Optimizing Storage of RDF Archives using Bidirectional Delta Chains

Ruben Taelman, Thibault Mahieu and Ruben Verborgh
IDLab, Department of Electronics and Information Systems, Ghent University – imec
E-mail: ruben.taelman@ugent.be

Abstract.
Linked Open Datasets on the Web that are published as RDF can evolve over time. There is a need to be able to store such evolving RDF datasets, and query across their versions. Different storage strategies are available for managing such versioned datasets, each being efficient for specific types of versioned queries. In recent work, a hybrid storage strategy has been introduced that combines these different strategies to lead to more efficient query execution for all versioned query types at the cost increased ingestion time. While this trade-off is beneficial in the context of Web querying, it suffers from exponential ingestion times in terms of the number of versions, which becomes problematic for RDF datasets with many versions. As such, there is a need for an improved storage strategy that scales better in terms of ingestion time for many versions. We have designed, implemented, and evaluated a change to the hybrid storage strategy where we make use of a *bidirectional delta chain* instead of the default *unidirectional delta chain*. In this article, we introduce a concrete architecture for this change, together with accompanying ingestion and querying algorithms. Experimental results from our implementation show that the ingestion time scales significantly better. As an additional benefit, this change also leads to lower total storage size and improved query execution performance for most cases. This work shows that modifying the structure of delta chains within the hybrid storage strategy can be highly beneficial for RDF archives. In future work, other modifications to this delta chain structure deserve to be investigated, as they may be able to provide additional benefits.

Keywords: Linked Data, RDF archiving, Semantic Data Versioning, storage, indexing

1. Introduction

Even though the RDF [1] data model itself is atemporal, RDF datasets typically change over time [2]. Such changes can include additions, modifications, or deletions of individual facts, ontologies, or even complete datasets. While some evolving datasets such as DBpedia [3] are published as separate dumps per version, more direct and efficient access to prior versions can be desired, so that versioned queries in, between, and across different versions can be done efficiently.

While RDF archiving systems have emerged in the past that can handle such evolving datasets, a survey on archiving Linked Open Data [4] illustrated the need for improved versioning capabilities in order to preserve RDF on the Web and expose queryable access. Concretely, there is a need for systems that can store and query such datasets with low cost and effort on Web servers, so that they can cope with the large scale of RDF datasets on the Web, and their velocity. In previous work, we introduced a new hybrid archiving approach, implemented as a system called OSTRICH [5]. The approach consists of efficient triple pattern queries for different versioned query types, while still keeping storage requirements reasonable. OSTRICH was designed to run on average machines, so it can be used as a back-end for low-cost Web query interfaces such as Triple Pattern Fragments [6]. Since it exposes a triple pattern query interface, it can also be used as an index inside SPAR-

QL query engines [7]. As such, this hybrid storage approach is a step towards solving the need for properly preserving RDF on the Web.

A recent survey [8] has shown that existing RDF archiving solutions fail to handle large RDF archives with many versions. It was shown that the hybrid approach employed by OSTRICH is the only one capable of storing large RDF archives, but that it suffers from a scalability issue in terms of ingestion time for many versions. This is an inherent consequence of the storage strategy of OSTRICH, which is employed to achieve performant query execution. Concretely, after ingesting many versions, the ingestion process starts slowing down significantly, which makes OSTRICH unusable for datasets with a large number of versions, which is crucial for preserving RDF datasets on the Web. The reason for this is that the hybrid storage approach from OSTRICH only consists of a single version *snapshot* at the start, followed by an *aggregated delta chain* that keeps growing longer for every new version. Since every additional delta requires all preceding deltas to be checked during ingestion, this process becomes slower for every new version. In order to solve this problem, we propose a storage strategy modification, where there still is a single version snapshot, but we place it in the middle of the delta chain, instead of at the beginning, leading to a *bidirectional delta chain*. While this complicates ingestion and querying, it leads to two shorter delta chains. This will require less effort than one long delta chain, and lead to faster ingestion and querying.

In the next section, we discuss the related work, and give more details on OSTRICH. Next, in Section 3, we present our problem statement, followed by our proposed solution in Section 4. After that, we present our experimental setup and results in Section 5, and we conclude in Section 6.

2. Related Work

In this section, we discuss the fundamentals on RDF archiving, which RDF archiving solutions already exist, and benchmarks for RDF archiving. Finally, we discuss OSTRICH in more detail, since we build upon this approach in this work.

2.1. RDF Archiving

Various techniques have been introduced to store RDF datasets [9, 10]. These techniques make use of various indexing and compression techniques to optimize query execution and storage size. There is a need to maintain the history of these datasets [2, 4], which gave rise to the research domain of *RDF archiving*. An *RDF archive* [11] has been defined as *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 composed of all triples with a given version label.

RDF archives allow multiple versions to exist in parallel, which leads to a range of new querying possibilities. Instead of only querying within the latest version of a dataset, also previous versions can be queried, or even differences between different versions. To cover this new range of querying possibilities, five foundational query types were introduced [11], which are referred to as *query atoms*. For brevity, we refer to the article in which they were introduced [11] their formal details. In this scope of this article, we only discuss three of the five query atoms, as they can be expressed in terms of each other [12]. The three relevant query atoms are defined as follows:

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 query 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 query results with the versions in which they are valid. Example: *At what times was book X present in the library?*

2.2. RDF Archiving Solutions

In the recent years, several techniques and solutions have been proposed to allow storing and querying RDF archives. RDF archiving systems are typically categorized into three non-orthogonal storage strategies [4]:

- The **Independent Copies (IC)** approach creates separate instantiations of datasets for each change or set of changes.
- The **Change-Based (CB)** approach instead only stores change sets between versions.

- The **Timestamp-Based (TB)** approach stores the temporal validity of facts.

There exists a correspondence between these query atoms and the independent copies (IC), change-based (CB), and timestamp-based (TB) storage strategies. Namely, IC typically leads to efficient VM queries, CB is better for DM queries, and TB is best for VQ queries. No single strategy leads to good performance of all query atoms.

Table 1 shows an overview of the primary RDF archiving systems, and which storage strategy they follow. These are explained in more detail hereafter.

Name	IC	CB	TB
SemVersion [13]	✓		
Cassidy et al. [14]		✓	
R&WBase [15]		✓	
R43ples [16]		✓	
Hauptman et al. [17]			✓
X-RDF-3X [18]			✓
RDF-TX [19]			✓
v-RDFCSA [20]			✓
Dydra [21]			✓
Quit Store [22]	✓		
TailR [23]	✓	✓	
OSTRICH [5]	✓	✓	✓

Table 1: Overview of RDF archiving solutions with their corresponding storage strategy: Individual copies (IC), Change-based (CB), or Timestamp-based (TB).

2.2.1. Independent Copies Approaches

SemVersion [13] tracks different versions of RDF graphs, using Concurrent Versions System (CVS) concepts to maintain different versions of ontologies, such as diff, branching and merging. Quit Store [22] is a system that is built on top of Git, which allows these same features by considering each version to be a commit.

2.2.2. Change-Based Approaches

Cassidy et al. [14] propose a system to store changes to graphs as a series of patches, which makes it a CB approach. They describe operations on versioned graphs such as reverse, revert and merge. A preliminary evaluation shows that their implementation is significantly slower than a native RDF store. Im et al. [24] propose a CB patching system based on a relational database. In their approach, they use a storage scheme called *aggregated deltas* which associates the latest version with each of the previous ones. While aggregated deltas result in fast delta queries, they introduce a higher storage overhead. R&WBase [15] is a CB versioning system that uses the graph component of quad-stores to build a versioning layer. It supports tagging, branching and merging. R43ples [16] follows a similar approach to R&WBase, but they additionally introduce new SPARQL keywords, such as `REVISION`, `BRANCH` and `TAG`.

2.2.3. Timestamp-Based Approaches

Hauptman et al. [17] store each triple in a different named graph as a TB storage approach. The identifying graph of each triple is used in a commit graph for SPARQL query evaluation at a certain version. X-RDF-3X [18] is a versioning extension of RDF-3X [10], where each triple is annotated with a creation and deletion timestamp. RDF-TX [19] is an in-memory query engine that supports a temporal SPARQL querying extension, which makes use of a compressed multi-version B+Trees that outperforms similar systems such as X-RDF-3X in terms of querying efficiency, while having similar storage requirements. v-RDFCSA [20] is a self-indexing RDF archive mechanism, that enables versioning queries on top of compressed RDF archives as a TB approach. Dydra [21] is an RDF graph storage platform with dataset versioning support. They introduce the `REVISION` keyword, which is similar to the SPARQL keyword `GRAPH` for referring to different dataset versions.

2.2.4. Hybrid Approaches

TailR [23] is an HTTP archive for Linked Data pages for retrieving prior versions of certain HTTP resources. It is a hybrid CB/IC approach as it starts by storing a dataset snapshot, after which only deltas are stored for each consecutive version, as shown in . When the chain becomes too long, or other conditions are fulfilled, a new snapshot is created for the next version to avoid long version reconstruction times. OSTRICH [5] is a hybrid IC/CB/TB approach that ex-

ploits the advantages of each strategy to provide a trade-off between storage requirements and querying efficiency. Experiments show that OSTRICH achieves good querying performance for all query atoms, but suffers from scalability issues in terms of ingestion time for many versions. As such, we build upon OSTRICH in this work, and attempt to solve this problem.

2.3. RDF Archiving Benchmarks

BEAR [11] is a benchmark for RDF archive systems. that is based on real-world datasets from different domains. The 58 versions of BEAR-A contain between 30M and 66M triples per version, with an average change ratio of 31%. BEAR-A provides triple pattern queries for three different query atoms for both result sets with a low and a high cardinality. The BEAR-B dataset contains the 100 most volatile resources from DBpedia Live as three different granularities (instant, hour and day), and provides a small collection of triple pattern queries corresponding to the real-world usage of DBpedia. Due to BEAR covering all query atoms we work with, and it providing baseline implementations for the different storage strategies, we make use of BEAR for our experiments.

2.4. OSTRICH

As mentioned before, OSTRICH [5] make us of a hybrid IC/CB/TB storage approach with the goal of providing a trade-off between storage size and querying efficiency. The main motivation for OSTRICH is to serve as a back-end of a low-cost Web APIs for exposing RDF archives [25], where query execution must be sufficiently fast, without requiring too much storage.

Concretely, OSTRICH always starts by storing the initial version as a fully materialized version, following the IC strategy. This initial version is stored using HDT [9], which enables high compression and efficient querying. Based on this initial version, all following versions are stored as deltas, following the CB strategy. To solve the problem of increasing query execution times for increasing numbers of versions, OSTRICH makes use of the aggregated deltas [24] approach, by making each delta relative to the initial snapshot instead of the previous version. Due to the storage redundancies that are introduced because of these aggregated deltas, OSTRICH uses a B+tree-based approach to store all aggregated deltas in a single store. This single store annotates each added and deleted triple with the delta version in which it exists, thereby following the timestamp-based strategy. To further reduce storage requirements and query execution times, all triple components inside this store are dictionary-encoded, similar to the approach followed by HDT.

On top of this storage approach, OSTRICH introduces algorithms for VM, DM and VQ triple pattern queries. Only triple pattern queries are supported instead of full SPARQL queries, since triple pattern queries are the foundational building blocks for more expressive SPARQL queries. These query algorithms produce streaming results, where the streams can start from an arbitrary offset, which is valuable for SPARQL query features such as `OFFSET` and `LIMIT`. Additionally, OSTRICH provides algorithms for cardinality estimation for these queries, which are valuable for query planning within query engines. OSTRICH has been implemented in C/C++, with bindings existing for Node.JS (JavaScript). The triple pattern index provided by OSTRICH has been demonstrated to be usable within a full SPARQL query engine such as Comunica [26, 27].

Experimental results on OSTRICH with the BEAR benchmark show that this hybrid strategy is more beneficial than having just a single storage strategy, as it allows efficient execution of all query atoms. The main downside of this approach is that it leads to scalability issues in terms of ingestion time for many versions. Concretely, the BEAR-B-hourly dataset—which contains 1,299 versions— starts showing high ingestion times starting around version 1,100. The reason for this is that the aggregated deltas start becoming too large. As such, we aim to resolve this problem in this work by improving the hybrid storage strategy from OSTRICH through fundamental changes to the delta chain structure.

3. Problem Statement

As mentioned in Section 1, RDF archiving solutions suffer are not sufficiently capable of handing large RDF archives with many versions. While the hybrid storage approach as proposed by OSTRICH can handle the largest archives among all currently existing approaches, it does not scale sufficiently to a large number of versions due to its long delta chains. Our

goal in this work is to investigate if we can build on top of this hybrid storage approach and modify its delta chain structure to be able to handle RDF archives with more versions.

We formulate our research question as follows: "How can we improve the storage of RDF archives under the hybrid storage strategy by modification of the delta chain structure?"

Concretely, we start from the hybrid storage approach from OSTRICH, and we modify its current (forward) *unidirectional delta chain* (UDC) into a *bidirectional delta chain* (BDC). This bidirectional delta chain consists of two smaller delta chains, with respectively reverse and forward deltas, all pointing to one common intermediary snapshot. Since these modifications will reduce the maximum length of a delta chain, without requiring more snapshots, we expect that this will reduce ingestion time, overall storage size, and query execution time for all query atoms. Under the assumption of typical RDF archives provided by standard RDF archiving benchmarks, we define the following hypotheses:

1. Storage size is lower for a BDC compared to a UDC.
2. In-order ingestion time is lower for a BDC compared to a UDC.
3. VM query execution is faster for a BDC compared to a UDC.
4. DM query execution is faster for a BDC to a UDC.
5. VQ query execution is faster for a BDC to a UDC.

4. Bidirectional Delta Chain

In this section, we explain our bidirectional delta chain approach. We start by explaining the general idea behind a bidirectional delta chain. After that, we explain its implication on storage. Finally, we discuss querying algorithms for the foundational query atoms based on this storage approach.

4.1. Delta Chain Approaches

In the scope of this work, we distinguish between 6 different delta chain approaches, as can be seen in Table 2. We decompose these approaches into 2 axes: *directionality* and *aggregation*.

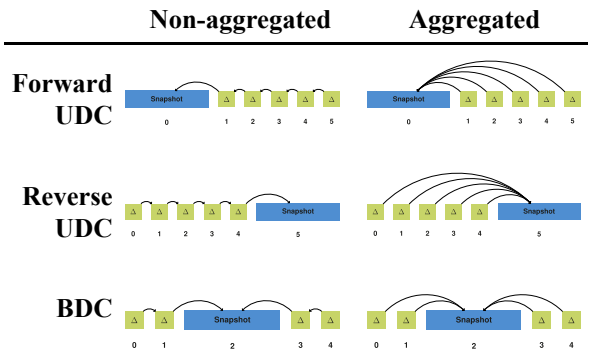


Table 2: Overview of unidirectional forward, unidirectional reverse, and bidirectional delta chain approaches, both with and without aggregated deltas.

Along the *directionality* axis, we distinguish 3 forms:

1. The simplest form is the **forward unidirectional** delta chain, where the snapshot comes first, and is followed by deltas that are relative to the previous delta.
2. The **reverse unidirectional** delta chain is a variant of this where everything is reversed. Concretely, the snapshot comes last, and is preceded by deltas, where each delta is relative to the next delta.
3. These forward and reverse unidirectional approaches can be combined with each other to form a **bidirectional delta chain**, where a first set of deltas exist before the snapshot, and a second set of deltas exists after the snapshot.

Along the *aggregation* axis, we consider 2 forms:

- In the **non-aggregated form**, each delta is relative to the delta immediately before or after it.
- In the **aggregated form** [24], each delta is relative to the snapshot before or after it, where other deltas may occur in-between.

The aggregated delta approach leads to lower version materialization times, since each delta can be directly applied to a snapshot, as opposed to non-aggregated deltas where multiple deltas need to be combined before a version can be materialized. As such, the version materialization time for aggregated deltas is `O(1)` with respect to the number of versions, while it is `O(n)` for non-aggregated deltas with respect to the number of versions. This shows how aggregated deltas lead to better query execution times. The major down-

side of aggregated deltas is however that storage size increases due to the redundancies between the different deltas. The longer the delta chain, the larger these redundancies become.

OSTRICH [5] is an example that follows the unidirectional forward aggregated delta chain approach, while RCS [28] (non-RDF-based) follows the unidirectional reverse non-aggregated delta chain approach. In this work, we will investigate the use of the *bidirectional aggregated* delta chain approach, for reasons explained in the next section.

4.2. Motivations For A Bidirectional Delta Chain

Experiments on the unidirectional forward aggregated delta chain approach from OSTRICH [5] have shown that this approach leads to ingestion times that increase linearly with chain length, assuming (non-aggregated) deltas as inputs. This is an expected consequence of the aggregated delta approach, as they grow in size for each new version. The goal of this work is to investigate how these problems can be solved, without losing the advantages of aggregated deltas with respect to query execution times. We would not achieve any lower ingestion times by reversing our delta chain, as the additions and deletions would just be swapped, but would not be smaller. Instead, we aim to reduce ingestion time by lowering storage through the reduction of the number of required snapshots.

One straightforward way of reducing ingestion time would be to create a new snapshot and delta chain once the ingestion time or size has crossed a certain threshold. One example of such a threshold could be that a new snapshot is created once the ingestion time of a delta became larger than the time for ingesting a snapshot. For instance, we can lower the total ingestion time to half the original time by splitting one delta chain into two delta chains, or even to one third by splitting it up into three delta chains. In the extreme, each version would be form its own snapshot, which would lead to the independent copies storage strategy, at the cost of increased storage size. As such, there is a trade-off between ingestion time and storage size, and new delta chains should only be started once ingestion times become higher than desired.

Since the creation of a snapshot can be costly in terms of storage size, it should be avoided until absolutely necessary. As explained in the previous paragraph, splitting up a delta chain into two separate delta chains would lead to two snapshots, each followed by a chain of deltas. We can however reduce the number of required snapshots by combining the forward and reverse approaches into a *bidirectional* approach, by allowing two sets of deltas to make use of the same snapshot. Intuitively, one bidirectional delta chain is equivalent to two forward delta chains, where the second delta chain is reversed. The snapshots of these two chains are therefore shared, so that it only has to be created and stored once.

As such, the main advantage of a bidirectional delta chain is that it can more efficiently make use of snapshots. Instead of only allowing deltas in one direction to make use of it, also deltas in the opposite direction can make use of it. This is especially advantageous for aggregated deltas, as these grow in size for longer chains. In the scope of this research, we continue working with a bidirectional *aggregated* delta chain due to the non-increasing query execution times for increasing numbers of versions.

One disadvantage of the bidirectional approach is that it complicates ingestion, since we can not build a reverse delta chain directly, as we can not always know beforehand what a future version will look like. We tackle this problem in Subsection 4.4.

4.3. Storage Approach

As mentioned before, our goal is to improve storage efficiency of RDF archives. For this, we build on top of the hybrid storage approach from OSTRICH, and we fundamentally modify this storage approach to use a bidirectional aggregated delta chain instead of a unidirectional aggregated delta chain. Concretely, this means that not only deltas exist *after* the snapshot, but also *before* the snapshot.

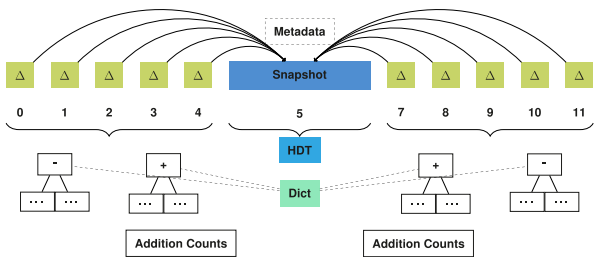


Fig. 1: Overview of the main components of our storage approach consisting of a bidirectional aggregated delta chain.

Fig. 1 shows an overview of the main components of our storage approach. Note in particular the delta chain on the left side of the snapshot, while OSTRICH only has a single delta chain on the right side of the snapshot. All other components are inherited from OSTRICH, which we briefly summarize in the next paragraph.

We store the snapshot using HDT [9], due to its highly performant triple pattern queries, cardinality estimates, and high compression rate. Furthermore, metadata about the archive is stored, containing details such as the total number of versions. To avoid storage overhead due to redundancies between different aggregated deltas, each delta chain is compressed into timestamp-based B+tree indexes where additions and deletions are stored separately. This separation is done to speed up query evaluation since additions and deletions are not always needed at the same time. To enable efficient triple pattern queries for all possible combinations, each addition and deletion index is stored three times for different triple components orders (SPO, POS, OSP). To compress each triple component further, a shared dictionary is used. In order to allow efficient cardinality estimate retrieval for deletions, the SPO deletion index contains additional metadata about the relative position of each triple inside the snapshot. To enable cardinality estimates for additions, we make use of a dedicated addition count index. For the sake of brevity, we omit further details about the components that can be found in the OSTRICH article [5].

4.4. Ingestion Approach

In this section, we introduce an approach to enable ingestion of new versions within our *bidirectional* aggregated storage approach. For this, we build upon the streaming ingestion algorithm and DM query algorithm for *unidirectional* forward aggregated delta chains [5], which allows us to insert deltas *after* the snapshot.

In order to insert deltas *before* the snapshot, our approach for constructing the reverse delta chain involves a temporary forward delta chain. This is because we can not start building our reverse delta chain directly, as we can not predict what triples will be in the snapshot later down the line. For each new version, our temporary forward delta chain will be built up, and can be queried in the meantime. From the moment that this delta chain becomes too long, or some other threshold has been exceeded, then an offline fix-up algorithm is triggered that will effectively *reverse* this delta chain, and place a snapshot at the end, where a new forward delta chain can be built upon when new versions arrive.

Algorithm 1 shows a sketch of our fix-up algorithm in pseudo-code. First, the aggregated deltas in the chain will be extracted as non-aggregated deltas by invoking a DM query over the current unidirectional aggregated delta chain. We store the deletions as additions, and the additions as deletions. Next, we create a new delta chain, and insert these reversed deltas by invoking the streaming ingestion algorithm for unidirectional aggregated delta chains. Once ingestion is done, the existing delta chain is replaced by our new delta chain.

```
FUNCTION fixUp(store)
# Recreate deltas, but in reverse
additions = []
deletions = []
FOR v IN store.getVersions()
  deletions.push(store.getAdditions(v, v + 1))
  additions.push(store.getDeletions(v, v + 1))
# Ingest reversed deltas into a new delta chain
newChain = store.newDeltaChain({ reversed: true })
FOR v IN store.getVersions()
  newChain.ingest(v, additions[v], deletions[v])
# Replace delta chain
existingChain = store.getDeltaChain(store.getVersions())
store.replaceChain(existingChain, newChain)
```

Algorithm 1: Fix-up algorithm for reversing an existing bidirectional aggregated delta chain.

The main advantage of this fix-up approach is that it avoids query unavailability of the archive. The fix-up algorithm can run at any time, preferably when the server is experiencing a lower query load. During the execution of this process, the temporary forward delta chain is still available, so queries are still possible during this time. Only after the fix-up process is done, query executions will be delegated to this new reverse delta chain, and the temporary forward delta chain can be deleted.

4.5. Query Algorithms

In this section, we discuss triple pattern query algorithms for the three query atoms discussed in Section 2 (VM, DM, VQ). For simplicity, we assume the existence of a (bidirectional) delta chain with one snapshot. We consider multiple snapshots and delta chains future work. We build upon the existing algorithms for unidirectional (aggregated) delta chains [5], and thereby inherit their properties of streaming, offset support, and cardinality estimators. Below, we briefly discuss

the relevant parts of these existing algorithms. For more details, we refer to the OSTRICH article [5].

4.5.1. Version Materialization

Version Materialization (VM) allows retrieval of triples in a given version. In summary, VM over a unidirectional delta chain works by either querying a snapshot directly, if the requested version is a snapshot, or applying a given delta on the closest preceding snapshot otherwise. In our bidirectional delta chain, a snapshot can not only exist before a delta, but also after a delta. Nevertheless, the algorithm itself remains the same as for a unidirectional delta chain, as the delta will have to be applied onto the snapshot in both cases. As such, we do not discuss this VM case any further.

4.5.2. Delta Materialization

Delta Materialization (DM) allows differences between two given versions to be retrieved. The DM algorithm over a unidirectional delta chain distinguishes two cases for this; either the start version is a snapshot or a delta, where the end version will always be a delta. If the start (or end) version is a snapshot, then the result is simply a query within the aggregated delta of the end version. Otherwise, the addition and deletion indexes for the two delta versions are iterated in a sort-merge join-like operation, and only emits the triples that have a different addition/deletion flag for the two versions.

In our bidirectional storage approach, one additional case can occur: when the start and end version correspond to deltas in the bidirectional delta chain *before* and *after* the snapshot, i.e., the DM query crosses the snapshot boundary. For this, we split up our query into two queries: a DM query from the start version until the snapshot, and a DM query from the snapshot until the end version. These two queries can be resolved over the two respective delta chains using the DM algorithm over a unidirectional delta chain. As the results from these two queries are sorted, we can merge them in a sort-merge join way, where triples are only emitted if they don't exist in both streams (ignoring the addition/deletion flag). Algorithm 2 illustrates this algorithm in pseudocode. Following the patch notation for DARCS [29], with `o` being the start version, `e` being the end version and `s` being the snapshot, our delta split corresponds to ${}^{o}D^{e} = {}^{o}D1^{s}D2^{e}$.

```
FUNCTION queryDmCase3(store, start, end)
snapshotVersion = store.getSnapshotBetween(start, end)
reverseStream = store.getDeltaStream(start, snapshotVersion)
forwardStream = store.getDeltaStream(snapshotVersion, end)
return sortMerge(reverseStream, forwardStream)
```

Algorithm 2: Delta Materialization algorithm for triple patterns that produces a triple stream when the version range crosses the snapshot boundary.

In order to estimate the cardinality of this third case, the same idea is followed where the counts of the part of the delta chain before and after the snapshot are added. Just like the existing DM cardinality estimator over a unidirectional delta chain, this can be an overestimation, since certain triples may occur in both delta chains that would be omitted from the final result stream.

4.5.3. Version Query

A Version Query (VQ) enables querying across all versions, with results being annotated with the version in which they occur. VQ over a unidirectional delta chain is done by iterating over the snapshot for a given triple pattern in a streaming manner. Every snapshot triple is queried within the deletion index. For every discovered deletion, their respective version annotations are removed from the result. If no such deletion value was found, the triple was never deleted, so the versions annotation will contain all versions of the store. Once the snapshot stream has finished, the addition index are iterated in a similar way, where the version annotation of every addition triple is again updated based on its presence in the deletion index.

Our case is a trivial extension of this algorithm. Instead of checking single addition and deletion streams, two addition and deletion streams have to be checked. This will produce distinct version annotations, for which we apply the union.

To estimate the cardinality, the unidirectional delta chain approach can again be extended by adding the snapshot cardinality with the addition cardinality for both delta chains for the given triple pattern. As some triples could occur in both delta chains, this can lead to an overestimation.

5. Evaluation

In this section, we evaluate our bidirectional archiving approach by comparing our implementation to

native OSTRICH.

5.1. Implementation

We have implemented our storage approach and query algorithms as a tool called COBRA (Change-Based Offset-Enabled Bidirectional RDF Archive). COBRA is an extension of OSTRICH, has been implemented in C/C++, and is available under the MIT license on GitHub *(https://github.com/rdfostrich/cobra)*. Our implementation uses HDT [9] as snapshot technology, and makes use of the highly efficient memory-mapped B+Tree implementation Kyoto Cabinet *(http://fallabs.com/kyotocabinet/)* for storing our indexes. The delta dictionary is encoded with gzip, which requires decompression during querying and ingestion.

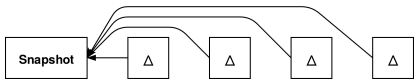
5.2. Experimental Setup

In order to evaluate the ingestion and triple pattern query execution of COBRA, we make use of the BEAR benchmark *(https://aic.ai.wu.ac.at/qadlod/bear.html)*. To test the scalability of our approach for datasets with few and large versions, we use the BEAR-A benchmark. We use the first eight versions of the BEAR-A dataset (more versions cause memory issues), which contains 30M to 66M triples per version. This dataset was compiled from the Dynamic Linked Data Observatory. To test for datasets with many smaller versions, we use BEAR-B with the daily and hourly granularities. For the daily dataset we use 89 versions and for hourly dataset 400 versions, both of them have around 48K triples per version. All experiments were performed on a 64-bit Ubuntu 14.04 machine with a 6-core 2.40 GHz CPU and 48 GB of RAM. Our experimental setup and its raw results are available on GitHub *(https://github.com/rdfostrich/cobra/tree/master/Experiments/)*.

Considering we aim to measure the benefits of the bidirectional aggregated delta chain compared to the unidirectional aggregated delta chain under the hybrid storage strategy, we distinguish between the following storage approaches:

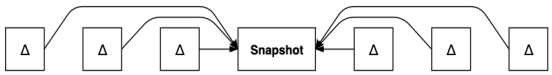
- **OSTRICH**: Forward unidirectional aggregated delta chain (Subfig. 2.1)
- **COBRA***: Bidirectional aggregated delta chain before fix-up (Subfig. 2.2)
- **COBRA**: Bidirectional aggregated delta chain after fix-up (Subfig. 2.3)

As such, we consider comparing against other systems with different storage strategies out of scope for this work. For an extensive comparison of the hybrid storage strategy with other systems, we refer to the OSTRICH article [5].



Subfig. 2.1: OSTRICH with a forward unidirectional aggregated delta chain

Subfig. 2.2: COBRA* with a bidirectional aggregated delta chain *before* fix-up



Subfig. 2.3: COBRA with a bidirectional aggregated delta chain *after* fix-up (ingested out-of-order starting with snapshot)

Fig. 2: The different storage approaches used in our experiments.

In the scope of this work, we work with at most two delta chains. For simplicity of these experiments, we always start a new delta chain in the middle version of the dataset (4 for BEAR-A, 45 for BEAR-B Daily, 200 for BEAR-B Hourly). Note that for the COBRA storage approach, we assume that all versions are available beforehand, so they can be stored out of order, starting with the middle snapshot. In practise, this may not always be possible, which is why we report on the additional fix-up time during ingestion separately that would be required when ingestion in order (COBRA*).

To evaluate triple pattern query performance, we make use of the query sets provided by BEAR. BEAR-A provides 7 query sets containing around 100 triple patterns that are further divided into high result cardinality and low result cardinality. BEAR-B provides two query sets that contain `?P?` and `?PO` queries.

We evaluate these queries as VM queries for all version, DM queries between the first and all other versions and a VQ query. In order to minimize outliers, we replicate the queries five times and take the mean results. Furthermore, we perform a warm-up period before the first query of each triple pattern. Since neither OSTRICH nor COBRA support multiple snapshots for all query atoms, we limit our experiments to OSTRICH's unidirectional storage layout and COBRA's bidirectional storage layout.

5.3. Measurements

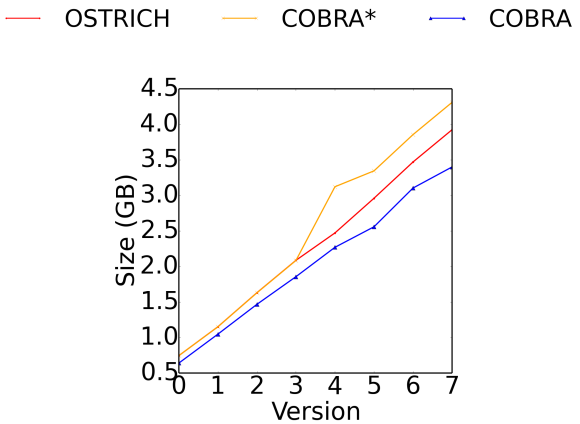
In this section, we discuss the results of our experiments on ingestion and query evaluation, which we then analyze in the next section.

5.3.1. Ingestion

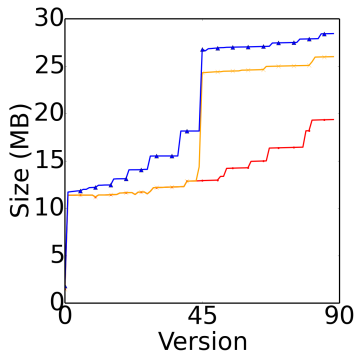
Table 3 show the total storage sizes and ingestion times for BEAR-A, BEAR-B Daily, and BEAR-B Hourly under the different storage approaches. These tables show that COBRA requires less ingestion time than OSTRICH in all cases (41% less on average). Furthermore, COBRA requires less storage space than OSTRICH for BEAR-A and BEAR-B Hourly, but not for BEAR-B Daily. COBRA* requires more storage space than both COBRA and OSTRICH with BEAR-A, but it requires less ingestion time. For BEAR-B Daily, OSTRICH requires less storage, but COBRA* has the lowest ingestion time. For BEAR-B Hourly, COBRA* is lower in terms of storage size and ingestion time than both COBRA and OSTRICH.

	OSTRICH	**COBRA***	**COBRA**
BEAR-A			
Size (GB)	3.92	4.31	*3.36*
Time (hours)	23.66	*12.92*	14.63
BEAR-B Daily			
Size (MB)	*19.37*	26.01	28.44
Time (minutes)	6.53	*3.28*	4.24
BEAR-B Hourly			
Size (MB)	61.02	*46.42*	53.26
Time (minutes)	34.47	*14.87*	18.30

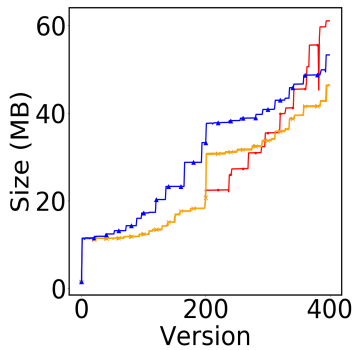
Table 3: Total storage size and ingestion time for the different datasets. COBRA* is always the fastest, with no consistent winner for total storage size.



Subfig. 3.1: BEAR-A

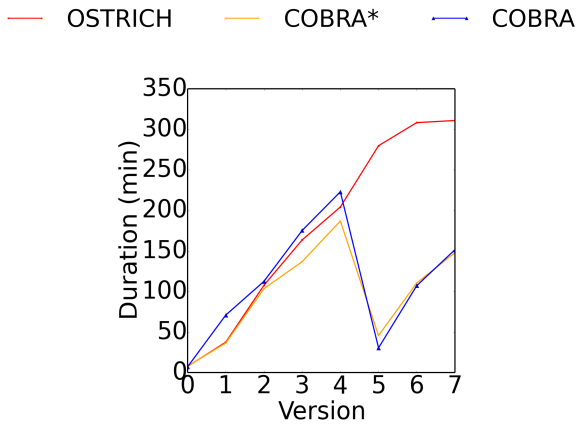


Subfig. 3.2: BEAR-B Daily

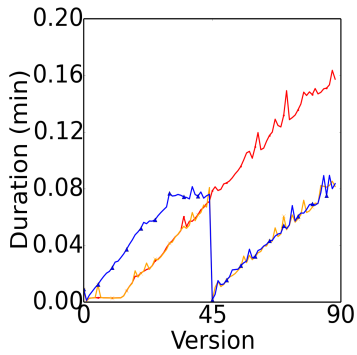


Subfig. 3.3: BEAR-B Hourly

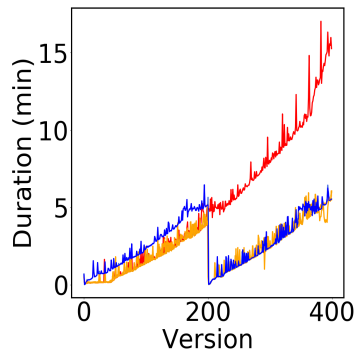
Fig. 3: Cumulative storage sizes for BEAR-A, BEAR-B Daily, and BEAR-B Hourly under the different storage approaches. COBRA requires less storage space than OSTRICH for BEAR-A. For BEAR-B Daily and Hourly, the middle snapshot leads to a significant increase in storage size.



Subfig. 4.1: BEAR-A



Subfig. 4.2: BEAR-B Daily



Subfig. 4.3: BEAR-B Hourly

Fig. 4: Ingestion times per version for BEAR-A, BEAR-B Daily, and BEAR-B Hourly under the different storage approaches. COBRA resets ingestion time from the snapshot version, while ingestion time for OSTRICH keeps increasing.

In order to provide more details on the evolution of storage size and ingestion time, Fig. 3 shows the cumulative storage size for the different datasets, and Fig. 4 shows the ingestion time for these datasets. These figures show the impact of the middle snapshots within the bidirectional chain. For BEAR-B Daily and Hourly, the storage size significantly increases at the middle version, but the ingestion times for all later versions reset to low values.

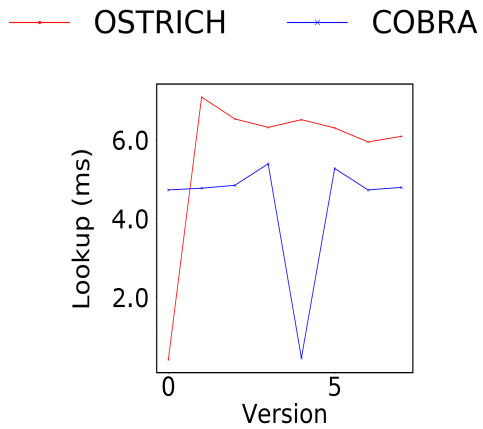
Finally, Table 4 show the fix-up times, which are measured as a separate offline process. This is the time it would take to transition from the COBRA* to COBRA storage approach, when the versions cannot be inserted out of order. On average, this fix-up requires 3,6 times more time relative to to the overhead of COBRA compared to COBRA*, showing that out-of-order ingestion is still preferred when possible.

Dataset	**Time**
BEAR-A	8.38 hours
BEAR-B Daily	2.48 minutes
BEAR-B Hourly	11.41 minutes

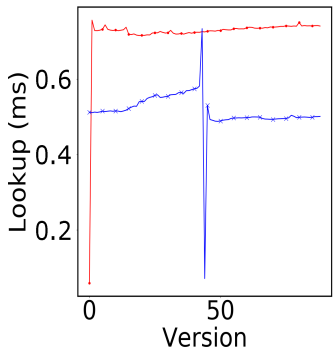
Table 4: Fix-up duration for the different datasets.

5.3.2. Query Evaluation

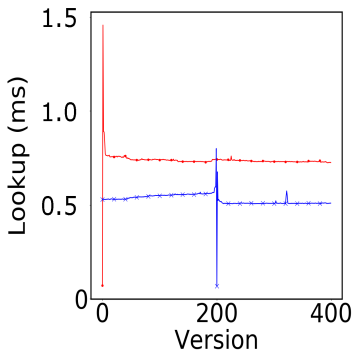
Fig. 5, Fig. 6 and Fig. 7 show the query evaluation times for COBRA (after fix-up) and OSTRICH for respectively VM, DM and VQ. These figures show that for VM, COBRA is faster than OSTRICH minus a few outliers around the middle version. For DM, COBRA is always faster than OSTRICH when querying within the first half of its delta chain. For the second half, COBRA becomes slower, and for BEAR-B Daily even becomes slower than OSTRICH. For VQ, COBRA is faster than OSTRICH for BEAR-B Hourly, slightly faster for BEAR-B Daily, and slower for BEAR-A.



Subfig. 5.1: BEAR-A

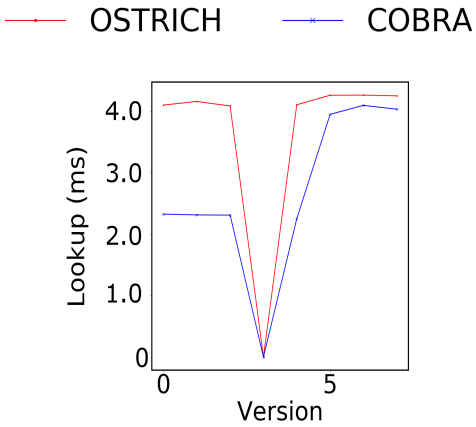


Subfig. 5.2: BEAR-B Daily

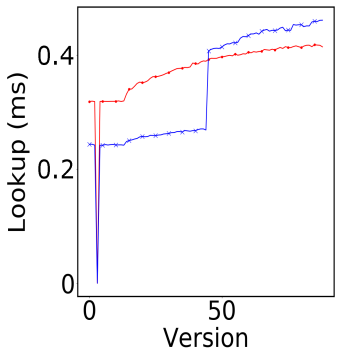


Subfig. 5.3: BEAR-B Hourly

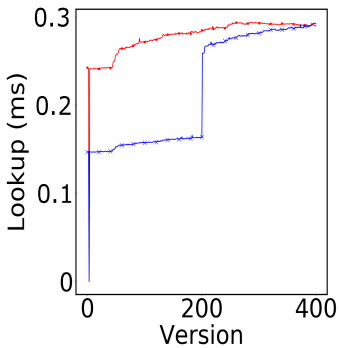
Fig. 5: Version Materialization evaluation times per version for BEAR-A, BEAR-B Daily, and BEAR-B Hourly under the different storage approaches. For most versions, COBRA has is faster than OSTRICH.



Subfig. 6.1: BEAR-A

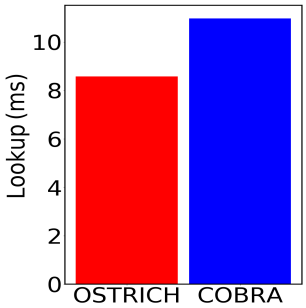


Subfig. 6.2: BEAR-B Daily

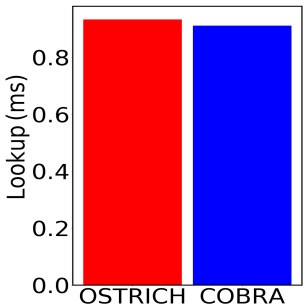


Subfig. 6.3: BEAR-B Hourly

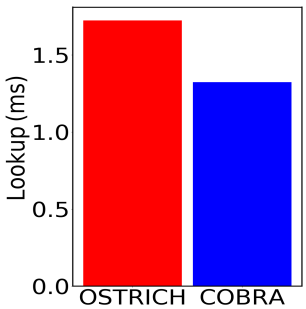
Fig. 6: Delta Materialization evaluation times between the first version and all other versions for BEAR-A, BEAR-B Daily, and BEAR-B Hourly under the different storage approaches. For the first half of versions, COBRA is faster than OSTRICH, but slows down in the second half.



Subfig. 7.1: BEAR-A



Subfig. 7.2: BEAR-B Daily



Subfig. 7.3: BEAR-B Hourly

Fig. 7: Version Query evaluation times across all versions for BEAR-A, BEAR-B Daily, and BEAR-B Hourly under the different storage approaches. COBRA is faster than OSTRICH for the BEAR-B datasets, but slower for BEAR-A.

Table 5 show the average overall query evaluation times for BEAR-A, BEAR-B Daily and BEAR-B

Hourly. This shows that on average, COBRA is faster than OSTRICH, except for VQ in BEAR-A.

	VM	DM	VQ
BEAR-A			
OSTRICH	5.64	4,15	*8,60*
COBRA	*4.37*	*2,93*	10,62
BEAR-B Daily			
OSTRICH	0,71	0,38	0.90
COBRA	*0,51*	*0,31*	*0.89*
BEAR-B Hourly			
OSTRICH	0.73	0.27	1,72
COBRA	*0.53*	*0.19*	*1,34*

Table 5: Average query evaluation times for OSTRICH and COBRA for VM, DM and VQ for the different datasets (ms).

5.4. Result Analysis

In this section, we discuss the findings of our results regarding ingestion and query evaluation, and we test our hypotheses.

5.4.1. Ingestion

Our experimental results show that the usage of a bidirectional delta chain has a significant beneficial impact on storage size and ingestion time compared to a unidirectional delta chain. While the unidirectional delta chain leads to increasing ingestion times for every new version, initiating a new snapshot (COBRA*) can effectively *reset* these ingestion times. The downside of this is that there can be an increase in storage size due to this, which is more significant for datasets that have many small versions (BEAR-B). As such, for those datasets (BEAR-B), it is recommended to wait longer before initiating a new snapshot in the delta chain, since ingestion times are typically much lower compared to datasets with fewer large versions (BEAR-A). Given the capabilities and query load of the server and affordable storage overhead, a certain ingestion time threshold could be defined, which would initiate a new snapshot when this threshold is exceeded.

Once there are two unidirectional delta chains, the first one could optionally be reversed so that both can share one snapshot through a fix-up process (COBRA). Our results show that this can further reduce storage size for datasets with few large versions (BEAR-A). However, for many small versions (BEAR-B), this leads to overhead in terms of storage size. This fix-up process does however require a significant execution time. Since this could easily run in a separate process can happen in an offline process, this additional time is typically not a problem. As such, when the server encounters a dataset with large versions (millions of triples per version), then the fix-up approach should be followed.

The results also show that if all versions are known beforehand, they should be ingested out-of-order into a bidirectional delta chain. Because this leads to a significantly lower total ingestion time compared to in-order ingestion followed by the fix-up process.

5.4.2. Query Evaluation

Regarding query performance, our results show that the bidirectional delta chain also has a large impact here. Since two shorter delta chains lead to two smaller addition and deletion indexes compared to one longer delta chain, VM and DM times become lower, since less data needs to be iterated. We see that DM times for the second half of the bidirectional delta chain become slower compared to the first half. This is because in these cases we need to query within the two parts of the delta chain, i.e., we need to search through two addition and deletion indexes instead of just one. For datasets with many small versions (BEAR-B), VQ also becomes faster with a bidirectional delta chain, but this does not apply when the dataset has few large versions (BEAR-A). This is again caused by the fact that we now have two delta chains, and two addition and deletion indexes to query in. When we have many small versions, these two delta chains are worth it, as the benefit of the shared snapshot outweighs the overhead of the delta chains. However, for few large versions, the overhead of two delta chains is too large for VQ, and one delta chain performs better. In summary, a bidirectional delta chain is most effective for optimizing VM, largely beneficial for DM, and beneficial for VQ (assuming many small versions).

5.4.3. Hypotheses

In Section 3, we defined research hypotheses, which we will now answer based on our experimental results.

In our first hypothesis, we expected storage size to become lower with a bidirectional delta chain compared to a unidirectional delta chain. While this is true for BEAR-A and BEAR-B Hourly, this is not true for BEAR-B Daily. As such, we *reject* this hypotheses. In our second hypothesis, we expected ingestion time to be lower with a bidirectional delta chain. Our results show that this is true. As such, we *accept* this hypothesis. Our other hypotheses expect that evaluation times for VM, DM and VQ with a bidirectional delta chain would be lower. Our results show that this is true, expect for VQ. As such, we *accept* our third and fourth hypothesis, and *reject* our fifth hypothesis.

6. Conclusions

In this work, we improved the storage of RDF archives under the hybrid storage strategy (OSTRICH) by making use of a *bidirectional delta chain*. Based on our implementation of this new approach (COBRA), our experimental results show that this modification solves the main scalability problem of a unidirectional delta chain (OSTRICH) regarding its ingestion times (41% faster). This change also reduces total storage size (13% lower) for two out of three datasets. Furthermore, all versioned query types achieve a performance boost (21% faster), except for VQ under the BEAR-A dataset. With query execution times in the order of 1 millisecond or less, the bidirectional delta chain strategy from COBRA is an ideal back-end for RDF archives in the context of Web querying, as network latency is typically slower than that.

As such, the bidirectional delta chain is a viable alternative to the unidirectional delta chain, as it is beneficial across nearly all metrics. We **recommend bidirectional delta chains** when any of the following is needed (in order of importance):

- **Lower ingestion times**
- **Faster VM and DM**
- **Lower storage sizes**

On the other hand, we **do not recommend bidirectional delta chains** in the following cases:

- **Fast VQ is needed over datasets with very large versions**: Bidirectional chains slow down VQ when versions are large.
- **Dataset has only a few small versions**: Unidirectional chain should be used until the ingestion of a new version exceeds the ingestion time of a new snapshot.

These limitations of a bidirectional delta chain may be resolvable in future work through more intelligent strategies on when to convert a unidirectional delta chain into a bidirectional delta chain. Next to this, the beneficial impact of the bidirectional delta chain opens up questions as to what respect other transformations of the delta chain in terms of delta directionality and snapshot placement may be beneficial to ingestion time, storage size, and query performance. First, deltas may inherit from two or more surrounding versions, instead of just one. Second, aggregated and non-aggregated deltas are just two extremes of delta organization. A range of valuable possibilities in between may exist, such as inheriting from the nth largest preceding version. Third, the impact of multiple snapshots and strategies to decide when to create them still remain as open questions, which we suspect will be crucial for RDF archiving for indefinitely increasing numbers of versions.

We have shown that modifying the structure of the delta chain can be highly beneficial for RDF archiving. This brings us closer to an efficient queryable Semantic Web that can evolve and maintain its history.

References

[1] Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1: Concepts and Abstract Syntax. W3C, http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (2014).

[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] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: Dbpedia: A nucleus for a web of open data. The semantic web. 722–735 (2007).

[4] Fernández, J.D., Polleres, A., Umbrich, J.: Towards efficient archiving of Dynamic Linked Open Data. In: Proceedings of te First DIACHRON Workshop on Managing the Evolution and Preservation of the Data Web. pp. 34–49 (2015).

[5] Taelman, R., Vander Sande, M., Van Herwegen, J., Mannens, E., Verborgh, R.: Triple storage for random-access versioned querying of RDF archives. Journal of Web Semantics. 54, 4–28 (2019).

[6] 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. 37–38, (2016).

[7] Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C, http://www.w3.org/TR/2013/REC-

sparql11-query-20130321/ (2013).
[8] Pelgrin, O., Galàrraga, L., Hose, K.: Towards Fully-fledged Archiving for RDF Datasets. Semantic Web Journal. (2020).
[9] 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).
[10] Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment. 1, 647–659 (2008).
[11] Fernández, J.D., Umbrich, J., Polleres, A., Knuth, M.: Evaluating Query and Storage Strategies for RDF Archives. Semantic Web Journal. (2018).
[12] Taelman, R., Verborgh, R., Mannens, E.: Exposing RDF Archives using Triple Pattern Fragments. In: Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management: Posters and Demos (2016).
[13] 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, ESWC 2005, Heraklion, Crete, Greece, May 29–June 1, 2005. Proceedings (2005).
[14] Cassidy, S., Ballantine, J.: Version Control for RDF Triple Stores. ICSOFT (ISDM/EHST/DC). 7, 5–12 (2007).
[15] 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).
[16] Graube, M., Hensel, S., Urbas, L.: R43ples: Revisions for triples. In: Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014) (2014).
[17] Hauptmann, C., Brocco, M., Wörndl, W.: Scalable Semantic Version Control for Linked Data Management. In: Proceedings of the 2nd Workshop on Linked Data Quality co-located with 12th Extended Semantic Web Conference (ESWC 2015) (2015).
[18] Neumann, T., Weikum, G.: x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. Proceedings of the VLDB Endowment. 3, 256–263 (2010).
[19] Gao, S., Gu, J., Zaniolo, C.: RDF-TX: A fast, user-friendly system for querying the history of RDF knowledge bases. In: Proceedings of the 19th International Conference on Extending DatabaseTechnology. pp. 269–280 (2016).
[20] Cerdeira-Pena, A., Farina, A., Fernández, J.D., Martínez-Prieto Miguel A: Self-indexing RDF archives. In: Data Compression Conference (DCC), 2016. pp. 526–535. IEEE (2016).
[21] Anderson, J., Bendiken, A.: Transaction-time queries in dydra. In: Joint Proceedings of the 2nd Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW 2016) and the 3rd Workshop on Linked Data Quality (LDQ 2016). pp. 11–19 (2016).
[22] Arndt, N., Naumann, P., Radtke, N., Martin, M., Marx, E.: Decentralized Collaborative Knowledge Management using Git. Journal of Web Semantics. (2018).
[23] 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).
[24] 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).
[25] 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).
[26] 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).
[27] Taelman, R., Vander Sande, M., Verborgh, R.: Versioned Querying with OSTRICH and Comunica in MOCHA 2018. In: Proceedings of the 5th SemWebEval Challenge at ESWC 2018 (2018).
[28] Tichy, W.F.: RCS—a system for version control. Software: Practice and Experience. 15, 637–654 (1985).
[29] Roundy, D.: Darcs. http://darcs.net (2008).