11th Extended Semantic Web Conference (ESWC), Springer, Cham, LNCS 8465, pp 52–67, https://doi.org/10.1007/978-3-319-07443-6_5

[27] Gomes D, Costa M, Cruz D, et al (2013) Creating a Billion-scale Searchable Web Archive. In: Proceedings of the 22nd International Conference on World Wide Web (WWW Companion). Association for Computing Machinery, New York, NY, USA, pp 1059–1066, https://doi.org/10.1145/2487788.2488118

[28] González R, Grabowski S, Mäkinen V, et al (2005) Practical implementation of rank and select queries. In: Poster Proceedings of the 4th Workshop on Efficient and Experimental Algorithms (WEA). CTI Press and Ellinika Grammata, pp 27–38

[29] Graube M, Hensel S, Urbas L (2014) R43ples: Revisions for triples. In: Proceedings of the 1st Workshop on Linked Data Quality (LQD)

[30] Grossi R, Gupta A, Vitter JS (2003) High-order entropy-compressed text indexes. In: Proceedings of the 14th annual ACM-SIAM Symposium on Discrete Algorithms (SODA). Society for Industrial and Applied Mathematics, USA, pp 841–850, https://doi.org/10.5555/644108.644250

[31] Harris S, Seaborne A (2013) SPARQL 1.1 Query Language. W3C Recommendation, http://www.w3.org/TR/sparql11-query/

[32] Hasemann H, Kröller A, Pagel M (2012) Rdf provisioning for the internet of things. In: Proceedings of the 3rd IEEE International Conference on the Internet of Things (IOT), pp 143–150, https://doi.org/10.1109/IOT.2012.6402316

[33] Hernández-Illera A, Martínez-Prieto M, Fernández J (2015) Serializing RDF in Compressed Space. In: Proceedings of the Data Compression Conference (DCC). IEEE Computer Society, USA, pp 363–372, https://doi.org/10.1109/DCC.2015.16

[34] Hernández-Illera A, Martínez-Prieto M, Fernández J, et al (2020) iHDT++: improving HDT for SPARQL triple pattern resolution. Journal of Intelligent & Fuzzy Systems 39(2):2249–2261. https://doi.org/10.3233/JIFS-179888

[35] Käfer T, Abdelrahman A, Umbrich J, et al (2013) Observing Linked Data Dynamics. In: Proceedings of the 10th Extended Semantic Web Conference (ESWC). Springer, Berlin, Heidelberg, LNCS 7882, pp 213–227, https://doi.org/10.1007/978-3-642-38288-8_15

[36] Klein M, Fensel D, Kiryakov A, et al (2002) Ontology Versioning and Change Detection on the Web. In: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW). Springer, Berlin, Heidelberg, LNCS 2473, pp 197–212, https://doi.org/10.1007/3-540-45810-7_20

[37] Lhez J, Ren X, Belabbess B, et al (2017) A compressed, inference-enabled encoding scheme for RDF stream processing. In: Proceedings of the 14th Extended Semantic Web Conference (ESWC). Springer, Berlin, Heidelberg, LNCS 10250, pp 79–93, https://doi.org/10.1007/978-3-319-58451-5_6

[38] Mäkinen V, Navarro G (2008) Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4(3):article 32. https://doi.org/10.1145/1367064.1367072

[39] Manber U, Myers G (1993) Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5):935–948. https://doi.org/10.1137/0222058

[40] Martínez-Prieto M, Brisaboa N, Cánovas R, et al (2016) Practical Compressed String Dictionaries. Information Systems 56:73–108. https://doi.org/10.1016/j.is.2015.08.008

[41] Martínez-Prieto MA, Arias Gallego M, Fernández JD (2012) Exchange and consumption of huge RDF data. In: Proceedings of the 9th Extented Semantic Web Conference (ESWC). Springer, Berlin, Heidelberg, LNCS 7295, pp 437–452, https://doi.org/10.1007/978-3-642-30284-8_36

[42] Martínez-Prieto MA, Fernández JD, Hernández-Illera A, et al (2018) RDF Compression, Springer, Cham, pp 1–11. https://doi.org/10.1007/978-3-319-63962-8_62-1

[43] Martínez-Prieto MA, Fernández JD, Hernández-Illera A, et al (2020) Knowledge Graph Compression for Big Semantic Data, Springer, Cham, pp 1–13. https://doi.org/10.1007/978-3-319-63962-8_62-2

[44] Meinhardt P, Knuth M, Sack H (2015) Tailr: a platform for preserving history on the web of data. In: Proceedings of the 11th International Conference on Semantic Systems (SEMANTICS). Association for Computing Machinery, New York, NY, USA, pp 57–64, https://doi.org/10.1145/2814864.2814875

[45] Munro JI, Nekrich Y, Vitter JS (2015) Dynamic data structures for document collections and graphs. In: Proceedings of the 34th ACM Symposium on Principles of Database Systems (PODS). Association for Computing Machinery, New York, NY, USA, pp 277–289, https://doi.org/10.1145/2745754.2745778

[46] Navarro G (2016) Compact Data Structures – A practical approach. Cambridge University Press, NY, USA, https://doi.org/10.1017/CBO9781316588284

[47] Navarro G, Providel E (2012) Fast, small, simple rank/select on bitmaps. In: Proceedings of the 11th International Conference on Experimental Algorithms (SEA). Springer, Berlin, Heidelberg, LNCS 7276, pp 295–306, https://doi.org/10.1007/978-3-642-30850-5_26

[48] Neumann T, Weikum G (2010) The RDF-3X engine for scalable management of RDF data. The VLDB Journal 19(1):91–113. https://doi.org/10.1007/s00778-009-0165-y

[49] Neumann T, Weikum G (2010) x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. Proceedings of the VLDB Endowment 3(1-2):256–263. https://doi.org/10.14778/1920841.1920877

[50] Okanohara D, Sadakane K (2007) Practical entropy-compressed rank/select dictionary. In: Proceedings of the Meeting on Algorithm Engineering & Expermiments (ALENEX). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp 60–70, https://doi.org/10.5555/2791188.2791194

[51] Pelgrin O, Galárraga L, Hose K (2021) Towards Fully-fledged Archiving for RDF Datasets. Semantic Web Journal Pre-press:1–24. https://doi.org/10.3233/sw-210434

[52] Pibiri GE, Perego R, Venturini R (2020) Compressed indexes for fast search of semantic data. IEEE Transactions on Knowledge and Data Engineering https://doi.org/10.1109/TKDE.2020.2966609

[53] Raman R, Raman V, Rao S (2002) Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of the 13th annual ACM-SIAM Symposium on Discrete Algorithms (SODA). Society for Industrial and Applied Mathematics, USA, pp 233–242, https://doi.org/10.5555/545381.545411

[54] Sadakane K (2003) New Text Indexing Functionalities of the Compressed Suffix Arrays. Journal of Algorithms 48(2):294–313. https://doi.org/10.1016/S0196-6774(03)00087-7

[55] Schreiber G, Raimond Y (2014) RDF Primer. W3C Recommendation, https://www.w3.org/TR/rdf11-primer/

[56] Taelman R, Vander Sande M, Van Herwegen J, et al (2019) Triple storage for random-access versioned querying of RDF archives. Journal of Web Semantics 54:4–28. https://doi.org/10.1016/j.websem.2018.08.001

[57] Thompson BB, Personick M, Cutcher M (2014) The Bigdata® RDF graph database. In: Linked Data Management. Chapman and Hall/CRC, chap 8, p 1–46, https://doi.org/10.1201/b16859

[58] Vander Sander M, Colpaert P, Verborgh R, et al (2013) R&Wbase: Git for Triples. In: Proceedings of the WWW2013 Workshop on Linked Data on the Web (LDOW), vol CEUR-WS 996, LDOW paper 1. CEUR-WS.org, p 5, URL http://ceur-ws.org/Vol-996

[59] Völkel M, Groza T (2006) Semversion: An RDF-based ontology versioning system. In: Proceedings of the IADIS international conference WWW/Internet (ICWI), pp 195–202, URL http://www.iadisportal.org/

digital-library/semversion-an-rdf-based-ontology-versioning-system

[60] Weiss C, Karras P, Bernstein A (2008) Hexastore: Sextuple indexing for semantic web data management. Proc VLDB Endowment 1(1):1008–1019. https://doi.org/10.14778/1453856.1453965