Reflections on: Triple Storage for Random-Access Versioned Querying of RDF Archives

Short version of article in Journal of Web Semantics

Ruben Taelman, Miel Vander Sande, Joachim Van Herwegen, Erik Mannens, Ruben Verborgh

IDLab, Department of Electronics and Information Systems, Ghent University – imec
Identifier: https://rdfostrich.github.io/article-iswc2019-journal-ostrich/

Abstract. In addition to their latest version, Linked Open Datasets on the Web can also contain useful information in or between previous versions. In order to exploit this information, we can maintain history in RDF archives. Existing approaches either require much storage space, or they do not meet sufficiently expressive querying demands. In this extended abstract, we discuss an RDF archive indexing technique that has a low storage overhead, and adds metadata for reducing lookup times. We introduce algorithms based on this technique for efficiently evaluating versioned queries. Using the BEAR RDF archiving benchmark, we evaluate our implementation, called OSTRICH. Results show that OSTRICH introduces a new trade-off regarding storage space, ingestion time, and querying efficiency. By processing and storing more metadata during ingestion time, it significantly lowers the average lookup time for versioning queries. Our storage technique reduces query evaluation time through a preprocessing step during ingestion, which only in some cases increases storage space when compared to other approaches. This allows data owners to store and query multiple versions of their dataset efficiently, lowering the barrier to historical dataset publication and analysis.

1. Introduction

In the area of data analysis, there is an ongoing need for maintaining the history of datasets. Such archives can be used for looking up data at certain points in time, for requesting evolving changes, or for checking the temporal validity of these data [1]. While the RDF data model itself is atemporal, Linked Datasets typically change over time [2] on dataset, schema, and/or instance level [3]. Such changes can include additions, modifications, or deletions of complete datasets, ontologies, and separate facts. While some evolving datasets, such as DBpedia [4], are published as separate dumps per version, more direct and efficient access to prior versions is desired.

In 2015, a survey on archiving Linked Open Data [1] illustrated the need for improved versioning capabilities, as current approaches have scalability issues at Web-scale. They either perform well for versioned query evaluation—at the cost of large storage space requirements—or require less storage space—at the cost of slower querying. Furthermore, no existing solution performs well for all existing versioned query types. An RDF archive solution should have a scalable *storage model*, efficient *compression*, and *indexing methods* that enable expressive versioned querying [1].

In this article, we argue that supporting both RDF archiving and SPARQL at once is difficult to scale due to their combined complexity. Instead, we propose an elementary but efficient versioned *triple pattern* index that can be used as a basis for query engines. We focus on the performance-critical features of *stream-based results*, query result *offsets*, and *cardinality estimation*, which allow more memory-efficient processing and more efficient query planning [5, 6]. We offer an open-source implementation of this approach, called OSTRICH, which we evaluate using an existing RDF archiving benchmark.

This article is structured as follows. In the following section, we start by introducing the related work in Section 2. Next, we introduce our storage approach in Section 3, and our querying algorithms in Section 4. After that, we present and discuss the evaluation of our implementation in Section 5. Finally, we present our conclusions in Section 6.

2. Related Work

In this section, we discuss existing solutions and techniques for indexing and compression in RDF storage, without archiving support. Then, we compare different RDF archiving solutions. Finally, we discuss suitable benchmarks and different query types for RDF archives. This section does not contain an exhaustive list of all relevant solutions and techniques, instead, only those that are most relevant to this work are mentioned.

2.1. General RDF Indexing and Compression

RDF storage systems typically use indexing and compression techniques for reducing query times and storage space.

RDF-3X [6] is an RDF storage technique that is based on a clustered B+Tree with 18 indexes in which triples are sorted lexicographically. These indexes correspond to different triple component orders. A dictionary is used to compress common triple components. When evaluating SPARQL queries, optimal indexes can be selected based on the query's triple patterns. In our storage approach, we will reuse the concept of multiple indexes and encoding triple components in a dictionary.

HDT [7] is a binary RDF representation that is highly compressed and provides indexing structures that enable efficient querying. Furthermore, it also uses a dictionary to reduce storage requirements. HDT archives are read-only, which leads to high efficiency and compressibility, but makes them unsuitable for cases where datasets change frequently. Because of these reasons, we will reuse HDT snapshots as part of our storage solution.

2.2. RDF Archiving

Fernández et al. formally define an *RDF archive* [8] as follows: *An RDF archive graph A is a set of version-annotated triples.* Where a *version-annotated triple* is defined as *an RDF triple with a label representing the version in which this triple holds.*

Furthermore, *an RDF version of an RDF archive is the set of triples that exist at a given version in the archive.*

Systems for archiving Linked Open Data are categorized into three storage strategies [1]:

- The **Independent Copies (IC)** approach creates separate instantiations of datasets for each change or set of changes. Example: SemVersion [9]
- The **Change-Based (CB)** approach instead only stores change sets between versions. Example: R&WBase [10]
- The **Timestamp-Based (TB)** approach stores the temporal validity of facts. Example: X-RDF-3X [11]

These storage strategies can also be combined into *hybrid approaches*, such as TailR [12] that combines the CB and IC approaches.

2.3. RDF Archiving Benchmarks

BEAR [8] is a benchmark for RDF archive systems. It offers three different categories of datasets with varying dataset sizes and temporal granularity. Next to that, triple pattern queries for different versioned query types are provided. BEAR provides baseline RDF archive implementations based on HDT [7] and Jena's [13] TDB store for the IC, CB, and TB approaches, but also hybrid IC/CB and TB/CB approaches. We use this benchmark for evaluation our approach.

2.4. Query atoms

To cover the retrieval demands in RDF archiving, three foundational query types were introduced [8], which are referred to as *query atoms*:

1. **Version materialization (VM)** retrieves data using a query targeted at a single version. Example: *Which books were present in the library yesterday?*
2. **Delta materialization (DM)** retrieves a query's result change sets between two versions. Example: *Which books were returned or taken from the library between yesterday and now?*
3. **Version query (VQ)** annotates a query's results with the versions (of RDF archive A) in which they are valid. Example: *At what times was book X present in the library?*

Typically, VM queries are efficient in storage solutions based on IC. DM queries are efficient in CB solutions, and VQ queries perform well in TB solutions. However, these query types typically perform sub-optimally in the other approaches. With our solution, we aim to make all query types sufficiently efficient.

3. Hybrid Multiversion Storage Approach

In order to efficiently evaluate all query types, we introduce a hybrid IC/CB/TB storage solution. Our approach consists of an *initial dataset snapshot*—stored in HDT [7]—followed by a *delta chain*. The delta chain uses multiple compressed

B+Trees for a TB-storage strategy (similar to X-RDF-3X [11]), applies dictionary-encoding to triples, and stores additional metadata to improve lookup times.

Our storage technique is partially based on a hybrid IC/CB approach similar to the approach followed by TailR [12]. This approach starts by a snapshot, and stores changes as deltas that are related to each other as can be seen in Fig. 1. In order to avoid increasing reconstruction times, we construct the delta chain in an aggregated deltas [14] fashion: each delta is *independent* of a preceding delta and relative to the closest preceding snapshot in the chain, as shown in Fig. 2. Hence, for any version, reconstruction in our approach only requires at most one delta and one snapshot, as opposed to the whole delta chain as needed for approaches like TailR.

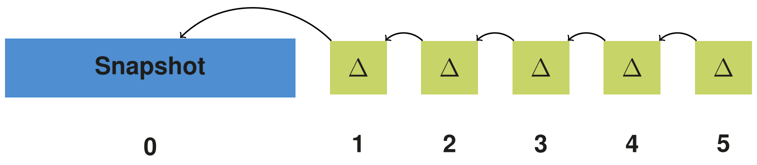

Fig. 1: Delta chain in which deltas are relative to the previous delta or snapshot, as done by TailR [12].

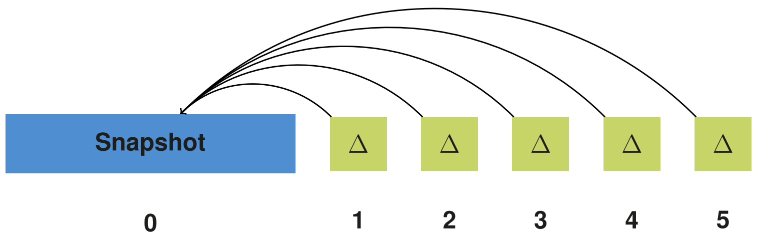

Fig. 2: Delta chain in which deltas are relative to the snapshot at the start of the chain, as part of our approach.

In order to cope with the newly introduced redundancies in our delta chain structure, we introduce a delta storage method similar to the TB storage strategy, which is able to compress redundancies within consecutive deltas. In contrast to a regular TB approach, which stores plain timestamped triples, we store timestamped triples in a separate store for additions and deletions. Each addition and deletion store uses three B+tree indexes with a different triple component order (SPO, POS and OSP) to improve lookup efficiency when querying.

For deletions specifically, we also store the *relative position* of each triple inside the delta to the deletion stores. When querying, this speeds up the process of patching a snapshot's triple pattern subset for any given offset. This position information serves two purposes: 1) it allows the querying algorithm to exploit offset capabilities of the snapshot store to resolve offsets for any triple pattern against any version; and 2) it allows deletion counts for any triple pattern and version to be determined efficiently. The use of the relative position during querying will be further explained in Section 4.

4. Versioned Query Algorithms

In this section, we introduce algorithms for performing VM, DM and VQ triple pattern queries based on the storage structure introduced in Section 3. Each of these querying algorithms are based on result streams, enabling efficient offsets and limits, by exploiting the index structure from Section 3.

4.1. Version Materialization

Version Materialization (VM) is the most straightforward versioned query type, it allows you to query against a certain dataset version. Our algorithm takes a triple pattern, a version, and a numerical offset as input, and returns a triple stream as output.

Starting from the initial snapshot, it does a sort-merge join of the deletions stream for the given version. At the end of the stream, all additions for the given version are appended. In order to apply the proper offset to the stream, we iteratively jump to the correct position in the snapshot and deletions streams based on the relative position that was stored as metadata in each triple.

4.2. Delta Materialization

The goal of delta materialization (DM) queries is to query the triple differences between two versions. Furthermore, each triple in the result stream is annotated with either being an addition or deletion between the given version range. Within the scope of this work, we limit ourselves to delta materialization within a single snapshot and delta chain. Because of this, we distinguish between two different cases for our DM algorithm in which we can query triple patterns between a start and end version, the start version of the query can either correspond to the snapshot version or it can come after that.

For the first query case, where the start version corresponds to the snapshot version, the algorithm is straightforward. Since we always store our deltas relative to the snapshot, filtering the delta of the given end version based on the given triple pattern directly corresponds to the desired result stream. Furthermore, we filter out local changes, as we are only interested in actual change with respect to the snapshot.

For the second case, the start version does not correspond to the snapshot version. The algorithm iterates over the triple pattern iteration scope of the addition and deletion trees in a sort-merge join-like operation, and only emits the triples that have a different addition/deletion flag for the two versions.

4.3. Version Query

For version querying (VQ), the final query atom, we have to retrieve all triples across all versions, annotated with the versions in which they exist. In this work, we again focus on version queries for a single snapshot and delta chain. For multiple snapshots and delta chains, the following algorithms can simply be applied once for each snapshot and delta chain.

Our version querying algorithm is again based on a sort-merge join-like operation. We start by iterating over the snapshot for the given triple pattern. Each snapshot triple is queried within the deletion tree. If such a deletion value can be found, the versions annotation contains all versions except for the versions for which the given triple was deleted with respect to the given snapshot. If no such deletion value was found, the triple was never deleted, so the versions annotation simply contains all versions of the store. Result stream offsetting can happen efficiently as long as the snapshot allows efficient offsets. When the snapshot iterator is finished, we iterate over the addition tree in a similar way. Each addition triple is again queried within the deletions tree and the versions annotation can equivalently be derived.

5. Evaluation

In this section, we evaluate our proposed storage technique and querying algorithms. We start by introducing OSTRICH, an implementation of our proposed solution. After that, we describe the setup of our experiments, followed by presenting our results. Finally, we discuss these results.

5.1. Implementation

OSTRICH stands for *Offset-enabled STore for TRIple CHangesets*, and it is a software implementation of the storage and querying techniques described in this article It is implemented in C/C++ and available on GitHub *(https://zenodo.org/record/883008)* under an open license. In the scope of this work, OSTRICH currently supports a single snapshot and delta chain. OSTRICH uses HDT [7] as snapshot technology. Furthermore, for our indexes we use Kyoto Cabinet *(http://fallabs.com/kyotocabinet/)*, which provides a highly efficient memory-mapped B+Tree implementation with compression support. For our dictionary, we use and extend HDT's dictionary implementation. We compress this delta dictionary with gzip, which requires decompression during querying and ingestion.

We provide a developer-friendly C/C++ API for ingesting and querying data based on an OSTRICH store. Additionally, we provide command-line tools for ingesting data into an OSTRICH store, or evaluating VM, DM or VQ triple pattern queries for any given limit and offset against a store. Furthermore, we implemented Node JavaScript bindings *(https://zenodo.org/record/883010)* that expose the OSTRICH API for ingesting and querying to JavaScript applications such as Comunica [15]. We used these bindings to expose an OSTRICH store *(http://versioned.linkeddatafragments.org/bear)* containing a dataset with 30M triples in 10 versions using TPF [5], with the VTPF feature [16].

5.2. Experimental Setup

As mentioned before in Section 2, we evaluate our approach using the BEAR benchmark. For a complete comparison with other approaches, we re-evaluated BEAR's Jena and HDT-based RDF archive implementations. After that, we evaluated OSTRICH for the same queries and datasets. We were not able to extend this benchmark with other similar systems such as X-RDF-3X, RDF-TX and Dydra, because the source code of systems was either not publicly available, or the system would require additional implementation work to support the required query interfaces. Our experiments were executed on a 64-bit Ubuntu 14.04 machine with 128 GB of memory and a 24-core 2.40 GHz CPU.

5.3. Results

In this section, we present the results of our evaluation. We report the ingestion results, compressibility, query evaluation times for all cases and offset result. All raw results and the scripts that were used to process them are available on GitHub *(https://github.com/rdfostrich/ostrich-bear-results/)*.

Figures 3, 4 and 5 show the query duration results for the BEAR-B queries on the complete BEAR-B-hourly dataset for all approaches. OSTRICH again outperforms Jena-based approaches in all cases. HDT-IC is faster for VM queries than OSTRICH, but HDT-CB is significantly slower, except for the first 100 versions. For DM queries, OSTRICH is comparable to HDT-IC, and faster than HDT-CB, except for the first 100 versions. Finally, OSTRICH outperforms all HDT-based approaches for VQ queries by almost an order of magnitude.

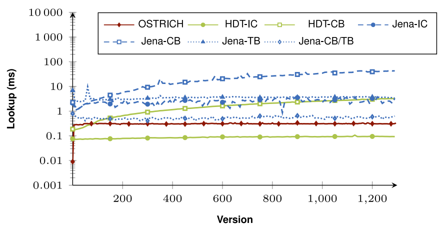

Fig. 3: Median BEAR-B-hourly VM query results for all triple patterns for all versions.

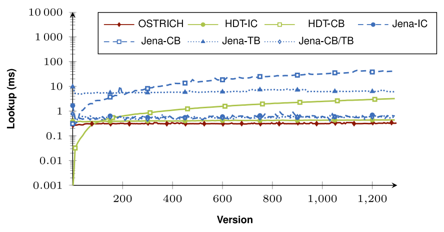

Fig. 4: Median BEAR-B-hourly DM query results for all triple patterns from version 0 to all other versions.

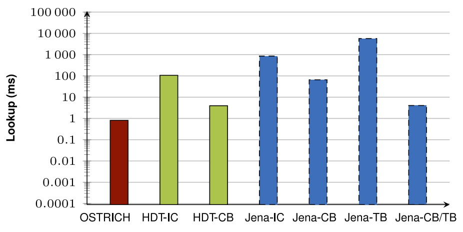

Fig. 5: Median BEAR-B-hourly VQ query results for all triple patterns.

5.4. Discussion

The results from previous section show that the OSTRICH query evaluation efficiency is faster than all Jena-based approaches, mostly faster than HDT-CB, and mostly slower than HDT-IC. VM queries in OSTRICH are always slower than HDT-IC, because HDT can very efficiently query a single materialized snapshot in this case, while OSTRICH requires more operations for materializing. VM queries in OSTRICH are however always faster than HDT-CB, because the latter has to reconstruct complete delta chains, while OSTRICH only has to reconstruct a single delta relative to the snapshot. For DM queries, OSTRICH is slower or comparable to HDT-IC, slower than HDT-CB for early versions, but faster for later versions. This slowing down of HDT-CB for DM queries is again caused by reconstruction of delta chains. For VQ queries, OSTRICH outperforms all other approaches for datasets with larger amounts of versions. For BEAR-A, which contains only 10 versions in our case, the HDT-based approaches are slightly faster because only a small amount of versions need to be iterated.

6. Conclusions

In this article, we introduced an RDF archive storage method with accompanied algorithms for evaluating versioned queries, with efficient result offsets. By storing additional data during ingestion, we achieve a significant query efficiency improvement.

With OSTRICH, we provide a technique for publishing and querying RDF archives at Web-scale. And with lookup times of 1ms or less in most cases, OSTRICH is an ideal candidate for Web querying, as the network latency will typically be higher than that. At the cost of increased ingestion times, lookups are fast. Several opportunities exist for advancing this technique in future work, such as improving the ingestion efficiency, increasing the DM offset efficiency, and supporting dynamic snapshot creation. Furthermore, branching and merging of different version can be investigated.

Our approach succeeds in reducing the cost of publishing RDF archives on the Web. It lowers the barrier towards intelligent clients that require *evolving* data, with the goal of time-sensitive querying over the ever-evolving Web of data.

References

1. Fernández, J.D., Polleres, A., Umbrich, J.: 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
2. Umbrich, J., Decker, S., Hausenblas, M., Polleres, A., Hogan, A.: Towards dataset dynamics: Change frequency of Linked Open Data sources. 3rd International Workshop on Linked Data on the Web (LDOW). (2010).
3. Meimaris, M., Papastefanatos, G., Viglas, S., Stavrakas, Y., Pateritsas, C., Anagnostopoulos, I.: A Query Language for Multi-version Data Web Archives. Expert Systems. 33, 383–404 (2016).
4. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A nucleus for a Web of open data. In: The semantic web
5. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. Journal of Web Semantics. (2016).
6. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment. 1, 647–659 (2008).
7. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF Representation for Publication and Exchange (HDT). Web Semantics: Science, Services and Agents on the World Wide Web. 19, 22–41 (2013).
8. Fernández, J.D., Umbrich, J., Polleres, A., Knuth, M.: Evaluating Query and Storage Strategies for RDF Archives. Semantic Web Journal. (2018).
9. Volkel, M., Winkler, W., Sure, Y., Kruk, S.R., Synak, M.: Semversion: A versioning system for RDF and ontologies. In: Second European Semantic Web Conference May 29–June 1, 2005 (2005).
10. Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., Van de Walle, R.: R&Wbase: git for triples. In: Proceedings of the 6th Workshop on Linked Data on the Web (2013).
11. Neumann, T., Weikum, G.: x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. Proceedings of the VLDB Endowment.
12. Meinhardt, P., Knuth, M., Sack, H.: TailR: a platform for preserving history on the web of data. In: Proceedings of the 11th International Conference on Semantic Systems. pp. 57–64. ACM (2015).
13. McBride, B.: Jena: A semantic web toolkit. IEEE Internet computing. 6, (2002).
14. Im, D.-H., Lee, S.-W., Kim, H.-J.: A version management framework for RDF triple stores. International Journal of Software Engineering and Knowledge Engineering. 22, 85–106 (2012).
15. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a Modular SPARQL Query Engine for the Web. In: Proceedings of the 17th International Semantic Web Conference (2018).
16. Taelman, R., Vander Sande, M., Verborgh, R., Mannens, E.: Versioned Triple Pattern Fragments: A Low-cost Linked Data Interface Feature for Web Archives. In: Proceedings of the 3rd Workshop on Managing the Evolution and Preservation of the Data Web (2017).