Triple Storage for Random-Access Versioned Querying of RDF Archives

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

IDLab, Department of Electronics and Information Systems, Ghent University – imec

Abstract

When publishing Linked Open Datasets on the Web, most attention is typically directed to their latest version. Nevertheless, useful information is present in or between previous versions. In order to exploit this historical information in dataset analysis, we can maintain history in RDF archives. Existing approaches either require much storage space, or they expose an insufficiently expressive or efficient interface with respect to querying demands. In this article, we introduce an RDF archive indexing technique that is able to store datasets with a low storage overhead, by compressing consecutive versions and adding metadata for reducing lookup times. We introduce algorithms based on this technique for efficiently evaluating queries at a certain version, between any two versions, and for versions. 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. OSTRICH performs better for many smaller dataset versions than for few larger dataset versions. Furthermore, it enables efficient offsets in query result streams, which facilitates random access in results. Our storage technique reduces query evaluation time for versioned queries 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.

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

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]. With the continuously increasing number of Linked Open Datasets [2], archiving has become an issue for RDF [3] data as well. While the RDF data model itself is atemporal, Linked Datasets typically change over time [4] on dataset, schema, and/or instance level [5]. Such changes can include additions, modifications, or deletions of complete datasets, ontologies, and separate facts. While some evolving datasets, such as DBpedia [6], are published as separate dumps per version, more direct and efficient access to prior versions is desired.

Consequently, RDF archiving systems emerged that, for instance, support query engines that use the standard SPARQL query language [7]. In 2015, however, 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 query evaluation. Furthermore, no existing solution performs well for all versioned query types, namely querying at, between, and for different versions. An efficient 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. Since triple patterns are the basic element of SPARQL, such indexes can serve as an entry point for query en-
gines. Our solution is applicable as: (a) an alternative index with efficient triple-pattern-based access for existing engines, in order to improve the efficiency of more expressive SPARQL queries; and (b) a data source for the Web-friendly Triple Pattern Fragments [8] (TPF) interface, i.e., a Web API that provides access to RDF datasets by triple pattern and partitions the results in pages. We focus on the performance-critical features of stream-based results, query result offsets, and cardinality estimation. Stream-based results allow more memory-efficient processing when query results are plentiful. The capability to efficiently offset (and limit) a large stream reduces processing time if only a subset is needed. Cardinality estimation is essential for efficient query planning [8, 9] in many query engines.

Concretely, this work introduces a storage technique with the following contributions:

- a scalable versioned and compressed RDF index with offset support and result streaming;
- efficient query algorithms to evaluate triple pattern queries and perform cardinality estimation at, between, and for different versions, with optional offsets;
- an open-source implementation of this approach called OSTRICH;
- an extensive evaluation of OSTRICH compared to other approaches using an existing RDF archiving benchmark.

The main novelty of this work is the combination of efficient offset-enabled queries over a new index structure for RDF archives. We do not aim to compete with existing versioned SPARQL engines—full access to the language can instead be leveraged by different engines, or by using alternative RDF publication and querying methods such as the HTTP interface-based TPF approach. Optional versioning capabilities are possible for TPF by using VTPF [10], or datatime content-negotiation [11] through Memento [12].

This article is structured as follows. In the following section, we start by introducing the related work and our problem statement in Section 3. Next, in Section 4, we introduce the basic concepts of our approach, followed by our storage approach in Section 5, our ingestion algorithms in Section 6, and the accompanying querying algorithms in Section 7. After that, we present and discuss the evaluation of our implementation in Section 8. Finally, we present our conclusions in Section 9.

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. These systems can either be based on existing database technologies, such as relational databases [13] or document stores [14], or on techniques tailored to RDF. These technologies can even be combined, such as approaches that detect emergent schemas [15, 16] in RDF datasets, which allow parts of the data to be stored in relational databases in order to increase compression and improve the efficiency of query evaluation. These emergent schemas are recently being exploited as characteristics sets in native RDF approaches [17, 18]. For the remainder of this article, we focus the RDF-specific techniques that have direct relevance to our approach.

RDF-3X [9] is an RDF storage technique that is based on a clustered B+Tree with 18 indexes in which triples are sorted lexicographically. Given that a triple consists of a subject (S), predicate (P) and object (O), it includes six indexes for different triple component orders (SPO, SOP, OSP, OPS, PSO and POS), six aggregated indexes (SP, SO, PS, PO, OS, and OP), and three one-valued indexes (S, P, and O). 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. Furthermore, the store allows update operations. In our storage approach, we will reuse the concept of multiple indexes and encoding triple components in a dictionary.

Hexastore [19] is a similar approach as it uses six different sorted lists, one for each possible triple component order. Also, it uses dictionary encoding to compress common triple components. An alternative is Triplebit [20], which is based on a two-dimensional storage matrix. Columns correspond to predicates, and
rows to subjects and objects. This sparse matrix is compressed and dictionary-encoded to reduce storage requirements. Furthermore, it uses auxiliary index structures to improve index selection during query evaluation.

K2-Triples [21] is another RDF storage technique that uses k2-tree structures to the data, which results in high compression rates. These structures allow SPARQL queries to be evaluated in memory without decompressing the structures.

RDFCSA [22] is a compact RDF storage technique. It is a self-index that stores the data together with its index, which results in less storage space than raw storage. Furthermore, it is built on the concept of compressed suffix arrays, which compresses text while still allowing efficient pattern-based search on it. RDFCSA requires about twice the storage space compared to K2-Triples, but it is faster for most queries.

HDT [23] is a binary RDF representation that is highly compressed and provides indexing structures that enable efficient querying. It consists of three main components:

**Header** metadata describing the dataset

**Dictionary** mapping between triple components and unique IDs for reducing storage requirements of triples

**Storage** actual triples based on the IDs of the triple components

The dictionary component encodes triple components in four subsets. The first subset consists of triple components that exist both as subject and objects. The second and third subset respectively consists of the non-common subject and object component. The last subset consists of the predicate components. The storage part encodes triple components using the dictionary, compacts the triples in a sorted predicate and object adjacency list, and stores these adjacency list in a bitmap structure that efficiently indicates the borders of these consecutive adjacency list. By default, HDT only stores triples in the SPO-order. When querying is required, enhanced triple indexes are constructed to allow any triple pattern to be resolved efficiently based on the HDT-FoQ [24] approach. HDT archives are read-only, which leads to high efficiency and compressibility, but makes them unsuitable for cases where datasets change frequently. Its fast triple pattern queries and high compression rate make it an appropriate backend storage method for TPF [8] servers.

Approaches like LOD Laundromat [25] combine HDT and TPF for hosting and publishing 650K+ Linked Datasets containing 38B+ triples, proving its usefulness at large scale. Because of these reasons, we will reuse HDT snapshots as part of our storage solution.

### 2.2. RDF Archiving

Linked Open Datasets typically change over time [4], creating a need for maintaining the history of the datasets [1]. Hence, RDF archiving has been an active area of research over the last couple of years. In the domain of non-RDF graph databases, several graph database extensions exist. These extensions are either wrapper-based [26, 27], which leads to sub-optimal querying due to the lack of indexing, or they are based on changing the graph model [28, 29], which complicates the writing of queries. Furthermore, none of the existing non-RDF graph stores offer native versioning capabilities at the time of writing. We therefore only discuss RDF archiving for the remainder of this section.

Fernández et al. formally define an RDF archive [30] as follows: An RDF archive graph $A$ is a set of version-annotated triples. Where a version-annotated triple $(s, p, o):[i]$ is defined as an RDF triple $(s, p, o)$ with a label $i \in \mathbb{N}$ representing the version in which this triple holds. The set of all RDF triples $[3]$ is defined as $(U \cup B) \times U \times (U \cup B \cup L)$, where $U$, $B$, and $L$ respectively represent the disjoint, infinite sets of URIs, blank nodes, and literals. Furthermore, an RDF version of an RDF archive $A$ at snapshot $i$ is the RDF graph $A(i) = \{(s, p, o):[i] \in A\}$. For the remainder of this article, we use the notation $V_i$ to refer to the RDF version $A(i)$.

The DIACHRON data model [5] introduces the concept of diachronic datasets, i.e., datasets that contain diachronic entities, which are semantic entities that evolve over time. This data model formally defines a diachronic dataset as a set of dataset versions together with metadata annotations about this dataset. Each dataset version is defined as a set of records (i.e., tuples or triples), an associated schema, temporal information about this version and metadata specific to this version. Domain data must be reified in order to store it in the DIACHRON model. Due to the simplicity of RDF archive model compared to the domain-specific DIACHRON data model, we will use the model of Fernández et al. for the remainder of this document.
Systems for archiving Linked Open Data are categorized into three non-orthogonal storage strategies [1]:

- 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.

In the following sections, we discuss several existing RDF archiving systems, which use either pure IC, CB or TB, or hybrid IC/CB. Table 1 shows an overview of the discussed systems.

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

<table>
<thead>
<tr>
<th>Name</th>
<th>IC</th>
<th>CB</th>
<th>TB</th>
</tr>
</thead>
<tbody>
<tr>
<td>SemVersion [31]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cassidy et. al. [32]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R&amp;WBase [33]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R43ples [34]</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Hauptman et. al. [35]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-RDF-3X [36]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDF-TX [37]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v-RDFCSA [38]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dydra [39]</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TailR [40]</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

2.2.2.1 Independent Copies Approaches

SemVersion [31] was one of the first works to look into tracking different versions of RDF graphs. SemVersion is based on Concurrent Versions System (CVS) concepts to maintain different versions of ontologies, such as diff, branching and merging. Their approach consists of a separation of language-specific features with ontology versioning from general features together with RDF versioning. Unfortunately, the implementation details on triple storage and retrieval are unknown.

2.2.2 Change-Based Approaches

Based on the Theory of Patches from the Darcs software management system [41], Cassidy et. al. [32] propose 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. An implementation of their approach is provided using the Redland python library and MySQL by representing each patch as named graphs and serializing them in TriG [42]. Furthermore, a preliminary evaluation shows that their implementation is significantly slower than a native RDF store. They suggest a native implementation of the approach to avoid some of the overhead.

Im et. al. [43] 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 much storage overhead.

R&WBase [33] is a CB versioning system that adds an additional versioning layer to existing quad-stores. It adds the functionality of tagging, branching and merging for datasets. The graph element is used to represent the additions and deletions of patches, which are respectively the even and uneven graph IDs. Queries are resolved by looking at the highest even graph number of triples.

Graube et. al. introduce R43ples [34] which stores change sets as separate named graphs, making it a CB system. It supports the same versioning features as R&WBase and introduces new SPARQL keywords for these, such as REVISION, BRANCH and TAG. As reconstructing a version requires combining all change sets that came before, queries at a certain version are only usable for medium-sized datasets.

2.2.3 Timestamp-Based Approaches

Hauptman et. al. introduce a similar delta-based storage approach [35] by storing 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. Their implementation is based on Sesame [44] and Blazegraph [45] and is slower than snapshot-based approaches, but uses less disk space.

X-RDF-3X [36] is an extension of RDF-3X [9] which adds versioning support using the TB approach. On storage-level, each triple is annotated with a creation and deletion timestamp. This enables time-travel
queries where only triples valid at the given time are returned.

RDF-TX [37] is an in-memory query engine that supports a temporal SPARQL querying extension. The system is based on compressed multi-version B+Trees that outperforms similar systems such as X-RDF-3X in terms of querying efficiency. The required storage space after indexing is similar to that of X-RDF-3X.

v-RDFCSA [38] is a self-indexing RDF archive mechanism, based on the RDF self-index RDFCSA [22], that enables versioning queries on top of compressed RDF archives as a TB approach. They evaluate their approach using the BEAR [30] benchmark and show that they can reduce storage space requirements 60 times compared to raw storage. Furthermore, they reduce query evaluation times more than an order of magnitude compared to state of the art solutions.

Dydra [39] is an RDF graph storage platform with dataset versioning support. They introduce the REVISION keyword, which is similar to the GRAPH SPARQL keyword for referring to different dataset versions. Their implementation is based on B+Trees that are indexed in six ways GSPO, GPOS, GOSP, SPOG, POSG, OSPG. Each B+Tree value indicates the revisions in which a particular quad exists, which makes it a TB approach.

2.2.4 Hybrid Approaches

TailR [40] is an HTTP archive for Linked Data pages based on the Memento protocol [12] 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 Fig. 1. 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.

Results show that this is an effective way of reducing version reconstruction times [40], in particular for many versions. Within the delta chain, however, an increase in version reconstruction times can still be observed. Furthermore, it requires more storage space than pure delta-based approaches.

The authors’ implementation is based on a relational database system. Evaluation shows that resource lookup times for any version ranges between 1 and 50 ms for 10 versions containing around 500K triples. In total, these versions require ~64MB of storage space.

Fig. 1: Delta chain in which deltas are relative to the previous delta, as is done in TailR [40].

2.3. RDF Archiving Benchmarks

BEAR [30] is a benchmark for RDF archive systems. The BEAR benchmark is based on three real-world datasets from different domains:

BEAR-A 58 weekly snapshots from the Dynamic Linked Data Observatory [4]. This is the main dataset from the article on BEAR [30].

BEAR-B The 100 most volatile resources from DBpedia Live [46] over the course of three months as three different granularities: instant, hour and day.

BEAR-C Dataset descriptions from the Open Data Portal Watch [47] project over the course of 32 weeks.

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 versioned query types for both result sets with a low and a high cardinality. The queries are selected in such a way that they will be evaluated over triples of a certain dynamicity, which requires the benchmarked systems to handle this dynamicity well. BEAR-B provides a small collection of triple pattern queries corresponding to the real-world usage of DBpedia. Finally, BEAR-C provides 10 complex SPARQL queries that were created with the help of Open Data experts.

BEAR provides baseline RDF archive implementations based on HDT [23] and Jena’s [48] TDB store for the IC, CB, and TB approaches, but also hybrid IC/CB and TB/CB approaches. The hybrid approaches are based on snapshots followed by delta chains, as implemented by TailR [40]. Due to HDT not supporting quads, the TB and TB/CB approaches could not be implemented in the HDT baseline implementations.

Results show that IC for both Jena and HDT requires more storage space than the compressed deltas for the three datasets. CB results in less storage space for both approaches for BEAR-A and BEAR-B, but not for BEAR-C because that dataset is so dynamic that the deltas require more storage space than they
would in with IC. Jena-TB results in the least storage space of Jena-based approaches, however, it fails for BEAR-B-instant because of the large amount of versions as Jena is less efficient for many graphs.

The hybrid approaches are evaluated with different delta chain lengths and expectedly show that shorter delta chains lead to results similar to IC, and longer delta chains lead are similar to CB or TB. The queries for BEAR-A and BEAR-B show that IC results in constant evaluation times for any version, CB times increase for each following version, and TB also result in constant times. The HDT-based approaches outperform Jena in all cases because of its compressed nature. The IC/CB hybrid approaches similarly show increasing evaluation times for each version, with a drop each time a new snapshot is created. The IC/TB hybrid Jena approach has slowly increasing evaluation times for each version, but they are significantly lower than the regular TB approach.

The queries of BEAR-C currently can not be solved by the archiving strategies in a straightforward way, but they are designed to help foster the development of future RDF archiving solutions. While queries of BEAR-A and BEAR-B are just triple pattern queries and therefore do not cover the full SPARQL spectrum, they provide the basis for more complex queries, as is proven by the TPF framework [8], which makes them sufficient for benchmarking.

EvoGen [49] is an RDF archive systems benchmark that is based on the synthetic LUBM dataset generator [50]. It is an extension of the LUBM generator with additional classes and properties for introducing dataset evolution on schema-level. EvoGen enables the user to tweak parameters of the dataset and query generation process, for example to change the dataset dynamicity and the number of versions.

While EvoGen offers more flexibility than BEAR in terms of configurability, BEAR provides real-world datasets and baseline implementations which lowers the barrier towards its usage. Hence, we will use the BEAR dataset in this work for benchmarking our system.

2.4. Query Atoms

The query atoms that will be introduced in this section are based on the RDF data model [3] and SPARQL query language [7]. In these models, a triple pattern is defined as \((U \cup V) \times (U \cup V) \times (U \cup L \cup V)\), with \(V\) being the infinite set of variables. A set of triple patterns is called a Basic Graph Pattern, which forms the basis of a SPARQL query. The evaluation of a SPARQL query \(Q\) on an RDF graph \(G\) containing RDF triples, produces a bag of solution mappings \([[Q]]_G\).

To cover the retrieval demands in RDF archiving, five foundational query types were introduced [30], which are referred to as query atoms:

1. **Version materialization (VM)** retrieves data using a query \(Q\) targeted at a single version \(V_i\). Formally: \(VM(Q, V_i) = [[Q]]_V_i\). Example: Which books were present in the library yesterday?

2. **Delta materialization (DM)** retrieves query \(Q\)’s result change sets between two versions \(V_i\) and \(V_j\). Formally: \(DM(Q, V_i, V_j) = (\Omega', \Omega)\). With \(\Omega' = [[Q]]_V_i \setminus [[Q]]_V_j\) and \(\Omega = [[Q]]_V_j \setminus [[Q]]_V_i\). Example: Which books were returned or taken from the library between yesterday and now?

3. **Version query (VQ)** annotates query \(Q\)’s results with the versions (of RDF archive \(A\)) in which they are valid. Formally: \(VQ(Q, A) = \{(\Omega, W) | W = \{A(i) | \Omega=[[Q]]_A(i), i \in N \land \Omega \neq \emptyset\}\}\). Example: At what times was book X present in the library?

4. **Cross-version join (CV)** joins the results of two queries \((Q_1\) and \(Q_2\)) between versions \(V_i\) and \(V_j\). Formally: \(VM(Q_1, V_i) \bowtie VM(Q_2, V_j)\). Example: What books were present in the library yesterday and today?

5. **Change materialization (CM)** returns a list of versions in which a given query \(Q\) produces consecutively different results. Formally: \(\{(i, j) | i, j \in N, i < j, DM(Q, A(i), A(j)) = (\Omega', \Omega), \Omega' \cup \Omega \neq \emptyset, \exists k \in N : i < k < j\}\). Example: At what times was book X returned or taken from the library?

There exists a correspondence between these query atoms and the independent copies (IC), change-based (CB), and timestamp-based (TB) storage strategies.

Namely, VM queries are efficient in storage solutions that are based on IC, because there is indexing on version. On the other hand, IC-based solutions may introduce a large amount of overhead in terms of storage space because each version is stored separately. Furthermore, DM and VQ queries are less efficient for IC solutions. That is because DM queries require two fully-materialized versions to be compared on-the-fly, and VQ requires all versions to be queried at the same time.
DM queries can be efficient in CB solutions if the query version ranges correspond to the stored delta ranges. In all other cases, as well as for VM and VQ queries, the desired versions must be materialized on-the-fly, which will take increasingly more time for longer delta chains. CB solutions do however typically require less storage space than VM if there is sufficient overlap between each consecutive version.

Finally, VQ queries perform well for TB solutions because the timestamp annotation directly corresponds to VQ’s result format. VM and DM queries in this case are typically less efficient than for IC approaches, due to the missing version index. Furthermore, TB solutions can require less storage space compared to VM if the change ratio of the dataset is not too large.

In summary, IC, CB and TB approaches can perform well for certain query types, but they can be slow for others. On the other hand, this efficiency typically comes at the cost of a large storage overhead, as is the case for IC-based approaches.

DIACHRON QL [5] is a SPARQL query language extension based on the DIACHRON data model that provides functionality similar to these query atoms in order to query specific versions, changesets, or all versions.

3. Problem statement

As mentioned in Section 1, no RDF archiving solutions exist that allow efficient triple pattern querying at, between, and for different versions, in combination with a scalable storage model and efficient compression. In the context of query engines, streams are typically used to return query results, on which offsets and limits can be applied to reduce processing time if only a subset is needed. Offsets are used to skip a certain amount of elements, while limits are used to restrict the number of elements to a given amount. As such, RDF archiving solutions should also allow query results to be returned as offsettable streams. The ability to achieve such stream subsets is limited in existing solutions.

This leads us to the following research question: “How can we store RDF archives to enable efficient VM, DM and VQ triple pattern queries with offsets?”

The focus of this article is evaluating version materialization (VM), delta materialization (DM), and version (VQ) queries efficiently, as CV and CM queries can be expressed in terms of the other ones [51]. In total, our research question indentifies the following requirements:

- an efficient RDF archive storage technique;
- VM, DM and VQ triple pattern querying algorithms on top of this storage technique;
- efficient offsetting of the VM, DM, and VQ query result streams.

In this work, we lower query evaluation times by processing and storing more metadata during ingestion time. Instead of processing metadata during every lookup, this happens only once per version. This will increase ingestion times, but will improve the efficiency of performance-critical features within query engines and Linked Data interfaces, such as querying with offsets. To this end, we introduce the following hypotheses:

1. Our approach shows no influence of the selected versions on the querying efficiency of VM and DM triple pattern queries.
2. Our approach requires less storage space than state-of-the-art IC-based approaches.
3. For our approach, querying is slower for VM and equal or faster for DM and VQ than in state-of-the-art IC-based approaches.
4. Our approach requires more storage space than state-of-the-art CB-based approaches.
5. For our approach, querying is equal or faster than in state-of-the-art CB-based approaches.
6. Our approach reduces average query time compared to other non-IC approaches at the cost of increased ingestion time.

4. Overview of Storage and Querying approach

In this section, we lay the groundwork for the following sections. We introduce fundamental concepts that are required in our storage approach and its accompanying querying algorithms, which will be explained in Section 5 and Section 7, respectively.

To combine smart use of storage space with efficient processing of VM, DM, and VQ triple pattern queries, we employ a hybrid approach between the individual copies (IC), change-based (CB), and timestamp-based (TB) storage techniques (as discussed in Section 2). In summary, intermittent fully materialized snapshots are followed by delta chains. Each delta chain is stored in six tree-based indexes, where values are dictionary-encoded and timestamped to reduce
storage requirements and lookup times. These six indexes correspond to the combinations for storing three triple component orders separately for additions and deletions. The indexes for the three different triple component orders ensure that any triple pattern query can be resolved quickly. The additions and deletions are stored separately because access patterns to additions and deletions in deltas differ between VM, DM, and VQ queries. To efficiently support inter-delta DM queries, each addition and deletion value contains a local change flag that indicates if the change is not relative to the snapshot. Finally, in order to provide cardinality estimation for any triple pattern, we store an additional count data structure.

In the following sections, we discuss the most important distinguishing features of our approach. We elaborate on the novel hybrid IC/CB/TB storage technique that our approach is based on, the reason for using multiple indexes, having local change metadata, and methods for storing addition and deletion counts.

4.1. Snapshot And Delta Chain

Our storage technique is partially based on a hybrid IC/CB approach similar to Fig. 1. To avoid increasing reconstruction times, we construct the delta chain in an aggregated deltas [43] 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 only requires at most one delta and one snapshot. Although this does increase possible redundancies within delta chains, due to each delta inheriting the changes of its preceding delta, the overhead can be compensated with compression, which we discuss in Section 5.

Our storage approach consists of six different indexes that are used for separately storing additions and deletions in three different triple component orders, namely: SP, POS and OSP. These indexes are B+Trees, thereby, the starting triple for any triple pattern can be found in logarithmic time. Consequently, the next triples can be found by iterating through the links between each tree leaf. Table 2 shows an overview of which triple patterns can be mapped to which index. In contrast to other approaches [9, 19] that ensure certain triple orders, we use three indexes instead of all six possible component orders, because we only aim to reduce the iteration scope of the lookup tree for any triple pattern. For each possible triple pattern, we now have an index that locates the first triple component in logarithmic time, and identifies the terminating element of the result stream without necessarily having iterate to the last value of the tree. For some scenarios, it might be beneficial to ensure the order of triples in the result stream, so that more efficient stream joining algorithms can be used, such as sort-merge join. If this would be needed, OPS, PS0 and SP0 indexes could optionally be added so that all possible triple orders would be available.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OSTRICH</td>
<td>SP0</td>
<td>SP0</td>
<td>OSP</td>
<td>SP0</td>
<td>POS</td>
<td>POS</td>
<td>OSP</td>
<td>SP0</td>
</tr>
<tr>
<td>HDT-FoQ</td>
<td>SP0</td>
<td>SP0</td>
<td>SP0</td>
<td>SOP</td>
<td>OPS</td>
<td>OPS</td>
<td>PS0</td>
<td>PS0</td>
</tr>
</tbody>
</table>

Table 2: Overview of which triple patterns are queried inside which index in OSTRICH and HDT-FoQ.

Our approach could also act as a dedicated RDF archiving solution without (necessarily efficient) querying capabilities. In this case, only a single index would be required, such as SP0, which would reduce the required storage space even further. If querying would become required afterwards, the auxiliary SP0 and POS indexes could still be derived from this main index during a one-time, pre-querying processing phase.

This technique is similar to the HDT-FoQ [24] extension for HDT that adds additional indexes to a basic HDT file to enable faster querying for any triple pattern. The main difference is that HDT-FoQ uses the indexes OSP, PS0 and OPS, with a different triple pattern to index mapping as shown in Table 2. We chose our indexes in order to achieve a more balanced distribution from triple patterns to index, which could lead...

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

4.2. Multiple Indexes
to improved load balancing between indexes when queries are parallelized. HDT-FoQ uses SP0 for five triple pattern groups, OPS for two and PS0 for only a single group. Our approach uses SP0 for 4 groups, POS for two and OSP for only a single group. Future work is needed to evaluate the distribution for real-world queries. Additionally, the mapping from patterns SP0 to index SP0 in HDT-FoQ will lead to suboptimal query evaluation when a large number of distinct predicates is present.

4.3. Local Changes

A delta chain can contain multiple instances of the same triple, since it could be added in one version and removed in the next. Triples that revert a previous addition or deletion within the same delta chain, are called local changes, and are important for query evaluation. Determining the locality of changes can be costly, thus we pre-calculate this information during ingestion time and store it for each versioned triple, so that this does not have to happen during query-time.

When evaluating version materialization queries by combining a delta with its snapshot, all local changes should be filtered out. For example, a triple A that was deleted in version 1, but re-added in version 2, is cancelled out when materializing against version 2. For delta materialization, these local changes should be taken into account, because triple A should be marked as a deletion between versions 0 and 1, but as an addition between versions 1 and 2. Finally, for version queries, this information is also required so that the version ranges for each triple can be determined.

4.4. Addition And Deletion Counts

Parts of our querying algorithms depend on the ability to efficiently count the exact number of additions or deletions in a delta. Instead of naively counting triples by iterating over all of them, we propose two separate approaches for enabling efficient addition and deletion counting in deltas.

For additions, we store an additional mapping from triple pattern and version to number of additions so that counts can happen in constant time by just looking them up in the map. For deletions, we store additional metadata in the main deletions tree. Both of these approaches will be further explained in Section 5.

5. Hybrid Multiversion Storage Approach

In this section, we introduce our hybrid IC/CB/TB storage approach for storing multiple versions of an RDF dataset. Fig. 3 shows an overview of the main components. Our approach consists of an initial dataset snapshot—stored in HDT [23]—followed by a delta chain (similar to TailR [40]). The delta chain uses multiple compressed B+Trees for a TB-storage strategy (similar to Dydra [39]), applies dictionary-encoding to triples, and stores additional metadata to improve lookup times. In this section, we discuss each component in more detail. In the next section, we describe two ingestion algorithms based on this storage structure.

![Fig. 3: Overview of the main components of our hybrid IC/CB/TB storage approach.](image)

Throughout this section, we will use the example RDF archive from Table 3 to illustrate the different storage components with.

<table>
<thead>
<tr>
<th>Version</th>
<th>Triple</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>:Bob foaf:name &quot;Bobby&quot;</td>
</tr>
<tr>
<td>1</td>
<td>:Alice foaf:name &quot;Alice&quot;</td>
</tr>
<tr>
<td>1</td>
<td>:Bob foaf:name &quot;Bobby&quot;</td>
</tr>
<tr>
<td>2</td>
<td>:Bob foaf:name &quot;Bob&quot;</td>
</tr>
<tr>
<td>3</td>
<td>:Alice foaf:name &quot;Alice&quot;</td>
</tr>
<tr>
<td>3</td>
<td>:Bob foaf:name &quot;Bob&quot;</td>
</tr>
</tbody>
</table>

Table 3: Example of a small RDF archive with 4 versions. We assume that the following URI prefixes: : http://example.org, foaf: http://xmlns.com/foaf/0.1/
5.1. Snapshot Storage

As mentioned before, the start of each delta chain is a fully materialized snapshot. In order to provide sufficient efficiency for VM, DM and VQ querying with respect to all versions in the chain, we assume the following requirements for the snapshot storage:

- Any triple pattern query must be resolvable as triple streams.
- Offsets must be applicable to the result stream of any triple pattern query.
- Cardinality estimation for all triple pattern queries must be possible.

These requirements are needed for ensuring the efficiency of the querying algorithms that will be introduced in Section 7. For the implementation of snapshots, existing techniques such as HDT [23] fulfill all the requirements. Therefore, we do not introduce a new snapshot approach, but use HDT in our implementation. This will be explained further in Subsection 8.1.

5.2. Delta Chain Dictionary

A common technique in RDF indexes [23, 9, 20] is to use a dictionary for mapping triple components to numerical IDs. This is done for three main reasons: 1) reduce storage space if triple components are stored multiple times; 2) reducing I/O overhead when retrieving data; and 3) simplify and optimize querying. As our storage approach essentially stores each triple three or six times, a dictionary can definitely reduce storage space requirements.

Each delta chain consists of two dictionaries, one for the snapshot and one for the deltas. The snapshot dictionary consists of triple components that already existed in the snapshot. All other triple components are stored in the delta dictionary. This dictionary is shared between the additions and deletions, as the dictionary ignores whether or not the triple is an addition or deletion. How this distinction is made will be explained in Subsection 5.3. The snapshot dictionary can be optimized and sorted, as it will not change over time. The delta dictionary is volatile, as each new version can introduce new mappings.

During triple encoding (i.e., ingestion), the snapshot dictionary will always first be probed for existence of the triple component. If there is a match, that ID is used for storing the delta’s triple component. To identify the appropriate dictionary for triple decoding, a reserved bit is used where 1 indicates snapshot dictionary and 0 indicates the delta dictionary. The text-based dictionary values can be compressed to reduce storage space further, as they are likely to contain many redundancies.

Table 4 contains example encodings of the triple components.

| :Bob foaf:name "Bobby" :Alice "Alice" "Bob" |
| S0 | S1 | S2 | D0 | D1 | D2 |

Table 4: Example encoding of the triple components from Table 3. Instead of the reserved bit, IDs prefixed with S belong to the snapshot dictionary and those prefixed with D belong to the delta dictionary.

5.3. Delta Storage

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 annotated with a flag for addition or deletion. An overview of this storage technique is shown in Fig. 4, which will be explained in detail hereafter.

![Figure 4: Overview of the components for storing a delta chain. The value structure for the addition and deletion trees are indicated with the dashed nodes.](image)

The additions and deletions of deltas require different metadata in our querying algorithms, which will be
explained in Section 7. Additions and deletions are respectively stored in separate stores, which hold all additions and deletions from the complete delta chain. Each store uses B+Tree data structures, where a key corresponds to a triple and the value contains version information. The version information consists of a mapping from version to a local change flag as mentioned in Subsection 4.3 and, in case of deletions, also the relative position of the triple inside the delta. Even though triples can exist in multiple deltas in the same chain, they will only be stored once. Each addition and deletion store uses three trees with a different triple component order (SPO, POS and OSP), as discussed in Subsection 4.2.

The relative position of each triple inside the delta to the deletion trees speeds up the process of patching a snapshot’s triple pattern subset for any given offset. In fact, seven relative positions are stored for each deleted triple: one for each possible triple pattern (SP?, S?O, S??, ?PO, ?P?, ??O, ???), except for SPO since this position will always be 0 as each triple is stored only once. 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 and the local change flag during querying will be further explained in Section 7.

Table 5: Addition and deletion tree contents based on the example from Table 3 using the dictionary encoding from Table 4. Column + and - respectively represent the keys of the addition and deletion trees, which contains triples based on the encoded triple components. The remaining columns represent the values, i.e., a mapping from version (V) to the local change flag (L). For the deletion trees, values also include the relative positions for all essential triple patterns.

<table>
<thead>
<tr>
<th>+</th>
<th>V</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 S1 D1</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>S0 S1 D2</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>-</td>
<td>V</td>
<td>L</td>
</tr>
<tr>
<td>D0 S1 D1</td>
<td>2</td>
<td>T 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>S0 S1 S2</td>
<td>2</td>
<td>F 0 0 0 0 1 0 1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>F 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Table 5 represent the addition and deletion tree contents when the triples from the example in Table 3 are stored. The local change flag is enabled for D0 S1 D1 in the deletions tree for version 2, as it was previously added in version 1. The relative positions in the deletion tree for S0 S1 S2 is not the same for versions 2 and 3, because in version 2, the triple D0 S1 D1 also exists as a deletion, and when sorted, this comes before S0 S1 S2 for triple patterns ?P? and ???.

5.4. Addition Counts

As mentioned before in Subsection 4.4, in order to make the counting of matching addition triples for any triple pattern for any version more efficient, we propose to store an additional mapping from triple pattern and version to the number of matching additions. Furthermore, for being able to retrieve the total number of additions across all versions, we also propose to store this value for all triple patterns. This mapping must be calculated during ingestion time, so that counts during lookup time for any triple pattern at any version can be derived in constant time. For many triples and versions, the number of possible triple patterns can become very large, which can result in a large mapping store. To cope with this, we propose to only store the elements where their counts are larger than a certain
threshold. Elements that are not stored will have to be counted during lookup time. This is however not a problem for reasonably low thresholds, because the iteration scope in our indexes can be limited efficiently, as mentioned in Subsection 4.4. The count threshold introduces a trade-off between the storage requirements and the required triple counting during lookups.

5.5. Deletion Counts

As mentioned in Subsection 5.3, each deletion is annotated with its relative position in all deletions for that version. This position is exploited to perform deletion counting for any triple pattern and version. We look up the largest possible triple (sorted alphabetically) for the given triple pattern in the deletions tree, which can be done in logarithmic time by navigating in the tree to the largest possible match for the given triple pattern. If this does not result in a match for the triple pattern, no matches exist for the given triple pattern, and the count is zero. Otherwise, we take one plus the relative position of the matched deletion for the given triple pattern. Because we have queried the largest possible triple for that triple pattern in the given version, this will be the last deletion in the list, so this position corresponds to the total number of deletions in that case.

For example, when we want to determine the deletion count for `? foaf:name ?` (encoded: `? S1 ?`) in version 2 using the deletion tree contents from Table 5, we will find `S0 S1 S2` as largest triple in version 2. This triple has relative position 1 for `?P?`, so the total deletion count is 2 for this pattern. This is correct, as we have indeed two triples matching this pattern, namely `D0 S1 D1` and `S0 S1 S2`.

5.6. Metadata

Querying algorithms have to be able to detect the total number of versions across all delta chains. Therefore, we must store metadata regarding the delta chain version ranges. Assuming that version identifiers are numerical, a mapping can be maintained from version ID to delta chain. Additionally, a counter of the total number of versions must be maintained for when the last version must be identified.

6. Changeset Ingestion Algorithms

In this section, we discuss two ingestion algorithms: a memory-intensive batch algorithm and a memory-efficient streaming algorithm. These algorithms both take a changeset—containing additions and deletions—as input, and append it as a new version to the store. Note that the ingested changesets are regular changesets: they are relative to one another according to Fig. 1. Furthermore, we assume that the ingested changesets are valid changesets: they don’t contain impossible triple sequences such as a triple that is removed in two versions without having an addition in between. During ingestion, they will be transformed to the alternative delta chain structure as shown in Fig. 2. Within the scope of this article, we only discuss ingestion of deltas in a single delta chain following a snapshot.

Next to ingesting the added and removed triples, an ingestion algorithm for our storage approach must be able to calculate the appropriate metadata for the store as discussed in Subsection 5.3. More specifically, an ingestion algorithm has the following requirements:

- addition triples must be stored in all addition trees;
- additions and deletions must be annotated with their version;
- additions and deletions must be annotated with being a local change or not;
- deletions must be annotated with their relative position for all triple patterns.

6.1. Batch Ingestion

Our first algorithm to ingest data into the store naively loads everything in memory, and inserts the data accordingly. The advantage of this algorithm is its simplicity and the possibility to do straightforward optimizations during ingestion. The main disadvantage is the high memory consumption requirement for large versions.

Before we discuss the actual batch ingestion algorithm, we first introduce an in-memory changeset merging algorithm, which is required for the batch ingestion. Algorithm 1 contains the pseudocode of this algorithm. First, all contents of the original changeset are copied into the new changeset (line 3). After that, we iterate over all triples of the second changeset (line 4). If the changeset already contained the given triple (line 5), the local change flag is negated. Otherwise, the triple is added to the new changeset, and the local change flag is set to `false` (line 9,10). Finally, in both
cases the addition flag of the triple in the new changeset is copied from the second changeset (line 12).

```java
mergeChangsets(changesetOriginal, changesetIngest) {
    changesetNew = new Changeset()
    for (triple : changesetIngest.getTriples()) {
        if (changesetOriginal.contains(triple)) {
            localChange = !changesetOriginal.isLocalChange(triple)
            changesetNew.setLocalChange(triple, localChange)
            if (changesetOriginal.contains(triple)) {
                changesetNew.addAll(changesetOriginal)
            } else {
                changesetNew.setAddition(triple, changesetIngest.isAddition(triple))
            }
        } else {
            changesetNew.add(triple)
        }
        return changesetNew
    }
}
```

**Algorithm 1: In-memory changeset merging algorithm**

Because our querying algorithms require the relative position of each deletion within a changeset to be stored, we have to calculate these positions during ingestion. We do this using the helper function `calculatePositions(triple)`. This function depends on external mappings that persist over the duration of the ingestion phase that map from triple to a counter for each possible triple pattern. When this helper function is called for a certain triple, we increment the counters for the seven possible triple patterns of the triple. For the triple itself, we do not maintain a counter, as its value is always 1. Finally, the function returns a mapping for the current counter values of the seven triple patterns.

The batch ingestion algorithm starts by reading a complete changeset stream in-memory, sorting it in SPO order, and encoding all triple components using the dictionary. After that, it loads the changeset from the previous version in memory, which is required for merging it together with the new changeset using the algorithm from Algorithm 1. After that, we have the new changeset loaded in memory. Now, we load each added triple into the addition trees, together with their version and local change flag. After that, we load each deleted triple into the deletion trees with their version, local change flag, and relative positions. These positions are calculated using `calculatePositions(triple)`. For the sake of completeness, we included the batch algorithm in pseudocode in Appendix D.

Even though this algorithm is straightforward, it can require a large amount of memory for large changesets and long delta chains. The theoretical time complexity of this algorithm is $O(P + N \log(N))$ if the new changeset is already sorted, with $P$ the number of triples in the previous changeset, and $N$ the number of triples in the new changeset.

6.2. Streaming Ingestion

Because of the unbounded memory requirements of the batch ingestion algorithm, we introduce a more complex streaming ingestion algorithm. Just like the batch algorithm, it takes a changeset stream as input, with the additional requirement that the stream’s values must be sorted in SPO-order. This way the algorithm can assume a consistent order and act as a sort-merge join operation. Just as for the batch algorithm, we included this algorithm in pseudo-code in Appendix D.

In summary, the algorithm performs a sort-merge join over three streams in SPO-order: 1) the stream of input changeset elements that are encoded using the dictionary when each element is read, 2) the existing deletions over all versions and 3) the existing additions over all versions. The algorithm iterates over all streams together, until all of them are finished. The smallest triple (string-based) over all stream heads is handled in each iteration, and can be categorized in seven different cases where these stream heads are indicated by input, deletion and addition, respectively:

1. **Deletion is strictly smaller than both** input and addition.
   The current deletion is the smallest element. The unchanged deletion information can be copied to the new version. New relative positions must be calculated in this and all other cases where deletions are added.

2. **Addition is strictly smaller than both** input and deletion.
   Similar to the previous case, the current addition is now the smallest element, and its information can be copied to the new version.

3. **Input is strictly smaller than both** addition and deletion.
   A triple is added or removed that was not present before, so it can respectively be added as a non-local change addition or a non-local change deletion.

4. **Input and deletion are equal, but strictly smaller than** addition.
   In this case, the new triple already existed in the previous version as a deletion. If the new triple is an addition, it must be added as a local change.

5. **Input and addition are equal, but strictly smaller than** deletion.
Similar as in the previous case, the new triple now already existed as an addition. So the triple must be deleted as a local change if the new triple is a deletion.

6. Addition and deletion are equal, but strictly smaller than input.
   The triple existed as both an addition and deletion at some point. In this case, we copy over the one that existed at the latest version, as it will still apply in the new version.

7. Addition, deletion, and input are equal.
   Finally, the triple already existed as both an addition and deletion, and is equal to our new triple. This means that if the triple was an addition in the previous version, it becomes a deletion, and the other way around, and the local change flag can be inherited.

The theoretical memory requirement for this algorithm is much lower than the batch variant. That is because it only has to load at least three triples, i.e., the heads of each stream, in memory, instead of the complete new changeset. Furthermore, we still need to maintain the relative position counters for the deletions in all triple patterns. While these counters could also become large, a smart implementation could perform memory-mapping to avoid storing everything in memory. The lower memory requirements come at the cost of a higher logical complexity, but an equal time complexity (assuming sorted changesets).

7. 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 5. Each of these querying algorithms are based on result streams, enabling efficient offsets and limits, by exploiting the index structure from Section 5. Furthermore, we provide algorithms to provide count estimates for each query.

7.1. Version Materialization

Version Materialization (VM) is the most straightforward versioned query type, it allows you to query against a certain dataset version. In the following, we start by introducing our VM querying algorithm, after we give a simple example of this algorithm. After that, we prove the correctness of our VM algorithm and introduce a corresponding algorithm to provide count estimation for VM query results.

7.1.1 Query

Algorithm 2 introduces an algorithm for VM triple pattern queries based on our storage structure. It starts by determining the snapshot on which the given version is based (line 2). After that, this snapshot is queried for the given triple pattern and offset. If the given version is equal to the snapshot version, the snapshot iterator can be returned directly (line 3). In all other cases, this snapshot offset could only be an estimation, and the actual snapshot offset can be larger if deletions were introduced before the actual offset.

Our algorithm returns a stream where triples originating from the snapshot always come before the triples that were added in later additions. Because of that, the mechanism for determining the correct offset in the snapshot, additions and deletions streams can be split up into two cases. The given offset lies within the range of either snapshot minus deletion triples or within the range of addition triples. At this point, the additions and deletions streams are initialized to the start position for the given triple pattern and version.

```
queryVm(store, tp, version, originalOffset) {
    snapshot = store.getSnapshot(version).query(tp, originalOffset)
    if (snapshot.getVersion() = version) {
        return snapshot
    } else {
        additions = store.getAdditionsStream(tp, version)
        deletions = store.getDeletionsStream(tp, version)
        offset = 0
        do {
            if (originalOffset < snapshot.count(tp) - deletions.exactCount(tp)) {
                snapshot.offset(originalOffset + offset)
                offset = snapshot.peek()
                deletions = store.getDeletionsStream(tp, version, offsetTriple)
                offset = deletions.getOffset(tp)
            } else {
                snapshot.offset(snapshot.count(tp) + deletions.exactCount(tp))
            }
        } while (snapshot.getCurrentOffset() != originalOffset + offset)
        return PatchedSnapshotIterator(snapshot, deletions, additions)
    }
}
```

Algorithm 2: Version Materialization algorithm for triple patterns that produces a triple stream with an offset in a given version.

In the first case, when the offset lies within the snapshot and deletions range (line 11), we enter a loop that converges to the actual snapshot offset based on the deletions for the given triple pattern in the given version. This loop starts by determining the triple at
We then query the deletions tree for the given triple pattern and version (line 15), filter out local changes, and use the snapshot triple as offset. This triple-based offset is done by navigating through the tree to the smallest triple before or equal to the offset triple. We store an additional offset value (line 16), which corresponds to the current numerical offset inside the deletions stream. As long as the current snapshot offset is different from the sum of the original offset and the additional offset, we continue iterating this loop (line 17), which will continuously increase this additional offset value.

In the second case (line 19), the given offset lies within the additions range. Now, we terminate the snapshot stream by offsetting it after its last element (line 20), and we relatively offset the additions stream (line 21). This offset is calculated as the original offset subtracted with the number of snapshot triples incremented with the number of deletions.

Finally, we return a simple iterator starting from the three streams (line 25). This iterator performs a sort-merge join operation that removes each triple from the snapshot that also appears in the deletion stream, which can be done efficiently because of the consistent SPO-ordering. Once the snapshot and deletion streams have finished, the iterator will start emitting addition triples at the end of the stream. For all streams, local changes are filtered out because locally changed triples are cancelled out for the given version as explained in Subsection 4.3, so they should not be returned in materialized versions.

7.1.2 Example

We can use the deletion’s position in the delta as offset in the snapshot because this position represents the number of deletions that came before that triple inside the snapshot given a consistent triple order. Table 6 shows simplified storage contents where triples are represented as a single letter, and there is only a single snapshot and delta. In the following paragraphs, we explain the offset convergence loop of the algorithm in function of this data for different offsets.

<table>
<thead>
<tr>
<th>Snapshot</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deletions</td>
<td>B</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Positions</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Simplified storage contents example where triples are represented as a single letter. The snapshot contains six elements, and the next version contains three deletions. Each deletion is annotated with its position.

Offset 0
For offset zero, the snapshot is first queried for this offset, which results in a stream starting from A. Next, the deletions are queried with offset A, which results in no match, so the final snapshot stream starts from A.

Offset 1
For an offset of one, the snapshot stream initially starts from B. After that, the deletions stream is offset to B, which results in a match. The original offset (1), is increased with the position of B (0) and the constant 1, which results in a new snapshot offset of 2. We now apply this new snapshot offset. As the snapshot offset has changed, we enter a second iteration of the loop. Now, the head of the snapshot stream is C. We offset the deletions stream to C, which again results in B. As this offset results in the same snapshot offset, we stop iterating and use the snapshot stream with offset 2 starting from C.

Offset 2
For offset 2, the snapshot stream initially starts from C. After querying the deletions stream, we find B, with position 0. We update the snapshot offset to 2 + 0 + 1 = 3, which results in the snapshot stream with head D. Querying the deletions stream results in D with position 1. We now update the snapshot offset to 2 + 1 + 1 = 4, resulting in a stream with head E. We query the deletions again, resulting in E with position 2. Finally, we update the snapshot offset to 2 + 2 + 1 = 5 with stream head F. Querying the deletions results in the same E element, so we use this last offset in our final snapshot stream.

7.1.3 Estimated Count

In order to provide an estimated count for VM triple pattern queries, we introduce a straightforward algorithm that depends on the efficiency of the snapshot to provide count estimations for a given triple pattern.
Based on the snapshot count for a given triple pattern, the number of deletions for that version and triple pattern are subtracted and the number of additions are added. These last two can be resolved efficiently, as we precalculate and store expensive addition and deletion counts as explained in Subsection 5.4 and Subsection 5.5.

7.1.4 Correctness

In this section, we provide a proof that Algorithm 2 results in the correct stream offset for any given version and triple pattern. We do this by first introducing a set of notations, followed by several lemmas and corollaries, which lead up to our final theorem proof.

Notations:
We will make use of bracket notation to indicate lists (ordered sets):
- \( A[1] \) is the element at position 1 from the list \( A \).
- \( A + B \) is the concatenation of list \( A \) followed by list \( B \).

Furthermore, we will use the following definitions:
- \( \text{snapshot}(tp, \text{version}) \) is the ordered list of triples matching the given triple pattern \( tp \) in the corresponding snapshot, from here on shortened to \( \text{snapshot} \).
- \( \text{additions}(\text{version}) \) and \( \text{deletions}(\text{version}) \) are the corresponding ordered additions and deletions for the given version, from here on shortened to \( \text{additions} \) and \( \text{deletions} \).
- \( \text{originalOffset} \) is how much the versioned list should be shifted, from here on shortened to \( \text{ori} \).
- \( \text{PatchedSnapshotIterator}(\text{snapshot}, \text{deletions}, \text{additions}) \) is a function that returns the list \( \text{snapshot}\backslash\text{deletions} + \text{additions} \).

The following definitions correspond to elements from the loop on lines 12-17:
- \( \text{deletions}(x) \) is the ordered list \( \{d \mid d \in \text{deletions}, d \geq x\} \), with \( x \) a triple.
- \( \text{offset}(x) = \text{|deletions|} - \text{|deletions}(x)| \), with \( x \) a triple.
- \( t(l) \) is the triple generated at line 13-14 for iteration \( l \).
- \( \text{off}(l) \) is the offset generated at line 16 for iteration \( l \).

**Lemma 1**: \( \text{off}(n) \geq \text{off}(n-1) \)

Proof:

We prove this by induction over the iterations of the loop. For \( n=1 \) this follows from line 9 and \( \forall x \text{offset}(x) \geq 0 \).

For \( n+1 \) we know by induction that \( \text{off}(n) \geq \text{off}(n-1) \). Since \( \text{snapshot} \) is ordered, \( \text{snapshot}[	ext{ori} + \text{off}(n)] \geq \text{snapshot}[	ext{ori} + \text{off}(n-1)] \). From lines 13-14 follows that \( t(n) = \text{snapshot}[	ext{ori} + \text{off}(n-1)] \), together this gives \( t(n+1) \geq t(n) \).

From this, we get:
- \( \{d \mid d \in \text{deletions}, d \geq t(n+1)\} \subseteq \{d \mid d \in \text{deletions}, d \geq t(n)\} \)
- \( \text{deletions}(t(n+1)) \subseteq \text{deletions}(t(n)) \)
- \( |\text{deletions}(t(n+1))| \leq |\text{deletions}(t(n))| \)
- \( |\text{deletions}| - |\text{deletions}(t(n+1))| \geq |\text{deletions}| - |\text{deletions}(t(n))| \)
- \( \text{offset}(t(n+1)) \leq \text{offset}(t(n)) \)

Together with lines 15-16 this gives us \( \text{off}(n+1) \geq \text{off}(n) \).

**Corollary 1**: The loop on lines 12-17 always terminates.

Proof:
Following the definitions, the end condition of the loop is \( \text{ori} + \text{off}(n) = \text{ori} + \text{off}(n+1) \). From Lemma 1 we know that \( \text{off} \) is a non-decreasing function. Since \( \text{deletions} \) is a finite list of triples, there is an upper limit for \( \text{off}(\text{|deletions|}) \), causing \( \text{off} \) to stop increasing at some point which triggers the end condition.

**Corollary 2**: When the loop on lines 12-17 terminates, \( \text{offset} = |\{d \mid d \in \text{deletions}, d \leq \text{snapshot}[	ext{ori} + \text{offset}]\}| \) and \( \text{ori} + \text{offset} < |\text{snapshot}| \)

Proof:
The first part follows from the definition of \( \text{deletions} \) and \( \text{offset} \). The second part follows from \( \text{offset} = |\text{deletions}| \) and line 11.

**Theorem 1**: \( \text{queryVm} \) returns a sublist of \( (\text{snapshot}\backslash\text{deletions} + \text{additions}) \), starting at the given offset.

Proof:
If the given version is equal to a snapshot, there are no additions or deletions so this follows directly from lines 2-4.

Following the definition of \( \text{deletions} \), \( \forall x \in \text{deletions}: x \in \text{snapshot} \) and thus \( |\text{snapshot}\backslash\text{deletions}| = |\text{snapshot}| - |\text{deletions}| \).

Due to the ordered nature of \( \text{snapshot} \) and \( \text{deletions} \), if \( \text{ori} < |\text{snapshot}\backslash\text{deletions}| \),
version[ori] = snapshot[ori + |D|] with D = {d | d ∈ deletions, d < snapshot[ori + |D|]}. Due to |snapshot\deletions| = |snapshot| - |deletions|, this corresponds to the if-statement on line 11. From Corollary 1 we know that the loop terminates and from Corollary 2 and line 13 that snapshot points to the element at position ori + |{d | d ∈ deletions, d ≤ snapshot[ori + offset]}| which, together with additions starting at index 0 and line 25, returns the requested result.

If ori ≥ |snapshot\deletions|, version[ori] = additions[ori - |snapshot\deletions|]. From lines 20-22 follows that snapshot gets emptied and additions gets shifted for the remaining required elements (ori - |snapshot\deletions|) which then also returns the requested result on line 25.

7.2. 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. Furthermore, we introduce an equivalent algorithm for estimating the number of results for these queries.

7.2.1 Query

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.

7.2.2 Estimated Count

For the first case, the start version corresponds to the snapshot version. The estimated number of results is then the number of snapshot triples for the pattern summed up with the exact number of deletions and additions for the pattern.

In the second case the start version does not correspond to the snapshot version. We estimate the total count as the sum of the additions and deletions for the given triple pattern in both versions. This may only be a rough estimate, but will always be an upper bound, as the triples that were changed twice within the version range and negate each other are also counted. For exact counting, this number of negated triples should be subtracted.

7.3. 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. In the following sections, we introduce an algorithm for performing triple pattern version queries and an algorithm for estimating the total number of matching triples for the former queries.

7.3.1 Query

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.

7.3.2 Estimated Count
Calculating the number of unique triples matching any triple pattern version query is trivial. We simply retrieve the count for the given triple pattern in the given snapshot and add the number of additions for the given triple pattern over all versions. The number of deletions should not be taken into account here, as this information is only required for determining the version annotation in the version query results.

8. 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.

8.1. 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 [23] as snapshot technology as it conforms to all the requirements for our approach. 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. OSTRICH immediately generates the main SP index and the auxiliary OS5 and PO5 indexes. In future work, OSTRICH could be modified to only generate the main index and delay auxiliary index generation to a later stage. Memory-mapping is required so that not all data must be loaded in-memory when queries are evaluated, which would not always be possible for large datasets. For our delta dictionary, we extend HDT’s dictionary implementation with adjustments to make it work with unsorted triple components. We compress this delta dictionary with gzip, which requires decompression during querying and ingestion. Finally, for storing our addition counts, we use the Hash Database of Kyoto Cabinet, which is also memory-mapped.

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. 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 [8], with the VTPF feature [10].

8.2. Experimental Setup

As mentioned before in Subsection 2.3, we evaluate our approach using the BEAR benchmark. We chose for this benchmark because it provides a complete set of tools and data for benchmarking RDF versioning systems, containing datasets, queries and easy-to-use engines to compare with.

We extended the existing BEAR implementation for the evaluation of offsets. We did this by implementing custom offset features into each of the BEAR approaches. Only for VM queries in HDT-IC an efficient implementation (HDT-IC+) could be made because of HDT’s native offset capabilities. In all other cases, naive offsets had to be implemented by iterating over the result stream until a number of elements equal to the desired offset were consumed. This modified implementation is available on GitHub (https://github.com/rdiostrich/bear/tree/ostrich-eval-journal). To test the scalability of our approach for datasets with few and large versions, we use the BEAR-A benchmark. We use the ten first versions of the BEAR-A dataset, 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. The daily dataset contains 89 versions and the hourly dataset contains 1,299 versions, both of them have around 48K triples per version. We did not evaluate BEAR-B-instant, because OSTRICH requires increasingly more time for each new version ingestion, as will be shown in the next section. As BEAR-B-hourly with 1,299 versions already takes more than three days to ingest, the 21,046 versions from BEAR-B-instant would require too much time to ingest. 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.

For BEAR-A, we use all 7 of the provided query-sets, each containing at most 50 triple pattern queries,
one with a high result cardinality and one with a low result cardinality. These querysets correspond to all possible triple pattern materializations, except for triple patterns where each component is blank. For BEAR-B, only two querysets are provided, those that correspond to \(?P?\) and \(?P0\) queries. The number of BEAR-B queries is more limited, but they are derived from real-world DBpedia queries which makes them useful for testing real-world applicability. All of these queries are evaluated as VM queries on all versions, as DM between the first version and all other versions, and as VQ.

For a complete comparison with other approaches, we re-evaluated BEAR’s Jena and HDT-based RDF archive implementations. More specifically, we ran all BEAR-A queries against Jena with the IC, CB, TB and hybrid CB/TB implementation, and HDT with the IC and CB implementations using the BEAR-A dataset for ten versions. We did the same for BEAR-B with the daily and hourly dataset. 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.

Additionally, we evaluated the ingestion rates and storage sizes for all approaches. Furthermore, we compared the ingestion rate for the two different ingestion algorithms of OSTRICH. The batch-based algorithm expectedly ran out of memory for larger amounts of versions, so we used the streaming-based algorithm for all further evaluations.

Finally, we evaluated the offset capabilities of OSTRICH by comparing it with custom offset implementations for the other approaches. We evaluated the blank triple pattern query with offsets ranging from 2 to 4,096 with a limit of 10 results.

8.3. 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/).

8.3.1 Ingestion

Table 7 and Table 8 respectively show the storage requirements and ingestion times for the different approaches for the three different benchmarks. For BEAR-A, the HDT-based approaches outperform OSTRICH in terms of ingestion time, they are about two orders of magnitude faster. Only HDT-CB requires slightly less storage space. The Jena-based approaches ingest one order of magnitude faster than OSTRICH, but require more storage space. For BEAR-B-daily, OSTRICH requires less storage space than all other approaches except for HDT-CB at the cost of slower ingestion. For BEAR-B-hourly, only HDT-CB and Jena-CB/TB require about 8 to 4 times less space than OSTRICH. For BEAR-B-daily and BEAR-B-hourly, OSTRICH even requires less storage space than gzip on raw N-Triples.

As mentioned in Subsection 5.4, we use a threshold to define which addition count values should be stored, and which ones should be evaluated at query time. For our experiments, we fixed this count threshold at 200, which has been empirically determined through various experiments as a good value. For values higher than 200, the addition counts started having a noticeable impact on the performance of count estimation. This threshold value means that when a triple pattern has 200 matching additions, then this count will be stored. Table 9 shows that the storage space of the addition count datastructure in the case of BEAR-A and BEAR-B-hourly is insignificant compared to the total space requirements. However, for BEAR-B-daily, addition counts take up 37.05% of the total size with still an acceptable absolute size, as the addition and deletion trees require relatively less space, because of the lower amount of versions. Within the scope of this work, we use this fixed threshold of 200. We consider investigating the impact of different threshold levels and methods for dynamically determining optimal levels future work.

Fig. 5 shows linearly increasing ingestion rate for each consecutive version for BEAR-A, while Fig. 6 shows corresponding linearly increasing storage sizes. Analogously, Fig. 7 shows the ingestion rate for BEAR-B-hourly, which increases linearly until around version 1100, after which it increases significantly. Fig. 8 shows near-linearly increasing storage sizes.

Fig. 9 compares the BEAR-A ingestion rate of the streaming and batch algorithms. The streaming algorithm starts of slower than the batch algorithm but grows linearly, while the batch algorithm consumes a large amount of memory, resulting in slower ingestion.
after version 8 and an out-of-memory error after version 10.

<table>
<thead>
<tr>
<th>Approach</th>
<th>BEAR-A</th>
<th>BEAR-B-daily</th>
<th>BEAR-B-hourly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw (N-Triples)</td>
<td>46,069.76</td>
<td>556.44</td>
<td>8,314.86</td>
</tr>
<tr>
<td>Raw (gzip)</td>
<td>3,194.88</td>
<td>30.98</td>
<td>466.35</td>
</tr>
<tr>
<td>OSTRICH</td>
<td>3,102.72</td>
<td>12.32</td>
<td>187.46</td>
</tr>
<tr>
<td>Jena-IC</td>
<td>32,808.96</td>
<td>415.32</td>
<td>6,233.92</td>
</tr>
<tr>
<td>Jena-CB</td>
<td>18,216.96</td>
<td>42.82</td>
<td>473.41</td>
</tr>
<tr>
<td>Jena-CB/TB</td>
<td>31,160.32</td>
<td>22.83</td>
<td>53.84</td>
</tr>
<tr>
<td>HDT-IC</td>
<td>5,335.04</td>
<td>142.08</td>
<td>2,127.57</td>
</tr>
<tr>
<td>HDT-CB</td>
<td>2,682.88</td>
<td>5.96</td>
<td>24.39</td>
</tr>
</tbody>
</table>

Table 7: Storage sizes for each of the RDF archive approaches in MB with BEAR-A, BEAR-B-daily and BEAR-B-hourly. The additional storage size for the auxiliary OSTRICH and HDT indexes are provided as separate rows. The lowest sizes per dataset are indicated in italics.

<table>
<thead>
<tr>
<th>Approach</th>
<th>BEAR-A</th>
<th>BEAR-B-daily</th>
<th>BEAR-B-hourly</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSTRICH</td>
<td>2,256</td>
<td>12.36</td>
<td>4,497.32</td>
</tr>
<tr>
<td>Jena-IC</td>
<td>443</td>
<td>8.91</td>
<td>142.26</td>
</tr>
<tr>
<td>Jena-CB</td>
<td>226</td>
<td>9.53</td>
<td>173.48</td>
</tr>
<tr>
<td>Jena-TB</td>
<td>1,746</td>
<td>0.35</td>
<td>70.56</td>
</tr>
<tr>
<td>Jena-CB/TB</td>
<td>679</td>
<td>0.35</td>
<td>0.65</td>
</tr>
<tr>
<td>HDT-IC</td>
<td>34</td>
<td>0.39</td>
<td>5.89</td>
</tr>
<tr>
<td>HDT-CB</td>
<td>18</td>
<td>0.02</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 8: Ingestion times for each of the RDF archive approaches with BEAR-A, BEAR-B-daily and BEAR-B-hourly. The lowest times per dataset are indicated in italics.

<table>
<thead>
<tr>
<th>Approach</th>
<th>BEAR-A</th>
<th>BEAR-B-daily</th>
<th>BEAR-B-hourly</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSTRICH</td>
<td>13.69 (0.29%)</td>
<td>6.25 (37.05%)</td>
<td>15.62 (3.46%)</td>
</tr>
</tbody>
</table>

Table 9: Storage sizes of the OSTRICH addition count component in MB with BEAR-A, BEAR-B-daily and BEAR-B-hourly. The percentage of storage space that this component requires compared to the complete store is indicated between brackets.

Fig. 5: OSTRICH ingestion durations for each consecutive BEAR-A version in minutes for an increasing number of versions, showing a linear growth.

Fig. 6: Cumulative OSTRICH store sizes for each consecutive BEAR-A version in GB for an increasing number of versions, showing a linear growth.

Fig. 7: OSTRICH ingestion durations for each consecutive BEAR-B-hourly version in minutes for an increasing number of versions.
Table 10 presents the compressibility of datasets without auxiliary indexes, showing that OSTRICH
and the HDT-based approaches significantly improve compressibility compared to the original N-Triples serialization. We omitted the results from the Jena-based approaches in this table, as all compressed sizes were in all cases two to three times larger than the N-Triples compression.

8.3.3 QuerEvaluationOn

Figures 10, 11 and 12 respectively summarize the VM, DM and VQ query durations of all BEAR-A queries on the ten first versions of the BEAR-A dataset for the different approaches. HDT-IC clearly outperforms all other approaches in all cases, while the Jena-based approaches are orders of magnitude slower than the HDT-based approaches and OSTRICH in all cases. OSTRICH is about two times faster than HDT-CB for VM queries, and slightly slower for both DM and VQ queries. For DM queries, HDT-CB does however continuously become slower for larger versions, while the lookup times for OSTRICH remain constant. From version 7, OSTRICH is faster than HDT-CB. Appendix A contains more detailed plots for each BEAR-A queryset, in which we can see that all approaches collectively become slower for queries with a higher result cardinality, and that predicate-queries are also significantly slower for all approaches.
Figures 13, 14 and 15 contain the query duration results for the BEAR-B queries on the complete BEAR-B-daily dataset for the different approaches. Jena-based approaches are again slower than both the HDT-based ones and OSTRICH. For VM queries, OSTRICH is slower than HDT-IC, but faster than HDT-CB, which becomes slower for larger versions. For DM queries, OSTRICH is faster than HDT-CB for the second half of the versions, and slightly faster HDT-IC. The difference between HDT-IC and OSTRICH is however insignificant in this case, as can be seen in Appendix B. For VQ queries, OSTRICH is significantly faster than all other approaches. Appendix B contains more detailed plots for this case, in which we can see that predicate-queries are again consistently slower for all approaches.

Figures 16, 17 and 18 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. Appendix C contains the more de-
tailed plots with the same conclusion as before that predicate-queries are slower.

Fig. 16: Median BEAR-B-hourly VM query results for all triple patterns for all versions.

Fig. 17: Median BEAR-B-hourly DM query results for all triple patterns from version 0 to all other versions.

Fig. 18: Median BEAR-B-hourly VQ query results for all triple patterns.

8.3.4 Offset

From our evaluation of offsets, Fig. 19 shows that OSTRICH offset evaluation remain below 1ms, while other approaches grow beyond that for larger offsets, except for HDT-IC+. HDT-CB, Jena-CB and Jena-CB/TB are not included in this and the following figures because they require full materialization before offsets can be applied, which is expensive and therefore take a very long time to evaluate. For DM queries, all approaches have growing evaluating times for larger offsets including OSTRICH, as can be seen in Fig. 20. Finally, OSTRICH has VQ evaluation times that are approximately independent of the offset value, while other approaches again have growing evaluation times, as shown in Fig. 21.
8.4. Discussion

In this section, we interpret and discuss the results from previous section. We discuss the ingestion, compressibility, query evaluation, offset efficiency and test our hypotheses.

8.4.1 Ingestion

For all evaluated cases, OSTRICH requires less storage space than most non-CB approaches. The CB and CB/TB approaches in most cases outperform OSTRICH in terms of storage space efficiency due to the additional metadata that OSTRICH stores per triple. Because of this, most other approaches require less time to ingest new data. These timing results should however be interpreted correctly, because all other approaches receive their input data in the appropriate format (IC, CB, TB, CB/TB), while OSTRICH does not. OSTRICH must convert CB input at runtime to the alternative CB structure where deltas are relative to the snapshot, which explains the larger ingestion times. As an example, Fig. 22 shows the number of triples in each BEAR-B-hourly version where the deltas have been transformed to the alternative delta structure that OSTRICH uses. Just like the first part of Fig. 7, this graph also increases linearly, which indicates that the large number of triples that need to be handled for long delta chains is one of the main bottlenecks for OSTRICH. This is also the reason why OSTRICH has memory issues during ingestion at the end of such chains. One future optimization could be to maintain the last version of each chain in a separate index for faster patching. Or a new ingestion algorithm could be implemented that accepts input in the correct alternative CB format. Alternatively, a new snapshot could dynamically be created when ingestion time becomes too large, which could for example for BEAR-B-hourly take place around version 1000.

Fig. 22: Total number of triples for each BEAR-B-hourly version when converted to the alternative CB structure used by OSTRICH, i.e., each triple is an addition or deletion relative to the first version instead of the previous version.

The BEAR-A and BEAR-B-hourly datasets indicate the limitations of the ingestion algorithm in our system. The results for BEAR-A show that OSTRICH ingests slowly for many very large versions, but it is still possible because of the memory-efficient streaming algorithm. The results for BEAR-B-hourly show that OSTRICH should not be used when the number of versions is very large. Furthermore, for each additional version in a dataset, the ingestion time increases. This is a direct consequence of our alternative delta chain method where all deltas are relative to a snapshot. That is the reason why when new deltas are inserted, the previous one must be fully materialized by iterating over all existing triples, because no version index exists.

In Fig. 7, we can observe large fluctuations in ingestion time around version 1,200 of BEAR-B-hourly. This is caused by the large amount of versions that are stored for each tree value. Since each version requires a mapping to seven triple pattern indexes and one local change flag in the deletion tree, value sizes become non-negligible for large amounts of versions. Each version value requires 28 uncompressed bytes, which results in more than 32KB for a triple in 1,200 versions. At that point, the values start to form a bottleneck as only 1,024 elements can be loaded in-memory using the default page cache size of 32MB, which causes a large amount of swapping. This could be solved by either tweaking the B+Tree parameters for this large amount of versions, reducing storage requirements for each value, or by dynamically creating a new snapshot.

We compared the streaming and batch-based ingestion algorithm in Fig. 9. The batch algorithm is initially faster because most operations can happen in memory, while the streaming algorithm only uses a small fraction of that memory, which makes the latter usable for very large datasets that don’t fit in memory. In fu-
ture work, a hybrid between the current streaming and batch algorithm could be investigated, i.e., a streaming algorithm with a larger buffer size, which is faster, but doesn’t require unbounded amounts of memory.

8.4.2 Compressibility

As shown in Table 10, when applying gzip directly on the raw N-Triples input already achieves significant space savings. However, OSTRICH, HDT-IC and HDT-CB are able to reduce the required storage space even further when they are used as a pre-processing step before applying gzip. This shows that these approaches are better—storage-wise—for the archival of versioned datasets. This table also shows that OSTRICH datasets with more versions are more prone to space savings using compression techniques like gzip compared to OSTRICH datasets with fewer versions.

8.4.3 QuEvaluationTion

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.

8.4.4 Offsets

One of our initial requirements was to design a system that allows efficient offsetting of VM, DM and VQ result streams. As shown in last section, for both VM and VQ queries, the lookup times for various offsets remain approximately constant. For VM queries, this can fluctuate slightly for certain offsets due to the loop section inside the VM algorithm for determining the starting position inside the snapshot and deletion tree. For DM queries, we do however observe an increase in lookup times for larger offsets. That is because the current DM algorithm naively offsets these streams by iterating over the stream until a number of elements equal to the desired offset have been consumed. Furthermore, other IC and TB approaches outperform OSTRICH’s DM result stream offsetting. This introduces a new point of improvement for future work, seeing whether or not OSTRICH would allow more efficient DM offsets by adjusting either the algorithm or the storage format.

8.4.5 Hypotheses

In Section 3, we introduced six hypotheses, which we will validate in this section based on our experimental results. We will only consider the comparison between OSTRICH and HDT-based approaches, as OSTRICH outperforms the Jena-based approaches for all cases in terms of lookup times. These validations were done using R, for which the source code can be found on GitHub (https://github.com/rdfostrich/os­trich-bear-results/). Tables containing p-values of the results can be found in Appendix E.

For our first hypothesis, we expect OSTRICH lookup times to remain independent of version for VM and DM queries. We validate this hypothesis by building a linear regression model with as response the lookup time, and as factors version and number of results. Table 11 in the appendix contains the influence of each factor, which shows that for all cases, we can accept the null hypothesis that the version factor has no influence on the models with a confidence of 99%. Based on these results, we accept our first hypothesis.

Hypothesis 2 states that OSTRICH requires less storage space than IC-based approaches, and Hypothesis 3 correspondingly states that query evaluation is slower for VM and faster or equal for DM and VQ. Results from previous section showed that for BEAR-A, BEAR-B-daily and BEAR-B-hourly, OSTRICH requires less space than HDT-IC, which means that we accept Hypothesis 2. In order to validate that query evaluation is slower for VM but faster or equal for DM and VQ, we compared the means using the two-sample t-test, for which the results can be found in Table 12 in the appendix. In all cases, the means are not equal with a confidence of 95%. For BEAR-B-daily and BEAR-B-hourly, HDT-IC is faster for VM queries, but slower for DM and VQ queries. For
BEAR-A, HDT-IC is faster for all query types. We therefore reject Hypothesis 3, as it does not apply for BEAR-A, but it is valid for BEAR-B-daily and BEAR-B-hourly. This means that OSTRICH typically requires less storage space than IC-based approaches, and outperforms other approaches in terms of querying efficiency unless the number of versions is small or for VM queries.

In Hypothesis 4, we stated that OSTRICH requires more storage space than CB-based approaches, and in Hypothesis 5 that query evaluation is faster or equal. In all cases OSTRICH requires more storage space than HDT-CB, which is why we accept Hypothesis 4. For the query evaluation, we again compare the means in Table 13 in the appendix using the same test. In BEAR-A, VQ queries in OSTRICH are not faster for BEAR-A, and VM queries in OSTRICH are not faster for BEAR-B-daily, which is why we reject Hypothesis 5. However, only one in three query atoms are not fulfilled, and OSTRICH is faster than HDT-CB for BEAR-B-hourly. In general, OSTRICH requires more storage space than CB-based approaches, and query evaluation is faster unless the number of versions is low.

Finally, in our last hypothesis, we state that average query evaluation times are lower than other non-IC approaches at the cost of increased ingestion times. In all cases, the ingestion time for OSTRICH is higher than the other approaches, and as shown in Table 13 in the appendix, query evaluation times for non-IC approaches are lower for BEAR-B-hourly. This means that we reject Hypothesis 6 because it only holds for BEAR-B-hourly and not for BEAR-A and BEAR-B-daily. In general, OSTRICH ingestion is slower than other approaches, but improves query evaluation time compared to other non-IC approaches, unless the number of versions is low.

In this section, we accepted three of the six hypotheses. As these are statistical hypotheses, these do not necessarily indicate negative results of our approach. Instead, they allow us to provide general guidelines on where our approach can be used effectively, and where not.

9. Conclusions

In this article, we introduced an RDF archive storage method with accompanied algorithms for evaluating VM, DM, and VQ queries, with efficient result offsets. Our novel storage technique is a hybrid of the IC/CB/TB approaches, because we store sequences of snapshots followed by delta chains. The evaluation of our OSTRICH implementation shows that this technique offers a new trade-off in terms of ingestion time, storage size and lookup times. By preprocessing and storing additional data during ingestion, we can reduce lookup times for VM, DM and VQ queries compared to CB and TB approaches. Our approach requires less storage space than IC approaches, at the cost of slightly slower VM queries, but comparable DM queries. Furthermore, our technique is faster than CB approaches, at the cost of more storage space. Additionally, VQ queries become increasingly more efficient for datasets with larger amounts of versions compared to IC, CB and TB approaches. Our current implementation supports a single snapshot and delta chain as a proof of concept, but production environments would normally incorporate more frequent snapshots, balancing between storage and querying requirements.

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. Furthermore, by reusing the highly efficient HDT format for snapshots, existing HDT files can directly be loaded by OSTRICH and patched with additional versions afterwards.

OSTRICH fulfills the requirements [51] for a back-end RDF archive storage solution for supporting versioning queries in the TPF framework. Together with the VTPF [10] interface feature, RDF archives can be queried on the Web at a low cost, as demonstrated on our public VTPF entrypoint (http://versioned.linkeddatafragments.org/ bear). TPF only requires triple pattern indexes with count metadata, which means that TPF clients are able to evaluate full VM SPARQL queries using OSTRICH and VTPF. In future work, the TPF client will be extended to also support DM and VQ SPARQL queries.

With OSTRICH, we provide a technique for publishing and querying RDF archives at Web-scale. 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. Solutions could be based on existing cost models [52] for determining whether a new snapshot or delta should be created based on quantified time and space parameters. Furthermore, branching and merging of different version chains can be investigated.
Our approach succeeds in reducing the cost for publishing RDF archives on the Web. This lowers the barrier towards intelligent clients in the Semantic Web [53] that require evolving data, with the goal of time-sensitive querying over the ever-evolving Web of data.

Acknowledgements

We would like to thank Christophe Billiet for providing his insights into temporal databases. We thank Giorgos Flouris for his comments on the structure and contents of this article, and Javier D. Fernández for his help in setting up and running the BEAR benchmark. The described research activities were funded by Ghent University, imec, Flanders Innovation & Entrepreneurship (AIO), and the European Union. Ruben Verborgh is a postdoctoral fellow of the Research Foundation – Flanders.

References


In this appendix section, we list all measured BEAR-A query evaluation durations. Each figure contains the durations for each storage approach, being OSTRICH, HDT-IC, HDT-CB, Jena-IC, Jena-CB, Jena-TB and Jena-CB/TB.

**Fig. 23:** Average VM query results for S?? triple patterns with a low cardinality for all versions.

**Fig. 24:** Average VM query results for S?? triple patterns with a high cardinality for all versions.

**Fig. 25:** Average VM query results for ?P?? triple patterns with a low cardinality for all versions.

**Fig. 26:** Average VM query results for ?P?? triple patterns with a high cardinality for all versions.

---

**Appendix**

**A. BEAR-A Query Evaluation results**


Fig. 27: Average VM query results for ?O triple patterns with a low cardinality for all versions.

Fig. 28: Average VM query results for ?O triple patterns with a high cardinality for all versions.

Fig. 29: Average VM query results for SP? triple patterns with a low cardinality for all versions.

Fig. 30: Average VM query results for SP? triple patterns with a high cardinality for all versions.

Fig. 31: Average VM query results for S?O triple patterns with a low cardinality for all versions.

Fig. 32: Average VM query results for ?PO triple patterns with a low cardinality for all versions.

Fig. 33: Average VM query results for ?PO triple patterns with a high cardinality for all versions.

Fig. 34: Average VM query results for SPO triple patterns for all versions.
Fig. 35: Average DM query results for S?? triple patterns with a low cardinality from version 0 to all other versions.

Fig. 36: Average DM query results for S?? triple patterns with a high cardinality from version 0 to all other versions.

Fig. 37: Average DM query results for ?P? triple patterns with a low cardinality from version 0 to all other versions.

Fig. 38: Average DM query results for ?P? triple patterns with a high cardinality from version 0 to all other versions.

Fig. 39: Average DM query results for ??O triple patterns with a low cardinality from version 0 to all other versions.

Fig. 40: Average DM query results for ??O triple patterns with a high cardinality from version 0 to all other versions.

Fig. 41: Average DM query results for SP? triple patterns with a low cardinality from version 0 to all other versions.

Fig. 42: Average DM query results for SP? triple patterns with a high cardinality from version 0 to all other versions.
Fig. 43: Average DM query results for S?O triple patterns with a low cardinality from version 0 to all other versions.

Fig. 44: Average DM query results for ?PO triple patterns with a low cardinality from version 0 to all other versions.

Fig. 45: Average DM query results for ?PO triple patterns with a high cardinality from version 0 to all other versions.

Fig. 46: Average DM query results for SPO triple patterns from version 0 to all other versions.

Fig. 47: Average VQ query results for S?? triple patterns with a low cardinality.

Fig. 48: Average VQ query results for S?? triple patterns with a high cardinality.

Fig. 49: Average VQ query results for ?P? triple patterns with a low cardinality.

Fig. 50: Average VQ query results for ?P? triple patterns with a high cardinality.

Fig. 51: Average VQ query results for ??O triple patterns with a low cardinality.
In this appendix section, we list all measured BEAR-B-daily query evaluation durations. Each figure contains the durations for each storage approach, being OSTRICH, HDT-IC, HDT-CB, Jena-IC, Jena-CB, Jena-TB and Jena-CB/TB.

B. BEAR-B-daily Query Evaluation results

Fig. 52: Average VQ query results for ??O triple patterns with a high cardinality.

Fig. 53: Average VQ query results for SP? triple patterns with a low cardinality.

Fig. 54: Average VQ query results for SP? triple patterns with a high cardinality.

Fig. 55: Average VQ query results for S?O triple patterns with a low cardinality.

Fig. 56: Average VQ query results for ?PO triple patterns with a low cardinality.

Fig. 57: Average VQ query results for ?PO triple patterns with a high cardinality.

Fig. 58: Average VQ query results for SPO triple patterns.

Fig. 59: Average VM query results for ?P? triple patterns for all versions.

Fig. 60: Average VM query results for ?PO triple patterns for all versions.
In this appendix section, we list all measured BEAR-B-hourly query evaluation durations. Each figure contains the durations for each storage approach, being OSTRICH, HDT-IC, HDT-CB, Jena-IC, Jena-CB, Jena-TB and Jena-CB/TB.
In this appendix section, we show the different ingestion algorithms in pseudo-code.

Algorithm 3: Streaming ingestion algorithm

```java
ingestStreaming(new, store) {
    newVersion = store.getLastVersion() + 1
    additions = store.getAdditionsStream(store.getLastVersion())
    deletions = store.getDeletionsStream(store.getLastVersion())
    while (new.hasNext() || additions.hasNext() || deletions.hasNext()) {
        if (deletions.peek() < new.peek() && deletions.peek() < additions.peek()) {
            deletion = deletions.next()
            store.addDeletion(deletion.getTriple(), newVersion, deletion.isLocalChange(), calculatePositions(deletion.getTriple()))
        } else if (additions.peek() < new.peek() && additions.peek() < deletions.peek()) {
            addition = additions.next()
            store.addAddition(addition.getTriple(), newVersion, addVersion, addition.isLocalChange())
        } else if (new.peek() < additions.peek() && new.peek() < deletions.peek()) {
            element = new.next()
            if (element.isAddition()) {
                store.addAddition(element.getTriple(), newVersion, false)
            } else {
                store.addDeletion(element.getTriple(), newVersion, false, calculatePositions(element.getTriple()))
            }
        } else if (new.peek() = deletions.peek() && new.peek() < additions.peek()) {
            element = new.next()
            deletion = deletions.next()
            if (element.isAddition()) {
                store.addAddition(element.getTriple(), newVersion, true)
            } else {
                store.addDeletion(element.getTriple(), newVersion, deletion.isLocalChange(), calculatePositions(element.getTriple()))
            }
        } else if (new.peek() = additions.peek() && new.peek() < deletions.peek()) {
            element = new.next()
            addition = addition.next()
            if (element.isAddition()) {
                store.addAddition(element.getTriple(), newVersion, addition.isLocalChange())
            } else {
                store.addDeletion(element.getTriple(), newVersion, deletion.isLocalChange(), calculatePositions(deletion.getTriple()))
            }
        } else {
            addition = addition.next()
            deletion = deletions.next()
            element = new.next()
            if (addition < element) {
                if (addition.getLastVersion() > deletion.getLastVersion()) {
                    store.addAddition(addition.getTriple(), newVersion, addition.isLocalChange())
                } else {
                    store.addDeletion(deletion.getTriple(), newVersion, deletion.isLocalChange(), calculatePositions(deletion.getTriple()))
                }
            } else {
                if (addition.getLastVersion() > deletion.getLastVersion()) {
                    store.addDeletion(element.getTriple(), newVersion, addition.isLocalChange(), calculatePositions(element.getTriple()))
                } else {
                    store.addAddition(element.getTriple(), newVersion, deletion.isLocalChange())
                }
            }
        }
    }
}
```
E. Statistical Tests

This appendix section contains the p-values for the different statistical tests about our hypotheses.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Query</th>
<th>Version (p)</th>
<th>Results (p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEAR-A</td>
<td>VM</td>
<td>0.960</td>
<td>0.570</td>
</tr>
<tr>
<td>BEAR-A</td>
<td>DM</td>
<td>0.301</td>
<td>0.320</td>
</tr>
<tr>
<td>BEAR-B-daily</td>
<td>VM</td>
<td>0.694</td>
<td>0.697</td>
</tr>
<tr>
<td>BEAR-B-daily</td>
<td>DM</td>
<td>0.0391</td>
<td>2.13e-09</td>
</tr>
<tr>
<td>BEAR-B-hourly</td>
<td>VM</td>
<td>0.568319</td>
<td>0.000574</td>
</tr>
<tr>
<td>BEAR-B-hourly</td>
<td>DM</td>
<td>0.259</td>
<td>2e-16</td>
</tr>
</tbody>
</table>

Table 11: P-values for the linear regression model factors with the lookup time as response, and version and number of results as factors for each of the three benchmarks for VM and DM queries. For all cases, the version factor has no significant influence on the models, the number of results has an influence in the last three cases.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Query</th>
<th>p</th>
<th>O ≤ H</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEAR-A</td>
<td>VM</td>
<td>1.387e-05</td>
<td>✗</td>
</tr>
<tr>
<td>BEAR-A</td>
<td>DM</td>
<td>&lt; 2.2e-16</td>
<td>✗</td>
</tr>
<tr>
<td>BEAR-A</td>
<td>VQ</td>
<td>&lt; 2.2e-16</td>
<td>✗</td>
</tr>
<tr>
<td>BEAR-B-daily</td>
<td>VM</td>
<td>&lt; 2.2e-16</td>
<td>✗</td>
</tr>
<tr>
<td>BEAR-B-daily</td>
<td>DM</td>
<td>0.02863</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-B-daily</td>
<td>VQ</td>
<td>&lt; 2.2e-16</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-B-hourly</td>
<td>VM</td>
<td>&lt; 2.2e-16</td>
<td>✗</td>
</tr>
<tr>
<td>BEAR-B-hourly</td>
<td>DM</td>
<td>&lt; 2.2e-16</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-B-hourly</td>
<td>VQ</td>
<td>&lt; 2.2e-16</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 12: P-values for the two-sample t-test for testing equal means between OSTRICH and HDT-IC lookup times for VM, DM and VQ queries in BEAR-B-daily and BEAR-B-hourly. The last column indicates whether or not the actual lookup time mean of OSTRICH is less than or equal to HDT-IC.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Query</th>
<th>p</th>
<th>O ≤ H</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEAR-A</td>
<td>VM</td>
<td>1.68e-05</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-A</td>
<td>DM</td>
<td>&lt; 2.2e-16</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-A</td>
<td>VQ</td>
<td>&lt; 2.2e-16</td>
<td>✗</td>
</tr>
<tr>
<td>BEAR-B-daily</td>
<td>VM</td>
<td>&lt; 2.2e-16</td>
<td>✗</td>
</tr>
<tr>
<td>BEAR-B-daily</td>
<td>DM</td>
<td>0.02863</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-B-daily</td>
<td>VQ</td>
<td>&lt; 2.2e-16</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-B-hourly</td>
<td>VM</td>
<td>&lt; 2.2e-16</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-B-hourly</td>
<td>DM</td>
<td>&lt; 2.2e-16</td>
<td>✓</td>
</tr>
<tr>
<td>BEAR-B-hourly</td>
<td>VQ</td>
<td>&lt; 2.2e-16</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 13: P-values for the two-sample t-test for testing equal means between OSTRICH and HDT-CB lookup times for VM, DM and VQ queries in BEAR-A, BEAR-B-daily and BEAR-B-hourly. The last column indicates whether or not the actual lookup time mean of OSTRICH is less than or equal to HDT-CB.