Self-Indexing RDF Archives

Ana Cerdeira-Pena*, Antonio Fariña*, Javier D. Fernández†, and
Miguel A. Martínez-Prieto‡

Database Lab.
Facultade de Informática
Univ. of A Coruña, Spain

† Institute for Information Business
Vienna University of Economics and
Business (WU), Austria

‡DataWeb Research
Dept. of Computer Science
Univ. of Valladolid, Spain

acerdeira@udc.es, fari@udc.es, jfernand@wu.ac.at, migumar2@infor.uva.es

Abstract

Although Big RDF management is an emerging topic in the so-called Web of Data, existing techniques disregard the dynamic nature of RDF data. These RDF archives evolve over time and need to be preserved and queried across it. This paper presents `v-RDFCSA`, an RDF archiving solution that extends RDFCSA (an RDF self-index) to provide version-based queries on top of compressed RDF archives. Our experiments show that `v-RDFCSA` reduces space requirements up to $35 - 60$ times over a state-of-the-art baseline, and gets more than one order of magnitude ahead over it for query resolution.

1 Introduction

The last decade has seen an impressive growth of the use of RDF (*Resource Description Framework*) [13] in the Web. Projects such as *Linked Open Data* or *schema.org* have promoted RDF as *de facto* standard to describe facts about any field of knowledge in the Web. This knowledge is continuously evolving in this emerging Web of Data [11], and these changes bring new dataset versions which should be archived to cover needs such as studying the data evolution across time or reverting changes. However, previous experiences in Web archiving highlight scalability problems when managing evolving information at Web-scale [8]. The Semantic Web is starting to face similar challenges when archiving historical RDF data (referred to as *RDF archives*), making the task of longitudinal querying across time a challenge [5].

Although different strategies have been proposed to manage RDF archives their storage requirements are far from being considered effective. For instance, these strategies need up to 15 times the space used by gzip for storing a state-of-the-art RDF archive [7]. In this scenario, RDF compression arise as a natural choice as many techniques provide efficient query resolution [1, 2, 6]. However, they consider a static RDF datasets and, to the best of our knowledge, they have not been tailored to be used in RDF archives.

This paper describes `v-RDFCSA`, the first approach that natively provides efficient query resolution on top of compressed RDF archives. It represents all different RDF triples in the archive using RDFCSA [2], a self-index that ensures SPARQL resolution on the compressed data. In turn, information about versions is succinctly encoded using bitsequences that allow version-based queries to be efficiently resolved. Experiments on BEAR [7], the state-of-the-art benchmark for RDF archives, show that `v-RDFCSA` encodes an archive of 325GB in just $5.7 - 7.3$GB, and provide query resolution one order of magnitude faster than a reference baseline.

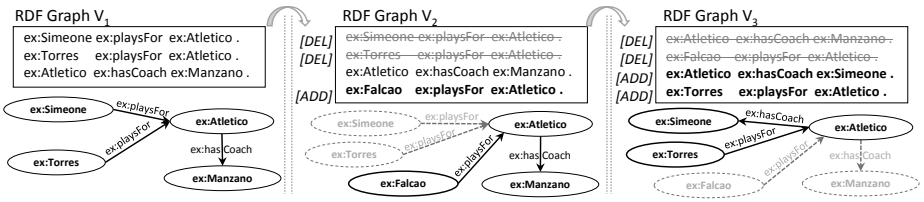

Figure 1: Example of RDF graph versions.

The rest of the paper is organized as follows. Section 2 provides the necessary background to understand our approach, which is explained in Section 3. Section 4 compares `v-RDFCSA` performance with other reference strategies used for RDF archiving. Finally, Section 5 includes conclusions and devises future lines of research.

2 Background

RDF and SPARQL are the main technologies for archiving applications. RDF [13] states facts in the form of ternary structures (triples). Each triple relates a *subject* data entity and an *object* value using a *predicate* property. In practice, each triple is a simple graph in which the predicate labels the edge from the subject to the object node. Thus, a set of RDF triples is a labelled directed graph which connects multiple descriptive facts about subject entities. SPARQL [10] is a graph-pattern matching language designed for querying RDF datasets. It combines triple patterns (RDF triples in which each component may be variable) using joins, unions, etc.

RDF archive. An RDF archive organizes versions of an RDF dataset by annotating triples with version labels. Following [7], a *version-annotated triple* is an RDF triple (s, p, o) with a label $i \in [1, \mathcal{N}]$ representing the version in which this triple holds. Thus, an *RDF archive* $\mathcal{A}$, is a set of version-annotated triples.

Figure 1 illustrates a 3-version RDF archive that describes facts about players and coaches of the *"Atlético de Madrid"* soccer team. First version, V_1, models two players: `ex:Torres` and `ex:Simeone`, and the coach `ex:Manzano`. Both players leave the team in V_2, but `ex:Falcao` is hired as a new player. Finally, the former player `ex:Simeone` replaces `ex:Manzano` as the coach in V_3, whereas `ex:Falcao` leaves the team in favour of `ex:Torres`, who rejoins `ex:Atletico`.

Retrieval Functionality. RDF archives provide SPARQL queries focused on particular *versions* or on differences (*deltas*) between two or more given versions in the archive. All these possible operations can be built using three primitive queries [5]:

- *Version materialisation* queries: $Mat(Q, V_i)$, returns bindings for the SPARQL query Q in version V_i. For instance, $Mat($`(ex:Atletico,ex:hasCoach,?x)`$, V_2)$ obtains `ex:Manzano` as Atlético's coach in V_2.
- *Delta materialisation* queries: $Diff(Q, V_i, V_j)$, retrieves bindings that match Q in V_i but not in V_j (*deleted bindings*), and vice versa (*added bindings*); e.g.

$Diff($`(?x,playsFor,ex:Atletico)`$, V_1, V_2)$ returns `ex:Simeone` and `ex:Torres` as deleted bindings in V_2, and `ex:Falcao` as an added binding.

- *Version* queries: $Ver(Q)$, resolves Q and annotates bindings with versions in which each of them holds; e.g. $Ver($`(ex:Atletico,hasCoach,?x)`$)$ returns that `ex:Manzano` is present in V_1 and V_2, and `ex:Simeone` is the coach in V_3.

More complex queries can be built on top of these primitives. For instance, *which players have been also coaching the team?* This query can be resolved as i) $Mat($`(?x,ex:playsFor,ex:Atletico`$, V_i) \bowtie Mat($`(ex:Atletico,ex:hasCoach,?x)`$,V_j)$, $\forall i, j \in \mathcal{N}$; or ii) by joining two version operations: $Ver($`(?x, ex:playsFor,ex:Atletico)`$) \bowtie Ver($`(ex:Atletico,ex:hasCoach,?x)`$)$.

State of the Art of RDF Archiving

RDF archiving has been approached from three different strategies [5]. Techniques keeping *independent copies* (IC) manage versions as different datasets [12]. This organization enables version materialisation queries to be efficiently resolved, but penalizes the other ones, and brings an obvious space overhead because of duplicated triples among the versions. *Change-based* (CB) approaches perform a differential encoding between consecutive versions. That is, V_i is stored as the differences (added and deleted triples) with respect to the previous version. This reorganization reduces space requirements and speeds up delta queries, but heavily penalizes version queries because of the need for delta propagation. Practical CB approaches [4] store fresh, fully materialized versions every k deltas. Finally, *timestamp-based* (TB) approaches [17] regard the archive as a single dataset comprising all different triples, which are annotated with version information (when they are added or deleted). Version queries exploit this triple arrangement, but practical TB deployments demand to index this additional version-based dimension, which leads to significant space overheads.

A recent archiving benchmark, BEAR [7], shows that each strategy excels for their expected query, at the price of huge space requirements (they needed $\approx 200-350$GB for encoding an archive whose gzipped size is only 23GB). This scenario clearly claims for a more compact and effective representation.

State of the Art of RDF Compression

HDT [6] was the precursor of RDF compressors based on succinct data structures. It transforms the RDF graph into a forest of subject-rooted trees, which are encoded using rank/select bitsequences [9]. It reports good compression ratios and is able to efficiently resolve SPARQL triple patterns. K^2-triples [1] exploits structural redundancy from the graph topology to outperform HDT effectiveness. It performs a predicate-based partition of the dataset into disjoint subsets of (subject, object) pairs, which are compressed as (sparse) binary matrices using k^2-trees [3]. The most recent RDF compressor is RDFCSA [2]. It revisits compressed suffix-arrays to index triples as cyclic strings. Although it does not report the best compression ratios, it offers stable and predictable times to solve SPARQL triple patterns. This feature drives us to choose RDFCSA as the core of our current approach.

3 Self-Indexing RDF archives (v-RDFCSA)

v-RDFCSA is designed as a *lightweight* TB approach which encodes independently i) the set of different triples in the archive, and ii) their versioning information. A self-index is chosen for triples, and succinct bitsequences are used for versions. We design efficient retrieval algorithms which exploit RDFCSA [2] self-indexing capabilities for accessing the compressed triples, and then perform bit-based operations on versioning information. Compression decisions and retrieval algorithms are explained below.

RDF triples encoding

v-RDFCSA manages only the set of different triples in the archive. These *version-oblivious triples* [7] are a small fraction of the total triples, e.g. the BEAR archive [7] contains ≈ 2 billion triples, but only ≈ 376 million are version-oblivious. Thus, the problem of RDF triples encoding is reduced to the compression of version-oblivious triples, as a regular RDF dataset. We choose RDFCSA [2], a self-index based on Sadakane's Compressed Suffix Array (CSA) [16], which indexes a set of triples rather than a text, and retains CSA functionality for pattern searching.

RDFCSA first performs the dictionary transformation on the original triples. It replaces RDF terms by integer IDs in ranges $[1, n_s]$ for subjects, $[1, n_p]$ for predicates, and $[1, n_o]$ for objects. Figure 2 (left) shows the set of version-oblivious triples used in the archive described before. It comprises $n = 5$ different triples which, in the middle, are transformed into their ID-based representation[1] (mappings between RDF terms and IDs are independently indexed using compressed string dictionaries [14]).

The right side of the figure illustrates the four steps to turn ID-based triples into an RDFCSA self-index. The sequence S_{id}, in step 1, organizes the list of sorted ID-triples. The first triple is stored in $S_{id}[1, 3]$, the second in $S_{id}[4, 6]$, and so on. Triple IDs are rewritten in step 2 to avoid ID overlappings between subjects, predicates, and objects. Subject values are not changed, so their IDs are in $[1, n_s]$. Predicate IDs range now from $[n_s + 1, n_s + n_p]$, and object values are in $[n_s + n_p, n_s + n_p + n_o]$. For instance, the triple $(1, 1, 2)$ is transformed into $(1, 5, 8)$. This ID rearrangement ensures that subject IDs are smaller than predicate IDs, and that these are smaller than object IDs. This fact leads to a particular suffix array configuration (step 3) where subjects appear in $SA[1, n]$, predicates in $SA[n + 1, 2n]$, and objects in $SA[2n + 1, 3n]$. This suffix array is finally compressed, in step 4, using CSA structures: D and ψ [16]. $D[1, n]$ is a bitmap where 1 bits mark the first suffix in SA starting with each different symbol in the alphabet. On the other hand, the array $\psi[1, n]$ enables the suffix array to be traversed by exploiting that $SA[\psi[i]] = SA[i] + 1$. That is, if $SA[i] = j$ points to the suffix $S[j, n]$, then $SA[\psi[i]] = j + 1$ points to the next text suffix $S[j + 1, n]$.

RDFCSA modifies $\psi[2n + 1, 3n]$, the region encoding jump information from objects. Originally, ψ allows for jumping from the object of the k^{th} triple to the subject of the $(k + 1)^{th}$ triple. It is not useful for SPARQL needs because it relates two independent triples. RDFCSA modifies ψ so that values $\psi[2n+1, 3n]$ point to the subject of the same triple. That is, $\psi[i] \leftarrow \psi[i] - 1, \forall i \in [2n + 1, 3n]$ (or $\psi[i] \leftarrow n$ if $\psi[i] = 1$).

[1] Note that $n_s = 4$, $n_p = 2$, $n_o = 3$, and IDs 1 and 2 are used both for subject and objects.

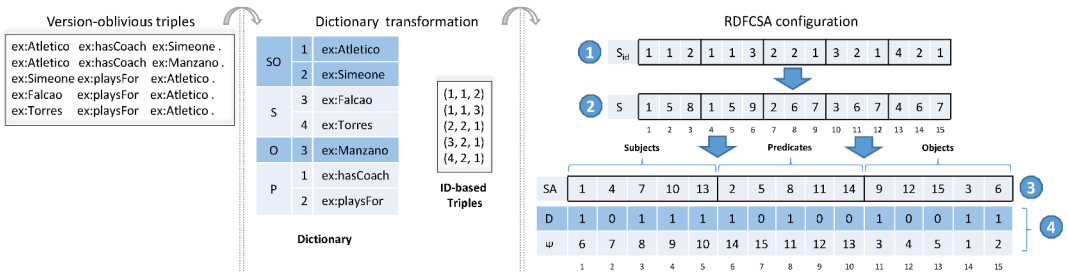

Figure 2: Step-by-step RDFCSA construction for the RDF archive.

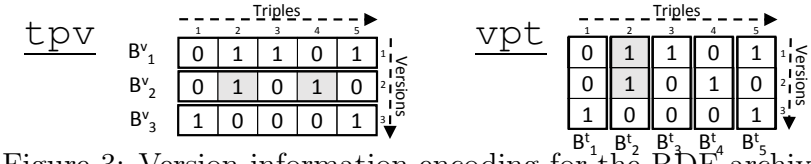

Figure 3: Version information encoding for the RDF archive.

D and ψ allow SPARQL triple patterns to be resolved by an initial binary search followed by a traversal to recover the matching triples ([2] describes these algorithms).

Version information encoding

A subtle yet interesting feature of RDFCSA is that it allows any triple to be identified by the position of its subject within SA. If (s_a,p_b,o_c) is the k^{th} triple ($1 \leq k \leq n$), we can recover it by retrieving its subject as[2] $s_a \leftarrow S[SA[k]]$, its predicate as $p_b \leftarrow S[SA[\psi[k]]]$, and its object as $o_c \leftarrow S[SA[\psi[\psi[k]]]]$.

We propose two encoding strategies which exploit this feature to store versioning data. Let us assume an archive $\mathcal{A}$ which comprises N different versions and a set of n *version-oblivious triples*. The first encoding strategy (called `tpv`: *triples per version*) uses N bitsequences $\mathcal{B}_i^v[1,n]$ to encode what triples appear in the corresponding version i. That is, if $\mathcal{B}_i^v[k] = 1$, it means the k^{th} triple appears in the i^{th} version; otherwise $\mathcal{B}_i^v[k] \leftarrow 0$ is set. Figure 3 (left) illustrates the `tpv` encoding for the example archive (e.g. the 2^{nd} version contains triples 2, and 4). Our second strategy (called `vpt`: *versions per triple*) considers n bitsequences $\mathcal{B}_i^t[1,N]$ to encode versions where the k^{th} triple occurs. If $\mathcal{B}_k^t$ describes the k^{th} triple, then $\mathcal{B}_k^t[i] = 1$ means the k^{th} triple ocurrs in the i^{th} version; otherwise, we set $\mathcal{B}_k^t[i] \leftarrow 0$. Figure 3 (right) illustrates `vpt` encoding for the example archive (e.g. the 2^{nd} triple is used in versions 1, and 2).

Note that `tpv` includes N bitsequences of n bits, and `vpt` uses n bitsequences of N bits. Thus, both strategies use $N * n$ bits. However, they show an asymmetric performance for retrieval purposes. We explain the corresponding algorithms below.

Retrieval algorithms

This explanation leaves apart dictionary management and assumes queries and their results are composed of the IDs that make up S_{id} in Figure 2. First, all our algorithms query RDFCSA to retrieve all candidate triples that match a given pattern Q'. We keep track of all positions $SA[k]$ where the subject of each candidate triple appears in the suffix array. From there on, a traversal of the candidate triples uses versioning

[2]Recall $S[SA[i]] = rank_1(D, i)$, where $rank_1$ indicates the number of ones in $D[1, i]$.

information to check: if the k^{th} triple occurs in version i ($Mat(Q', i)$); if a triple changes from version i to j ($Diff(Q', i, j)$); or to gather all the versions in which such triple occurs ($Ver(Q')$). Note that, the subjects of all candidate triples for operations `(s??)`, `(sp?)` and `(spo)` make up a contiguous range suffix array $SA[l, r]$ [2]. This allows us to optimize the access to versioning data during triples traversal.

Version materialisation queries: $Mat(Q', i)$, retrieves triples matching Q' in a given version i. Once subject positions are retrieved from RDFCSA, we use the versioning information to discard triples that do not occur in version i. For `vpt`, each candidate triple k is checked by accessing to the bitsequence $\mathcal{B}^t_k$. If $\mathcal{B}^t_k[i] = 1$ that triple is returned; otherwise it is discarded. `tpv` performs similarly: if $\mathcal{B}^v_i[k]$ is set to 1, we return the triple pointed to from $SA[k]$; otherwise it is discarded. Patterns `(s??)`, `(sp?)` and `(spo)` can be optimized in `tpv`, since the subjects of the candidate triples are contiguous in $SA[l, r]$. Let $c_l \leftarrow rank_1(\mathcal{B}^v_i, l)$ and $c_r \leftarrow rank_1(\mathcal{B}^v_i, r)$, the number of active triples for version i in range [l,r] is $c = c_r - c_l + 1$. Thus, we can solve $Mat(Q', i)$ by only returning triples at positions $k \leftarrow select_1(\mathcal{B}^v_i, j), \forall j \in [c_l, c_r]$[3].

Delta materialisation queries: $Diff(Q', i, j)$, looks for triples that match Q' and have been changed (either added or removed) between versions i and j. Again, the algorithm queries RDFCSA to retrieve all the candidate triples that match Q', and then, for each candidate triple (whose subject is pointed to from $SA[k]$), its versioning information is processed. For `vpt`, it implies two accesses to $\mathcal{B}^t_k$. We retrieve bit values: $x \leftarrow \mathcal{B}^t_k[i]$ and $y \leftarrow \mathcal{B}^t_k[j]$, that indicate if triple k occurs in versions i and j. Similarly, in the case of `tpv` approach, we gather values $x \leftarrow \mathcal{B}^v_i[k]$ and $y \leftarrow \mathcal{B}^v_j[k]$

We use operators $\overline{\texttt{xor}}$ and $\overline{\texttt{and}}$ to check each candidate triple: *i)* if $(0 \neq ((x \ \overline{\texttt{xor}} \ y) \ \overline{\texttt{and}} \ x))$ the triple was active in version i and removed in version j, so it is returned as a *removed* triple. *ii)* if $(0 \neq ((x \ \overline{\texttt{xor}} \ y) \ \overline{\texttt{and}} \ y))$ the triple was not in version i but it is active in version j (it is returned as an *added* triple). *iii)* Otherwise, the candidate triple is discarded.

There is room for optimization in `tpv` when candidate triples are consecutive in $SA[l, r]$. Being a bitsequence B, let $getNext_1(B, pos)$ be a function that returns pos if $B[pos] = 1$; otherwise, it returns the position of the next 1 after pos in B as $select_1(B, 1 + rank_1(B, pos))$. Given the range [l,r], and using $getNext_1$ function, we can solve $Diff(Q', i, j)$ in `tpv` as follows. We set $p_1 \leftarrow getNext_1(\mathcal{B}^v_i)$ and $p_2 \leftarrow getNext_1(\mathcal{B}^v_j)$. Then, we traverse in parallel $\mathcal{B}^v_i$ and $\mathcal{B}^v_j$ at positions with ones p_1 and p_2, while it holds $((p1 \leq r) \ or \ (p2 \leq r))$. At each iteration, we check: *i)* if $(p_1 < p_2)$ then triple at position p_1 is removed in version j, and we set $p_1 \leftarrow getNext_1(\mathcal{B}^v_i, p_1)$; *ii)* if $(p_2 < p_1)$ then triple at position p_2 is added in version j, and we set $p_2 \leftarrow getNext_1(\mathcal{B}^v_j, p_2)$; *iii)* otherwise, we set $p_1 \leftarrow getNext_1(\mathcal{B}^v_i, p_1)$, $p_2 \leftarrow getNext_1(\mathcal{B}^v_j, p_2)$, and continue with the next iteration.

Version queries: $Ver(Q')$, retrieves all triples matching Q', and for all of them, returns the list of versions in which they were active. Thus, for each candidate triple

[3]Given a bitsequence B, $p \leftarrow select_1(B, j)$, indicates the position p of the j^{th} one in B.

k, we check if such triple appears in the i^{th} version. That is, $\forall i \in [1, N]$, we check either if bit $\mathcal{B}_k^t[i]$ is set to 1 for `vpt` approach, or if bit $\mathcal{B}_i^v[k]$ is set to 1 in `tpv` approach. Optimizations are also possible for triple patterns `(s??)`, `(sp?)` and `(spo)` in `tpv`. We could iterate over the N version bitsequences $\mathcal{B}_i^v, i \in [1, N]$; for each of them, we set $p \leftarrow l$, and then use an inner loop that, while $p \leq r$, computes $p \leftarrow getNext_1(\mathcal{B}_i^v, p)$ and reports that triple p is active at version i.

4 Experiments

We evaluate `v-RDFCSA` using BEAR [7], an benchmark that provides an RDF archive, retrieval queries and a baseline archiving implementation. The BEAR archive contains an heterogeneous corpus of 58 versions with a total of $|\mathcal{A}| = 2,073$ million triples, 376 millions of version-oblivious triples, and 3.5 millions of triples appearing in all versions. In turn, a mean of 31% triples changes between versions, and version sizes grow from ≈ 33 million triples in $|V_0|$ to ≈ 66 millions in $|V_{57}|$. The raw archive size (in NTriples) is `325GB`, and it can be reduced up to `23GB` using gzip.

BEAR also contains a varied set of Mat, $Diff$, and Ver queries. For a fair comparison, it designs queries which return a similar number of results per version, and classifies them into Q_L (low number of results), and Q_H (high number of results). For each class, it provides queries for subject: `(s??)`, predicate: `(?p?)`, and object lookups: `(??o)`. Thus, the query set considers six scenarios: Q_L^S, Q_L^P, Q_L^O describe queries with low cardinality, and Q_H^S, Q_H^P, Q_H^O with high cardinality for subject, predicate, and object lookups respectively. BEAR provides 50 different queries for each scenario, except for predicates (6 and 10 queries for Q_L^P and Q_H^P respectively).

Finally, BEAR deploys a baseline archiving system based of Jena TDB store[4], which implements the three archiving strategies described above: i) Jena-Ic indexes each version in an independent store; ii) Jena-Cb creates an index for added and deleted statements per version; and iii) Jena-Tb uses two named graphs per version, annotating when each triple is added or deleted and indexes all in a single TDB store.

We implemented `v-RDFCSA` in C, considering two `v-RDFCSA` variants (implementing `vpt` and `tpv` strategies) which tune their RDFCSA self-index with ψ sampling $t_\psi = \{16, 64, 256\}$. Versioning information is stored in plain form or compressed with RRR [15]. Note that RRR is not able to achieve compression when it is directly applied to individual `vpt` bitsequences. Thus, we concatenate all `vpt` bitsequences and create a single RRR to compress them. Additionally, we exploit rank/select features of RRR to test optimized subject lookups in `tpv-RRR-OPT`.

All experiments were performed on an Intel Xeon E5-2650v2 @ 2.60GHz (32 cores), 256GB RAM. Debian 7.8, and our prototype is compiled using gcc 4.7.2 (option -O9).

`v-RDFCSA` space/time tradeoffs. We first study `v-RDFCSA` performance for all its variants. We only show the most representative results for lack of space. On the one hand, we choose those scenarios using high cardinality queries because they report similar conclusions than for low cardinality queries. On the other hand, we discard predicate lookups because their numbers are just between subject and object ones.

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

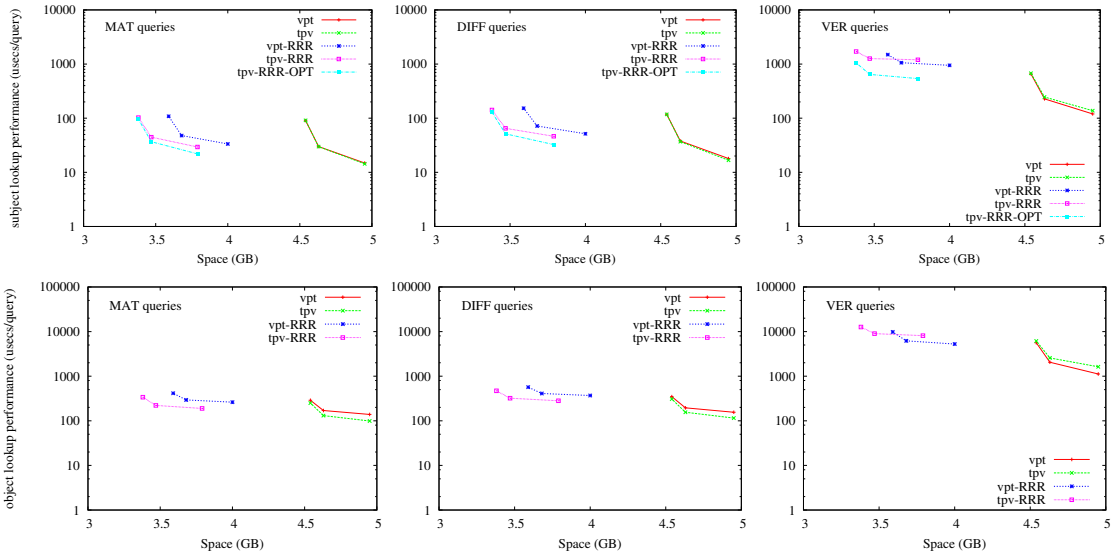

Figure 4: Subject (top) and object (bottom) lookups: Mat, $Diff$, and Ver queries.

Figure 4 summarizes space/time tradeoffs for `v-RDFCSA`. Space requirements (in GB) are described on the X axis, and query times (in μs per query) on the Y axis. Focusing on space requirements, plain `vpt` and `tpv` variants need $4.54 - 4.95$GB (the space increases from $t_\psi = 256$ to $t_\psi = 16$), but using RRR reduces up to $3.38 - 3.79$GB (`tpv`) and $3.59 - 4.00$GB (`vpt`). It demonstrates that versioning information is also compressible, and compressed bitsequences can save more than 1GB in the best case. However, RRR times are slower than those reported on plain bitsequences. Even so, they compete for Mat and $Diff$ queries, but the difference is more important for Ver queries. The comparison among `vpt` and `tpv` shows that they perform similarly for subject lookups due to the locality of accesses to the bitsequences in both configurations. The difference increases for object lookups due to candidate results are scattered in the suffix array, meaning more cache misses in bitsequence accesses. This fact discovers what bitsequence configuration is better for each class of query. On the one hand, `tpv` is the fastest choice for Mat and $Diff$ queries because it only checks one or two versions respectively. On the other hand, `vpt` leads the comparison for Ver queries, in which all versions are checked for all triples retrieved from RDFCSA.

For subject queries, `tpv` averages 14-91μs and 17-118μs, for Mat and $Diff$ individual operations, and `tpv-RRR-OPT` 22-95μs and 32-128μs, respectively. In Ver queries, `vpt` averages 120-660μs and `tpv-RRR-OPT` reports 539-1055μs. Regarding object queries, higher times are reported because of the greater complexity of resolving such queries in RDFCSA. In this case, plain `tpv` averages 99-250μs and 115-307μs, for Mat and $Diff$ operations, and `tpv-RRR` 189-338μs and 281-470μs, respectively. For Ver queries, `vpt` needs 1.1-5.6ms per query whereas `vpt-RRR` reports 5.2-9.8ms.

Comparison with the baseline. We compare `v-RDFCSA` variants (with $t_\psi = 64$, which provides a balanced space/time tradeoff) with the Jena baseline deployed in BEAR. For a fair comparison, we integrate a standard *Front-Coding* dictionary [14] to

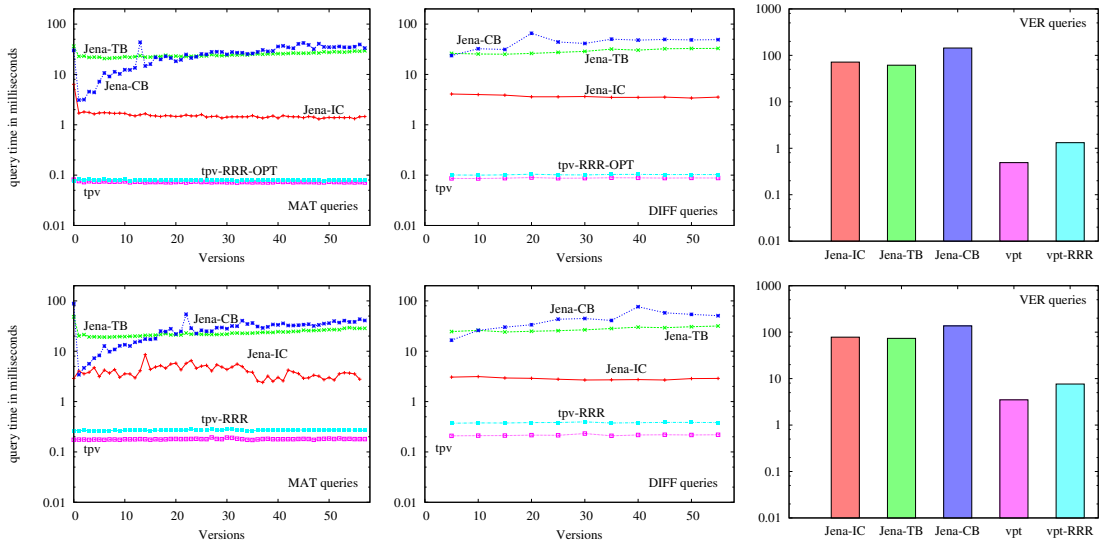

Figure 5: Subject (top) and object (bottom) lookups: Mat, $Diff$, and Ver queries.

transform ID result sets into RDF terms. Thus, we deliver the same result sets as the Jena baseline. This dictionary adds 2.3GB on top of `v-RDFCSA` space requirements.

`v-RDFCSA` shows important space improvements against the baseline. Our variants use $\approx$ 5.7-7.3GB, whereas Jena-IC, Jena-CB, and Jena-TB need 225GB, 196GB, and 353GB, respectively. These important space savings are complemented with a highly efficient query performance. Figure 5 compares query performance for subject and object lookups with high cardinality. The Mat plot (left) shows average query times (Y axis) for each version in the archive (X axis), whereas, the $Diff$ plot (middle) provides average query times for diffs between the initial version and increasing intervals of 5 versions (as defined in BEAR). For clarity on `v-RDFCSA`, we only include `tpv` and `tpv-RRR(-OPT)` lines in these plots because all our variants overlap on the same order of magnitude. Finally, the right plot shows the average query times of Ver queries, considering our `vpt` variants on `v-RDFCSA`.

Jena-IC is clearly the best choice from the baseline, although its performance is similar to Jena-TB for Ver queries. However, our variants get more than one order of magnitude ahead over the baseline approaches for Mat and $Diff$ queries (subject and object lookups). Jena-IC reports times in the orders of 1-4ms/query whereas `v-RDFCSA` reports 0.1-0.4ms/query. The comparison is similar for Ver queries, although our advantage is slightly lower in object lookups. Our best variant (`vpt`) reports 0.5ms/query and 3.5ms/query for subject and object lookups, whereas Jena-TB needs 61.5ms/query and 73.5ms/query respectively.

In conclusion, `v-RDFCSA` demonstrates that managing RDF compressed archives is not only an advantage in terms of space, but also in query performance.

5 Conclusions and Future Work

This paper presents `v-RDFCSA`, to the best of our knowledge, the first native strategy to manage and query RDF archives in compressed space. Our approach, built on

an RDFCSA self-index, proposes two bit-based strategies to compress versions in RDF archives and provides retrieval functionalities. A deep evaluation with BEAR, a state-of-the-art benchmark for RDF archives, shows that both `v-RDFCSA` variants outperform a reference baseline by reducing space requirements up to 60 times and performing more than an order of magnitude faster for query resolution.

These results open up interesting issues for future. Our on-going work focuses on exploiting our current achievement to serve advanced SPARQL queries which increase `v-RDFCSA` retrieval functionality for practical purposes.

6 References

[1] S. Álvarez-García, N. Brisaboa, J.D. Fernández, M.A. Martínez-Prieto, and G. Navarro. Compressed Vertical Partitioning for Efficient RDF Management. *Knowl. Inf. Syst.*, 44(2):439–474, 2015.

[2] N. Brisaboa, A. Cerdeira, A. Fariña, and G. Navarro. A compact RDF store using suffix arrays. In *Proc. of SPIRE*, pages 103–115, 2015.

[3] N. Brisaboa, S. Ladra, and G. Navarro. Compact Representation of Web Graphs with Extended Functionality. *Infor. Syst.*, 39(1):152–174, 2014.

[4] I. Dong-Hyuk, L. Sang-Won, and K. Hyoung-Joo. A Version Management Framework for RDF Triple Stores. *Int. J. Softw. Eng. Know.*, 22(1):85–106, 2012.

[5] J. D. Fernández, A. Polleres, and J. Umbrich. Towards Efficient Archiving of Dynamic Linked Open Data. In *Proc. of DIACHRON*, pages 34–49, 2015.

[6] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange. *J. Web Semant.*, 19:22–41, 2013.

[7] J.D. Fernández, J. Umbrich, and A. Polleres. BEAR: Benchmarking the Efficiency of RDF Archiving. Technical report, 2015. `Available at http://epub.wu.ac.at/4615/.`

[8] D. Gomes, M. Costa, D. Cruz, J. Miranda, and S. Fontes. Creating a Billion-scale Searchable Web Archive. In *Proc. of WWW Companion*, pages 1059–1066, 2013.

[9] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. of WEA*, pages 27–38, 2005.

[10] S. Harris and A. Seaborne. *SPARQL 1.1 Query Language*. W3C Recomm., 2013. `http://www.w3.org/TR/sparql11-query/.`

[11] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan. Observing Linked Data Dynamics. In *Proc. of ISWC*, pages 213–227, 2013.

[12] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. Ontology Versioning and Change Detection on the Web. In *Proc. of EKAW*, pages 197–212, 2002.

[13] F. Manola and E. Miller. *RDF Primer*. W3C Recomm., 2004. `www.w3.org/TR/rdf-primer/.`

[14] M.A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical Compressed String Dictionaries. *Infor. Syst.*, 56:73–108, 2016.

[15] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of SODA*, pages 233–242, 2002.

[16] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *J. Algorithm*, 48(2):294–313, 2003.

[17] M. Vander Sander, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. Van de Walle. R&Wbase: Git for Triples. In *Proc. of LDOW*, 2013. CEUR-WS 996, paper 1.