Efficient Management of Large RDF Archives

Pelgrin, Olivier Paul

DOI (link to publication from Publisher):
10.54337/aau715850299

Publication date:
2024

Document Version
Publisher's PDF, also known as Version of record

EFFICIENT MANAGEMENT OF LARGE RDF ARCHIVES

BY
OLIVIER PAUL PELGRIN

PhD Thesis 2024

AALBORG UNIVERSITY
DENMARK

Efficient Management of Large RDF Archives

Ph.D. Dissertation

Olivier Paul Pelgrin

Dissertation submitted February, 2024

Submitted: February 2024

Main Supervisor: Professor Katja Hose
Aalborg University / TU Wien

Co-supervisor: Luis Galárraga
INRIA

Assessment: Professor Kristian Torp (chair)
Aalborg University, Denmark

Senior Researcher Fabien Gandon
Centre Inria d'Université Côte D'Azur, France

Professor Andreas Harth
Friedrich-Alexander-Universität Erlangen-Nürnberg,
FAU, Germany

PhD Series: Technical Faculty of IT and Design, Aalborg University

Department: Department of Computer Science

ISSN: 2446-1628
ISBN: 978-87-94563-27-7

Published by:
Aalborg University Open Publishing
Kroghstræde 1-3
DK – 9220 Aalborg Øst
aauopen@aau.dk

Abstract

The Semantic Web has greatly grown in popularity in recent years, partly due to the popularization of collaborative datasets as well as a multiplication of applications and use cases. Among these, the Knowledge Graph (KG) data model has seen a multiplication of use cases and applications due to its flexibility to represent the semantics of relations between diverse entities. The Resource Description Framework (RDF) has been widely adopted by the community as a way to represent knowledge graphs.

With this increasing popularity, building and maintaining RDF datasets has become more difficult for data producers. Similarly, many RDF datasets are continuously updated and changed as potential errors are corrected and new facts are added to them. For example, DBpedia sees an almost continuous stream of updates that accumulates to millions of changes for every new public release of the graph. Several applications require effective solutions to keep track of these changes, such as version control systems, historical data analytics, and knowledge graph building tools.

Traditional solutions for RDF management are insufficient for dealing with such applications, as existing methods cannot efficiently handle the increase in data to manage caused by having multiple versions. Similarly, tracking versioning data also implies novel ways to access them, including new kinds of queries that can be run over the entire history of an RDF graph. These challenges and new usages have sparked the development of new methods, algorithms, and systems.

In this thesis, we investigate the problem of managing large and evolving knowledge graphs. To understand how popular knowledge graphs evolve in practice, we propose an analysis of some of the most widely used open knowledge graphs. In addition to this analysis, we investigate current solutions for the management of evolving RDF datasets. We show that existing solutions use a variety of architecture and storage paradigms. However, the scalability of these solutions remains limited and cannot handle real-world KGs with several versions due to their size.

Subsequently, I explore how indexing techniques can be adapted to the dynamics of evolving knowledge graphs. We propose an in-memory index-

ing scheme and dictionary, inspired by the *trie* tree data structure. This indexing scheme is highly flexible and allows the versioning of RDF datasets with any other type of metadata. Our experimental results show promising performance against equivalent approaches. However, being an in-memory solution, scalability remains a challenge for large datasets that could overwhelm the main memory of a system. Similarly, the lack of on-disk storage makes data persistence impossible.

To handle larger datasets, on-disk solutions are needed. We propose a novel hybrid storage paradigm that uses multiple snapshots and delta chains to scale to much larger datasets and more versions than was previously possible. In addition, we introduce several strategies to decide when to make a new snapshot and delta chain. Using these strategies, we can optimize our storage for ingestion speed, disk usage, or query performance. Our experiments demonstrate the capability of this architecture to handle much larger datasets than existing state-of-the-art solutions. Despite that, the querying capabilities are limited and running complex queries is impossible.

Finally, we explore complex query processing on versioned RDF datasets. Existing state-of-the-art solutions either do not have such querying capabilities or are unable to handle real-world datasets. We combine our hybrid multiple delta chain architecture introduced earlier with the Comunica SPARQL query engine that we adapted for processing versioned queries. SPARQL queries are essential for complex applications relying on RDF data. Our solution combines the scalability improvements brought about by our architecture with the ability to execute complex SPARQL queries. Our evaluation shows that we can process complex queries on RDF datasets impossible before using existing solutions.

Overall, this thesis proposes a comprehensive overview of the field of versioned RDF management. The contributions from the papers included in this thesis allow the management of much larger versioned knowledge graphs than previously possible. With the implementation of full SPARQL processing over versioned RDF graphs, we tackle several blockers to the adoption of versioning for RDF datasets.

Resumé

Det Semantiske Web er blevet meget populært i de seneste år, delvist på grund af populariseringen af samarbejdsdatasæt samt en mangfoldighed af applikationer og anvendelser. Blandt disse har data modellen for Vidensgraf (VG) set en mangfoldighed af anvendelser og applikationer på grund af dens fleksibilitet til at repræsentere semantikken af relationer mellem forskelligartede enheder. Ressource Description Framework (RDF) er blevet bredt adopteret af fællesskabet som en måde at repræsentere vidensgrafer på.

Med denne stigende popularitet er opbygning og vedligeholdelse af RDF-datasæt blevet mere vanskelig for dataproducenter. På samme måde opdateres og ændres mange RDF-datasæt løbende, mens potentielle fejl rettes, og nye fakta tilføjes til dem. For eksempel oplever DBpedia en næsten kontinuerlig strøm af opdateringer, der summerer sig til millioner af ændringer for hver ny offentliggørelse af grafen. Flere applikationer kræver effektive løsninger til at holde styr på disse ændringer, såsom versionsstyringssystemer, historisk dataanalyse og værktøjer til opbygning af vidensgrafer.

Traditionelle løsninger til RDF-administration er utilstrækkelige til at håndtere sådanne applikationer, da eksisterende metoder ikke effektivt kan håndtere den stigende mængde data forårsaget af flere versioner. På samme måde indebærer sporing af versionsdata også nye måder at få adgang til dem på, herunder nye typer af forespørgsler, der kan køres over hele historien af en RDF-graf. Disse udfordringer og nye anvendelser har ført til udviklingen af nye metoder, algoritmer og systemer.

I denne PhD-afhandling undersøger vi problemet med at administrere store og udviklende vidensgrafer. For at forstå hvordan populære vidensgrafer udvikler sig i praksis, foreslår vi en analyse af nogle af de mest anvendte åbne vidensgrafer. Ud over denne analyse undersøger vi nuværende løsninger til administration af udviklende RDF-datasæt. Vi viser, at eksisterende løsninger bruger en række forskellige arkitekturer og lagringsparadigmer. Dog er skalerbarheden af disse løsninger stadig begrænset og kan ikke håndtere virkelige vidensgrafer med flere versioner på grund af deres størrelse.

Derefter undersøger jeg, hvordan indekseringsteknikker kan tilpasses dynamikken i udviklende vidensgrafer. Vi foreslår en indekseringsskema og

et ordbogsord inspireret af *trie* trædatastrukturen. Dette indekseringsskema er meget fleksibelt og tillader versionering af RDF-datasæt med enhver anden type metadata. Vores eksperimentelle resultater viser lovende ydeevne i forhold til tilsvarende tilgange. Dog er skalerbarhed stadig en udfordring for store datasæt, der kunne overvælde hovedhukommelsen i et system. På samme måde gør manglen på diskbaseret lagring dataudholdenhed umulig.

For at håndtere større datasæt er der brug for løsninger baseret på disk. Vi foreslår et nyt hybridt lagringsparadigme, der bruger flere øjebliksbilleder og delta-kæder til at skalere til meget større datasæt og flere versioner end tidligere var muligt. Derudover introducerer vi flere strategier til at beslutte, hvornår der skal oprettes et nyt øjebliksbillede og en delta-kæde. Ved at bruge disse strategier kan vi optimere vores lagring til indtags- hastighed, diskforbrug eller forespørgselsydelse. Vores eksperimenter viser, at denne arkitektur kan håndtere meget større datasæt end eksisterende state-of-the-art løsninger. Trods det er forespørgselsmulighederne begrænsede, og det er umuligt at køre komplekse forespørgsler.

Endelig udforsker vi kompleks forespørgselsbehandling på versionerede RDF-datasæt. Eksisterende state-of-the-art løsninger har enten ikke sådanne forespørgselsmuligheder eller er ude af stand til at håndtere datasæt fra den virkelige verden. Vi kombinerer vores hybridmultiple deltakædearkitektur, der blev introduceret tidligere, med Comunica SPARQL-forespørgselsmotoren, som vi har tilpasset til at behandle versionerede forespørgsler. SPARQL-forespørgsler er afgørende for komplekse applikationer, der er afhængige af RDF-data. Vores løsning kombinerer de skaleringsforbedringer, vores arkitektur bringer med sig, med evnen til at udføre komplekse SPARQL-forespørgsler. Vores evaluering viser, at vi kan behandle komplekse forespørgsler på RDF-datasæt, som var umulige før ved hjælp af eksisterende løsninger.

Alt i alt foreslår denne afhandling et omfattende overblik over feltet for versioneret RDF-administration. Bidragene fra de papirer, der er inkluderet i denne afhandling, tillader administration af meget større versionerede vidensgrafer end tidligere muligt. Med implementeringen af fuld SPARQL-behandling over versionerede RDF-grafer tackler vi flere forhindringer for vedtagelsen af versionering til RDF-datasæt.

Acknowledgments

I would like to thank the following people for their help and support during my PhD. First, I would like to express my gratitude to my supervisor Katja Hose, for giving me the opportunity to come to Denmark and start a PhD, and for her help and ideas during the PhD. I would like to extend my thanks to my co-supervisors, Luis Galárraga, for his invaluable help during the thesis and for always providing insightful feedback on my work. Both of my supervisors have been key to the completion of this thesis and I am especially grateful for the countless hours they spent discussing ideas and reviewing my work.

Secondly, I especially thank Ruben Taelman from Ghent University for accepting to collaborate on multiple papers, despite the inability to initially come to visit in Belgium because of COVID travel restrictions. This was a very pleasant and productive collaboration, thanks to Ruben's expertise and friendliness.

A significant amount of my PhD was done during the peak of the COVID pandemic, which made it difficult to work and keep motivated. I would like to express a special thanks to my family and friends, who were a tremendous help in going through those times and keeping me motivated to continue. Finally, I would like to thank my colleagues from the DKW group for being always friendly and providing a pleasant work environment.

Olivier Paul Pelgrin
Aalborg University, Thursday 29th February, 2024

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B and the Poul Due Jensen Foundation.

Contents

Thesis Details

Thesis Title:	Efficient Management of Large RDF Archives
Ph.D. Student:	Olivier Paul Pelgrin
Supervisor:	Professor Katja Hose, Aalborg University / TU Wien
Co-Supervisor:	Luis Galárraga, INRIA

The main body of this thesis consists of the following six papers.

[A] **Olivier Pelgrin**, Luis Galárraga, Katja Hose, "Towards Fully-fledged Archiving for RDF Datasets", *In the Semantic Web Journal*, vol. 12, pp. 903-925, 2021.

[B] **Olivier Pelgrin**, Luis Galárraga, Katja Hose, "TrieDF: Efficient In-memory Indexing for Metadata-augmented RDF", *Managing the Evolution and Preservation of the Data Web (MEPDaW@ISWC)*, vol. 3225, pp. 20-29, 2021.

[C] **Olivier Pelgrin**, Ruben Taelman, Luis Galárraga, Katja Hose, "Scaling Large RDF Archives To Very Long Histories", *In Proceedings of 17th IEEE International Conference on Semantic Computing (ICSC)*, pp. 41–48, 2023.

[D] **Olivier Pelgrin**, Ruben Taelman, Luis Galárraga, Katja Hose, "Expressive Querying and Scalable Management of Large RDF Archives", *Manuscript under review at the Semantic Web Journal*.

[E] **Olivier Pelgrin**, Ruben Taelman, Luis Galárraga, Katja Hose, "GLENDA: Querying RDF Archives with full SPARQL", *In Proceedings of the 20th Extended Semantic Web Conference (ESWC)*, vol. 13998, pp. 75–80, 2023.

[F] **Olivier Pelgrin**, Ruben Taelman, Luis Galárraga, Katja Hose, "The Need for Better RDF Archiving Benchmarks", *In Managing the Evolution and Preservation of the Data Web (MEPDaW@ISWC)*, vol. 3565, pp. 50–54, 2023.

This thesis has been submitted for assessment in partial fulfillment of the PhD degree. The thesis is based on the scientific papers that are listed above. Parts of the papers are used directly or indirectly in the extended summary of the thesis. As part of the assessment, co-author statements have been made available to the assessment committee and are also available at the Technical Faculty of IT and Design at Aalborg University. The permission for using the published and accepted articles in the thesis has been obtained from the corresponding publishers with the conditions that they are cited and DOI pointers and/or copyright/credits are placed prominently in the references.

Part I

Thesis Summary

Thesis Summary

1 Introduction

In this section, we introduce the context of the work presented in this thesis. We first discuss the Semantic Web concepts and technologies and then describe the novel challenges and use cases related to the versioning of semantic web data. Finally, we give a brief overview of the structure and contributions of the remaining of this thesis.

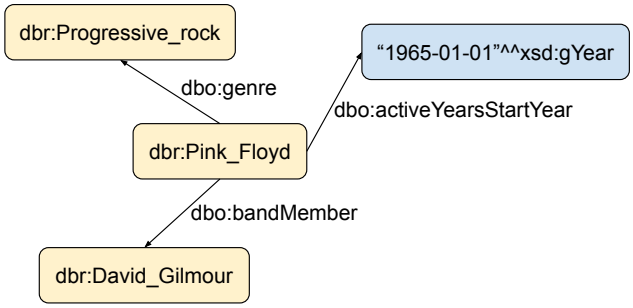
1.1 Background and Motivation

The Semantic Web [9] is based on a set of standards proposed by the World Wide Web Consortium[1] to facilitate the sharing and use of data on the Web. The Semantic Web has been designed to complement the World Wide Web (WWW) in several ways. More specifically, the Semantic Web provides ways for data processing by machines, as opposed to the WWW which focuses on delivering human-readable content in various formats such as HTML, XML, and plain text. Data accessible through the Web usually do not have clear semantics, which is not the case with the Semantic Web.

RDF and SPARQL While there is no single data format used in a Semantic Web context, the most common is the Resource Description Framework (RDF) [51], the W3C-recommended data model for the Semantic Web. RDF provides an expressive framework for representing and interlinking diverse forms of knowledge and data in a machine-readable format. At its core, RDF represents information through *subject-predicate-object* triples, which are often referred to as RDF statements or simply *triples*. Triples are ordered in collections named graphs, and as such RDF graphs are the basis for all RDF datasets on the web.

Figure 1 illustrates and presents an RDF graph along its triple representation, as represented in the DBpedia [6] knowledge graph. Entities in an

[1] http://www.w3.com/



(a) RDF Graph example. Nodes in yellow are IRIs, nodes in blue are literals.

```
<dbr:Pink_Floyd, dbo:genre, dbr:Progressive_rock> .
<dbr:Pink_Floyd, dbo:bandMember, dbr:David_Gilmour> .
<dbr:Pink_Floyd, dbo:activeYearsStartYear, “1965-01-01”^^xsd:gYear> .
```

(b) RDF Graph triples

Fig. 1: RDF Graph example showcasing information about the Pink Floyd band as represented in DBpedia [6]

RDF knowledge graph are identified by Internationalized Resource Identifiers (IRI), which are built on top of the Uniform Resource Identifier (URI) standard. For convenience, IRIs are often shortened to *namespace:ressource* pairs. In our example Figure 1, the IRI *"http://dbpedia.org/ontology/genre"* can be shorten to *"dbo:genre"*. Data and property values are represented as strings that can optionally be typed, called literals.

The W3C recommended standard for querying and managing RDF graphs is the SPARQL [55] language. SPARQL is the equivalent of SQL in the relational database world and is a specialized declarative language for handling RDF data. The most elementary query atom in SPARQL is the triple pattern, which is matched against the queried RDF graph. Triple patterns are grouped into Basic Graph Patterns or BGPs. A SPARQL query contains at least one BGP, which can be combined with other BGPs with optional language constructs, such as, for example, *"UNION"* or *"OPTIONAL"*.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT * WHERE
{
        ?band dbo:genre ?genre .
        ?band dbo:bandMember ?member .
}
```

Fig. 2: SPARQL query retrieving the name, genre, and members of bands in DBpedia.

Figure 2 shows an example SPARQL query that retrieves the name, genre, and members of bands in the DBpedia [6] knowledge graph. This query is simple and features a single BGP.

Data Maintenance and Archiving for RDF The amount of RDF data has steadily grown since the conception of the Semantic Web in 2001 [9] as more and more organizations opt for RDF [51] as the format to publish semantic data. For example, by July 2009 the Linked Open Data (LOD) cloud counted more than 90 RDF datasets, adding up to almost 6.7B triples [10]. By 2023, these numbers have increased to more than 650k datasets[2] and at least 28B triples[3]. This boom is due in part to the increasing number of data providers, but also to the constant evolution of the data in the LOD cloud.

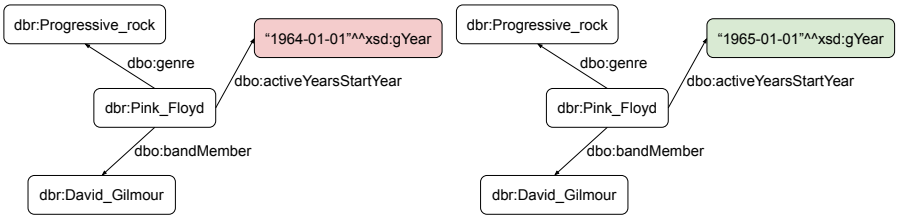
This increase in popularity, as well as the amount of data that producers need to maintain, has led to new use cases and needs. In fact, Semantic Web datasets are not static; many of them are updated regularly. Some applications, such as version control systems, collaborative knowledge graph building tools, or simply data archiving systems, require full access to the entire edition history of an RDF dataset [5, 24, 49, 54]. However, mainstream management systems for RDF datasets are usually unable to handle and store updates in a graceful way. As such, the need for better techniques and systems to handle versioned RDF datasets has emerged in recent years and has initiated research in the *RDF Archiving* field.

The example in Figure 3 displays a common case of update that occurs during the maintenance of an RDF graph. In that case, incorrect information was stored in the graph (the date is incorrect) and has to be corrected. Data maintainers will delete the wrong triple and add a new, correct, triple. This is illustrated in Figure 3c, the deleted triple is denoted by $\Delta-$ and the new added triple by $\Delta+$. The set of added and deleted triples is called a changeset. Now, data maintainers have two different versions of the graph, the old version and the new version, and are faced with the challenge of how to deal with these data. Without any other options, the choice falls between simply deleting the old version and losing the history of changes in the graph or to keep both versions of the graph stored separately. The former means that data is always lost after any change, which prevents any rollback if mistakes are made during updating, and removes the ability to access the history of the data. The latter option naturally has a significant cost in terms of storage resources needed and can be costly for data maintainers.

Many use cases are enabled by RDF archiving, such as data analytics or version control. However, these require efficient storage of the versioned RDF graphs as well as expressive querying and management systems. This

[2] `https://lod-cloud.net/`

[3] `http://lod-a-lot.lod.labs.vu.nl/`



(a) RDF graph version 1 **(b)** RDF graph version 2

<dbr:Pink_Floyd, dbo:genre, dbr:Progressive_rock> .
<dbr:Pink_Floyd, dbo:bandMember, dbr:David_Gilmour> .

Δ-
<dbr:Pink_Floyd, dbo:activeYearsStartYear, "1964-01-01"^^xsd:gYear> .
Δ+
<dbr:Pink_Floyd, dbo:activeYearsStartYear, "1965-01-01"^^xsd:gYear> .

(c) RDF graph triple update

Fig. 3: RDF graph example with an update to one triple to correct wrong information.

makes the design of RDF archiving systems particularly difficult: static RDF datasets can already scale to very large sizes, with multiple billions of triples, which is further accentuated by archiving, which multiply the amount of data along the time axis. This requires RDF archiving systems to carefully consider their storage architecture and also their querying algorithms. Both go hand in hand and impact each other and need to be carefully considered in the development of solutions for RDF archiving.

This thesis first offers a comprehensive examination of the state-of-the-art in RDF archiving. This is used to draw the design of new data structures, indexing schemes, and algorithms, to deal with the challenge of archiving large RDF graphs, while supporting expressive querying capabilities.

1.2 Thesis Structure

This thesis is structured as follows. Part I motivates the thesis and provides a summary of the contributions of the included papers. In Section 2, we investigate the current state of the RDF archiving literature. We provide a comprehensive overview of existing RDF archiving systems and evaluate their functionality and availability. Moreover, we present a framework for analyzing the evolution of RDF datasets over time through a set of metrics and use it to analyze several major publicly available RDF datasets. This analysis is used to draw design lessons for RDF archiving systems, which are paramount in the following of this thesis. These contributions are presented in more detail in Paper A. In Section 3, we present an in-memory indexing scheme inspired

by Tries, designed for RDF data with additional metadata, such as versioning and provenance. This proposed data structure and the corresponding algorithms offer a very flexible solution with an attractive trade-off in memory usage and query performance. This system is described in full detail in Paper B. Our analysis of state-of-the-art systems showed the significant challenges that remain for handling RDF archives with a long revision history. We address this issue by proposing in Section 4 a hybrid storage paradigm that combines multiple delta chains with metadata compression. This permits the management of large RDF archives with long revision histories at scales that are not achievable by existing state-of-the-art systems. The hybrid storage architecture is introduced in Paper C while the improvements to metadata compression can be found in Paper D. In order to support more advanced applications, a system requires expressive querying capabilities. In Section 5, we describe our proposed implementation of versioned SPARQL query processing on top of the hybrid storage architecture discussed previously. We evaluated our solution on the BEAR-C benchmark [19], a first at the time of writing and to the best of our knowledge. Details of this contribution can be found in Paper D. To illustrate these querying capabilities, in Section 6 we show a demonstration system that offers a user-friendly way to express and execute versioned queries over an RDF archive. This system is presented in further detail as part of Paper E. In Section 7, we discuss existing difficulties in evaluating RDF archiving systems with standard, community-accepted, benchmarks and propose directions for future work in that area. This is discussed in more details in Paper F. Finally, Section 8 summarizes the content of this thesis and discusses possible future work avenues.

Part II, proposes a complete reproduction of the six papers that compose this thesis, only modified in their layout to fit the format of the thesis. While each paper is self-contained, it is recommended to read them sequentially, as some contributions build on previous ones.

2 Archiving for RDF Datasets

This section gives an overview of Paper A [39] and reuses its content.

2.1 Motivation and Problem Statement

As mentioned in Section 1.1, the growth and wide availability of RDF datasets on the Web led to the emergence of new challenges. The storage and querying of the entire revision history of RDF datasets is prominent among them and is called *RDF archiving* in the literature.

RDF archives demonstrate multifaceted utility within collaborative projects, serving as a back-end for version control in collaborative environ-

ments [3, 5, 21, 24, 32, 49, 54]. They also facilitate data evolution analysis for providers [19], offering an avenue for error tracking and debugging methodologies. Furthermore, RDF archives find application within the domain of RDF streaming applications relying on structured historical data [13, 27]. Moreover, the utility of these archives extends to consumer applications such as data analytics, for example, the discernment of correction patterns [45, 46], and historical trend analysis [28].

Consequently, a substantial body of literature has emerged that addresses the challenges associated with RDF archiving [39, 49]. The existing landscape of research endeavors encompasses a spectrum of solutions aimed at the storage and querying of RDF archives [2, 4, 5, 14, 22, 24, 36, 50, 54, 58, 63], as well as benchmarking frameworks [19, 31, 37] to facilitate the evaluation of these engines, and an array of temporal extensions for SPARQL, each tailored to address specific use cases [8, 20, 23, 48].

Despite the multitude of existing efforts, a comprehensive and fully developed solution for the effective management of large-scale and dynamic RDF datasets remains absent. This gap in the current landscape can be attributed to several contributory factors, including the inherent performance and functionality limitations within RDF engines pertaining to metadata handling and a lack of consideration for the evolution patterns observed within real-world RDF data.

As such, Paper A investigates those points by conducting an in-depth survey of the prevailing state of the art, highlighting the limitations and intricacies of existing approaches. Similarly, a comprehensive framework designed to analyze the evolution patterns of RDF data is proposed. This framework is applied in a study of three prominent large and dynamically evolving RDF datasets, namely DBpedia [6], YAGO [57], and Wikidata [17].

2.2 Preliminaries

In this section, we introduce the concepts and notation for RDF archiving and querying, which are used throughout the remainder of the thesis. This section is adapted from [39, 44].

RDF Graphs

An RDF graph, denoted G, is defined as a set of triples $t = \langle s, p, o \rangle$, with $s \in \mathcal{I} \cup \mathcal{B}$, $p \in \mathcal{I}$, and $o \in \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$ [51]. $\mathcal{I}$ represents the set of Internationalized Resource Identifiers (IRI), $\mathcal{L}$ the set of literals (strings, numbers, dates, ...), and $\mathcal{B}$ the set of blank nodes (anonymous entities). The concept of graphs for RDF comes from the fact that G can be modeled as a labeled directed graph. As such, s and o represent the nodes and p the edge that connects them. A

$\langle s, p, o\rangle$, $\langle s, p, o, \rho\rangle$	triple and 4-tuple: subject, predicate, object, graph revision
G	an RDF graph
g	a graph label
G_i	the i-th version or revision of the graph G
$A = \{G_0, G_1, \dots\}$	an RDF graph archive
$u = \{u^+, u^-\}$	an update or changeset with sets of added and deleted triples.
$u_{i,j} = \{u^+_{i,j}, u^-_{i,j}\}$	the changeset between the graph revisions i and j ($j > i$)
$rv(\rho)$	revision number of the graph revision ρ
$ts(\rho)$	commit time of graph revision ρ
$l(\rho)$, $l(G)$	labels of graph revision ρ and graph G

Table 1: RDF Graphs notations, from [39]

can graph G can be associated with a label $g \in \mathcal{I} \cup \mathcal{B}$ and become a *named graph*. Table 1 summarizes the notations related to *RDF graphs*.

RDF Graphs Archives

G_0	u_1	$G_1 = u_1(G_0)$
$\langle$:*USA*, *a*, :*Country*$\rangle$	$u_1^+ = \{\langle$:*France*, *a*, :*Country*$\rangle\}$	$\langle$:*USA*, *a*, :*Country*$\rangle$
$\langle$:*Cuba*, *a*, :*Country*$\rangle$	$u_1^- = \{\langle$:*USA*, :*dr*, :*Cuba*$\rangle\}\}$	$\langle$:*Cuba*, *a*, :*Country*$\rangle$
$\langle$:*USA*, :*dr*, :*Cuba*$\rangle$		$\langle$:*France*, : *a*, :*Country*$\rangle$

Fig. 4: Two revisions G_0, G_1 and a changeset u_1 of an RDF graph archive A. Taken from [39].

We now define an *RDF graph archive*, A, as the temporally ordered collection of states that an *RDF graph* has had since its creation. Each of the states can be viewed as an independent graph and consequently, $A = \{G_s, G_{s+i}, \dots, G_{s+n-1}\}$, with G_s the graph representing the state at the revision (or version) $s \in \mathcal{N}$. We define G_s with $s = 0$ as the first revision of the graph. Subsequent revisions G_i of the graph, with $i > s$, can be obtained from their previous state G_{i-1}, by applying an update (or changeset) $u_i = \langle u_i^+, u_i^-\rangle$. u_i^+ correspond to the set of triples added between G_{i-1} and G_i, and u_i^- corresponds to the set of deleted triples. As such, we can formulate $G_i = u_i(G_{i-1}) = (G_{i-1} \cup u_i^+) \setminus u_i^-$. The notion of changeset can be further

generalized to any pair of revisions i, j with $i < j$ and denote $u_{i,j} = \langle u^+_{i,j}, u^-_{i,j} \rangle$ the changeset between G_i and G_j.

A graph archive can also be described as a set of 4-tuples (or quads) $\langle s, p, o, \rho \rangle$, where $\rho \in \mathcal{I}$ is the revision identifier $i = rv(\rho)$. $rv \subset \mathcal{I} \times \mathcal{N}$ is a function that assigns a revision identifier ρ to its corresponding natural number identifier. We further define the function $ts \subset \mathcal{I} \times \mathcal{N}$ that assigns a revision identifier ρ to its commit time or timestamp. This is particularly useful for enabling time-travel queries or for timestamp-based approaches to indexing, which we will describe in more detail in Section 2.3.

We illustrate an example graph archive in Figure 4, representing information about countries and their diplomatic relationships (*:dr*).

RDF Dataset Archives

We now define the concept of the *RDF dataset* and summarize its corresponding notation in Table 2. An RDF dataset is defined as a collection of named graphs $D = \{G^0, G^1, \ldots, G^m\}$ with G^k the k-th graph in the dataset. Each graph $G^k \in D$ is associated with a label $l(G^k) = g^k \in \mathcal{I} \cup \mathcal{B}$ with the exception of G^0, the *default graph* [51].

Similarly to RDF graph archives, we define a *RDF dataset archive* as $\mathcal{A} = \{D_0, D_1, \ldots, D_{l-1}\}$, a temporally ordered set of RDF datasets. Like graph archives, it is possible to obtain the state D_j of the dataset archive by applying a *dataset update*, defined as $U_j = \{\hat{u}_j, u^0_j, \ldots u^m_j\}$, to the revision D_{j-1}. A dataset update consists of a set of updates, one for each graph in the dataset, as well as a *graph changeset* $\hat{u}_j = \langle \hat{u}^+_j, \hat{u}^-_j \rangle$. A graph changeset stores the added and deleted graphs (identified by their label) between revision $j-1$ and j. Consequently, any graph marked for deletion (in the set $\hat{u}^-_j$) cannot have a set of changes $u^k_j \in U_j$. Like for graph archives, the concept of dataset update can be further generalized to any pair of revisions i, j with $j > i$, which we denote $U_{i,j}$. We can further define $\zeta \in \mathcal{I}$ as the global revision identifier $j = rv(\zeta)$ of the RDF dataset archive. Therefore, the function $ts(\zeta)$ returns the commit timestamp of the revision ζ.

Archives Queries

Querying an RDF graph or dataset archive is different from standard querying in that results from different versions can be combined to form an answer. Several works in the literature have proposed to categorize the types of queries performed against RDF archives into several categories [19, 37]. Fernández et al. [19] identify five different types of queries, which we illustrate based on our previous RDF archive example in Figure 4:

$\langle s, p, o, æ, ι \rangle$	a 5-tuple subject, predicate, object, graph revision, and dataset revision
$D = \{G^0, G^1, \dots\}$	an RDF dataset
$\mathcal{A} = \{D_0, D_1, \dots\}$	an RDF dataset archive
D_j	the j-th version or revision of the dataset D
G_i^k	the i-th revision of the k-th graph in a dataset archive
$\hat{u} = \{\hat{u}^+, \hat{u}^-\}$	a graph changeset with sets of added and deleted graphs
$U = \{\hat{u}, u^0, \dots\}$	a dataset update or changeset consisting of a graph changeset $\hat{u}$ and changesets u^i associated with graphs G^i
U^+, U^-	the addition/deletion changes of U: $U^+ = \{\hat{u}^+, u^{0+}, u^{1+}, \dots\}$, $U^- = \{\hat{u}^-, u^{0-}, u^{1-}, \dots\}$
$U_{i,j}$	the dataset changeset between dataset revisions i and j ($j > i$)
$rv(\zeta)$	revision number of the dataset revision ζ
$ts(\zeta)$	commit time of dataset revision ζ
$\Upsilon(\cdot)$	the set of terms (IRIs, literals, and blank nodes) present in a graph G, dataset D, changeset u, and dataset changeset U.

Table 2: RDF Datasets notations, from [39]

- **Version Materialization (VM).** VM queries are standard queries which target a single revision, such as *"what was the list of countries present at revisions i?"*
- **Delta Materialization (DM).** DM queries are standard queries targeting a changeset $u_{i,j}$, e.g., *"which countries were added between revision i and j?"*
- **Version (V or VQ).** V queries are standard queries where results are annotated with revision validity. The V query *"what is the list of countries?"* would return the list of countries, each annotated with the revisions where they are part of the graph.
- **Cross-version (CV).** CV queries combine (e.g., via joins, unions, aggregations, differences, etc.) the information from multiple revisions, e.g., *"which of the current countries has diplomatic relationships with countries present at revision i?"*
- **Cross-delta (CD).** CD queries result from the combination of the results of multiple changesets, e.g., *"what are the revisions j with the largest number of country additions?"*

In practice, CV and CD queries can be implemented as a combination of VM, DM, or V queries, and as such, the minimal set of query types needed to support the full expressiveness consists of VM, DM, and V queries. Throughout the remainder of this thesis, we adopt the categorization of Fernández et al. [19], as it has seen greater adoption by the community than the other proposed categorizations [39, 58].

2.3 RDF Archiving Systems

Numerous systems have been developed to facilitate the storage and querying of RDF datasets and graph archives. However, most existing systems are designed to support the archiving of individual RDF graphs, with only a limited number of approaches [4, 5, 24, 63] designed to accommodate full dataset archives. For example, the OSTRICH system handles quads of the form $\langle s, p, o, rv(\rho)\rangle$. On the contrary, certain alternatives opt instead for the representation of temporal metadata, such as insertion and deletion timestamps, alongside validity timestamps for triples, by using the ρ-component with $\rho \in \mathcal{I}$. In practice and in the context of this work, both approaches can be considered equivalent in terms of functionalities.

In the remainder of this section, we will define a categorization of state-of-the-art approaches along several criteria.

- **Storage paradigm.** The storage paradigm is paramount to the architecture, performance, and functionalities of a system. In practice we can identify three main paradigms, independent copies (IC), change-based (CB), and timestamp-based (TB), with some modern systems using a combination of those. We discuss each of the paradigms in detail later

	Paradigm	Queries	BGPs	Multi-graph	Source
Dydra [4]	TB	all	+	+	-
Ostrich [58]	IC/CB/TB	VM, DM, V	-[c]	-	+
QuitStore [5]	FB	all	+	+	+
RDF-TX [22]	TB	all	+	-	-
R43ples [24]	CB	all	+	+	+[a]
R&WBase [54]	CB	all	+	-	+
RBDMS [29]	CB	all	+	-	-
SemVersion [63]	IC	VM, DM	-	-	-
Stardog [2]	CB	all	+	+	-
v-RDFCSA [14]	TB	VM, DM, V	-	-	-
x-RDF-3X [36]	TB	VM, V	+	-	+[b]

[a] It needs modifications to have the console client running and working [b] Old source code [c] Full BGP support is possible via integration with the Comunica query engine

Table 3: Functional-based categorization of existing RDF archiving systems, adapted from [39]

in this section.

- **Query types.** The types of archive queries supported natively by the system (see Section 2.2).
- **Full BGPs.** If the system supports complex queries, with more than one triple pattern.
- **Multi-graph.** Whether the system supports the archiving of multiple graphs, i.e. if it supports RDF dataset archives.
- **Source available.** Finally, we also indicate whether the system is available and open source.

A more complete list of criteria can be found in Paper A, this summary limits itself to the most important ones for the remainder of the thesis.

We now detail each storage paradigm in detail and discuss their trade-off in terms of querying performance and disk usage.

Independent Copies Systems (IC)

Independent Copies systems store each revision of a dataset archive D_i (or graph) as a fully materialized independent dataset. This approach has a significant cost in terms of disk usage, since redundancy is maximized, and is only adapted to the versioning of small and simple datasets [63]. However, IC approaches are particularly effective at answering VM and CV queries, since neither will require any materialization cost before querying. The most prominent system that implements an IC storage paradigm is SemVersion [63], which offers functionalities similar to version control systems such as CVS or SVN. All in all, IC approaches suffer from scalability issues, and research has since moved to more efficient paradigms.

Timestamp-based Systems (TB)

Timestamp-based solutions involve the storage of triples alongside associated temporal metadata, such as temporal validity intervals or timestamps denoting their insertion and deletion. VM and DM queries need a potentially expensive materialization step in order to be executed, whereas V queries are usually more straightforward. The effectiveness of the materialization step depends on the specific indexing strategies used within the system. TB systems are typically more storage efficient than IC approaches, and as such, have seen a lot more development.

x-RDF-3X [36] is an extension of the RDF-3X [35] engine which represents quads in the form $\langle s, p, o, \rho \rangle$ where ρ represents all the revisions where the triple exists, as well as the addition and deletion timestamps. x-RDF-3X, like RDF-3X, features a full SPARQL query engine.

Dydra [4] is a TB system that supports the archiving of dataset archives. Its data model consists of 5-tuples $\langle s, p, o, \rho, \zeta \rangle$, and consequently does not support a global revision identifier. The ρ component is mapped to addition and deletion timestamps. Dydra supports full SPARQL queries and extends the language with a *REVISION* clause which can be used to reference the desired revision.

v-RDFCSA [14] proposes a TB storage scheme based on compact suffix-array (CSA) [12] which supports graph archives. It offers unparalleled efficiency in terms of storage and querying at the expense of update support. As such, it is adapted for use on existing, static, graph archives. Contrary to Dydra and x-RDF-3X, v-RDFCSA only supports single triple pattern queries for the VM, DM, and V types.

Change-based Systems (CB)

Changed-based systems focus on minimizing redundancies by storing the initial revision of a dataset D as a full snapshot. Subsequent revisions D_j ($s < j$) are stored as deltas or changesets U_j. This sequence of deltas after the initial snapshot is called a *delta chain*. CB systems usually propose great storage efficiency, as long as the deltas do not become larger than the fully materialized dataset, and can be convenient for DM and CD queries.

R&WBase [54] is one of the oldest CB-based archiving systems for RDF graphs. Each update u_i is stored in two new named graphs G_g^{i+}, G_g^{i-} for the additions and deletions. R&WBase [54] does not support the versioning of multiple named graphs.

R43ples [24] draws inspiration from R&WBase and extends its functionalities to the versioning of named graphs. R43ples does not support global revision identifiers, and each graph in the dataset is versioned independently. R43ples offers full support for SPARQL and proposes the use of the *REVI-*

SION clause to extend the language.

Hybrid Systems

Finally, some systems chose to implement a storage strategy that takes aspects from several of the aforementioned paradigms.

OSTRICH [58] proposes a storage paradigm that combines the characteristics of CB and TB approaches. In practice, OSTRICH stores the initial revision of a graph as a fully materialized snapshot via HDT [18]. Subsequent revisions i are stored as *aggregated deltas* $u_{0,i}$ in B+Trees, together with TB-based versioning metadata. OSTRICH querying capabilities are limited to single-triple-pattern queries of the DM, VM, or V kind. Overall, OSTRICH shows through its experiments a good space and querying efficiency compared to other alternatives.

QuitStore [5] proposes a Git-like version control system for RDF datasets and makes use of PROV-O to model its metadata. In practice, QuitStore always materializes the latest revision in an in-memory quad store, based on the Python library RDFlib[4]. The data is otherwise stored on disk as N-triples files in a Git-versioned directory. QuitStore has been designed for collaborative dataset construction projects, and the need to materialize the latest revision in-memory makes it unsuited for the archiving of large RDF datasets.

2.4 Evaluation of the Existing Archiving Systems

After having presented the existing systems for RDF archiving in Section 2.3, we evaluated the testable systems on several RDF graph archives.

Datasets

We chose several publicly available knowledge graphs, YAGO [57], DBpedia [6] and Wikidata [17], for our evaluation. Each of these knowledge graphs has seen several versions since their initial publication, and we consider each as a separate revision in a graph archive. Due to the large size of these datasets, we limit our evaluation data to some selected *"themes"* and versions. For DBpedia, we include the *mapping-based objects* and *mapping-based literals*, as well as the *instance-types* and the *ontology*. For YAGO, we use the *facts*, *meta facts*, *literal facts*, *date facts*, and *labels* themes. Finally, for Wikidata, we chose *simple-statements* of the RDF Exports [1] for the period 2014-05 - 2016-08. Table 4 summarizes the versions used for each dataset.

[4] https://rdflib.dev/

Revision	DBpedia	YAGO	Wikidata
0	3.5	2s	2014-05-26
1	3.5.1	3.0.0	2014-08-04
2	3.6	3.0.1	2014-11-10
3	3.7	3.0.2	2015-02-23
4	3.8	3.1	2015-06-01
5	3.9		2015-08-17
6	2015-04		2015-10-12
7	2015-10		2015-12-28
8	2016-04		2016-03-28
9	2016-10		2016-06-21
10	2019-08		2016-08-01

Table 4: Datasets revision mapping, reproduced from [39]

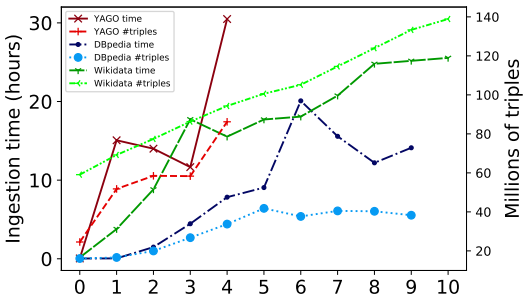


Fig. 5: OSTRICH's ingestion time on the evaluation datasets, reproduced from [39]

Tested Systems

As shown in Table 3, existing RDF archiving systems vary significantly in functionalities, but are all capable of versioning single graph archives. However, only five systems have their source code available to the public, which limits possible evaluation. Among these systems, R43ples [24] was unable to ingest any of the datasets after more than four days of execution. R&WBase [54] does not offer a bulk ingestion process, necessary to ingest the large changesets of our evaluation datasets. Similarly, x-RDF-3X [36] could not ingest any of the DBpedia changesets and needs modifications to its source code to function on modern systems. QuitStore [5], was unable to ingest our tested datasets in either its *persistence* or *lazy loading* mode due to crashes. Of all the available systems, only OSTRICH [58] was able to successfully ingest our evaluation datasets.

Evaluation Results

Because OSTRICH was the only system capable of ingesting our evaluation datasets, its the only system included on our evaluation. Figure 5 shows the ingestion time of OSTRICH for each of the evaluation datasets. The time needed to ingest each revision follows an upward tendency for each dataset with variations related to the size of the changesets. In general, the more revisions there are, the longer it takes for the ingestion process. This can be explained by the increase in redundancy within the deltas, since OSTRICH stores the version j as an aggregated delta $u_{0,j}$. As such the aggregated deltas can only increase in size, and become more expansive to construct due to the need to consider all previous changes. As a consequence, OSTRICH would have difficulties scaling to datasets with long revision histories.

2.5 RDF Data Evolution Analysis

We now describe the analysis of the evolution of several RDF datasets. This analysis will help establish the requirements for RDF archiving systems by taking into account the evolution patterns of real-world datasets. The datasets considered for this analysis are described in Section 2.4.

Metrics for RDF Archives Analysis

To describe the evolution of RDF archive datasets or graphs, we propose the use of a set of metrics to quantify the changes between two revisions. We divide those metrics into two categories: low-level change metrics and high-level change metrics. In this analysis, we will focus on the application of these metrics to RDF graph archives. A complete formalization for both graph and dataset archives can be found in Paper A. Finally, a tool has been developed to compute these metrics for any RDF archive.

Low-level changes metrics The low-level change metrics focus on the addition and deletion of triples as well as vocabulary elements. The vocabulary, $\mathrm{Y} \subset \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$, is the set of terms present in the triples of the graph or dataset. Those metrics have been adapted from the state-of-the-art, notably the work of Fernández et al. [19], and apply to arbitrary pairs of revisions i and j.

Change-ratio. The change-ratio represents the ratio of changes between two revisions against the joined size of those revisions, and is defined as:

$$\delta_{i,j}(G) = \frac{|u^{+}_{i,j}| + |u^{-}_{i,j}|}{|G_i \cup G_j|}. \tag{1}$$

Vocabulary dynamicity. The vocabulary dynamicity describes how much the vocabulary set of graph evolves between two versions and is defined as:

$$\text{vdyn}_{i,j}(G) = \frac{|\text{Y}(u_{i,j})|}{|\text{Y}(G_i) \cup \text{Y}(G_j)|} \tag{2}$$

Growth ratio. The growth ratio defines the pure change in size of a graph archive between two revisions and is defined as:

$$\Gamma_{i,j}(G) = \frac{|G_j|}{|G_i|} \tag{3}$$

High-level changes metrics Contrary to low-level metrics, high-level metrics aim to examine the semantics of the changes made to an RDF archive. The notion of a high-level change can be application-specific, as shown by [38, 52], however, here we focus on domain independent metrics, which can be applied to any RDF archive.
Entity changes. This metrics describe the changes in RDF entities (s in $\langle s, p, o\rangle$) between two revisions i and j. We define the *entity change* metric as:

$$ec_{i,j}(G) = |\sigma_{i,j}(G)| = |\sigma^{+}_{i,j}(G) \cup \sigma^{-}_{i,j}(G)| \tag{4}$$

With $\sigma^{+}_{i,j}(G)$ the set of added entities, and $\sigma^{-}_{i,j}(G)$ the set of deleted entities.
Triple-to-entity-change. The triple-to-entity-change score constitute the average number of triples being part of an *entity change*. We define it as:

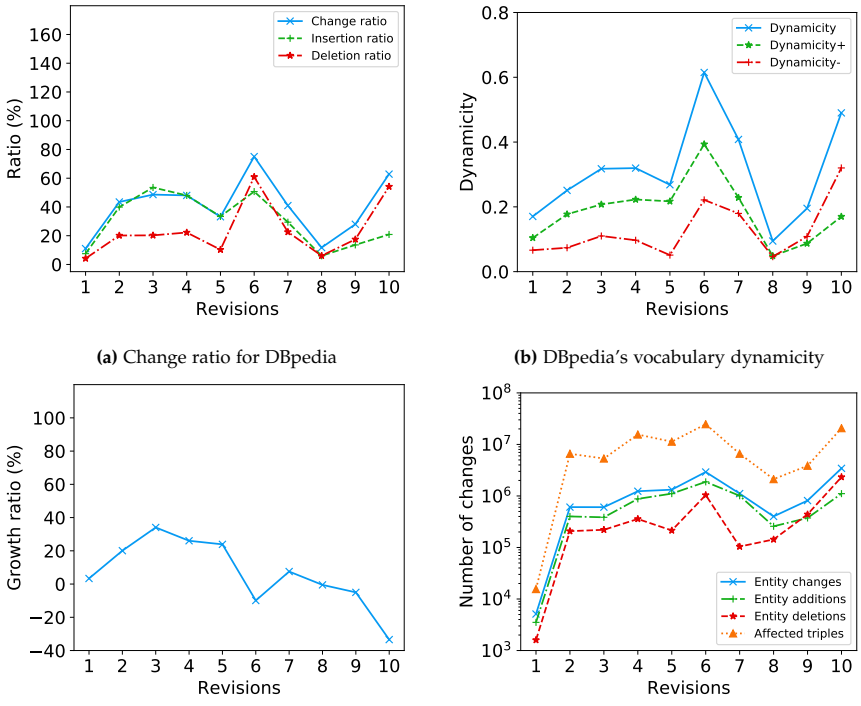
$$ect_{i,j}(G) = \frac{|\langle s, p, o\rangle \in u_{i,j} : s \in \sigma_{i,j}(G)|}{ec_{i,j}(G)} \tag{5}$$

Object Updates. This metric aims at representing identifying changes to a triple's object. In that case, a triple $\langle s, p, o\rangle$ is deleted and a triple $\langle s, p, o'\rangle$ added, with $o \neq o'$. Those changes can usually be interpreted as a correction to the data of the RDF graph.
Orphan Object Additions/Deletions. Any triple $\langle s, p, o\rangle \in u_{i,j}$ that are not part of any other *high-level* change are considered *orphan* additions and deletions.

Analysis of an RDF Graph Archive

We now analyze the evolution of the DBpedia RDF graph archive with the metrics described above. Paper A contains a more detailed analysis that includes Wikidata and YAGO, as described in Section 2.4.

Figure 6a shows the change-ratio of DBpedia over its revision history. The change-ratio is stable until revision 6 (release 2015-04) where a significant spike of changes occurs. A similar high number of changes can be observed



(a) Change ratio for DBpedia

(b) DBpedia's vocabulary dynamicity

(c) Growth ratio for DBpedia

(d) Entity changes in DBpedia

Fig. 6: Evolution of DBpedia, reproduced from [39]

for revision 10 (2019-08). This behavior can be found in the vocabulary dynamicity (Figure 6b) with distinctly higher values for revisions 6 and 10. For entity-changes, revisions 6 and 10 feature a notably high number of entity deletions, hinting at major refactoring work done by the DBpedia data maintainers for those revisions. All in all, we observe that DBpedia showcases a release pattern with several minor revisions and periodic major revisions (6 and 10). This confirms the possibility of unpredictable variability in the number of changes occurring throughout the revision history of an RDF archive. As such, any system designed to handle RDF archiving should be able to handle possibly highly variable number of changes between revisions.

2.6 Conclusion

In this section, we have discussed the significance of RDF archiving for both data maintainers and consumers. Our comprehensive survey of existing solutions and benchmarks in the domain of RDF archiving reveals a limited availability of downloadable and usable solutions. Among these, OSTRICH

emerges as the sole system capable of effectively storing the release history of notably large RDF datasets. However, it is imperative to note that, despite its capabilities, OSTRICH's design is not entirely up to the scaling requirements posed by extended revision histories. Alternative available solutions, such as R43ples [24], R&WBase [54], Quit Store [5], and x-RDF-3X [36], face significant challenges in managing extended revision histories. These challenges are primarily attributed to their orientation toward collaborative version control, thereby imposing scalability limitations.

In addition, we have introduced a series of metrics designed to analyze the evolution of RDF archives. The application of these metrics has been instrumental in conducting an analysis of the historical evolution of three RDF datasets, specifically DBpedia, YAGO, and Wikidata. Through this analysis, noticeable changes within their release histories have been identified, characteristic of major/minor revision patterns. These insights can be used to optimize resources in the archiving process. For example, the identification of large changesets could prompt the creation of a new snapshot. It should be noted that, within existing RDF archiving solutions, none currently leverages these evolution patterns. This provides a promising avenue for new archiving techniques capable of adapting to the dynamic nature of RDF archives.

3 Flexible In-memory Indexing for Metadata Augmented RDF Data

This section gives an overview of Paper B [40] and reuses its content.

3.1 Motivation and Problem Statement

As discussed previously in Section 2, RDF usages in recent years have been transformed from simple graphs to more advanced applications involving RDF datasets with multiple versions and graphs. The difficulty in handling such use cases comes from the lack of freely available solutions to index and query such RDF datasets.

In Section 2.3, we have discussed the current state-of-the-art of RDF archiving systems. Existing available solutions have several limitations, which make them suboptimal for modern applications involving the versioning of multigraph RDF datasets. This notably includes a lack of concurrent support for both versioning and named-graphs, to the lack of scalability for larger datasets. Moreover, other applications of RDF may require additional types of extensions to the triple model to accommodate additional metadata, such as provenance annotations.

Paper B investigates solutions for these problems by introducing an in-memory indexing scheme for arbitrary tuples, i.e. extended triples objects.

The proposed indexing scheme is inspired by the *trie* data structure, and we evaluate it on RDF datasets with a focus on versioning and provenance annotations. The contributions of Paper B are summarized in this Section. First, we introduce the concept of *metadata augmented triples* in Section 3.2, thereafter, in Section 3.3, we present our in-memory indexing scheme for such triples. Finally, we will discuss the evaluation experiments in Section 3.4 and conclude in Section 3.5.

3.2 Metadata Augmented RDF Triples

An RDF graph G is defined as a set of triples $\langle s, p, o \rangle$ with $s \in \mathcal{I} \cup \mathcal{B}$, $p \in \mathcal{I}$, and $o \in \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$. See Section 2.2 for a more extensive overview of the standard notation used throughout this thesis. We further define a metadata-augmented RDF graph as a set of triples with a k-tuple of additional RDF terms. In other words, a metadata-augmented RDF graph is a set of n-tuples $q = \langle$ `s`, `p`, `o`, ... $\rangle$ with $n = k + 3$. As an example, a versioned RDF quad, $\langle s, p, o, \rho \rangle$, is an n-tuple with $n = 4$ and the fourth component, ρ, a graph revision identifier.

To represent such metadata augmented with traditional RDF stores, one will usually resort to *reification*. Reification consists in encoding n-ary statements as a set of several binary statements. For example, a versioned triple $\langle$ `:Copenhagen`, `:capital`, `:Denmark`, `2` $\rangle$ would be instead identified through an IRI or blank node u. From there, statements about u can be expressed via four standards triples: $\langle$ u, `:subject`, `:Copenhagen` $\rangle$, $\langle$ u, `:predicate`, `:capital` $\rangle$, $\langle$ u, `:object`, `:Denmark` $\rangle$, and $\langle$ u, `:version`, `2` $\rangle$. Reification has a notorious cost for both storage usage and querying performance due to the extensive increase in statements needed to represent the same amount of information but has the benefit of working natively with any RDF triple store.

The limitations of reification have sparked the development of RDF-star [26], which, at the time of writing, is considered for integration into the RDF 1.2 standard[5]. RDF-star proposes nested triple statements, such as $\langle \langle$ `:Copenhagen, :capital, :Denmark` $\rangle$, `:version`, `2` $\rangle$. Support for RDF-star has increased in recent years among RDF systems; however, few approaches document their indexing scheme in detail [61], and many have limitations on the number of levels of nesting supported.

3.3 In-memory Indexing and Dictionary Encoding

We propose *TrieDF*, an in-memory indexing scheme for metadata-augmented RDF triples. In practice, TrieDF stores *RDF tuples* of arbitrary length $n > 3$.

[5] `https://www.w3.org/TR/rdf12-concepts/`

TrieDF takes inspiration from Tries – compact prefix trees for string representation – and seeks to employ similar techniques for the benefit of RDF. We approach RDF tuples as strings of items, where the items are RDF terms ($Y \subset \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$) instead of characters. In order to improve the compactness of the representation, we employ a similar approach to dictionary encoding, i.e. the substitution of strings by integer identifiers. An overview of the TrieDF index and dictionary can be found in Figure 7.

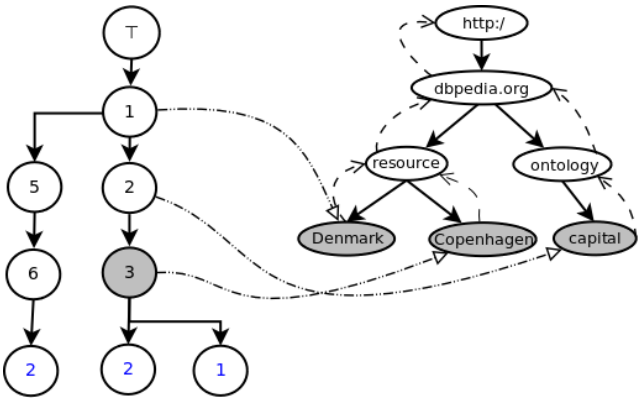


Fig. 7: TrieDF index and dictionary, reproduced from [40]

Trie-based Indexes

TrieDF represents RDF tuples in several trie-indexes. Consider the example in Figure 7, where versioned triples (quads) in the form $\langle$ s, p, o, v $\rangle$ are stored in an SPOV index. Each element of a tuple is associated with a node in the trie index. Tuples share nodes in the tree when they share a common prefix. For example, the tuples in Figure 7 all share node #1, since they all start with the same IRI (http://dbpedia.org/resource/Denmark).

Trie-based Dictionaries

The mapping of RDF terms to integers is handled by the *dictionary*. In TrieDF, in the same spirit as [7], we employ a trie-based approach for string representation. The inherent redundancy of RDF IRI make them ideal candidates for such representation. Indeed, IRI often share prefixes which end up duplicated in a traditional dictionary. Instead of using a node for each character, as done in traditional tries, we coalesce IRI fragments between the character "/" into single nodes as shown in Figure 7. Furthermore, the trie-based dictionary is made bidirectional in TrieDF to permit both IRI-to-integer lookups as well as integer-to-IRI *"reverse"* lookups in a single data structure.

3.4 Evaluation

TrieDF has been evaluated on the data loading time and the retrieval time of tuples that match a prefix. Three different use cases are used for the evaluation of these metrics: standard RDF triples, versioned RDF triples (quads), and versioned RDF triples with provenance annotation (5-tuples). We compare TrieDF with comparable in-memory approaches to tuple indexing. We select Jena and RDFlib for triples and quads. Both are industry standard systems to handle RDF data, and are used extensively by the community. For 5-tuples, due to the absence of RDF specific solutions, we resort to a relational database, SQLite, which offers efficient in-memory storage. Results from Paper B [40] are summarized in this Section. Additional analysis and discussion can be found in the original paper.

Datasets

We chose DBpedia 2016-10 [6], YAGO 3.1 [57], YAGO 4 [62], and Wikidata [17] for our standard triple experiments. For the quad experiments, we select a DBpedia archive composed of versions 3.5 to 2016-10, as well as the BEAR-B Hourly and BEAR-C datasets from the BEAR [19] benchmark. Finally, our 5-tuple evaluation is done with the NELL [33] dataset, which features provenance and versioning information. The full details of the datasets and their characteristics can be found in Paper B.

Loading Time

Table 5 shows the loading times of the different systems for triples, quads, and 5-tuples. We note that, for standard triples, Jena is very fast to load data compared to TrieDF and RDFlib. This can be explained by the mature batch-loading system employed by Jena, as opposed to TrieDF, where triples are currently loaded one by one into the indexes.

For quads, Jena is not able to load the DBpedia archive due to memory constraints. Both RDFlib and TrieDF can load all datasets, with TrieDF generally outperforming the other solutions, except for BEAR-B.

Finally, in the 5-tuple experiment, SQLite is the fastest approach when not using indexes. However, a more fair comparison is with indexes where SQLite loads the data slower than TrieDF.

Retrieval Time

Figure 8 shows the retrieval time of triple, quad, and 5-tuple patterns in the evaluation datasets. A full description on the choice and generation of query patterns can be found in the original paper, Paper B [40]. Overall, TrieDF displays a notably faster retrieval times than alternative approaches. This

	DBpedia	YAGO 3.1	YAGO 4	Wikidata
Jena	**587.14**	**1281.65**	**289.42**	**1665.48**
RDFLib	2816.16	7102.30	1626.85	9587.51
TrieDF	727.20	2105.37	358.20	2800.77

(a) Loading time of the triples evaluation in seconds.

	DBpedia	BEAR-B	BEAR-C
Jena	-	**1094.83**	418.74
RDFLib	26433.76	4851.09	1663.40
TrieDF	**16074.17**	3743.55	**387.68**

(b) Loading time of the quads evaluation in seconds.

	NELL
TrieDF	36.55
SQLite	**16.98**
SQLite w. indexes	47.27

(c) Loading time of the 5-tuple evaluation in seconds.

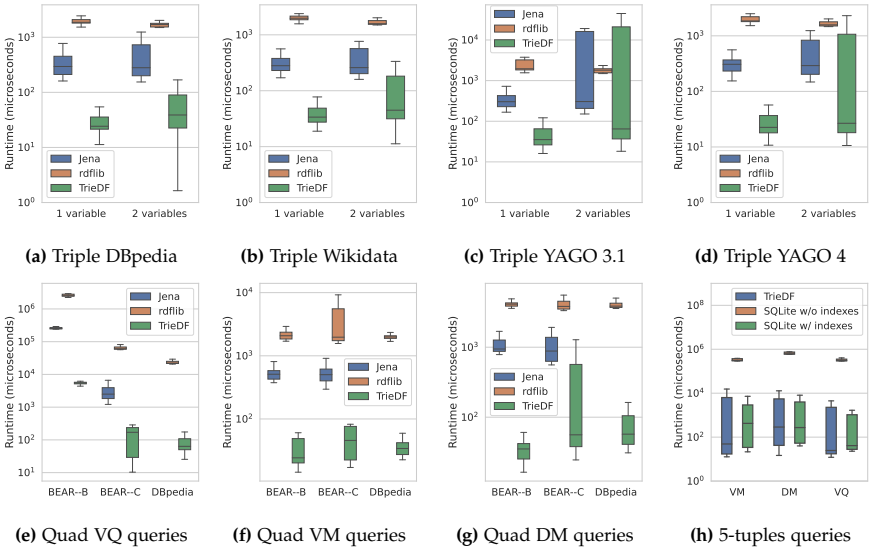
Table 5: Loading time for triples, quads, and 5-tuples, reproduced from [40].

is particularly noticeable on quad pattern queries, where all alternatives are at least one order of magnitude slower. For 5-tuple pattern queries, both TrieDF and SQLite with indexes are competitive with each other, with TrieDF generally having lower median runtime but more variability.

3.5 Discussion and Future Work

We have proposed in Paper B an in-memory approach for the indexing of metadata-annotated RDF triples. This approach, inspired by the trie-tree data structure, offers a flexible indexing scheme for arbitrary-sized tuples. The experimental evaluation shows that this approach can outperform competing approaches in retrieval time for tuple pattern queries, while keeping ingestion times reasonable. With the popularization of *quoted triples* with RDF-star (and RDF 1.2) enabling more use cases for extended RDF tuples, TrieDF could be used as baseline for the design of more general indexing scheme RDF data with support for various levels of metadata.

Some of the drawbacks of TrieDF include its current limitation to in-memory indexing, which is limiting when needing to handle larger datasets. Furthermore, retrieval capabilities are currently limited to simple tuple patterns queries. Although such queries can serve as the building blocks of more complex queries, for example, in SPARQL or SPARQL-star [26], it can be insufficient for some real-world usages. Finally, applications such as versioning



(a) Triple DBpedia **(b)** Triple Wikidata **(c)** Triple YAGO 3.1 **(d)** Triple YAGO 4

(e) Quad VQ queries **(f)** Quad VM queries **(g)** Quad DM queries **(h)** 5-tuples queries

Fig. 8: Retrieval time in microseconds for triples pattern queries and different types of quad and 5-tuple pattern queries (log scale), reproduced from [40].

have specific requirements as well as specific data semantics (see Section 2.2 for a detailed description) that are not easily captured by a general approach.

However, the versatility of the TrieDF approach is useful for general types and possibly heterogeneous metadata. In the future, the implementation of an on-disk representation would open the door for the support of larger datasets, as well as reducing the burden on the system memory. Support for full SPARQL and especially SPARQL-star would permit more complex real-world applications.

4 Scaling Large RDF Archives to Very Long Histories

This section gives an overview of our hybrid multiple delta chain architecture for on-disk RDF archives indexing. This is based on the work presented in Paper C [44]. In this section, we also describe some contributions from Paper D [41] on metadata compression, which are direct extensions of Paper C.

4.1 Motivation and Background

As discussed in Section 2, a large number of solutions for the storage and querying of RDF archives has been proposed throughout the years. However,

only a very limited number of them are available, most of them being either closed source or with unpublished sources. This greatly limits the practical use of RDF archiving solutions. Among the available solutions, scalability remains a major challenge. The experimental results presented in Section 2.4 show that only OSTRICH [58] is capable of ingesting medium-sized real-world RDF graph archives. Despite that, OSTRICH ingestion times increase indefinitely with the number of versions in the archive, making longer histories and larger graph archives impractical in practice.

In Paper B [40] (summarized in Section 3), we have proposed an in-memory indexing scheme capable of tackling the challenges of indexing RDF graph archives. Its flexibility allows for efficient handling of RDF archives, including datasets archives, with a good performance level. However, this solution is limited to in-memory indexing and is therefore unsuitable for larger archives that would exceed the memory capacity of a system. Moreover, lack of on-disk storage makes persistence impossible. As a consequence, there is a need for further advancements in on-disk solutions for RDF archiving.

In Paper C [44], we propose a new hybrid on-disk storage architecture, based on the CB/TB aggregated changeset architecture proposed by OSTRICH, to scale to much larger and longer histories for RDF graph archives. Furthermore, we propose in Paper D a new representation aimed at compressing the versioning metadata of the indexes by reducing the amount of redundancies. This improves both scalability in terms of the ingestion time of new data and disk usage, while having no negative impact on query performance. Both contributions from Paper C and Paper D allow handling large RDF archives at a scale that was impossible before [39].

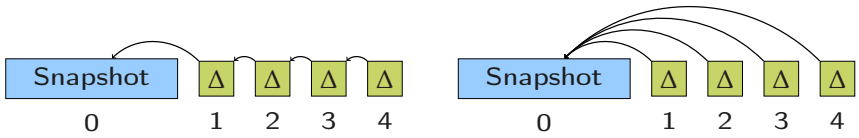
4.2 OSTRICH's Hybrid Architecture

In this section, we elaborate on our base storage architecture, which is based on the hybrid change-based(CB)/timestamp-based(TB) storage paradigm proposed by OSTRICH. We chose it as our base, as OSTRICH is the only available system capable of ingesting moderately sized RDF graph archives [39]. Hybrid storage paradigms have been introduced by the most modern RDF archiving systems to combine the strengths of different approaches and achieve better efficiency [39, 58].

Aggregated Delta Chain

Approaches using a CB storage paradigm store the change history of an RDF archive in a *delta chain*. A delta chain consists of an initial *fully materialized* snapshot of the first revision of the archive, while subsequent revisions are stored as deltas $u_{i,j}$ with $i = j - 1$. More details can be found in Section 2.3. Unlike standard CB systems, OSTRICH instead opts for *aggregated deltas*,

where revisions $i > 0$ are stored as deltas $u_{0,i}$ in the delta chain. Figure 9 illustrates the concepts of delta chain and aggregated delta chain.



(a) Delta chain **(b)** Aggregated delta chain

Fig. 9: Delta chain architectures, reproduced from [44]

The use of aggregated deltas aims to address some weaknesses of standard delta chains at the cost of increasing redundancy. In standard delta chains, since each delta is relative to the previous one, some queries can become increasingly expensive to run as the delta chain grows. This is particularly problematic for version-materialization (VM) queries, which require materialization of the desired data through the full iteration of the delta chain up to the target revision. As such, aggregated deltas do not suffer from the same drawback, as only a single delta is required to materialize any revision.

Versioning Metadata Representation

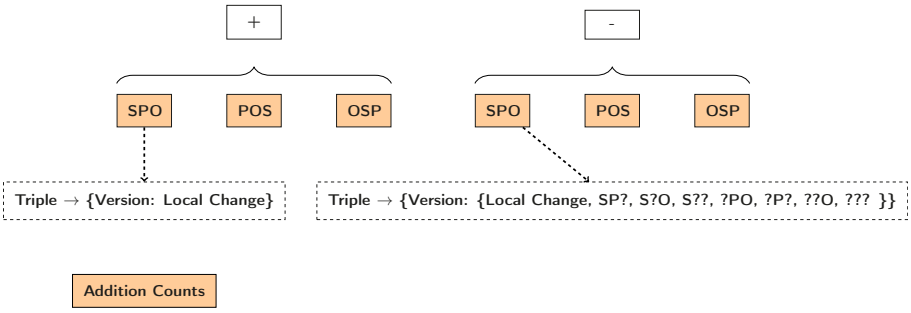


Fig. 10: OSTRICH delta chain storage overview, reproduced from [41]

Figure 10 illustrates how delta chains are structured in OSTRICH. At its core, triples are divided into two separate triple stores, depending on whether they have been added ($t \in u^{+}_{i,j}$) or deleted ($t \in u^{-}_{i,j}$) in a delta. Each triple store consists of three different indexes, in different orders: *SPO*, *POS*, and *OSP*. OSTRICH also combines aggregated deltas with aspects of the TB storage paradigm. In fact, triples within OSTRICH delta chains are also annotated with versioning metadata. The purpose of these metadata is to reduce redundancies, by storing each triple only once, and to improve querying runtimes. Versioning metadata differs between the additions and deletions triples, as

seen on Figure 10. Both additions and deletions are annotated with a collection of mappings of versions to a local-change flag. This contains the list of versions (i.e., aggregated delta) where the triple has been added (respectively, deleted) with respect to revision 0. The local-change flag indicates whether the triple reverts a previous change in the delta chain, e.g. a triple deleted and then added again in a later revision. Since the deltas are aggregated, this information can be duplicated between revisions: A triple added in an aggregated delta for revision i is also added in a revision $i+1$ if it has not been deleted instead. This creates redundancies within OSTRICH's indexes that can grow overtime, especially for long delta chains.

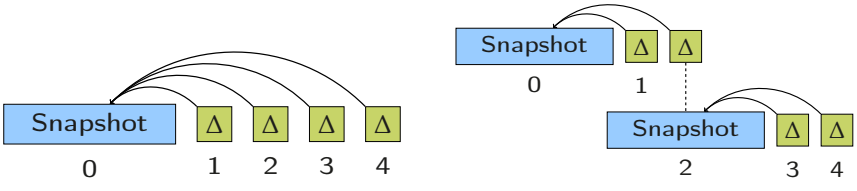
Furthermore, deletion triples are mapped to an additional vector of relative positions for each possible triple pattern order. This is used during querying for fast offset computations and count estimations for deletions. For additions, OSTRICH maintains a separate count index for each possible triple patterns. More details about OSTRICH can be found in the original paper [58].

It should be noted that aggregated deltas only grow with the number of revisions in the delta chain. Their ever increasing size and redundancy make the ingestion of long histories and large datasets prohibitive, as shown in Section 2.4.

4.3 Scaling to Long Histories

Multiple Aggregated Delta Chains

We propose in Paper C to solve the increasing cost of aggregated deltas by creating a new snapshot of the data that will serve as the starting point for a new delta chain. We illustrate this multi-snapshot approach in Figure 11. The main idea is that when the ingestion of new data becomes too prohibitive in a single delta chain, a fresh snapshot will reduce the ingestion time of subsequent revisions.



(a) Aggregated delta chain

(b) Multiple aggregated delta chain

Fig. 11: Aggregated delta chains and multiple aggregated delta chains, reproduced from [44]

The use of multiple snapshots may have an impact on disk usage, as multiple full snapshots of the data are now created. However, in cases

where inner delta chain redundancy is high, this impact can instead be beneficial. Querying will also be affected. Indeed, any version (V) or delta-materialization (DM) query, as described in Section 2.2, would now need to be evaluated across multiple delta chains. In practice, both query performance and disk usage depend on the data itself, especially its size in number of triples and revisions. This will be discussed in more detail in Section 4.5, during the analysis of the experimental results. As seen in Figure 11, in our proposed architecture, revisions stored as full snapshots are also stored as an aggregated delta in the previous delta chain. While this increases redundancy, it allows for some optimizations for DM queries.

When Should a Snapshot be Created?

With the multiple snapshot architecture discussed previously, the decision to materialize a snapshot at a specific point will impact both the ingestion time, disk usage, and query performance. This will be subject to a trade-off between all these aspects and, as such, we propose multiple *strategies* to tackle this problem. More formally, we define a *snapshot oracle* $f : \mathbb{A} \times \mathbb{U} \rightarrow \{0, 1\}$, with $\mathcal{A} \in \mathbb{A}$ an RDF archive with k revisions. The ingestion of a changeset $u_{k-1,k} \in \mathbb{U}$ triggers the decision of whether k should be materialized as a snapshot or only stored as an aggregated delta.

Paper B proposes several possible strategies for implementing the snapshot oracle, which are briefly described here.

- **Baseline.** The baseline strategy never triggers the creation of a new snapshot. This is tantamount to the behavior of OSTRICH [58].
- **Periodic.** The periodic strategy triggers the creation of a new snapshot when a fixed number d of revisions has been ingested since the last snapshot.
- **Change-ratio.** The change-ratio strategy takes into account the change dynamics of the data, which is quantified with the *change-ratio* metric [19] (see Section 2.5). The change-ratio, $\delta_{i,j}(\mathcal{A})$, is calculated between a pair of revisions i and j. In an aggregated delta chain, we estimate the overall level of changes by summing the change-ratio since the last snapshot s. As such, the strategy consists in materializing a new snapshot when $\sum_{i=s+1}^{k} \delta_{s,i} \geq \gamma$, where γ is a fixed predefined threshold.
- **Time.** The time strategy aims to bound the amount of time it takes to ingest a new revision. With t_k the time taken to ingest revision k, and $s + 1$ the first revision following the last snapshot s, the strategy materializes a new snapshot when $\frac{t_k}{t_{s+1}} > \theta$, with θ being a user-defined threshold.

Versioning Metadata Compression

As discussed previously in Section 4.2, OSTRICH adds versioning metadata to triples in aggregated deltas to prevent them from being stored multiple times, as well as optimizing some querying processes. The drawback of this representation is that it creates, in turn, metadata-level redundancies as metadata entries can be duplicated across multiple versions. This is illustrated in Tables 6a and 6c. We can observe that redundant information is stored both in the metadata associated with additions and in the metadata associated with deletions. In this section, we describe our solution to this problem, as proposed in Paper D [41].

Version	2	3	4	6
LC	T	T	T	T

(a) Original addition metadata in OSTRICH

Version	[2,4)	-	-	[5,∞)
LC	[2,∞)	-	-	-

(b) Compressed addition metadata

Version	2	3	4	6
LC	F	F	F	T
SP?	0	0	0	0
S?O	0	0	0	0
S??	4	6	6	0
?PO	0	0	0	1
?P?	6	8	8	0
??O	0	0	0	0
???	8	8	8	0

(c) Original deletion metadata in OSTRICH

Version	[2,5)	-	-	[6,∞)
LC	-	-	-	[6,∞)
SP?	0	-	-	-
S?O	0	-	-	-
S??	4	+2	-	-6
?PO	0	-	-	+1
?P?	6	+2	-	-8
??O	0	-	-	-
???	8	-	-	-8

(d) Compressed deletion metadata

Table 6: Representation of the versioning metadata in OSTRICH and compressed in our implementation. *LC* denotes the local change flag. Reproduced from [41].

Building and storing this metadata can become increasingly expensive for long delta chains – the metadata size grows linearly with the number of revisions – and is one of the scalability limitations of this storage architecture. Paper D proposes to replace this metadata representation by a new compressed representation, greatly reducing redundancies. This new representation is illustrated in Tables 6b and 6d. In this new representation, version numbers and local change flags are instead stored as numerical intervals, where their value does not change. As such, in the example in Table 6b, the local change flag is represented as an [2,∞) interval which indicate that the value is *true* from revision 2 onwards. Deletion metadata also contains a relative position vector, as described in Section 4.2. This data is now replaced by *delta compressed* vectors. The first vector in the metadata is stored plainly and the subsequent vectors are stored as relative deltas, as illustrated in Table 6d. With this representation, only the changes are actually stored in the

metadata, leading to great reductions in size, especially for long delta chains.

4.4 Querying a Multiple Delta Chain Architecture

The introduction of multiple delta chains creates new challenges for querying which need to be addressed by new algorithms. In this section, we detail our *multiple delta chains* querying algorithms, as proposed in Paper C [44]. For the sake of brevity, we assume that each algorithm has access to routines to query individual delta chains. The full details of the single delta chain query routines are available in Paper D [41] and are based on the work of Taelman et al. [58].

Version Materialization Queries

Algorithm 1 VM query algorithm, reproduced from [41, 44]

```
1:  function queryVM(i, p)                          ▷ version i, triple pattern p
2:      sid_i ← snapshot(i)
3:      q_i ← query(sid_i, p)        ▷ we query the triple pattern on the snapshot
4:      if sid_i = i then            ▷ the target version correspond to a snapshot
5:          return q_i
6:      u⁺ ← getAdditions(i, p)
7:      u⁻ ← getDeletions(i, p)
8:      vm_i ← q_i \ u⁻                          ▷ filter out the deleted triples
9:      vm_i ← vm_i ∪ u⁺                            ▷ add the added triples
10:     return vm_i
```

Version Materialization (VM) queries target a specific revision of the archive, and we detail the procedure to execute them in Algorithm 1. This algorithm is similar in both a single-delta chain and a multiple-delta chain contexts. This works by identifying the snapshot corresponding to the target version (line 2). In a single delta chain case, this snapshot is always 0, i.e. the first revision. From there, if the target revision is the snapshot (line 4), then it is sufficient to return the triples that match the query pattern in the snapshot. Otherwise, the triples matching the query pattern that were deleted w.r.t. the snapshot at the target revision are removed from the result set (line 8), while added triples are included in the results (line 9).

Delta Materialization Queries

Delta Materialization (DM) queries provide results for the delta between two revisions. DM queries in a multiple delta chain context are much more complicated. The procedure to execute DM queries for two revisions i and j on

Algorithm 2 DM query algorithm, reproduced from [41, 44]

```
1:  function queryDM(i, j, p)                          ▷ versions i, j, triple pattern p
2:      sid_i ← snapshot(i)
3:      sid_j ← snapshot(j)
4:      if sid_i = sid_j then                          ▷ i and j are in the same delta-chain
5:          delta ← singleDCQueryDM(i, j, p)
6:      else                                           ▷ i and j are not in the same delta-chain
7:          u_{si,sj} ← snapshotDiff(sid_i, sid_j, p)
8:          u_{si,i}, u_{sj,j} ← ∅
9:          if i ≠ sid_i then                          ▷ test if version i is a delta
10:             u_{si,i} ← singleDCQueryDM(sid_i, i, p)
11:         if j ≠ sid_j then                          ▷ test if version j is a delta
12:             u_{sj,j} ← singleDCQueryDM(sid_j, j, p)
13:         u_{i,sj} ← mergeBackwards(u_{si,i}, u_{si,sj})
14:         (u_i, u_j) ← mergeForward(u_{i,sj}, u_{sj,j})
15:     return u_i, u_j
```

multiple delta chains is described in Algorithm 2. The DM query algorithm for multiple delta chains relies on two external routines: The first, called *singleDCQueryDM*, executes a DM query on a single delta chain. The second, *snapshotDiff*, computes the difference between two different snapshots, i.e. the set of added and deleted triples matching the given query pattern.

The first step in the algorithm consists of finding the respective snapshots of the target revisions i and j (lines 2 and 3). If the snapshots are the same (line 4), i and j belong to the same delta chain and the query boils down to a single delta chain DM query (line 5). Otherwise, this means that the target revisions are on different delta chains and the difference between their reference snapshots is computed in line 7. Then, if either i or j do not correspond to a snapshot, the changesets between them and their reference snapshot are computed on lines 10 and 12. Finally, in lines 13 and 14, the intermediary results are merged in order to form the final result set. This merging procedure consists in two steps, *mergeBackwards* and *mergeForward*, which filter reverted and contradicting changes in the intermediary results. These are explained in more detail in Paper C.

Version Queries

Version Queries annotate results with the revision identifiers where they hold. We detail the querying procedure for V queries in a multiple delta chain context in Algorithm 3. This algorithm relies on the *singleDCQueryV* function to execute V queries on single delta chains. This works by iterating

Algorithm 3 V query algorithm, reproduced from [41, 44]

```
1: function queryV(p)                          ▷ p a triple pattern
2:     r ← ∅
3:     for c ∈ C do                            ▷ C the list of delta chains
4:         v ← singleDCQueryV(c, p)
5:         r ← merge(r, v)                     ▷ merge intermediate results
6:     return r
```

over all delta chains present in the archive (line 3) and executing *singleDCQueryV*. The obtained results are then merged with the existing results from the previous loop iterations (line 5). When all iterations are completed, r contains the final set of results.

4.5 Experimental Evaluation

In this Section, we summarize the experimental evaluation of our proposed architecture. The full evaluation of all the proposed strategies for snapshot materialization can be found in Paper C, while a complete analysis of the impact of the compressed metadata representation can be found in Paper D.

Experimental Setup

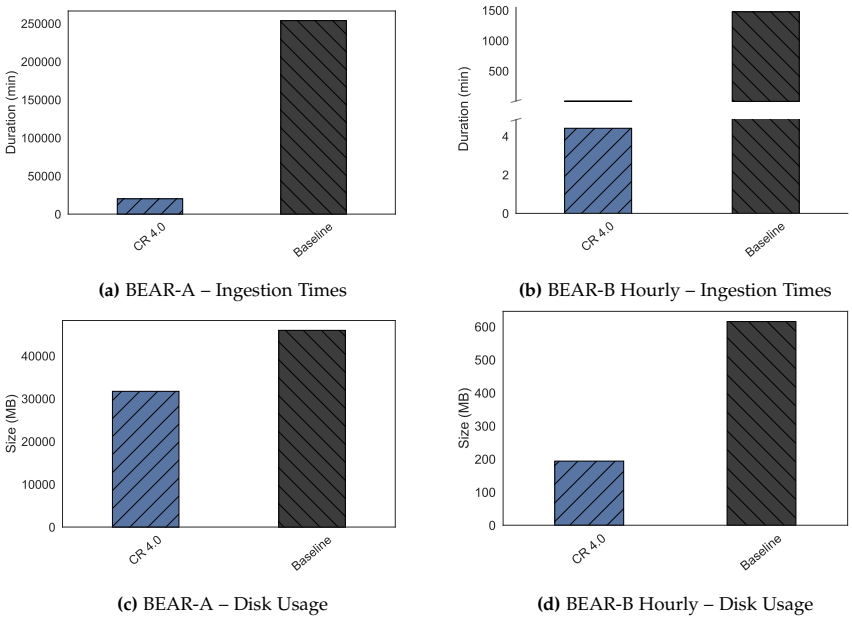
Experimental evaluation is carried out on the BEAR RDF archiving benchmark [19]. BEAR comes in three different variants: BEAR-A, BEAR-B, and BEAR-C. BEAR-A constitutes a large archive of 58 revisions containing up to 66M triples. BEAR-B is proposed in three different variants: Daily, Hourly, and Instant. Each variant has versions containing up to 44K triples. BEAR-B Daily proposes 89 revisions, BEAR-B Hourly 1299 revisions, and BEAR-B Instant proposes 21046 revisions. Both BEAR-A and BEAR-B only propose single triple pattern queries. In this summary of the experimental evaluation, the results for BEAR-A and BEAR-B Hourly are included due to their very different scale and challenges. The results of all other BEAR benchmark datasets are available in Paper C. The characteristics of the datasets used in this section are summarized in Table 7.

In this evaluation, two different strategies are included. First, the *Baseline* strategy, which correspond to OSTRICH [58], the current state-of-the-art for RDF archiving. Second, to represent the multiple delta chain architecture, the *Change-ratio (CR)* strategy with a threshold value $\gamma = 4.0$ is chosen due to its overall good performance in the experiments of Paper C. The CR results include the new compressed metadata representation described in Section 4.3.

	BEAR-A	BEAR-B Hourly
# versions	58	1299
$\lvert G_i \rvert$'s range	30M - 66M	33K - 44K
$\overline{\lvert \Delta \rvert}$	22M	198
# queries	368	62 (49 ?P? and 13 ?PO)

Table 7: Dataset characteristics. With $\lvert G_i \rvert$ the size of the individual revisions, and $\overline{\lvert \Delta \rvert}$ the average size of the changesets $u_{k-1,k}$. Adapted from [44].

Ingestion Time and Disk Usage



(a) BEAR-A – Ingestion Times

(b) BEAR-B Hourly – Ingestion Times

(c) BEAR-A – Disk Usage

(d) BEAR-B Hourly – Disk Usage

Fig. 12: Ingestion time and disk usage for BEAR-A and BEAR-B Hourly, using our different strategies.

Figure 12 displays the time needed and the disk used for ingestion of the datasets. First, the time required to ingest the datasets has been drastically reduced between the baseline and the multiple delta chain architecture. For BEAR-A (Figure 12a), the ingestion time goes from 253676 minutes to 20053, an order of magnitude speed-up. For BEAR-B Hourly (Figure 12b), the time required to ingest the data by the baseline is 1473 minutes, while the multiple delta chain architecture with the change-ratio strategy takes only 4.41 minutes. This is a speed-up by a factor of 334. This confirms that the usage of multiple delta chains is hugely beneficial for the scalability of the system.

Disk usage is also significantly improved by the CR strategy, although to

a lesser extent. BEAR-A (Figure 12c) requires 45.9 GB using the baseline, while the multiple delta chain CR strategy uses 31.7 GB. For BEAR-B Hourly (Figure 12d), disk usage goes from 615 MB with the baseline to only 193 MB with the multiple delta chain alternative. These improvements can be attributed in part by the use of multiple delta chains, which limit the size reached by the aggregated deltas, and thus limit redundancies, but more importantly by the use of compressed versioning metadata.

Querying Performance

The querying experiments are run for every revision present in the benchmark datasets: VM queries are run for each revision $i \in \mathcal{A}$, while DM queries are run on the changesets $u_{0,i}$ and $u_{1,i}$ for each revision $i \in \mathcal{A}$. Each query is run 5 times, and the runtimes averaged.

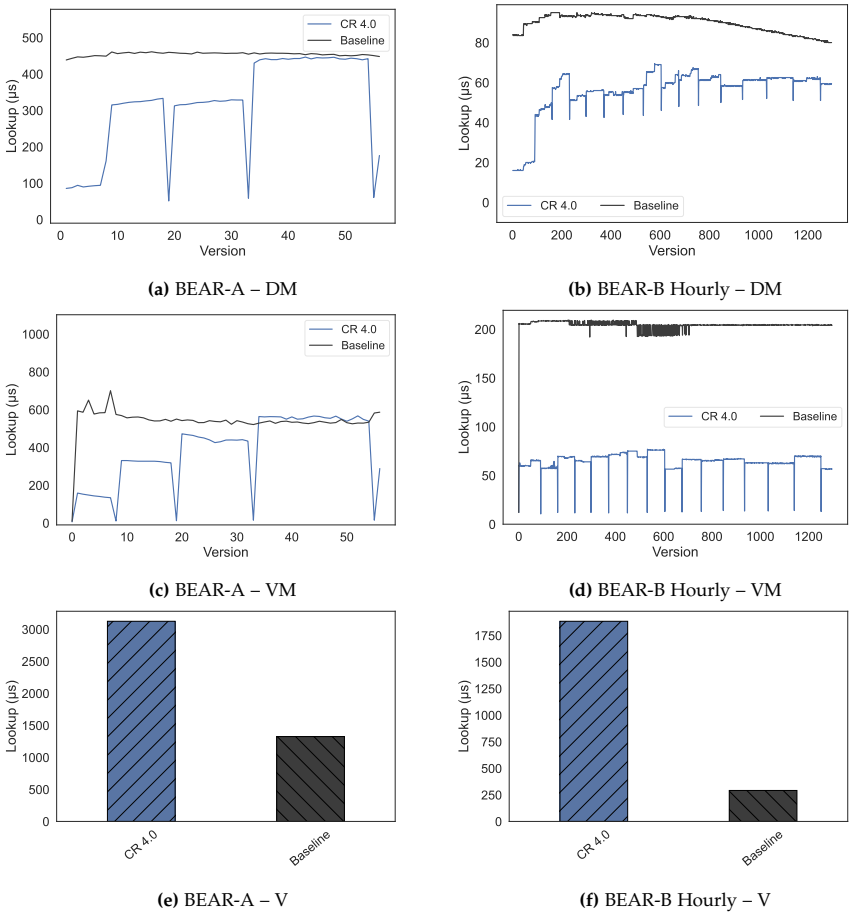
In Figure 13, the query runtime for DM, VM, and V queries on the BEAR-A and BEAR-B hourly data set are displayed. The runtime of DM queries for the multiple delta chain strategy is always better than the single delta chain baseline. When using multiple delta chains, the runtime of DM queries is reduced when the target revision i corresponds to a snapshot, as evidenced by the downward spikes in Figures 13a and 13b. Overall, the short delta chains and smaller aggregated deltas of the multiple delta chain approach show clear benefits in querying performance on the two datasets, despite their large differences in size and number of revisions.

For VM queries, the performance of the multiple delta chain approach is also overall better for both datasets, with the exception of the last 20 revisions of BEAR-A where the runtime is slightly worse than the baseline (Figure 13c). Like for DM queries, runtime is significantly improved when the query directly targets a snapshot.

Finally, V queries is where the multiple delta chain approach loses to the baseline. As described in Algorithm 3 (Section 4.4), V queries need to be evaluated independently on all delta chains before obtaining the final results. This is clearly detrimental to performance because the cost of the query increases linearly with the number of delta chains.

4.6 Discussion and Future Work

In the section, we described our hybrid multiple snapshot architecture based on aggregated changesets to provide indexing for RDF graph archives. This architecture permits to overcome the limitations of previous single delta chains approach, notably for the time needed to ingest new data. The creation of new snapshots, indicating the start of a new delta chain, is guided by *snapshot creation strategies*. We propose multiple implementations of strategies



(a) BEAR-A – DM

(b) BEAR-B Hourly – DM

(c) BEAR-A – VM

(d) BEAR-B Hourly – VM

(e) BEAR-A – V

(f) BEAR-B Hourly – V

Fig. 13: Querying runtime of DM, VM, and V queries for the BEAR-A and BEAR-B Hourly dataset.

adapted to different scenarios. The evaluation shows notable gains in the ingestion times of RDF archives with up to an order of magnitude faster times. This is especially notable for large RDF archives with long revision histories. Querying performance on multiple delta chains is comparable to or better than with a single delta chain, except for V queries, which are slower overall. More details can be found in Paper C, where these contributions were first introduced.

In addition to the multiple delta chain architecture, we introduce a new versioning metadata representation to replace the one used by OSTRICH [58] and Paper C. The new metadata representation offers the same expressivity and features necessary for aggregated delta chains, but with compression

and redundancy reductions. The results in greatly improved ingestion times, without notable drawbacks to query performance. This new compressed representation for versioning metadata is presented in more details in its original paper, Paper D.

Overall, the contributions of Paper C, along with the improvements proposed in Paper D, allows the archiving of much larger RDF archives than before with state-of-the-art archiving systems. However, querying is still limited to single triple pattern queries, which is not sufficient for many real-world applications, limiting the wider adoptions of archiving technologies by the RDF community.

5 SPARQL Processing over RDF Archives

In this section, we provide an overview of our solution for full SPARQL processing over RDF archives. The content of this section is based on Paper D [41] and reuses its content.

5.1 Motivation

Previously, we have described the challenges related to handling large RDF archives. In Section 2 (detailed in Paper A [39]), we have reviewed state-of-the-art systems for RDF archiving and concluded that, except for OSTRICH [58], no system was able to deal with real-world RDF archives. Even then, scalability remained a significant challenge. As such, in Section 4 (detailed in Paper C [44]) we have proposed a hybrid multiple delta chain architecture, built on top of OSTRICH, capable of significantly improving scalability for larger RDF archives. However, this system is still limited to single triple pattern queries, which is insufficient for many real-world applications. Most uses of RDF data require full SPARQL processing, usually via a SPARQL endpoint. A look at other available solutions for RDF archiving shows that among the openly available systems, several of them offer some kind of support for complex querying [5, 24, 36, 54]. However, their lack of scalability, discussed in Section 2 (Paper A), makes them difficult to use with real-world datasets and use cases.

In this section, we discuss our solution to the problem of SPARQL processing over RDF archives. First, in Section 5.2, we describe the current solutions for expressing versioned SPARQL queries. Subsequently, in Section 5.3, we describe our solution for SPARQL processing RDF archives. This includes our representation of versioned queries, as well as our concrete implementation. In Section 5.4, we evaluate our implementation on the BEAR-C benchmark for RDF archives. And finally, in Section 5.5, we will conclude with a discussion on our current implementation and future work. Additional details

about these contributions can be found in the original paper, Paper D.

5.2 Related Work

Multiple efforts have been made to express complex queries with versioning or temporal information. Many of such approaches rely on ad hoc extensions of the SPARQL language to support temporal annotations [8, 20, 23, 48]. T-SPARQL [23] proposes an extension to SPARQL, inspired by TSQL2 [56], which allows triples to be annotated with temporal constraints. Such constraints can be a commit timestamp or time intervals. SPARQL-LTL [20] on the other hand, works by annotating triples with version numbers, where each version is stored as a named graph. Other SPARQL extensions [8, 48] are more domain-specific and focus on the inclusion of temporal annotations with geo-spatial data.

The BEAR [19] benchmark for RDF archiving proposes instead the use of the AnQL [64] language to express versioned complex queries. AnQL is a superset of SPARQL operating on quads instead of triples. The quad component can be bound to any term $u \in \mathcal{I} \cup \mathcal{L}$. In practice, BEAR suggests the use of this component to represent time objects, such as timestamps or version identifiers.

All in all, no formal standard exists for representing complex queries for RDF archives, leaving existing systems with the responsibility of implementing their own solution. This lack of standard hinders the wider adoption of archiving technologies in the RDF community and limits the application of these technologies to the real world. In this section, we adopt the use of named graphs to represent version numbers, which we detail in Section 5.3. We leave the development of a new standard for SPARQL archive queries to future work.

5.3 SPARQL 1.1 for RDF Archives

In this section, we describe our implementation of versioned queries within SPARQL 1.1. Our solution is built on top of our architecture described in Section 4. First, we discuss how we tackle the formulation of versioned SPARQL queries. Then, the details of the implementation, architecture, and query engine will be discussed.

Versioned SPARQL Queries

As shown in Section 5.2, only a few efforts exists for the representation complex versioned queries in SPARQL. Most solutions are based on ad-hoc extensions for the SPARQL language, and none has been widely adopted by

the community. As such, we propose to stick to standard SPARQL, by making use of the *GRAPH* keyword, in order to express versioning information as named graphs. This is similar to other solutions, such as those proposed by [24].

Version Materialization (VM) Which countries were part of the EU in 2003?	**Delta Materialization (DM)** Which countries joined the EU in 2004?	**Version Query (V)** Which countries were part of the EU in each year?

SELECT * WHERE {
GRAPH <version:2003> {
?country rdf:type ex:country .
?country ex:member ex:EU .
}
}
	SELECT * WHERE {
	GRAPH <version:2004> {
	?country rdf:type ex:country .
	?country ex:member ex:EU .
	} FILTER (NOT EXISTS {
	GRAPH <version:2003> {
	?country rdf:type ex:country .
	?country ex:member ex:EU .
	}
	})
	}
		SELECT * WHERE {
		GRAPH ?version {
		?country rdf:type ex:country .
		?country ex:member ex:EU .
		}
		}

<ex:Austria> <ex:Belgium> <ex:Denmark> <ex:Finland> ...	<ex:Cyprus> <ex:Czech Republic> <ex:Estonia> <ex:Hungary> ...	<ex:Austria> <version:1995> <ex:Austria> <version:1996> ... <ex:Belgium> <version:1958> ...

Table 8: Example of SPARQL representation and results for VM, DM, and V queries. Reproduced from [41].

Table 8, provides examples of versioned full SPARQL queries for the VM, DM, and V types, using our formulation. In this example, queries are defined for an RDF archive $\mathcal{A}$ containing information on countries. Each graph revision $G_i \in \mathcal{A}$ represents the state of countries at a specific year, e.g., G_{2004} represents year 2004. As discussed earlier, we make use of the *GRAPH* keyword to describe versioning information. For VM queries, the graph IRI specifies the exact revision that is to be queried. DM queries are more complex and combine the *GRAPH* keyword with a *FILTER* clause to only select changes between revisions. Finally, V queries instead use a variable together with the *GRAPH* keyword.

Implementation

The implementation of the SPARQL formulation discussed previously is done on top of the multiple delta chain RDF archiving system presented in Section 4 (Paper C with improvements from Paper D). Because this system only supports single triple pattern queries, we chose the *Comunica* [60] query engine to support SPARQL queries. Comunica is a flexible and modular query engine with full support for SPARQL 1.1. Its modularity naturally allows for the inclusion of versioning into the engine via modules, and initial work has been conducted by Taelman et al. [59] for an initial implementation, although without V query support.

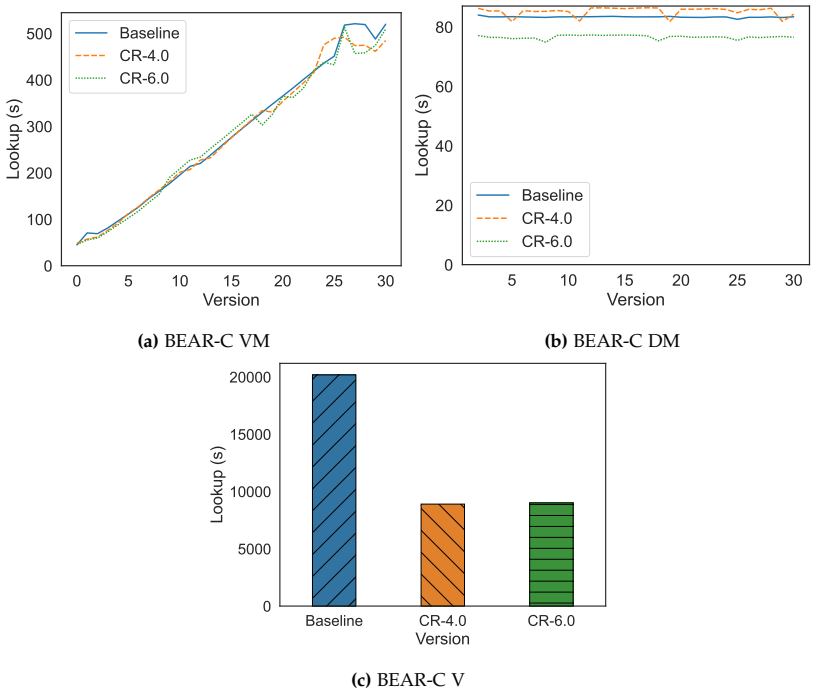
This implementation has been extended to support the new multiple delta chain architecture. This includes several optimizations to the communication between the storage system and the query engine to reduce the number of buffered triples during querying, which is beneficial for both memory usage and response time. Moreover, full support for V queries has been implemented, according to the syntax proposed in Section 5.3.

This implementation allows us to fully support the execution of arbitrary SPARQL queries on RDF archives, with full support for all three main versioned query types. Together with the improvements in performance and scalability provided by the multiple delta chain storage architecture, this enables complex querying on much larger archives, something that was not possible before.

5.4 Experimental Evaluation

We evaluated our versioned SPARQL implementation on the BEAR-C benchmark. BEAR-C is part of the BEAR benchmark suite [19] for RDF archives and focuses on SPARQL query processing. BEAR-C is made up of 32 snapshots from the European Open Data portal, obtained from the Open Data Portal Watch project [34]. In terms of size, BEAR-C is moderately large, with snapshots ranging from 485K to 563K triples. The query workload consists of 11 full SPARQL queries of various complexity and selectivity. As discussed previously, none of the other openly available RDF archiving systems can process SPARQL queries over larger RDF archives [19, 39], which limits the possible competitors for experimental evaluation. We selected both OSTRICH and our multiple delta chain architecture for our experiments. OSTRICH uses Comunica for SPARQL processing, with the same implementation described in Section 5.3. We opt for the change-ratio strategy with $\gamma = 4.0$ and $\gamma = 6.0$ for our multiple delta chain system.

Figure 14 shows the aggregated average runtimes for the BEAR-C benchmark. Due to its small number of revisions, the BEAR-C dataset is not well suited to a multiple delta chain approach, as it can already be handled efficiently by a single one. However, for DM queries (Figure 14b), the change ratio $\gamma = 6.0$ outperforms the baseline, while the change ratio $\gamma = 4.0$, which produces a higher number of delta chains, is slightly slower. For VM queries (Figure 14a), all systems perform closely to each other, except for the last few revisions (25 to 32) where the baseline appears to be slightly outperformed. Finally, V queries is where the largest performance difference exists. In that case, both multiple delta chains strategies significantly outperform the baseline. Overall, the performance differential between the multiple delta chain strategies and the baseline is small on this benchmark, mainly due to its relatively small size and reduced number of revisions. Nonetheless, using multiple delta chains is still beneficial for querying, which together with



(a) BEAR-C VM

(b) BEAR-C DM

(c) BEAR-C V

Fig. 14: BEAR-C average query execution time in seconds for VM, DM, and V queries.

its performance advantage during ingestion, makes this solution the overall best.

5.5 Conclusion and Future Work

In this section, we have described our implementation of full SPARQL querying over RDF archives. This implementation is built on top of our multiple delta chain architecture detailed in Section 4 (Paper C), with performance and compression improvements proposed in Paper D. Our experiments illustrate the capabilities of our implementation by fully completing the BEAR-C benchmark, a first to the best of our knowledge at the time of writing. Full SPARQL querying over RDF archives is a challenging task, and our implementation constitutes a first step towards more advanced applications. More details about our SPARQL processing implementation, as well as more experiments, can be found in Paper D. However, the lack of a formal standard for archiving queries currently limits advancements in that direction. Most solutions, including our own, rely on ad hoc or application-specific extensions to the SPARQL language, which limits their applicability to the real world. As future work, we envision to propose a standard for the syntax and semantics

of versioned queries.

6 Querying RDF Archives with GLENDA

This section describes a practical demonstration of SPARQL processing over RDF archives, built on top of the work presented in Section 5. This section is based on Paper E [42], and partially reuses its content.

6.1 Motivation

One of the key blocker to the adoption of archiving technologies by the wider semantic web community is the lack of proper solutions for running complex queries. In Section 5 (Paper D), we have described our solutions for processing full SPARQL queries over RDF archives, combining the Comunica [60] query engine with our multiple delta chain storage architecture [44]. Our experiments demonstrated our ability to effectively run complex queries for the first time on the BEAR-C benchmark for SPARQL processing over RDF archives. In this section, based on Paper E, we describe the use of our SPARQL implementation to develop a demonstration system that provides a practical and visual showcase of query processing over RDF archives. This system, named GLENDA, includes an SPARQL endpoint usable by any other application, as well as a front-end web application where the user can express versioned queries with a user-friendly interface (UI).

6.2 Functionalities and Implementation

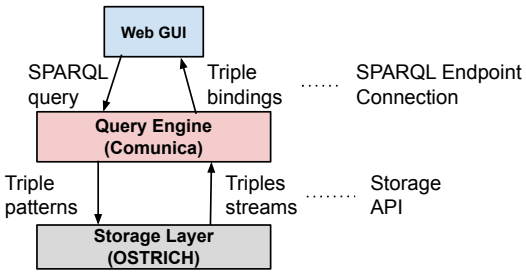


Fig. 15: GLENDA's components, reproduced from [42]

GLENDA is at its core a web application built on top of the Comunica query engine and our multiple delta chain storage architecture. A simplified overview of the different components necessary for the functioning of GLENDA is shown in Figure 15. As mentioned above, the storage layer is

composed of our architecture presented in Section 4 (Paper C) including improvements to the versioning of metadata compression (Paper D). The query engine is Comunica [60], with custom modules to enable the processing of versioned SPARQL queries, while the web application is a traditional website, implemented with HTML and JavaScript. Details about the processing of SPARQL queries over our multiple delta chain architecture can be found in Section 5 (Paper D).

The web application allows the user to run versioned queries in SPARQL of any of the Delta Materialization (DM), Version Materialization (VM), or Version Query (V) types (see Section 2.2 for a description of the query types). UI elements facilitate the choice of query type and selection of the target version(s) when relevant. Additionally, the interface offers the possibility to display various statistics about the underlying RDF archive.

6.3 Conclusion

In this section, based on Paper E, we describe a concrete example of an application for full SPARQL processing over RDF archives. Users can benefit from the UI to express versioned SPARQL queries in a natural way and display various relevant statistics about the evolution of the underlying archive. This tool constitutes a first step towards more concrete usage of archiving, with a focus on the expression and execution of versioned queries. The tool also exposes an SPARQL endpoint, which can be used directly by any other application. As discussed in the conclusion of Section 5, the lack of a formal standard to express SPARQL queries for RDF archives remains a crucial limitation.

7 The Limits of RDF Archiving System Evaluation

This section discusses the problems related to the evaluation of RDF archiving systems with standard benchmarks. We describe there the current solutions for benchmarking and their limitations with respect to the modern challenges faced by RDF archiving systems. This section is based on the work of Paper F [43] and partially reuses its content.

7.1 Motivation

Benchmarks are crucial for the scientific evaluation of systems and algorithms. They propose standardized workloads and metrics that can be repeated and reproduced across systems. Similarly, benchmarks are used to evaluate the compliance of systems and algorithms with community-defined

feature sets. As such, the availability of a diverse set of up-to-date benchmarks is a key element in the advancement of a field. Although there are several standard benchmarks for regular RDF systems [11, 16, 25], the same cannot be said for RDF archiving systems [37]. In this section, we examine the current state of RDF archiving benchmarks and analyze their strengths and limitations. We show how the current offer in benchmarks does not adequately cover novel challenges faced by RDF archiving systems, and we use this insight to sketch the requirements for future benchmarks.

7.2 Existing Benchmarks for RDF Archives

Several benchmarks for RDF archiving can be identified in the literature. EvoGen [31], BEAR [19], and SPBv [37].

EvoGen [31] is a synthetic benchmark based on the LUBM [25] data generator, which has been extended to support archiving. By being synthetic, this benchmark offers several configuration options for the number of versions and magnitude of changes between versions. Its querying workload is made up of the 14 SPARQL queries from LUBM.

BEAR [19] is a benchmark extracted from real-world RDF datasets, which we have used to evaluate our work in Papers C and D . It proposes three different archives, BEAR-A, BEAR-B, and BEAR-C, of different size and query workloads. BEAR-A and BEAR-B focus on data scalability with only single triple pattern queries, while BEAR-C proposes 11 full SPARQL queries. BEAR does not offer configuration options, all workloads being predefined and static.

SPBv [37] is a synthetic benchmark based on the Semantic Publishing Benchmark (SPB) [30]. Like EvoGen, SPBv offers various parameters that the user can configure to change the size of the generated data, the number of queries, and how the data need to be generated (snapshots, deltas, or both). The querying workload consists of full SPARQL queries.

7.3 Evaluating RDF Archives

Several aspects must be considered when designing a benchmark. We propose three overarching qualities to cover them:

- *Reproducibility*: how easy benchmarks results can be shared and reproduced.
- *Realism*: how well the benchmark emulates the real world through its choice of data and querying workload.
- *Configurability*: how many configuration options the benchmark provides to scale and tune its workload.

Existing benchmarks are either using synthetic data through a generator or derived from real-world datasets. Although synthetic benchmarks will usually be highly *configurable*, they are often unrealistic, as discussed by Duan et al. [16]. They propose, as a solution, the use of a *coherence* metric to enforce the realism of the generated data. However, no benchmarks for RDF archiving currently use this solution, which means that real-world-based benchmarks would be considered more realistic.

While testing early RDF archiving systems with single triple pattern queries was sufficient, current progresses in storage architectures and scalability suggest that a shift towards more comprehensive SPARQL query workloads is necessary. Many applications are using complex SPARQL queries, which benchmarks would need to emulate to comply with real-world usages. Indeed, SPARQL processing over RDF archives constitutes a significant challenge that RDF archiving systems need to solve. As such, a *realistic* benchmark should provide carefully crafted SPARQL query workloads. However, the lack of formal standards to formulate archiving queries in SPARQL is a major hurdle that needs to be addressed in order to progress in that direction.

	Dataset	Reproducibility	Realism (data)	Realism (queries)	Configurability
EvoGen [31]	Synthetic	-/+	-	+	+
BEAR [19]	Real-world	+	+	-	-
SPBv [37]	Synthetic	-/+	-	+	+

Table 9: Comparison table of existing RDF Archiving benchmarks, reproduced from [43].

We summarize our categorization of the existing benchmarks for RDF archives in Table 9. The main takeaway is that none of the benchmarks fulfills all of the three requirements. Synthetic benchmarks, namely EvoGen and SPBv cannot ensure the realism of their generated data, but provide full SPARQL query workloads. Oppositely, BEAR proposes real-world based datasets, but only the BEAR-C variant offers SPARQL queries, which limits BEAR ability to thoroughly evaluate RDF archiving systems with SPARQL support.

7.4 Conclusion

In this section, we have discussed benchmarks for RDF archives. We showed that the current offer for such benchmarks is limited, with only three available. Among these, none is fully satisfactory as a complete solution for evaluating the modern challenges in RDF archiving. We have proposed three qualities that the benchmarks should have and showed that none of them currently satisfies all of them. Benchmarks should be relevant to the novel challenges faced by RDF applications, which include the ability to execute complex queries over RDF archives. The lack of a formal standard for such

queries limits the possibility of standardization and hinders the adoption of archiving technologies by applications.

8 Conclusions and Future Work

The proliferation and widespread adoption of semantic web technologies has increased in recent years. This surge is notably characterized by exponential growth in RDF sources and datasets, thus accentuating the need for the tracking and management of metadata. Notably, a particular focus is observed on managing the provenance and changes of triples within knowledge graphs. These emergent tasks introduce substantial challenges for both data maintainers and end users, necessitating consideration of issues such as storage usage, the means to access and use these metadata, and incorporation of these metadata within diverse applications. The increasing volume of metadata, particularly when tracking changes to a knowledge graph, means that existing storage solutions and algorithms are insufficient. In particular, none can efficiently handle this kind of data on the scale required by real-world RDF datasets. This realization has sparked the development of dedicated methods, architectures, and algorithms by the scientific community.

In this thesis, we investigate the challenges in efficiently managing and querying the evolution history of RDF knowledge graphs. Existing solutions do not completely address the significant challenges involved in dealing with the scale of current RDF archives. Similarly, making RDF archives usable through expressive querying has not been addressed in the state of the art.
In summary, this thesis includes the following papers and contributions:

- In Paper A [39], we investigate the state-of-the-art in archiving RDF datasets. We discuss the strengths and weaknesses of all different storage paradigms employed by existing systems. We conducted an evaluation of available RDF archiving systems on real-world RDF archives, and showed how current solutions have limitations, notably in scalability. Furthermore, we propose a framework to analyze the evolution of RDF datasets over time and use it on the Wikidata, DBpedia and YAGO knowledge graph. We show that those KG often exhibit a *"minor"* and *"major"* update pattern, which archiving systems would need to take into account for the best efficiency.
- In Paper B [40], we investigate solutions for indexing RDF data with arbitrary levels of metadata. We propose an in-memory indexing scheme and dictionary, inspired by the *trie* data structure, capable of indexing triples with any amount of metadata. We evaluate this solution with versioning and provenance-annotated RDF triples and show promising performance against other in-memory solutions.

- In Paper C [44], we focus on the challenge of on-disk indexing and querying of large RDF archives. Building on a top of state-of-the-art RDF archiving system, OSTRICH [58], we propose a novel hybrid multiple delta chain storage architecture. We provide several strategies to automatically make the decision to start a new delta chain. The different strategies allow for optimizing disk usage, ingestion time, and affect query performance. Our evaluation shows significantly improved ingestion times for large RDF archives with competitive querying performance against our baseline, OSTRICH.
- In Paper D [41], we build further on top of the contributions from Paper C. We introduce a new compressed metadata representation for our triple indexes, which significantly improves ingestion speed and storage usage compared to the old metadata representation. Furthermore, we detail our implementation of full SPARQL processing with versioning support over RDF archives, built on top of our storage solution thanks to the Comunica query engine. We evaluated our SPARQL processing capabilities on the BEAR-C benchmark. This is a first, to the best of our knowledge and at the time of writing.
- In Paper E [42], we propose a demonstration of full SPARQL processing over RDF archives. The demonstration consists of a web application where users can express archive queries thanks to a specialized GUI. The application is built on top of the system proposed in Paper C.
- In Paper F [43], we discuss current limitations in benchmarks designed to evaluate RDF archiving systems. We show that the number of different benchmarks is limited, with only three options. Among the available ones, we show that they do not adequately cover the modern challenges faced by RDF archiving systems, notably with SPARQL query processing. We use these insights to outline the requirements of a modern benchmark for RDF archiving.

8.1 Future Work

The work presented in this thesis leads to several possible future work in the domain of RDF archiving and metadata management.

First, improvements to scalability are still possible, notably to enable the archiving of larger RDF archives, for example, the entirety of Wikidata or DBpedia. Modern indexing schemes, such as v-HDT/v-RDFCSA [14] show great efficiency at archiving static RDF archives. In the future, combining such techniques with the one developed in this thesis could enable greater storage efficiency and querying performance while still allowing live updates with the use of delta chains. Furthermore, novel approaches for compact representation of semantic data [47, 53] could be promising alternatives to the B+Trees indexes currently used. In Paper C, we have introduced sev-

eral strategies to dynamically decide when the materialization of new delta chains while ingesting data is necessary. However, these strategies remain straightforward. We envision the development of more advanced strategies, possibly using machine learning techniques, to better handle the dynamicity of RDF archives.

As discussed in Paper E and D, the options to express archive queries in SPARQL are limited. Most solutions consist of ad hoc extensions to the SPARQL languages, with various levels of expressivity and different semantics. Our current proposal, presented in Paper D, makes use of the *named graph* feature of SPARQL to refer to versions. However, this prevents the concurrent use of named graphs and versioning. In the future, a standardization of archive queries in SPARQL should be undertaken to permit a wider use of archiving technologies by the semantic web community. This effort could be inspired by the RSP-QL [15] standardization effort for the processing of RDF streams. During this process, the possible conceptual overlap between RDF streams' temporal graph and RDF archives should be carefully examined, together with an analysis of real-world RDF archive usages.

Finally, the integration of multiple types of metadata together remains to be fully explored. We have presented in Paper B a first step towards a more general support for an arbitrary amount of metadata with RDF. However, this solution remains limited to in-memory indexing, making large datasets prohibitively expensive to manage. Furthermore, querying capabilities are currently limited to single tuple queries. The recent development of RDF-star [26] has allowed for graceful representation of metadata in a unified format. At the time of writing this thesis, RDF-star is considered for inclusion in the upcoming RDF 1.2 standard. The development of an RDF-star compatible system with an archiving-optimized indexing scheme could enable the emergence of new RDF-star archives, i.e. the archiving of RDF data with an arbitrary number of triple metadata.

References

[1] "RDF Exports from Wikidata," Available at tools.wmflabs.org/wikidata-exports/rdf/index.html.

[2] "Stardog," http://stardog.com.

[3] C. Aebeloe, G. Montoya, and K. Hose, "Colchain: Collaborative linked data networks," in *WWW*. ACM / IW3C2, 2021, pp. 1385–1396.

[4] J. Anderson and A. Bendiken, "Transaction-time queries in dydra," in *MEPDaW/LDQ@ESWC*, ser. CEUR Workshop Proceedings, vol. 1585. CEUR-WS.org, 2016, pp. 11–19.

[5] N. Arndt, P. Naumann, N. Radtke, M. Martin, and E. Marx, "Decentralized collaborative knowledge management using git," *J. Web Semant.*, vol. 54, pp. 29–47, 2019.

[6] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A Nucleus for a Web of Open Data," in *The Semantic Web*, 2007, pp. 722–735.

[7] H. R. Bazoobandi, S. de Rooij, J. Urbani, A. ten Teije, F. van Harmelen, and H. E. Bal, "A Compact In-Memory Dictionary for RDF Data," in *ESWC*, 2015, pp. 205–220.

[8] K. Bereta, P. Smeros, and M. Koubarakis, "Representation and Querying of Valid Time of Triples in Linked Geospatial Data," in *ESWC*, 2013, pp. 259–274.

[9] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 28–37, 2001.

[10] C. Bizer, "The Emerging Web of Linked Data," *IEEE Intelligent Systems*, vol. 24, no. 5, pp. 87–92, 2009.

[11] P. A. Boncz, I. Fundulaki, A. Gubichev, J. L. Larriba-Pey, and T. Neumann, "The linked data benchmark council project," *Datenbank-Spektrum*, vol. 13, no. 2, pp. 121–129, 2013.

[12] N. R. Brisaboa, A. Cerdeira-Pena, A. Fariña, and G. Navarro, "A Compact RDF Store Using Suffix Arrays," in *String Processing and Information Retrieval*, 2015, pp. 103–115.

[13] J. Brunsmann, "Archiving Pushed Inferences from Sensor Data Streams," in *Proceedings of the International Workshop on Semantic Sensor Web*. INSTICC, 2010, pp. 38–46.

[14] A. Cerdeira-Pena, G. de Bernardo, A. Fariña, J. D. Fernández, and M. A. Martínez-Prieto, "Compressed and queryable self-indexes for rdf archives," *Knowledge and Information Systems*, pp. 1–37, 2023.

[15] D. Dell'Aglio, E. D. Valle, J. Calbimonte, and Ó. Corcho, "RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems," *Int. J. Semantic Web Inf. Syst.*, vol. 10, no. 4, pp. 17–44, 2014.

[16] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea, "Apples and oranges: a comparison of RDF benchmarks and real RDF datasets," in *SIGMOD*, 2011, pp. 145–156.

[17] F. Erxleben, M. Günther, M. Krötzsch, J. Mendez, and D. Vrandečić, "Introducing Wikidata to the Linked Data Web," in *International Semantic Web Conference*, 2014, pp. 50–65.

[18] J. D. Fernández, M. A. Martínez-Prieto, C. Gutierrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange (HDT)," *J. Web Semant.*, vol. 19, pp. 22–41, 2013.

[19] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for RDF archives," *J. Web Semant.*, vol. 10, no. 2, pp. 247–291, 2019.

[20] V. Fionda, M. W. Chekol, and G. Pirrò, "Gize: A Time Warp in the Web of Data," in *ISWC (Posters & Demos)*, vol. 1690, 2016.

[21] J. Frey, M. Hofer, D. Obraczka, J. Lehmann, and S. Hellmann, "Dbpedia flexifusion the best of wikipedia > wikidata > your data," in *ISWC (2)*, ser. LNCS, vol. 11779. Springer, 2019, pp. 96–112.

[22] S. Gao, J. Gu, and C. Zaniolo, "RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases." in *Proceedings of the Extended Semantic Web Conference*, 2016, pp. 269–280.

[23] F. Grandi, "T-SPARQL: A TSQL2-like Temporal Query Language for RDF," in *Local Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems*, 2010, pp. 21–30.

[24] M. Graube, S. Hensel, and L. Urbas, "R43ples: Revisions for triples - an approach for version control in the semantic web," in *LDQ@SEMANTICS*, 2014.

[25] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *J. Web Semant.*, vol. 3, no. 2-3, pp. 158–182, 2005.

[26] O. Hartig, "Foundations of rdf⋆ and sparql⋆ (an alternative approach to statement-level metadata in RDF)," in *AMW*, ser. CEUR Workshop Proceedings, vol. 1912. CEUR-WS.org, 2017.

[27] I. Horrocks, T. Hubauer, E. Jiménez-Ruiz, E. Kharlamov, M. Koubarakis, R. Möller, K. Bereta, C. Neuenstadt, O. Özçep, M. Roshchin, P. Smeros, and D. Zheleznyakov, "Addressing Streaming and Historical Data in OBDA Systems: Optique's Approach (Statement of Interest)," in *Workshop on Knowledge Discovery and Data Mining Meets Linked Open Data (Know@LOD)*, 2013.

[28] T. Huet, J. Biega, and F. M. Suchanek, "Mining History with Le Monde," in *Proceedings of the Workshop on Automated Knowledge Base Construction*, 2013, pp. 49–54.

[29] D. hyuk Im, S. won Lee, and H. joo kim, "A Version Management Framework for RDF Triple Stores," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, 04 2012.

[30] V. Kotsev, N. Minadakis, V. Papakonstantinou, O. Erling, I. Fundulaki, and A. Kiryakov, "Benchmarking RDF query engines: The LDBC semantic publishing benchmark," in *BLINK@ISWC*, 2016.

[31] M. Meimaris and G. Papastefanatos, "The EvoGen Benchmark Suite for Evolving RDF Data," in *MEPDaW/LDQ@ESWC*, 2016, pp. 20–35.

[32] S. Metzger, K. Hose, and R. Schenkel, "Colledge: a vision of collaborative knowledge networks," in *Proceedings of the 2nd International Workshop on Semantic Search over the Web*. ACM, 2012, pp. 1–8. [Online]. Available: https://doi.org/10.1145/2494068.2494069

[33] T. M. Mitchell and et al., "Never-Ending Learning," in *AAAI*, 2015, pp. 2302–2310.

[34] S. Neumaier, J. Umbrich, and A. Polleres, "Automated quality assessment of metadata across open data portals," *ACM J. Data Inf. Qual.*, vol. 8, no. 1, pp. 2:1–2:29, 2016.

[35] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 647–659, 2008.

[36] ——, "x-rdf-3x: Fast querying, high update rates, and consistency for RDF databases," *PVLDB*, vol. 3, no. 1, pp. 256–263, 2010.

[37] V. Papakonstantinou, G. Flouris, I. Fundulaki, K. Stefanidis, and Y. Roussakis, "Spbv: Benchmarking linked data archiving systems," in *BLINK/NLIWoD3@ISWC*, vol. 1932, 2017.

[38] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides, "On Detecting High-level Changes in RDF/S KBs," in *International Semantic Web Conference (ISWC)*, 2009, pp. 473–488.

[39] O. Pelgrin, L. Galárraga, and K. Hose, "Towards fully-fledged archiving for RDF datasets," *Semantic Web Journal*, vol. 12, no. 6, pp. 903–925, 2021.

[40] ——, "Triedf: Efficient in-memory indexing for metadata-augmented RDF," in *MEPDaW@ISWC*, ser. CEUR Workshop Proceedings, vol. 3225. CEUR-WS.org, 2021, pp. 20–29.

[41] O. Pelgrin, R. Taelman, L. Galárraga, and K. Hose, "Expressive querying and scalable management of large rdf archives," in *Under revision*, 2023.

[42] ——, "GLENDA: querying RDF archives with full SPARQL," in *ESWC (Satellite Events)*, ser. Lecture Notes in Computer Science, vol. 13998. Springer, 2023, pp. 75–80.

[43] ——, "The need for better rdf archiving benchmarks," in *MEPDaW@ISWC*, ser. CEUR Workshop Proceedings. CEUR-WS.org, 2023.

[44] ——, "Scaling large RDF archives to very long histories," in *ICSC*. IEEE, 2023, pp. 41–48.

[45] T. Pellissier Tanon, C. Bourgaux, and F. Suchanek, "Learning How to Correct a Knowledge Base from the Edit History," in *The World Wide Web Conference*, 2019, pp. 1465–1475.

[46] T. Pellissier Tanon and F. M. Suchanek, "Querying the Edit History of Wikidata," in *Extended Semantic Web Conference*, 2019.

[47] R. Perego, G. E. Pibiri, and R. Venturini, "Compressed indexes for fast search of semantic data," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 9, pp. 3187–3198, 2021.

[48] M. Perry, P. Jain, and A. P. Sheth, "SPARQL-ST: Extending SPARQL to Support Spatio-temporal Queries," in *Geospatial Semantics and the Semantic Web*, vol. 12, 2011, pp. 61–86.

[49] A. Polleres, R. Pernisch, A. Bonifati, D. Dell'Aglio, D. Dobriy, S. Dumbrava, L. Etcheverry, N. Ferranti, K. Hose, E. Jiménez-Ruiz, M. Lissandrini, A. Scherp, R. Tommasini, and J. Wachs, "How does knowledge evolve in open knowledge graphs?" *TGDK*, vol. 1, no. 1, pp. 11:1–11:59, 2023.

[50] M. Psaraki and Y. Tzitzikas, "CPOI: A Compact Method to Archive Versioned RDF Triple-Sets," 2019.

[51] Y. Raimond and G. Schreiber, "RDF 1.1 primer," W3C Recommendation, 2014, http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[52] Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris, and Y. Stavrakas, "A Flexible Framework for Understanding the Dynamics of Evolving RDF Datasets," in *International Semantic Web Conference (ISWC)*, 2015.

[53] T. Sagi, M. Lissandrini, T. B. Pedersen, and K. Hose, "A design space for RDF data representations," *VLDB J.*, vol. 31, no. 2, pp. 347–373, 2022.

[54] M. V. Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. V. de Walle, "R&wbase: git for triples," in *LDOW*, ser. CEUR Workshop Proceedings, vol. 996. CEUR-WS.org, 2013.

[55] A. Seaborne and S. Harris, "SPARQL 1.1 query language," W3C, W3C Recommendation, 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[56] R. T. Snodgrass, I. Ahn, G. Ariav, D. S. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. G. Kulkarni, T. Y. C. Leung, N. A. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada, "TSQL2 Language Specification," *SIGMOD Record*, vol. 23, no. 1, pp. 65–86, 1994.

[57] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A Large Ontology from Wikipedia and Wordnet," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 3, pp. 203–217, 2008.

[58] R. Taelman, M. V. Sande, J. V. Herwegen, E. Mannens, and R. Verborgh, "Triple Storage for Random-access Versioned Querying of RDF Archives," *J. Web Semant.*, vol. 54, pp. 4–28, 2019.

[59] R. Taelman, M. V. Sande, and R. Verborgh, "Versioned querying with OSTRICH and comunica in MOCHA 2018," in *SemWebEval@ESWC*, vol. 927, 2018, pp. 17–23.

[60] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, "Comunica: a modular sparql query engine for the web," in *ISWC*, 2018.

[61] R. Taelman and R. Verborgh, "In-memory indexing of quoted rdf triples," in *QuWeDa@ISWC*, 2023.

[62] T. P. Tanon, G. Weikum, and F. M. Suchanek, "YAGO 4: A Reason-able Knowledge Base," in *ESWC*, 2020, pp. 583–596.

[63] M. Volkel, W. Winkler, Y. Sure, S. R. Kruk, and M. Synak, "SemVersion: A Versioning System for RDF and Ontologies," *ESWC*, 2005.

[64] A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia, "A general framework for representing, reasoning and querying with annotated semantic web data," *J. Web Semant.*, vol. 11, pp. 72–95, 2012.

Part II

Papers

Paper A

Towards Fully-fledged Archiving for RDF Datasets

Olivier Pelgrin, Luis Galárraga, Katja Hose

The paper has been published in the
Semantic Web Journal, pp. 903-925, 2021.
DOI: 10.3233/SW-210434

Abstract

The dynamicity of RDF data has motivated the development of solutions for archiving, i.e., the task of storing and querying previous versions of an RDF dataset. Querying the history of a dataset finds applications in data maintenance and analytics. Notwithstanding the value of RDF archiving, the state of the art in this field is under-developed: (i) most existing systems are neither scalable nor easy to use, (ii) there is no standard way to query RDF archives, and (iii) solutions do not exploit the evolution patterns of real RDF data. On these grounds, this paper surveys the existing works in RDF archiving in order to characterize the gap between the state of the art and a fully-fledged solution. It also provides RDFev*, a framework to study the dynamicity of RDF data. We use RDFev to study the evolution of YAGO, DBpedia, and Wikidata, three dynamic and prominent datasets on the Semantic Web. These insights set the ground for the sketch of a fully-fledged archiving solution for RDF data.*

Towards Fully-fledged Archiving for RDF Datasets. In the Semantic Web Journal, Volume 12, Pages 903–925, 2021.
https://doi.org/10.3233/SW-210434
The layout has been revised.

1 Introduction

The amount of RDF data has steadily grown since the conception of the Semantic Web in 2001 [17], as more and more organizations opt for RDF [69] as the format to publish and manage semantic data [42, 44]. For example, by July 2009 the Linked Open Data (LOD) cloud counted a few more than 90 RDF datasets adding up to almost 6.7B triples [18]. By 2020, these numbers have catapulted to 1200+ datasets[1] and at least 28B triples[2], although estimates based on LODStats [25] suggest more than 10K datasets and 150B+ triples if we consider the datasets with errors omitted by the LOD Cloud [62]. This boom does not only owe credit to the increasing number of data providers and availability of Open Government Data [1, 4, 10], but also to the constant evolution of the datasets in the LOD cloud. This phenomenon is especially true for community-driven initiatives such as DBpedia [13], YAGO [77], or Wikidata [26], and also applies to automatically ever-growing projects such as NELL [21].

Storing and querying the entire edition history of an RDF dataset, a task we call *RDF archiving*, has plenty of applications for data producers. For instance, RDF archives can serve as a backend for fine-grained version control in collaborative projects [9, 12, 32, 36, 52, 72]. They also allow data providers to study the evolution of the data [29] and track errors for debugging purposes. Likewise, they can be of use to RDF streaming applications that rely on a structured history of the data [20, 43]. But archives are also of great value for consumer applications such as data analytics, e.g., mining correction patterns [64, 65] or historical trend analysis [45].

For all the aforementioned reasons, a significant body of literature has started to tackle the problem of RDF archiving. The current state of the art ranges from systems to store and query RDF archives [3, 11, 12, 23, 34, 36, 59, 67, 72, 78, 81], to benchmarks to evaluate such engines [29, 51], as well as temporal extensions for SPARQL [16, 30, 35, 66]. Diverse in architecture and aim, all these works respond to particular use cases. Examples are solutions such as R&Wbase [72], R43ples [36], and Quit Store [12] that provide data maintainers with distributed version control management in the spirit of Git. Conversely, other works [34, 66] target data consumers who need to answer time-aware queries such as "obtain the list of house members who sponsored a bill from 2008". In this case the metadata associated to the actual triples is used to answer domain-specific requirements.

Despite this plethora of work, there is currently no available fully-fledged solution for the management of large and dynamic RDF datasets. This situation originates from multiple factors such as (i) the performance and func-

[1] `https://lod-cloud.net/`

[2] `http://lod-a-lot.lod.labs.vu.nl/`

tionality limitations of RDF engines to handle metadata, (ii) the absence of a standard for querying RDF archives, and (iii) a disregard of the actual evolution of real RDF data. This paper elaborates on factors (i) and (ii) through a survey of the state of the art that sheds light on what aspects have not yet been explored. Factor (iii) is addressed by means of a framework to study the evolution of RDF data applied to three large and ever-changing RDF datasets, namely DBpedia, YAGO, and Wikidata. The idea is to identify the most challenging settings and derive a set of design lessons for fully-fledged RDF archive management. We therefore summarize our contributions as follows:

1. *RDFev*, a metric-based framework to analyze the evolution of RDF datasets;
2. A study of the evolution of DBpedia, YAGO, and Wikidata using *RDFev*;
3. A detailed survey of existing work on RDF archive management systems and SPARQL temporal extensions;
4. An evaluation of Ostrich [78] on the history of DBpedia, YAGO, and Wikidata. This was the only system that could be tested on the experimental datasets;
5. The sketch of a fully-fledged RDF archiving system that can satisfy the needs not addressed in the literature, as well as a discussion about the challenges in the design and implementation of such a system.

This paper is organized as follows. In Section 2 we introduce preliminary concepts. Then, Section 3 presents *RDFev*, addressing contribution (1). Contribution (2) is elaborated in Section 4. In the light of the evolution of real-world RDF data, we then survey the strengths and weaknesses of the different state-of-the-art solutions in Section 5 (contribution 3). Section 6 addresses contribution (4). The insights from the previous sections are then used to drive the sketch of an optimal RDF archiving system in Section 7, which addresses contribution (5). Section 8 concludes the paper.

2 Preliminaries

This section introduces the basic concepts in RDF archive storage and querying, and proposes some formalizations for the design of RDF archives.

2.1 RDF Graphs

We define an *RDF graph* G as a set of triples $t = \langle s, p, o \rangle$, where $s \in \mathcal{I} \cup \mathcal{B}$, $p \in \mathcal{I}$, and $o \in \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$ are the subject, predicate, and object of t, respectively. Here, $\mathcal{I}$, $\mathcal{L}$, and $\mathcal{B}$ are sets of IRIs (entity identifiers), literal values (e.g.,

$\langle s, p, o \rangle$, $\langle s, p, o, \rho \rangle$	triple and 4-tuple: subject, predicate, object, graph revision
G	an RDF graph
g	a graph label
G_i	the i-th version or revision of graph G
$A = \{G_0, G_1, \dots\}$	an RDF graph archive
$u = \{u^+, u^-\}$	an update or changeset with sets of added and deleted triples.
$u_{i,j} = \{u^+_{i,j}, u^-_{i,j}\}$	the changeset between graph revisions i and j ($j > i$)
$rv(\rho)$	revision number of graph revision ρ
$ts(\rho)$	commit time of graph revision ρ
$l(\rho)$, $l(G)$	labels of graph revision ρ and graph G

Table A.1: Notation related to RDF Graphs.

$\langle s, p, o, æ, ı \rangle$	a 5-tuple subject, predicate, object, graph revision, and dataset revision
$D = \{G^0, G^1, \dots\}$	an RDF dataset
$\mathcal{A} = \{D_0, D_1, \dots\}$	an RDF dataset archive
D_j	the j-th version or revision of dataset D
G_i^k	the i-th revision of the k-th graph in a dataset archive
$\hat{u} = \{\hat{u}^+, \hat{u}^-\}$	a graph changeset with sets of added and deleted graphs
$U = \{\hat{u}, u^0, \dots\}$	a dataset update or changeset consisting of a graph changeset $\hat{u}$ and changesets u^i associated to graphs G^i
U^+, U^-	the addition/deletion changes of U: $U^+ = \{\hat{u}^+, u^{0+}, u^{1+}, \dots\}$, $U^- = \{\hat{u}^-, u^{0-}, u^{1-}, \dots\}$
$U_{i,j}$	the dataset changeset between dataset revisions i and j ($j > i$)
$rv(\zeta)$	revision number of dataset revision ζ
$ts(\zeta)$	commit time of dataset revision ζ
$\Upsilon(\cdot)$	the set of terms (IRIs, literals, and blank nodes) present in a graph G, dataset D, changeset u, and dataset changeset U.

Table A.2: Notation related to RDF Datasets.

strings, integers, dates), and blank nodes (anonymous entities) [69]. The notion of a *graph* is based on the fact that G can be modeled as a directed labeled graph where the predicate p of a triple denotes a directed labeled edge from node s to node o. The RDF W3C standard [69] defines a *named graph* as an RDF graph that has been associated to a label $l(G) = g \in \mathcal{I} \cup \mathcal{B}$. The function $l(\cdot)$ returns the associated label of an RDF graph, if any. Table A.1 provides the relevant notation related to RDF graphs.

2.2 RDF Graph Archives

Intuitively, an *RDF graph archive* is a temporally-ordered collection of all the states an RDF graph has gone through since its creation. More formally, a graph archive $A = \{G_s, G_{s+i}, \ldots, G_{s+n-1}\}$ is an ordered set of RDF graphs, where each G_i is a *revision* or *version* with *revision number* $i \in \mathcal{N}$, and G_s ($s \geq 0$) is the graph archive's *initial revision*. A non-initial revision G_i ($i > s$) is obtained by applying an *update* or *changeset* $u_i = \langle u_i^+, u_i^- \rangle$ to revision G_{i-1}. The sets u_i^+, u_i^- consist of triples that should be added and deleted respectively to and from revision G_{i-1} such that $u_i^+ \cap u_i^- = \emptyset$. In other words, $G_i = u_i(G_{i-1}) = (G_{i-1} \cup u_i^+) \setminus u_i^-$. Figure A.1 provides a toy RDF graph archive A that models the evolution of the information about the country members of the United Nations (UN) and their diplomatic relationships (*:dr*). The archive stores triples such as $\langle$ `:USA`, `a`, `:Country` $\rangle$ or $\langle$ `:USA`, `:dr`, `:Cuba` $\rangle$, and consists of two revisions $\{G_0, G_1\}$. G_1 is obtained by applying update u_1 to the initial revision G_0. We extend the notion of changesets to arbitrary pairs of revisions i, j with $i < j$, and denote by $u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$ the changeset such that $G_j = u_{i,j}(G_i)$.

We remark that a graph archive can also be modeled as a collection of 4-tuples $\langle s, p, o, \rho \rangle$, where $\rho \in \mathcal{I}$ is the RDF identifier of revision $i = rv(\rho)$ and $rv \subset \mathcal{I} \times \mathcal{N}$ is a function that maps revision identifiers to natural numbers. We also define the function $ts \subset \mathcal{I} \times \mathcal{N}$ that associates a revision identifier ρ to its commit time, i.e., the timestamp of application of changeset u_i. Some solutions for RDF archiving [12, 34, 36, 59, 72, 78] implement this logical model in different ways and to different extents. For example, R43ples [36], R&WBase [72] and Quit Store [12] store changesets and/or their associated metadata in additional named graphs using PROV-O [24]. In contrast, x-RDF-3X [59] stores the temporal metadata in special indexes that optimize for concurrent updates at the expense of temporal consistency, i.e., revision numbers may not always be in concordance with the timestamps.

2.3 RDF Dataset Archives

In contrast to an RDF graph archive, an *RDF dataset* is a set $D = \{G^0, G^1, \ldots, G^m\}$ of named graphs. Differently from revisions in a graph

G_0	u_1	$G_1 = u_1(G_0)$
$\langle$:*USA*, *a*, :*Country*$\rangle$	$u_1^+ = \{\langle$:*France*, *a*, :*Country*$\rangle\}$	$\langle$:*USA*, *a*, :*Country*$\rangle$
$\langle$:*Cuba*, *a*, :*Country*$\rangle$	$u_1^- = \{\langle$:*USA*, :*dr*, :*Cuba*$\rangle\}\}$	$\langle$:*Cuba*, *a*, :*Country*$\rangle$
$\langle$:*USA*, :*dr*, :*Cuba*$\rangle$		$\langle$:*France*, : *a*, :*Country*$\rangle$

Fig. A.1: Two revisions G_0, G_1 and a changeset u_1 of an RDF graph archive A

archive, we use the notation G^k for the k-th graph in a dataset, whereas G_i^k denotes the i-th revision of G^k. The notation related to RDF datasets is detailed in Table A.2. Each graph $G^k \in D$ has a label $l(G^k) = g^k \in \mathcal{I} \cup \mathcal{B}$. The exception to this rule is G^0, known as the *default graph* [69], which is unlabeled.

Most of the existing solutions for RDF archiving can handle the history of a single graph. However, scenarios such as data warehousing [33, 38, 39, 47–50, 56, 57] may require to keep track of the common evolution of an RDF dataset, for example, by storing the addition and removal timestamps of the different RDF graphs in the dataset. Analogously to the definition of graph archives, we define a *dataset archive* $\mathcal{A} = \{D_0, D_1, \ldots, D_{l-1}\}$ as a temporally ordered collection of RDF datasets. The j-th revision of $\mathcal{A}$ ($j > 1$) can be obtained by applying a *dataset update* $U_j = \{\hat{u}_j, u_j^0, u_j^1, \ldots u_j^m\}$ to revision $D_{j-1} = \{G_{j-1}^0, G_{j-1}^1, \ldots G_{j-1}^m\}$. U_j consists of an update per graph plus a special changeset $\hat{u}_j = \langle \hat{u}_j^+, \hat{u}_j^- \rangle$ that we call the *graph changeset* ($\hat{u}_j^+ \cap \hat{u}_j^- = \emptyset$). The sets $\hat{u}_j^+, \hat{u}_j^-$ store the labels of the graphs that should be added and deleted in revision j respectively. If a graph G^k is in $\hat{u}_j^-$ (i.e., it is scheduled for removal), then G^k as well as its corresponding changeset $u_j^k \in U_j$ must be empty. It follows that we can obtain revision D_j by (i) applying the individual changesets $u_j^k(G_{j-1}^k)$ for each $0 \leq k \leq m$, (ii) removing the graphs in $\hat{u}_j^-$, and (iii) adding the graphs in $\hat{u}_j^+$.

Figure A.2 illustrates an example of a dataset archive with two revisions D_0 and D_1. D_0 is a dataset with graphs $\{G_0^0, G_0^1\}$ both at local revision 0. The dataset update U_1 generates a new global dataset revision D_1. U_1 consists of three changesets: u_1^0 that modifies the default graph G^0, u_1^1 that leaves G^1 untouched, and the graph update $\hat{u}_2$ that adds graph G^2 to the dataset and initializes it at revision $s = 1$ (denoted by G_1^2).

As proposed by some RDF engines [31, 76], we define the master graph $G^M \in D$ (with label M) as the RDF graph that stores the metadata about all the graphs in an RDF dataset D. If we associate the creation of a graph G^k with label g^k to a triple of the form $\langle g^k$, *rdf:type*, *η:Graph*$\rangle$ in G^M for some

$$
\begin{array}{ccc}
D_0 & U_1 & D_1 = U_1(D_0) \\
G_0^0 = \{\langle :USA, a, :Country\rangle, & \hat{u}_1 = \{\hat{u}_1^+ = \{G^2\},\ \hat{u}_1^- = \emptyset\} & G_1^0 = \{\langle :USA, a, :Country\rangle \\
\langle :Cuba, a, :Country\rangle, & u_1^0 = \{u_1^{0+} = \{\langle :France, a, :Country\rangle\}, & \langle :Cuba, a, :Country\rangle \\
\langle :USA, :dr, :Cuba\rangle\} & u_1^{0-} = \{\langle :USA, :dr, :Cuba\rangle\}\} & \langle :France, a, :Country\rangle\} \\
G_0^1 = \{\langle x{:}JFK, a, x{:}Airport\rangle\} & u_1^1 = \{\emptyset, \emptyset\} & G_1^1 = \{\langle x{:}JFK, a, x{:}Airport\rangle\} \\
 & & G_1^2 = \emptyset
\end{array}
$$

Fig. A.2: A dataset archive $\mathcal{A}$ with two revisions D_0, D_1. The first revision contains two graphs, the default graph G^0 and G^1. The dataset update $\hat{U}_1$ (i) modifies G^0, (ii) leaves G^1 untouched, and (iii) and creates a new graph G^2, all with local revision 1.

namespace η, then we can model a dataset archive as a set of 5-tuples $\langle s, p, o, \rho, \zeta\rangle$. Here, $\rho \in \mathcal{I}$ is the RDF identifier of the local revision of the triple in an RDF graph with label $g = l(\rho)$ (Table A.2). Conversely, $\zeta \in \mathcal{I}$ identifies a (global) dataset revision $j = rv(\zeta)$. Likewise, we overload the function $ts(\zeta)$ (defined originally in Table A.1) so that it returns the timestamp associated to the dataset revision identifier ζ. Last, we notice that the addition of a non-empty graph to a dataset archive generates two revisions: one for creating the graph, and one for populating it. A similar logic applies to graph deletion.

2.4 SPARQL

SPARQL 1.1 is the W3C standard language to query RDF data [74]. For the sake of brevity, we do not provide a rigorous definition of the syntax and semantics of SPARQL queries; instead we briefly introduce the syntax of a subset of SELECT queries and refer the reader to the official specification [74]. SPARQL is a graph-based language whose building blocks are triple patterns. A *triple pattern* $\hat{t}$ is a triple $\langle \hat{s}, \hat{p}, \hat{o}\rangle \in (\mathcal{I} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$, where $\mathcal{V}$ is a set of variables such that $(\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}) \cap \mathcal{V} = \emptyset$ (variables are always prefixed with ? or $). A *basic graph pattern* (abbreviated BGP) $\hat{G}$ is the conjunction of a set of triple patterns $\{\ \hat{t}_1\ .\ \hat{t}_2\ \ldots\ .\ \hat{t}_m\ \}$, e.g.,

$$\{\ ?s\ a\ :Person\ .\ ?s\ :nationality\ :France\ \}$$

When no named graph is specified, the SPARQL standard assumes that the BGP is matched against the *default graph* in the RDF dataset. Otherwise, for matches against specific graphs, SPARQL supports the syntax *GRAPH* $\bar{g}\ \{\hat{G}\}$, where $\bar{g} \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{V}$. In this paper we call this, a *named BGP* denoted by $\hat{G}_{\bar{g}}$. A SPARQL select query Q on an RDF dataset has the basic form "SELECT V (FROM NAMED $\bar{g}_1$ FROM NAMED $\bar{g}_2 \ldots$) WHERE $\{\hat{G}'\ \hat{G}''\ \ldots\ \hat{G}_{\bar{g}_1}\ \hat{G}_{\bar{g}_2} \ldots\}$, with projection variables $V \subset \mathcal{V}$. SPARQL supports named BGPs $\hat{G}_{\bar{g}}$ with variables $\bar{g} \in \mathcal{V}$. In some implementations [31, 76] the bindings for those

variables originate from the master graph G^M. The BGPs in the expression can contain FILTER conditions, be surrounded by OPTIONAL clauses, and be combined by means of UNION clauses.

2.5 Queries on Archives

Queries on graph/dataset archives may combine results coming from different revisions in the history of the data collection in order to answer an information need. The literature defines five types of queries on RDF archives [29, 78]. We illustrate them by means of our example graph archive from Figure A.1.

- **Version Materialization.** VM queries are standard queries run against a single revision, such as *what was the list of countries according to the UN at revision j?*
- **Delta Materialization.** DM queries are standard queries defined on a changeset $u_j = \langle u_j^+, u_j^- \rangle$, e.g., *which countries were added to the list at revision j?*
- **Version.** V queries ask for the revisions where a particular query yields results. An example of a V query is: *in which revisions j did USA and Cuba have diplomatic relationships?*
- **Cross-version.** CV queries result from the combination (e.g., via joins, unions, aggregations, differences, etc.) of the information from multiple revisions, e.g., *which of the current countries was not in the original list of UN members?*
- **Cross-delta.** CD queries result from the combination of the information from multiple sets of changes, e.g., *what are the revisions j with the largest number of UN member adhesions?*

Existing solutions differ in the types of queries they support. For example, Ostrich [78] provides native support for queries of types VM, DM, and V on single triple patterns, and can handle multiple triple patterns via integration with external query engines. Dydra [11], in contrast, has native support for all types of queries on BGPs of any size. Even though our examples use the revision number $rv(\rho)$ to identify a revision, some solutions may directly use the revision identifier ρ or the revision's commit time $ts(\rho)$. This depends on the system's data model.

3 Framework for the Evolution of RDF Data

This section proposes *RDFev*, a framework to understand the evolution of RDF data. The framework consists of a set of metrics and a software tool to calculate those metrics throughout the history of the data. The metrics quan-

tify the changes between two revisions of an RDF graph or dataset and can be categorized into two families: metrics for low-level changes, and metrics for high-level changes. Existing benchmarks, such as BEAR [29], focus on low-level changes, that is, additions and deletions of triples. This, however, may be of limited use to data maintainers, who may need to know the semantics of those changes, for instance, to understand whether additions are creating new entities or editing existing ones. On these grounds, we propose to quantify changes at the level of entities and object values, which we call high-level.

RDFev takes each version of an RDF dataset as an RDF dump in N-triples format (our implementation does not support multi-graph datasets and quads for the time being). The files must be provided in chronological order. *RDFev* then computes the different metrics for each consecutive pair of revisions. The tool is implemented in C++ and Python and uses the RocksBD[3] key-value store as storage and indexing backend. All metrics are originally defined for RDF graphs in the state of the art [29], and have been ported to RDF datasets in this paper. *RDFev*'s source code is available at our project website[4].

3.1 Low-level Changes

Low-level changes are changes at the triple level. Indicators for low-level changes focus on additions and deletions of triples and vocabulary elements. The vocabulary $\mathrm{Y}(D) \subset \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$ of an RDF dataset D is the set of all the terms occurring in triples of the dataset. Tracking changes in the number of triples rather than in the raw size of the RDF dumps is more informative for data analytics, as the latter option is sensitive to the serialization format. Moreover an increase in the vocabulary of a dataset can provide hints about the nature of the changes and the novelty of the data incorporated in a new revision. All metrics are defined by Fernández et al. [29] for pairs of revisions i, j with $j > i$.

Change ratio. The authors of BEAR [29] define the change ratio between two revisions i and j of an RDF graph G as

$$\delta_{i,j}(G) = \frac{|u_{i,j}^{+}| + |u_{i,j}^{-}|}{|G_i \cup G_j|}. \tag{A.1}$$

$\delta_{i,j}$ compares the size of the changes between two revisions w.r.t. the revisions' joint size. Large values for $\delta_{i,j}$ denote important changes in between

[3] http://rocksdb.org
[4] https://relweb.cs.aau.dk/rdfev

the revisions. For a more fine-grained analysis, Fernández et al. [29] also proposes the insertion and deletion ratios:

$$\delta^{+}_{i,j} = \frac{|u^{+}_{i,j}|}{|G_i|} \qquad \text{(A.2)} \qquad\qquad \delta^{-}_{i,j} = \frac{|u^{-}_{i,j}|}{|G_i|}. \qquad \text{(A.3)}$$

We now adapt these metrics for RDF datasets. For this purpose, we define the size of a dataset D as $sz(D) = \sum_{G \in D} |G|$ and the size of a dataset changeset U as $sz(U) = sz(U^+) + sz(U^-)$ with $sz(U^+) = \sum_{u \in U} |u^+|$ and $sz^-(U) = \sum_{u \in U} |u^-|$. With these definitions, the previous formulas can be ported to RDF datasets as follows:

$$\delta_{i,j}(D) = \frac{sz(U)}{\sum_{G \in D_i \cap D_j} |G_i \cup G_j| + \sum_{G \in D_i \triangle D_j} |G|} \qquad \text{(A.4)}$$

$$\delta^{+}_{i,j}(D) = \frac{sz(U^+)}{sz(D_i)} \qquad \text{(A.5)} \qquad\qquad \delta^{-}_{i,j}(D) = \frac{sz(U^-)}{sz(D_i)} \qquad \text{(A.6)}$$

Here, $D_i \triangle D_j$ denotes the symmetric difference between the sets of RDF graphs in revisions i and j.

Vocabulary dynamicity. The vocabulary dynamicity for two revisions i and j of an RDF graph is defined as [29]:

$$\text{vdyn}_{i,j}(G) = \frac{|\Upsilon(u_{i,j})|}{|\Upsilon(G_i) \cup \Upsilon(G_j)|} \qquad \text{(A.7)}$$

$\Upsilon(u_{i,j})$ is the set of vocabulary terms – IRIs, literals, or blank nodes – in the changeset $u_{i,j}$ (Table A.1). The literature also defines the vocabulary dynamicity for insertions (vdyn+$_{i,j}$) and deletions (vdyn-$_{i,j}$):

$$\text{vdyn+}_{i,j}(G) = \frac{|\Upsilon(u^{+}_{i,j})|}{|\Upsilon(G_i) \cup \Upsilon(G_j)|} \qquad \text{(A.8)}$$

$$\text{vdyn-}_{i,j}(G) = \frac{|\Upsilon(u^{-}_{i,j})|}{|\Upsilon(G_i) \cup \Upsilon(G_j)|}. \qquad \text{(A.9)}$$

The formulas are analogous for RDF datasets if we replace G by D and $u_{i,j}$ by $U_{i,j}$.

Growth ratio. The grow ratio is the ratio between the number of triples in two revisions i, j. It is calculated as follows for graphs and datasets:

$$\Gamma_{i,j}(G) = \frac{|G_j|}{|G_i|} \qquad \text{(A.10)} \qquad\qquad \Gamma_{i,j}(D) = \frac{sz(D_j)}{sz(D_i)}. \qquad \text{(A.11)}$$

3.2 High-level Changes

A high-level change confers semantics to a changeset. For example, if an update consists of the addition of triples about an unseen subject, we can interpret the triples as the addition of an entity to the dataset. High-level changes provide deeper insights about the development of an RDF dataset than low-level changes. In addition, they can be domain-dependent. Some approaches [63, 71] have proposed vocabularies to describe changesets in RDF data as high-level changes. Since our approach is oblivious to the domain of the data, we propose a set of metrics on domain-agnostic high-level changes.

Entity changes. RDF datasets describe real-world entities s by means of triples $\langle s, p, o\rangle$. Hence, an entity is a subject for the sake of this analysis. We define the metric *entity changes* between revisions i, j in an RDF graph as:

$$ec_{i,j}(G) = |\sigma_{i,j}(G)| = |\sigma^{+}_{i,j}(G) \cup \sigma^{-}_{i,j}(G)| \tag{A.12}$$

In the formula, $\sigma^{+}_{i,j}$ is the set of added entities, i.e., the subjects present in $\mathrm{Y}(G_j)$ but not in $\mathrm{Y}(G_i)$ (analogously the set of deleted entities $\sigma^{-}_{i,j}$ is defined by swapping the roles of i and j). This metric can easily be adapted to an RDF dataset D if we define $ec(G)$ (with no subscripts) as the number of different subjects in a graph G. It follows that,

$$ec_{i,j}(D) = \sum_{G \in D_i \cap D_j} ec_{i,j}(G) + \sum_{G \in D_i \triangle D_j} ec(G). \tag{A.13}$$

We also propose the *triple-to-entity-change score*, that is, the average number of triples that constitute a single entity change. It can be calculated as follows for RDF graphs:

$$ect_{i,j}(G) = \frac{|\langle s, p, o\rangle \in u^{+}_{i,j} \cup u^{-}_{i,j} : s \in \sigma_{i,j}(G)|}{ec_{i,j}(G)} \tag{A.14}$$

We port this metric to RDF datasets by first defining $\mathbf{U}^{+} = \bigcup_{u \in U^{+}} u$ and $\mathbf{U}^{-} = \bigcup_{u \in U^{-}} u$ and plugging them into the formula for $ect_{i,j}$:

$$ect_{i,j}(D) = \frac{|\langle s, p, o\rangle \in \mathbf{U}^{+}_{i,j} \cup \mathbf{U}^{-}_{i,j} : s \in \sigma_{i,j}(D)|}{ec_{i,j}(D)} \tag{A.15}$$

Object Updates and Orphan Object Additions/Deletions. An object update in a changeset $u_{i,j}$ is defined by the deletion of a triple $\langle s, p, o\rangle$ and the addition of a triple $\langle s, p, o'\rangle$ with $o \neq o'$. Once a triple in a changeset has been assigned to a high-level change, the triple is *consumed* and cannot be assigned

Low-level changes	Change ratio
	Insertion and deletion ratios
	Vocabulary dynamicity
	Growth ratio
High-level changes	Entity changes
	Triple-to-entity-change score
	Object updates
	Orphan object additions and deletions

Table A.3: RDFev's metrics

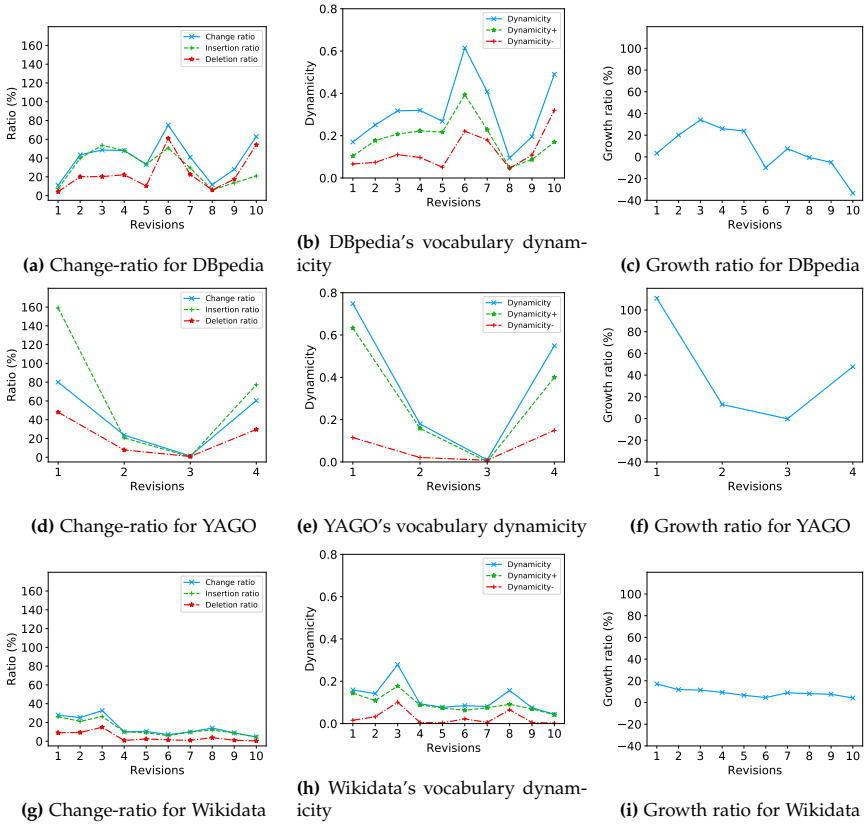
to any other high-level change. We define orphan object additions and deletions respectively as those triples $\langle s^+, p^+, o^+ \rangle \in u^+_{i,j}$ and $\langle s^-, p^-, o^- \rangle \in u^-_{i,j}$ that have not been consumed by any of the previous high-level changes. The dataset counterparts of these metrics for two revisions i, j can be calculated by summing the values for each of the graphs in $D_i \cap D_j$.
Table A.3 summarizes all the metrics defined by RDFev.

4 Evolution Analysis of RDF Datasets

Having introduced *RDFev*, we use it to conduct an analysis of the revision history of three large and publicly available RDF knowledge bases, namely YAGO, DBpedia, and Wikidata. The analysis resorts to the metrics defined in Sections 3.1 and 3.2 for every pair of consecutive revisions.

4.1 Data

We chose the YAGO [77], DBpedia [13], and Wikidata [26] knowledge bases for our analysis, because of their large size, dynamicity, and central role in the Linked Open Data initiative. We build an RDF graph archive by considering each release of the knowledge base as a revision. None of the datasets is provided as a monolithic file, instead they are divided into *themes*. These are subsets of triples of the same nature, e.g., triples with literal objects extracted with certain extraction methods. We thus focus on the most popular themes. For DBpedia we use the *mapping-based objects* and *mapping-based literals* themes, which are available from version 3.5 (2015) onwards. Additionally, we include the *instance-types* theme as well as the *ontology*. For YAGO, we use the knowledge base's core, namely, the themes *facts*, *meta facts*, *literal facts*, *date facts*, and *labels* available from version 2 (v.1.0 was not published in RDF). As for Wikidata, we use the *simple-statements* of the RDF Exports [2] in the period from 2014-05 to 2016-08. These dumps provide a clean subset



(a) Change-ratio for DBpedia

(b) DBpedia's vocabulary dynamicity

(c) Growth ratio for DBpedia

(d) Change-ratio for YAGO

(e) YAGO's vocabulary dynamicity

(f) Growth ratio for YAGO

(g) Change-ratio for Wikidata

(h) Wikidata's vocabulary dynamicity

(i) Growth ratio for Wikidata

Fig. A.3: Change ratio, vocabulary dynamicity, and growth ratio

of the dataset useful for applications that rely mainly on Wikidata's encyclopedic knowledge. All datasets are available for download in the RDFev's website `https://relweb.cs.aau.dk/rdfev`. Table A.4 maps revision numbers to releases for the sake of conciseness in the evolution analysis.

4.2 Low-level Evolution Analysis

Change ratio. Figures A.3a, A.3d and A.3g depict the evolution of the change, insertion, and deletion ratios for our experimental datasets. Up to the release 3.9 (rev. 5), DBpedia exhibits a steady growth with significantly more insertions than deletions. Minor releases such as 3.5.1 (rev. 1) are indeed minor in terms of low-level changes. Release 2015-04 (rev. 6) is an inflexion point not only in terms of naming scheme (see Table A.4): the deletion rate exceeds the insertion rate and subsequent revisions exhibit a tight difference between the rates. This suggests a major design shift in the construction of

Revision	DBpedia	YAGO	Wikidata
0	3.5	2s	2014-05-26
1	3.5.1	3.0.0	2014-08-04
2	3.6	3.0.1	2014-11-10
3	3.7	3.0.2	2015-02-23
4	3.8	3.1	2015-06-01
5	3.9		2015-08-17
6	2015-04		2015-10-12
7	2015-10		2015-12-28
8	2016-04		2016-03-28
9	2016-10		2016-06-21
10	2019-08		2016-08-01

Table A.4: Datasets revision mapping

DBpedia from revision 6.

As for YAGO, the evolution reflects a different release cycle. There is a clear distinction between major releases (3.0.0 and 3.1, i.e., rev. 1 and 4) and minor releases (3.0.1 and 3.0.2, i.e., rev. 2 and 3). The magnitude of the changes in major releases is significantly higher for YAGO than for any DBpedia release. Minor versions seem to be mostly focused on corrections, with a low number of changes.

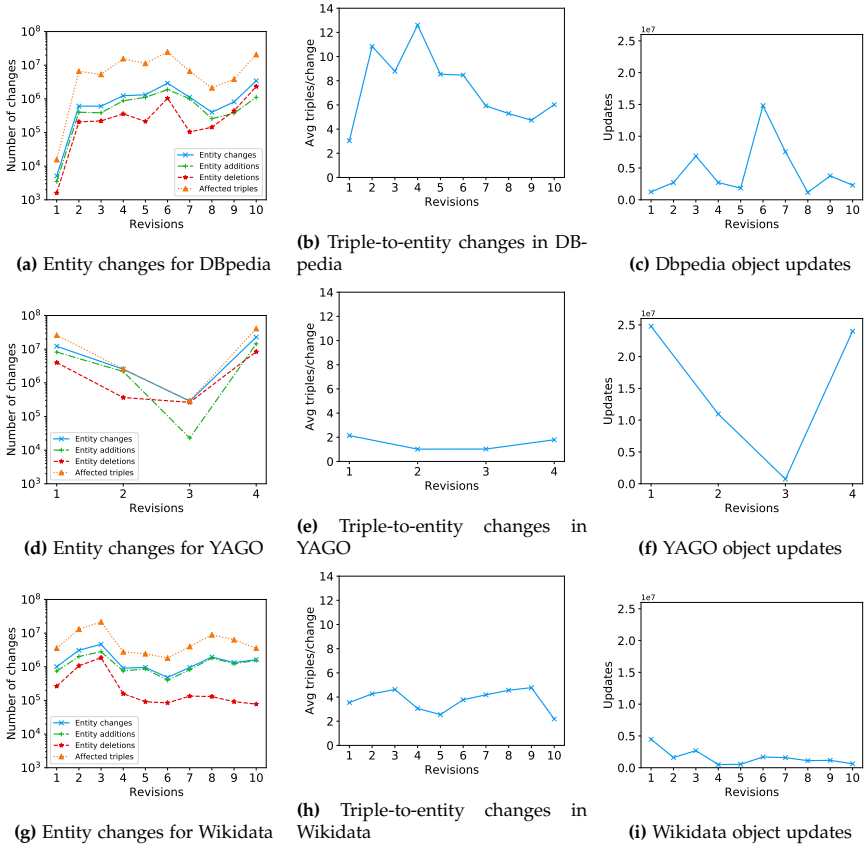
Contrary to the other datasets, Wikidata shows a slowly decreasing change ratio that fluctuates between 5% (rev. 10) and 33% (rev. 3) within the studied period of 2 years.

Vocabulary dynamicity. As shown in Figures A.3b, A.3e, and A.3h, the vocabulary dynamicity is, not surprisingly, correlated with the change ratio. Nevertheless, the vocabulary dynamicity between releases 3.9 and 2015-14 (rev. 5 and 6) in DBpedia did not decrease. This suggests that DBpedia 2015-04 contained more entities, but fewer – presumably noisy – triples about those entities. The major releases of YAGO (rev. 1 and 4) show a notably higher vocabulary dynamicity than the minor releases. As for Wikidata, slight spikes in dynamicity can be observed at revisions 4 and 9, however this metric remains relatively low in Wikidata compared to the others bases.

Growth ratio. Figures A.3c, A.3f, and A.3i depict the growth ratio of our experimental datasets. In all cases, this metric is mainly positive with low values for minor revisions. As pointed out by the change ratio, the 2015-04 release in DBpedia is remarkable as the dataset shrank and was succeeded by more conservative growth ratios. This may suggest that recent DBpedia releases are more curated. We observe that YAGO's growth ratio is significantly larger for major versions. This is especially true for the 3.0.0 (rev. 1)

release that doubled the size of the knowledge base.

4.3 High-level Evolution Analysis



(a) Entity changes for DBpedia

(b) Triple-to-entity changes in DBpedia

(c) Dbpedia object updates

(d) Entity changes for YAGO

(e) Triple-to-entity changes in YAGO

(f) YAGO object updates

(g) Entity changes for Wikidata

(h) Triple-to-entity changes in Wikidata

(i) Wikidata object updates

Fig. A.4: Entity changes and object updates

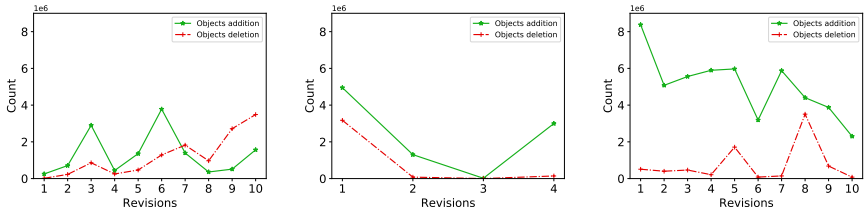
Entity changes

Figures A.4a, A.4d, and A.4g illustrate the evolution of the entity changes, additions, and deletions for DBpedia, YAGO and Wikidata. We also show the number of triples used to define these high-level changes (labeled as *affected triples*). We observe a stable behavior for these metrics in DBpedia except for the minor release 3.5.1 (rev. 1). Entity changes in Wikidata also display a monotonic behavior, even though the deletion rate tends to decrease from rev. 4. In YAGO, the number of entity changes peaks for the major revisions (rev.

1 and 4), and is one order of magnitude larger than for minor revisions. The minor release 3.0.2 (rev. 3) shows the lowest number of additions, whereas deletions remain stable w.r.t release 3.0.1 (rev. 2). This suggests that these two minor revisions focused on improving the information extraction process, which removed a large number of noisy entities.

Figure A.4b shows the triple-to-entity-change score in DBpedia. Before the 2015-14 release, this metric fluctuates between 2 and 12 triples without any apparent pattern. Conversely subsequent releases show a decline, which suggests a change in the extraction strategies for the descriptions of entities. The same cannot be said about YAGO and Wikidata (Figures A.4e and A.4h), where values for this metric are significantly lower than for DBpedia, and remain almost constant. This suggests that minor releases in YAGO improved the strategy to extract entities, but did not change much the amount of extracted triples per entity.

Object Updates and Orphan Object Additions/Deletions



(a) Dbpedia **(b)** YAGO **(c)** Wikidata

Fig. A.5: Orphan object additions and deletions

We present the evolution of the number of object updates for our experimental datasets in Figures A.4c, A.4f, and A.4i. For DBpedia, the curve is consistent with the change ratio (Figure A.3a). In addition to a drop in size, the 2015-04 release also shows the highest number of object updates, which corroborates the presence of a drastic redesign of the dataset.

The results for YAGO are depicted in Figure A.4f, where we see larger numbers of object updates compared to major releases in DBpedia. This is consistent with the previous results that show that YAGO goes through bigger changes between releases. The same trends are observed for the number of orphan object additions and deletions in Figures A.5a and A.5b. Compared to the other two datasets, Wikidata's number of object updates, shown in Figure A.4i, is much lower and constant throughout the stream of revisions.

Finally, we remark that in YAGO and DBpedia, object updates are 4.8 and 1.8 times more frequent than orphan additions and deletions. This entails

that the bulk of editions in these knowledge bases aims at updating existing object values. This behavior contrasts with Wikidata, where orphan object updates are 3.7 times more common than proper object updates. As depicted in Figure A.5c, Wikidata exhibits many more orphan object updates than the other knowledge bases. Moreover, orphan object additions are 19 times more common than orphan object deletions.

4.4 Conclusion

In this section we have conducted a study of the evolution of three large RDF knowledge bases using our proposed framework *RDFev*, which resorts to a domain-agnostic analysis from two perspectives: At the low-level it studies the dynamics of triples and vocabulary terms across different versions of an RDF dataset, whereas at the high-level it measures how those low-level changes translate into updates to the entities described in the experimental datasets. All in all, we have identified different patterns of evolution. On the one hand, Wikidata exhibits a stable release cycle in the studied period, as our metrics did not exhibit big fluctuations from release to release. On the other hand, YAGO and DBpedia have a release cycle that distinguishes between minor and major releases. Major releases are characterized by a large number of updates in the knowledge base and may not necessarily increase its size. Conversely, minor releases incur in at least one order of magnitude fewer changes than major releases and seem to focus on improving the quality of the knowledge base, for instance, by being more conservative in the number of triple and entity additions. Unlike YAGO, DBpedia has shown decreases in size across releases. We argue that an effective solution for large-scale RDF archiving should be able to adapt to different patterns of evolution.

5 Survey of RDF Archiving Solutions

We structure this section in three parts. Section 5.1 surveys the existing engines for RDF archiving and discusses their strengths and weaknesses. Section 5.2 presents the languages and SPARQL extensions to express queries on RDF archives. Finally, Section 5.3 introduces various endeavors on analysis and benchmarking of RDF archives.

5.1 RDF Archiving Systems

There are plenty of systems to store and query the history of an RDF dataset. Except for a few approaches [11, 12, 36, 81], most available systems support archiving of a single RDF graph. Ostrich [78], for instance, manages quads of the form $\langle s, p, o, rv(\rho)\rangle$. Other solutions do not support revision numbers

and use the ρ-component $\rho \in \mathcal{I}$ to model temporal metadata such as insertion, deletion, and validity timestamps for triples [34]. In this paper we make a distinction between insertion/deletion timestamps for triples and validity intervals. While the former are unlikely to change, the latter are subject to modifications because they constitute domain information, e.g., the validity of a marriage statement. This is why the general data model introduced in Section 2 only associates revision numbers and commit timestamps to the fourth component ρ, whereas other types of metadata are still attached to the graph label $g = l(\rho)$. We summarize the architectural spectrum of RDF archiving systems in Table A.5 where we characterize the state-of-the-art approaches according to the following criteria:

- **Storage paradigm.** The storage paradigm is probably the most important feature as it shapes the system's architecture. We identify three main paradigms in the literature [29], namely independent copies (IC), change-based (CB), and timestamp-based (TB). Some systems [78] may fall within multiple categories, whereas Quit Store [12] implements a fragment-based (FB) paradigm.
- **Data model.** It can be quads or 5-tuples with different semantics for the fourth and fifth component.
- **Full BGPs.** This feature determines whether the system supports BGPs with a single triple pattern or full BGPs with an unbounded number of triple patterns and filter conditions.
- **Query types.** This criterion lists the types of queries on RDF archives (see Section 2.5) natively supported by the solution.
- **Branch & tags.** It defines whether the system supports branching and tagging as in classical version control systems.
- **Multi-graph.** This feature determines if the system supports archiving of the history of multi-graph RDF datasets.
- **Concurrent updates.** This criterion determines whether the system supports concurrent updates. This is defined regardless of whether conflict management is done manually or automatically.
- **Source available.** We also specify whether the system's source code is available for download and is usable, that is, whether it can be compiled and run in modern platforms.

In the following, we discuss further details of the state-of-the-art systems, grouped by their storage paradigms.

Independent Copies Systems

In an IC-like approach, each revision D_i of a dataset archive $\mathcal{A} = \{D_1, D_2, \ldots, D_n\}$ is fully stored as an independent RDF dataset. IC approaches shine at the execution of VM and CV queries as they do not in-

	Storage paradigm	Data model	Full BGPs	Queries
Dydra [11]	TB	5-tuples	+	all
Ostrich [78]	IC/CB/TB	quads	+[a]	VM, DM, V
QuitStore [12]	FB	5-tuples	+	all
RDF-TX [34]	TB	quads	+	all
R43ples [36]	CB	5-tuples[b]	+	all
R&WBase [72]	CB	quads	+	all
RBDMS [46]	CB	quads	+	all
SemVersion [81]	IC	5-tuples[b]	-	VM, DM
Stardog [3]	CB	5-tuples	+	all
v-RDFCSA [23]	TB	quads	-	VM, DM, V
x-RDF-3X [59]	TB	quads	+	VM, V

[a] Full BGP support is possible via integration with the Comunica query engine
[b] Graph local revisions

	Branch & tags	Multi-graph	Concurrent Updates	Source available
Dydra [11]	-	+	-	-
Ostrich [78]	-	-	-	+
QuitStore [12]	+	+	+	+
RDF-TX [34]	-	-	-	-
R43ples [36]	+	+	+	+[c]
R&WBase [72]	+	-	+	+
RBDMS [46]	+	-	+	-
SemVersion [81]	+	-	+	-
Stardog [3]	+	-	-	
v-RDFCSA [23]	-	-	-	-
x-RDF-3X [59]	-	-	-	+[d]

[c] It needs modifications to have the console client running and working [d] Old source code

Table A.5: Existing RDF archiving systems

cur any materialization cost for such types of queries. Conversely, IC systems are inefficient in terms of disk usage. For this reason they have mainly been proposed for small datasets or schema version control [61, 81]. SemVersion [81], for instance, is a system that offers similar functionalities as classical version control systems (e.g., CVS or SVN), with support for multiple RDF graphs and branching. Logically, SemVersion supports 5-tuples of the form $\langle s, p, o, l(\rho), rv(\rho)\rangle$, in other words, revision numbers are local to each RDF graph. This makes it difficult to track the addition or deletion of named graphs in the history of the dataset. Lastly, SemVersion provides an HTTP interface to submit updates either as RDF graphs or as changesets. Despite this flexibility, new revisions are always stored as independent copies. This makes its disk-space consumption prohibitive for large datasets like the ones studied in this paper.

Change-based Systems

Solutions based on the CB paradigm store a subset $\hat{\mathcal{A}} \subset \mathcal{A}$ of the revisions of a dataset archive as independent copies or *snapshots*. On the contrary, all the intermediate revisions D_j ($p < j < q$) between two snapshots D_p and D_q, are stored as deltas or changesets U_j. The sequence of revisions stored as changesets between two snapshots is called a *delta chain*. CB systems are convenient for DM and CD queries. Besides, they are obviously considerably more storage-efficient than IC solutions. Their weakness lies in the high materialization cost for VM and CV queries, particularly for long delta chains.

R&WBase [72] is an archiving system that provides Git-like distributed version control with support for merging, branching, tagging, and concurrent updates with manual conflict resolution on top of a classical SPARQL endpoint. R&WBase supports all types of archive queries on full BGPs. The system uses the PROV-Ontology (PROV-O) [24] to model the metadata (e.g., timestamps, parent branches) about the updates of a single RDF graph. An update u_i generates two new named graphs G_g^{i+}, G_g^{i-} containing the added and deleted triples at revision i. Revisions can be materialized by processing the delta chain back to the initial snapshot, and they can be referenced via aliases called *virtual named graphs*. In the same spirit, tags and branches are implemented as aliases of a particular revision. R&WBase has inspired the design of R43ples [36]. Unlike the former, R43ples can version multiple graphs, although revision numbers are not defined at the dataset level, i.e., each graph manages its own history. Moreover, the system extends SPARQL with the clause *REVISION* j ($j \in \mathcal{N}$) used in conjunction with the GRAPH clause to match a BGP against a specific revision of a graph. Last, the approach presented by Dong-hyuk et al. [46] relies on an RDBMS to store snapshots and deltas of an RDF graph archive with support for branching and tagging. Its major drawback is the lack of support for SPARQL queries:

while it supports all the types of queries introduced in Section 2.5, they must be formulated in SQL, which can be very tedious for complex queries.

Stardog [3] is a commercial RDF data store with support for dataset snapshots, tags, and full SPARQL support. Unlike R43ples, Stardog keeps track of the global history of a dataset, hence its logical model consists of 5-tuples of the form $\langle s, p, o, l(\rho), \zeta \rangle$ (i.e., metadata is stored at the dataset level). While the details of Stardog's internal architecture are not public, the documentation[5] suggests a CB paradigm with a relational database backend.

Timestamp-based Systems

TB solutions store triples with their temporal metadata, such as domain temporal validity intervals or insertion/deletion timestamps. Like in CB solutions, revisions must be materialized at a high cost for VM and CV queries. V queries are usually better supported, whereas the efficiency of materializing deltas depends on the system's indexing strategies.

x-RDF-3X [59] is a system based on the RDF-3X [58] engine. Logically x-RDF-3X supports quads of the form $\langle s, p, o, \rho \rangle$ where $\rho \in \mathcal{I}$ is associated to all the revisions where the triple was present as well as to all addition and deletion timestamps. The system is a fully-fledged query engine optimized for highly concurrent updates with support for snapshot isolation in transactions. However, x-RDF-3X does not support versioning for multiple graphs, neither branching nor tagging.

Dydra [11] is a TB archiving system that supports archiving of multigraph datasets. Logically, Dydra stores 5-tuples of the form $\langle s, p, o, l(\rho), \zeta \rangle$, that is, revision metadata lies at the dataset level. In its physical design, Dydra indexes quads $\langle s, p, o, l(\rho) \rangle$ and associates them to visibility maps and creation/deletion timestamps that determine the revisions and points in time where the quad was present. The system relies on six indexes – *gspo*, *gpos*, *gosp*, *spog*, *posg*, and *ospg* implemented as B+ trees – to support arbitrary SPARQL queries ($g = l(\rho)$ is the graph label). Moreover, Dydra extends the query language with the clause *REVISION* x, where x can be a variable or a constant. This clause instructs the query engine to match a BGP against the contents of the data sources bound to x, namely a single database revision ζ, or a dataset changeset $U_{j,k}$. A revision can be identified by its IRI ζ, its revision number $rv(\zeta)$ or by a timestamp τ'. The latter case matches the revision ζ with the largest timestamp $\tau = ts(\zeta)$ such that $\tau \leq \tau'$. Alas, Dydra's source is not available for download and use.

RDF-TX [34] supports single RDF graphs and uses a multiversion B-tree (MVBT) to index triples and their time metadata (insertion and deletion timestamps). An MVBT is actually a forest where each tree indexes the triples

[5] https://github.com/stardog-union/stardog-examples/tree/d7ac8b562ecd0346306a266d9cc28063fde7edf2/examples/cli/versioning

that were inserted within a time interval. RDF-TX implements an efficient compression scheme for MVBTs, and proposes SPARQL-T, a SPARQL extension that adds a fourth component $\hat{g}$ to BGPs. This component can match only time objects τ of type timestamp or time interval. The attributes of such objects can be queried via built-in functions, e.g., $year(\tau)$. While RDF-TX offers interval semantics at the query level, it stores only timestamps.

v-RDFCSA [23] is a lightweight and storage-efficient TB approach that relies on suffix-array encoding [19] for efficient storage with basic retrieval capabilities (much in the spirit of HDT [27]). Each triple is associated to a bitsequence of length equals the number of revisions in the archive. That is, v-RDFCSA logically stores quads of the form $\langle s, p, o, rv(\rho)\rangle$. Its query functionalities are limited since it supports only VM, DM, and V queries on single triple patterns.

Hybrid and Fragment-based Systems

Some approaches can combine the strengths of the different storage paradigms. One example is Ostrich [78], which borrows inspirations from IC, CB, and TB systems. Logically, Ostrich supports quads of the form $\langle s, p, o, rv(\rho)\rangle$. Physically, it stores snapshots of an RDF graph using HDT [27] as serialization format. Delta chains are stored as B+ trees timestamped with revision numbers in a TB-fashion. These delta chains are redundant, i.e., each revision in the chain is stored as a changeset containing the changes w.r.t. the latest snapshot – and not the previous revision as proposed by Dong-hyuk et al. [46]. Ostrich alleviates the cost of redundancy using compression. All these design features make Ostrich query and space efficient, however its functionalities are limited. Its current implementation does not support more than one (initial) snapshot and a single delta chain, i.e., all revisions except for revision 0 are stored as changesets of the form $u_{0,i}$. Multi-graph archiving as well as branching/tagging are not possible. Moreover, the system's querying capabilities are restricted to VM, DM, and V queries on single triple patterns. Support for full BGPs is possible via integration with the Comunica query engine[6].

Like R43ples [36], Quit Store [12] provides collaborative Git-like version control for multi-graph RDF datasets, and uses PROV-O for metadata management. Unlike R43ples, Quit Store provides a global view of the evolution of a dataset, i.e., each commit to a graph generates a new dataset revision. The latest revision is always materialized in an in-memory quad store. Quit-Store is implemented in Python with RDFlib and provides full support for SPARQL 1.1. The dataset history (RDF graphs, commit tree, etc.) is physically stored in text files (i.e. N-quads files resp. N-triples files in the latest implementation) and is accessible via a SPARQL endpoint on a set of virtual

[6] https://github.com/rdfostrich/comunica-actor-init-sparql-ostrich

graphs. However, the system only stores snapshots of the modified files in the spirit of fragment-based storage. Quit Store is tailored for collaborative construction of RDF datasets, but its high memory requirements make it unsuitable as an archiving backend. As discussed in Section 7, fully-fledged RDF archiving can provide a backend for this type of applications

5.2 Languages to Query RDF Archives

Multiple research endeavors have proposed alternatives to succinctly formulate queries on RDF archives. The BEAR benchmark [29] uses AnQL to express the query types described in Section 2.5. AnQL [82] is a SPARQL extension based on quad patterns $\langle \hat{s}, \hat{p}, \hat{o}, \hat{g} \rangle$. AnQL is more general than SPARQL-T (proposed by RDF-TX [34]) because the $\hat{g}$-component can be bound to any term $u \in \mathcal{I} \cup \mathcal{L}$ (not only time objects). For instance, a DM query asking for the countries added at revision 1 to our example RDF dataset from Figure A.1 could be written as follows:

```
SELECT * WHERE {
  { (?x a :Country):  [1] } MINUS
  { (?x a :Country):  [0] }
}
```

T-SPARQL [35] is a SPARQL extension inspired by the query language TSQL2 [75]. T-SPARQL allows for the annotation of groups of triple patterns with constraints on temporal validity and commit time, i.e., it supports both time-intervals and timestamps as time objects. T-SPARQL defines several comparison operators between time objects, namely *equality*, *precedes*, *overlaps*, *meets*, and *contains*. Similar extensions [16, 66] also offer support for geospatial data.

SPARQ-LTL [30] is a SPARQL extension that makes two assumptions, namely that (i) triples are annotated with revision numbers, and (ii) revisions are accessible as named graphs. When no revision is specified, BGPs are iteratively matched against *every* revision. A set of clauses on BGPs can instruct the SPARQL engine to match a BGP against other revisions at each iteration. For instance the clause *PAST* in the expression *PAST*{ q } MINUS { q } with $q = \langle ?x\ a\ :Country \rangle$ will bind variable ?x to all the countries that were ever deleted from the RDF dataset, even if they were later added.

5.3 Benchmarks and Tools for RDF Archives

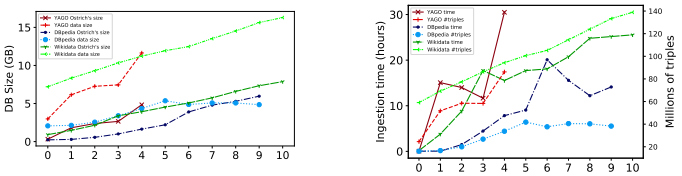
BEAR [29] is the state-of-the-art benchmark for RDF archive solutions. The benchmark provides three real-world RDF graphs (called BEAR-A, BEAR-B, and BEAR-C) with their corresponding history, as well as a set of VM, DM, and V queries on those histories. In addition, BEAR allows system designers to compare their solutions with baseline systems based on different storage

strategies (IC, CB, TB, and hybrids TB/CB, IC/CB) and platforms (Jena TDB and HDT). Despite its multiple functionalities and its dominant position in the domain, BEAR has some limitations: (i) It assumes single-graph RDF datasets; (ii) it does not support CV and CD queries, moreover VM, DM, and V queries are defined on single triple patterns; and (iii) it cannot simulate datasets of arbitrary size and query workloads.

EvoGen [51] tackles the latter limitation by extending the Lehigh University Benchmark (LUBM) [37] to a setting where both the schema and the data evolve. Users can not only control the size and frequency of that evolution, but can also define customized query workloads. EvoGen supports all the types of queries on archives presented in Section 2.5 on multiple triple patterns.

A recent approach [79] proposes to use FCA (Formal Concept Analysis) and several data fusion techniques to produce summaries of the evolution of entities across different revisions of an RDF archive. A summary can, for instance, describe groups of subjects with common properties that change over time. Such summaries are of great interest for data maintainers as they convey edition patterns in RDF data through time.

6 Evaluation of the Related Work



(a) Disk-space usage **(b)** Ingestion time

Fig. A.6: Ostrich's performance on multiple revisions of DBpedia and YAGO

In this section, we conduct an evaluation of the state-of-the-art RDF archiving engines. We first provide a global analysis of the systems' functionalities in Section 6.1. Section 6.2 then provides a performance evaluation of Ostrich (the only testable solution) on our experimental RDF archives from Table A.4. This evaluation is complementary to the Ostrich's evaluation on BEAR (available in [78]), as it shows the performance of the system in three real-world large RDF datasets.

6.1 Functionality Analysis

As depicted in Table A.5, existing RDF archiving solutions differ greatly in design and functionality. The first works [22, 59, 81] offered mostly storage of old revisions and support for basic VM queries. Consequently, subsequent efforts focused on extending the query capabilities and allowing for concurrent updates as in standard version control systems [12, 36, 46, 72]. Such solutions are attractive for data maintainers in collaborative projects, however they still lack scalability, e.g., they cannot handle large datasets and changesets, besides conflict management is still delegated to users. More recent works [23, 78] have therefore focused on improving storage and querying performance, alas, at the expense of features. For example, Ostrich [78] is limited to a single snapshot and delta chain. In addition to the limitations in functionality, Table A.5 shows that most of the existing systems are not available because their source code is not published. While this still leaves us with Ostrich [78], Quit Store [12], R&WBase [72], R43ples [36] and x-RDF-3X as testable solutions, only [78] was able to run on our experimental datasets. To carry out a fair comparison with the other systems, we tried Quit Store in the *persistence* mode, which ingests the data graphs into main memory at startup – allowing us to measure ingestion times. Unfortunately, the system crashes for all our experimental datasets[7]. We also tested Quit Store in its default lazy loading mode, which loads the data into main memory at query time. This option throws a Python *MemoryError* for our experimental queries. In regards to R43ples, we had to modify its source code to handle large files[8]. Despite this change, the system could not ingest a single revision of DBpedia after four days of execution. R&WBase, on the other hand, accepts updates only through a SPARQL endpoint, which cannot handle the millions of update statements required to ingest the changesets. Finally, x-RDF-3X's source code does not compile out of the box in modern platforms, and even after successful compilation, it is unable to ingest one DBpedia changeset.

6.2 Performance Analysis

We evaluate the performance of Ostrich on our experimental datasets in terms of storage space, ingestion time – the time to generate a new revision from an input changeset – and query response time. The changesets were computed with RDFev from the different versions of DBpedia, YAGO, and Wikidata (Table A.4). All the experiments were run on a server with a 4-core CPU (Intel Xeon E5-2680 v3@2.50GHz) and 64 GB of RAM.

[7] The Python interpreter reports a *UnboundLocalError*.
[8] The code creates an array that exceeds the maximal array size in Java.

Triple Patterns	DBpedia VM	DBpedia V	DBpedia DM	YAGO VM	YAGO V	YAGO DM
? p ?	92.81(0)	118.64(0)	91.78(0)	2.9(3)	- (5)	1.82(3)
? <top p> o	112.81(0)	283.59(0)	130.88(0)	35.08(2)	69.65(4)	137.16(4)
? p o	96.74(0)	92.04(0)	91.93(0)	2.38(4)	2.35(4)	2.35(2)
s p ?	94.99(0)	91.67(0)	92.81(0)	2.42(2)	2.36(3)	2.41(1)

Triple Patterns	Wikidata VM	Wikidata V	Wikidata DM
? p ?	281.41(0)	302.26(0)	303.73(0)
? <top p> o	347.06(0)	499.77(0)	285.02(0)
? p o	284.74(0)	281.4(0)	281.2(0)

Table A.6: Ostrich's Query Performance in seconds

Storage space. Figure A.6a shows the amount of storage space (in GB) used by Ostrich for the selected revisions of our experimental datasets. We provide the raw sizes of the RDF dumps of each revision for reference. Storing each version of YAGO separately requires 36 GB, while Ostrich uses only 4.84 GB. For DBpedia compression goes from 39 GB to 5.96 GB. As for Wikidata, it takes 131 GB to stores the raw files, but only 7.88 GB with Ostrich. This yields a compression rate of 87% for YAGO, 84% for DBpedia and 94% for Wikidata. This space efficiency is the result of using HDT [27] for snapshot storage, as well as compression for the delta chains.

Ingestion time. Figure A.6b shows Ostrich's ingestion times. We also provide the number of triples of each revision as reference. The results suggest that this measure depends both on the changeset size, and the length of the delta chain. However, the latter factor becomes more prominent as the length of the delta chain increases. For example, we can observe that Ostrich requires ∼22 hours to ingest revision 9 of DBpedia (2.43M added and 2.46M deleted triples) while it takes only ∼14 hours to ingest revision 5 (12.85M added and 5.95M deleted triples). This confirms the trends observed in [78] where ingestion time increases linearly with the number of revisions. This is explained by the fact that Ostrich stores the i-th revision of an archive as a changeset of the form $u_{0,i}$. In consequence, Ostrich's changesets are constructed from the triples in all previous revisions, and can only grow in size. This fact makes it unsuitable for very long histories.

Query runtime. We run Ostrich on 100 randomly generated VM, V, and DM queries on our experimental datasets. Ostrich does not support queries on full BGPs natively, hence the queries consisted of single triple patterns of the most common forms, namely ⟨ ?, p, ? ⟩, ⟨ s, p, ? ⟩, and ⟨ ?, p, o ⟩ in equal numbers. We also considered queries ⟨ ?, <top p>, o ⟩, where <*top*

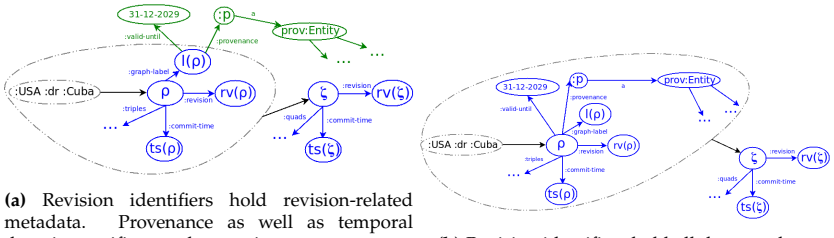
p> corresponds to the top 5 most common predicates in the dataset. Revision numbers for all queries were also randomly generated. Table A.6 shows Ostrich's average runtime in seconds for the different types of queries. We set a timeout of 1 hour for each query, and show the number of timeouts in parentheses next to the runtime, which excludes queries that timed out. We observe that Ostrich is roughly one order of magnitude faster on YAGO than on DBpedia and Wikidata. To further understand the factors that impact Ostrich's runtime, we computed the Spearman correlation score between Ostrich's query runtime and a set of features relevant to query execution. These features include the length of the delta chain, the average size of the relevant changesets, the size of the initial revision, the average number of deleted and added triples in the changesets, and the number of query results. The results show that the most correlated features are the length of the delta chain, the standard deviation of the changeset size, and the average number of deleted triples. This suggests that Ostrich's runtime performance will degrade as the history of the archive grows and that massive deletions actually aggravate that phenomenon. Finally, we observe some timeouts in YAGO in contrast to DBpedia and Wikidata. We believe this is mainly caused by the sizes of the changesets, which are on average of 3.72GB for YAGO, versus 2.09GB for DBpedia and 1.86GB for Wikidata. YAGO's changesets at revisions 1 and 4 are very large as shown in Section 4.

7 Towards Fully-fledged RDF Archiving

We now build upon the findings from previous sections to derive a set of lessons towards the design of a scalable fully-fledged solution for archiving of large RDF datasets. We structure this section in two parts. Section 7.1 discusses the most important functionalities that such a solution may offer, whereas Section 7.2 discusses the algorithmic and design challenges of providing those functionalities.

7.1 Functionalities

Global and local history. Our survey in Section 5.1 shows that R43ples [36] and Quit Store [12] are the only available solutions that support both archiving of the local and joint (global) history of multiple RDF graphs. We argue that such a feature is vital for proper RDF archiving: It is not only of great value for distributed version control in collaborative projects, but can also be useful for the users and maintainers of data warehouses. Conversely, existing solutions are strictly focused on distributed version control and their Git-based architectures make them unsuitable to archive the releases of large datasets such as YAGO, DBpedia, or Wikidata as explained in Section 6. From



(a) Revision identifiers hold revision-related metadata. Provenance as well as temporal domain-specific metadata are in green.

(b) Revision identifiers hold all the metadata.

Fig. A.7: Two logical models to handle metadata in RDF archives. The namespace *prov:* corresponds to the PROV-O namespace.

an engineering and algorithmic perspective, this implies to redesign RDF solutions to work with 5-tuples instead of triples. We discuss the technical challenges of such requirement in Section 7.2.

Temporal domain-specific vs. revision metadata. Systems and language extensions for queries with time constraints [34, 35], treat both domain-specific metadata (e.g., triple validity intervals) and revision-related annotations (e.g., revision numbers) in the same way. We highlight, however, that revision metadata is immutable and should therefore be logically placed at a different level. In this line of thought we propose to associate revision metadata for graphs and datasets, e.g., commit time, revision numbers, or branching & tagging information, to the local and global revision identifiers ρ and ζ, whereas depending on the application, domain-specific time objects could be modeled either as statements about the revisions or as statements about the graph labels $g = l(\rho)$. The former alternative enforces the same temporal domain-specific metadata to all the triples added in a changeset, whereas the latter option makes sense if all the triples with the same graph label are supposed to share the same domain-specific information – which can still be edited by another changeset on the master graph. We depict both alternatives in Figure A.7. We remark that such associations are only defined at the logical level.

Provenance. Revision metadata is part of the history of a triple within a dataset. Instead, its complete history is given by its workflow provenance. The W3C offers the PROV-O ontology [24] to model the history of a triple from its sources to its current state in an RDF dataset. Pretty much like temporal domain-specific metadata, provenance metadata can be logically linked to either the (local or global) revision identifiers or to the graph labels (Figure A.7). This depends on whether we want to define provenance for changesets because the triples added to an RDF graph may have different provenance workflows. A hybrid approach could associate a default prove-

nance history to a graph and use the revision identifiers to override or extend that default history for new triples. Moreover, the global revision identifier ζ provides an additional level of metadata that allow us to model the provenance of a dataset changeset.

Concurrent updates & modularity. We can group the existing state-of-the-art solutions in three categories regarding their support for concurrent updates, namely (i) solutions with limited or no support for concurrent updates [11, 23, 34, 67, 78], (ii) solutions inspired by version control systems such as Git [12, 36, 46, 72, 81], and (iii) approaches with full support for highly concurrent updates [59]. Git-like solutions are particularly interesting for collaborative efforts such as DBpedia, because it is feasible to delegate users the task of conflict management. Conversely, fully automatically constructed KBs such as NELL [21] or data-intensive (e.g., streaming) applications may need the features of solutions such as x-RDF-3X [59]. Consequently, we propose a modular design that separates the concurrency layer from the storage backend. Such a middleware could take care of enforcing a consistency model for concurrent commits either automatically or via user-based conflict management. The layer could also manage the additional metadata for features such as branching and tagging. In that light, collaborative version control systems for RDF [12, 36, 72] become an application of fully-fledged RDF archiving.

Formats for publication and querying. A fully functional archiving solution should support the most popular RDF serialization formats for data ingestion and dumping. For metadata enhanced RDF, this should include support for N-quads, singleton properties, and RDF-star. Among those, RDF-star [40] is the only one that can natively support multiple levels of metadata (still in a very verbose fashion). For example RDF-star could serialize the tuple $\langle$*:USA*, *:dr*, *:Cuba*, $\rho, \zeta\rangle$ with graph label (*:gl*) $l(\rho)$ = *:UN* and global timestamp (*:ts*) $ts(\zeta)$ =*2020-07-09* as follows:

<<<:USA :dr :Cuba> :gl :UN> :ts "2020-07-09"^^xsd:date>

The authors of [40] propose this serialization as part of the Turtle-star format. Moreover, they propose SPARQL-star that allows for nested triple patterns. While SPARQL-star enables the definition of metadata constraints at different levels, a fully archive-compliant language could offer further syntactic sugar such as the clauses *REVISION* [11, 36] or *DELTA* to bind the variables of a BGP to the data in particular revisions or deltas. We propose to build such an archive-compliant language upon SPARQL-star.

Support for different types of archive queries. Most of the studied archiving systems can answer all the query types defined in the literature of RDF

archives [29, 78]. That said, more complex queries such as CD and CV queries, or queries on full BGPs are sometimes supported via query middlewares and external libraries [12, 78]. We endorse this design philosophy because it eases modularity. Existing applications in mining archives [45, 64, 65] already benefit from support for V, VM, and DM queries on single triple patterns. By guaranteeing scalable runtime for such queries, we can indirectly improve the runtime of more complex queries. Further optimizations can be achieved by proper query planning.

7.2 Challenges

Trade-offs on storage, query runtime, and ingestion time. RDF archiving differs from standard RDF management in an even more imperative need for scalability, in particular storage efficiency. As shown by Taelman et al. [78], the existing storage paradigms shine at different types of queries. Hence, supporting arbitrary queries while being storage-efficient requires the best from the IC, CB, FB, and TB philosophies. A hybrid approach, however, will inevitably be more complex and introduce further parameters and trade-offs. The authors of Ostrich [78], for instance, chose to benefit faster version materialization via redundant deltas at the expense of larger ingestion times. Users requiring shorter ingestion times could, on the other hand, opt for non-redundant changesets, or lazy non-asynchronous ingestion (at the expense of data availability). We argue that the most crucial algorithmic challenge for a CB archiving solution is to decide when to store a revision as a snapshot or as a delta, which is tantamount to trading disk space for faster VM queries. This could be formulated as an (multi-objective) minimization problem whose objective might be a function of response time for triple patterns in VM, CV and V queries with constraints on available disk space and average ingestion time. When high concurrency is imperative, the objective function could also take query throughput into account. In the same vibe, a TB solution could trigger the construction of further indexes (e.g., new combinations of components, incremental indexes in the concurrent setting) based on a careful consideration of disk consumption and runtime gain. Such scenarios would not only require the conception of a novel cost model for query runtime in the archiving setting, but also the development of approximation algorithms for the underlying optimization problems, which are likely NP-Hard. Finally, since fresh data is likely to be queried more often than stale data, we believe that fetch time complexity[9] on the most recent(s) version(s) of the dataset should not depend on the size of the archive history. Hence, and depending on the host available main memory, an RDF archiving system could keep the latest revision(s) of a dataset (or parts of it) in main memory or in optimized

[9]For single triple patterns on VM queries

disk-based stores for faster query response time (as done by QuitStore [12]). Hence, main memory consumption could also be part of the optimization objective.

Internal serialization. Archiving multi-graph datasets requires the serialization of 5-tuples, which complexifies the trade-offs between space (i.e., disk and main memory consumption) and runtime efficiency (i.e., response time, ingestion time). For example, dealing with more columns increases the number of possible index combinations. Also, it leads to more data redundancy, since a triple can be associated to multiple values for the fourth and fifth component. Classical solutions for metadata in RDF include reification [70], singleton properties [60], and named graphs [69]. Reification assigns each RDF statement (triple or quad) an identifier $t \in \mathcal{I}$ that can be then used to link the triple to its ρ and ζ components in the 5-tuples data model introduced in Section 2.3. While simple and fully compatible with the existing RDF standards, reification is well-known to incur serious performance issues for storage and query efficiency, e.g., it would quintuple the number of triple patterns in SPARQL queries. On those grounds, Nguyen et al. [60] proposes singleton properties to piggyback the metadata in the predicate component. In this strategy, predicates take the form $p\#m \in \mathcal{I}$ for some $m \in \mathcal{N}$ and every triple with p in the dataset. This scheme gives $p\#m$ the role of ρ in the aforementioned data model reducing the overhead of reification. However, singleton properties would still require an additional level of reification for the fifth component ζ. The same is true for a solution based on named graphs. A more recent solution is HDTQ [28], which extends HDT with support for quads. An additional extension could account for a fifth component. Systems such as Dydra [11] or v-RDFCSA [23] resort to bit vectors and visibility maps for triples and quads. We argue that vector and matrix representations may be suitable for scalable RDF archiving as they allow for good compression in the presence of high redundancy: If we assume a binary matrix from triples (rows) to graph revisions (columns) where a one denotes the presence of a triple in a revision, we would expect rows and columns to contain many contiguous ones – the logic is analogous for removed triples.

Accounting for evolution patterns. As our study in Section 4 shows, the evolution patterns of RDF archives can change throughout time leading even, to decreases in dataset size. With that in mind, we envision an adaptive data-oriented system that adjusts its parameters according to the archive's evolution for the sake of efficient resource comsumption. Parameter tuning could rely on the metrics proposed in Section 3. Nonetheless, these desiderata translate into some design and engineering considerations. For example, we saw in Section 6 that a large number of deletions can negatively impact

Ostrich's query runtime, hence, such an event could trigger the construction of a complete snapshot of the dataset in order to speed-up VM queries (assuming the existence of a cost model for query runtime). In the same spirit and assuming some sort of dictionary encoding, an increase in the vocabulary dynamicity could increase the number of bits used to encode the identifiers of RDF terms in the dictionary. Those changes could be automatically carried out by the archiving engine, but could also be manually set up by the system administrator after an analysis with RDFev. A design philosophy that we envision to explore divides the history of each graph in the dataset in intervals such that each interval is associated to a block file. This file contains a full snapshot plus all the changesets in the interval. It follows that the application of a new changeset may update the latest block file or create a new one (old blocks could be merged into snapshots to save disk space). This action could be automatically executed by the engine or triggered by the system administrator. For instance, if the archive is the backend of a version control system, new branches may always trigger the creation of snapshots. This base architecture should be enhanced with additional indexes to speed up V queries and adapted compression for the dictionary and the triples.

Finally as we expect long dataset histories, it is vital for solutions to improve their ingestion time complexity, which should depend on the size of the changesets rather than on history size—contrary to what we observed in Section 6 for Ostrich. This constraint could be taken into account by the storage policy for the creation of storage structures such as deltas, snapshots, or indexes (e.g., by reducing the length of delta chains for redundant changesets). Nevertheless, very large changesets may still be challenging, specially in the concurrent scenario. This may justify the creation of temporary incremental (in-memory) indexes and data structures optimized for asynchronous batch updates as proposed in x-RDF-3X [59].

8 Conclusions

In this paper we have discussed the importance of RDF archiving for both maintainers and consumers of RDF data. Besides, we have discussed the importance of evolution patterns in the design of a fully-fledged RDF archiving solution. On these grounds, we have proposed a metric-based framework to characterize the evolution of RDF data, and we have applied our framework to study the history of three challenging RDF datasets, namely DBpedia, YAGO, and Wikidata. This study has allowed us to characterize the history of these datasets in terms of changes at the level of triples, vocabulary terms, and entities. It has also allowed us to identify design shifts in their release history. Those insights can be used to optimize the allocation of resources for archiving, for example, by triggering the creation of a new snapshot as a

response to a large changeset.

In other matters, our survey and study of the existing solutions and benchmarks for RDF archiving has shown that only a few solutions are available for download and use, and that among those, only Ostrich can store the release history of very large RDF datasets. Nonetheless, its design still does not scale to long histories and does not exploit the data evolution patterns. R43ples [36], R&WBase [72], Quit Store [12], and x-RDF-3X [59] are also available, however they are still far from tackling the major challenges of this task, mainly because, they are conceived for collaborative version control, which is an application of RDF archiving in itself. Our survey also reveals that the state of the art lacks a standard to query RDF archives. We think that a promising solution is to use SPARQL-star combined with additional syntactic sugar as proposed by some approaches [11, 30, 36]

Finally, we have used all these observations to derive a set of design lessons in order to overcome the gap between the literature and a fully functional solution for large RDF archives. All in all, we believe that such a solution should (i) support global histories for RDF datasets, (i) resort to a modular architecture that decouples the storage from the application layers, (iii) handle provenance and domain-specific temporal metadata, (iv) implement a SPARQL extension to query archives, (v) use a metric-based approach to monitor the data evolution and adapt resource consumption accordingly, and (vi) provide a performance that does not depend on the length of the history. The major algorithmic challenges in the field lie in how to handle the inherent trade-offs between disk usage, ingestion time, and query runtime. With this detailed study and the derived guidelines, we aim at paving the way towards an ultimate solution for this problem. In this sense, we envision archiving solutions to not only serve as standalone single server systems but also as components of the RDF ecosystem on the Web in all its flavors covering federated [5, 55, 68, 73] and client-server architectures [6, 14, 15, 41, 53, 54, 80] as well as peer-to-peer [7, 8] solutions.

References

[1] "European Open Data Portal," https://data.europa.eu/data/, accessed: 2020-06-09.

[2] "RDF Exports from Wikidata," Available at tools.wmflabs.org/wikidata-exports/rdf/index.html.

[3] "Stardog," http://stardog.com, accessed: 2020-06-09.

[4] "USA Data Gov. portal," https://www.data.gov/, accessed: 2020-06-09.

[5] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus, "ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints," in *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference,*

Bonn, Germany, October 23-27, 2011, 2011, pp. 18–34. [Online]. Available: https://doi.org/10.1007/978-3-642-25073-6_2

[6] C. Aebeloe, I. Keles, G. Montoya, and K. Hose, "Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns," *CoRR*, vol. abs/2002.09172, 2020. [Online]. Available: https://arxiv.org/abs/2002.09172

[7] C. Aebeloe, G. Montoya, and K. Hose, "A Decentralized Architecture for Sharing and Querying Semantic Data," in *ESWC*, vol. 11503. Springer, 2019, pp. 3–18. [Online]. Available: https://doi.org/10.1007/978-3-030-21348-0_1

[8] ——, "Decentralized Indexing over a Network of RDF Peers," in *ISWC*, vol. 11778. Springer, 2019, pp. 3–20. [Online]. Available: https://doi.org/10.1007/978-3-030-30793-6_1

[9] ——, "Colchain: Collaborative linked data networks," in *WWW*. ACM / IW3C2, 2021, pp. 1385–1396.

[10] A. B. Andersen, N. Gür, K. Hose, K. A. Jakobsen, and T. B. Pedersen, "Publishing Danish Agricultural Government Data as Semantic Web Data," in *Semantic Technology - 4th Joint International Conference, JIST 2014, Chiang Mai, Thailand, November 9-11, 2014. Revised Selected Papers*. Springer, 2014, pp. 178–186. [Online]. Available: https://doi.org/10.1007/978-3-319-15615-6_13

[11] J. Anderson and A. Bendiken, "Transaction-Time Queries in Dydra," in *MEPDaW/LDQ@ESWC*, ser. CEUR Workshop Proceedings, vol. 1585. CEUR-WS.org, 2016, pp. 11–19.

[12] N. Arndt, P. Naumann, N. Radtke, M. Martin, and E. Marx, "Decentralized collaborative knowledge management using git," *Journal of Web Semantics*, vol. 54, pp. 29 – 47, 2019, managing the Evolution and Preservation of the Data Web. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570826818300416

[13] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A Nucleus for a Web of Open Data," in *The Semantic Web*, 2007, pp. 722–735.

[14] A. Azzam, C. Aebeloe, G. Montoya, I. Keles, A. Polleres, and K. Hose, "Wisekg: Balanced access to web knowledge graphs," in *WWW*. ACM / IW3C2, 2021, pp. 1422–1434.

[15] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, and A. Polleres, "SMART-KG: Hybrid Shipping for SPARQL Querying on the Web," in *WWW '20: The Web Conference 2020*. ACM / IW3C2, 2020, pp. 984–994. [Online]. Available: https://doi.org/10.1145/3366423.3380177

[16] K. Bereta, P. Smeros, and M. Koubarakis, "Representation and Querying of Valid Time of Triples in Linked Geospatial Data," in *Extended Semantic Web Conference*, 2013, pp. 259–274.

[17] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 28–37, 2001.

[18] C. Bizer, "The Emerging Web of Linked Data," *IEEE Intelligent Systems*, vol. 24, no. 5, pp. 87–92, 2009.

[19] N. R. Brisaboa, A. Cerdeira-Pena, A. Fariña, and G. Navarro, "A Compact RDF Store Using Suffix Arrays," in *String Processing and Information Retrieval*, 2015, pp. 103–115.

[20] J. Brunsmann, "Archiving Pushed Inferences from Sensor Data Streams," in *Proceedings of the International Workshop on Semantic Sensor Web*. INSTICC, 2010, pp. 38–46.

[21] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. H. Jr., and T. M. Mitchell, "Toward an Architecture for Never-Ending Language Learning," in *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence*, 2010.

[22] S. Cassidy and J. Ballantine, "Version Control for RDF Triple Stores," *ICSOFT (ISDM/EHST/DC)*, vol. 7, pp. 5–12, 2007.

[23] A. Cerdeira-Pena, A. Fariña, J. D. Fernández, and M. A. Martínez-Prieto, "Self-Indexing RDF Archives," in *DCC*. IEEE, 2016, pp. 526–535.

[24] T. W. W. W. Consortium, "PROV-O: The PROV Ontology," http://www.w3.org/TR/prov-o, 2013.

[25] I. Ermilov, J. Lehmann, M. Martin, and S. Auer, "LODStats: The Data Web Census Dataset," in *Proceedings of 15th International Semantic Web Conference - Resources Track*, 2016. [Online]. Available: http://svn.aksw.org/papers/2016/ISWC_LODStats_Resource_Description/public.pdf

[26] F. Erxleben, M. Günther, M. Krötzsch, J. Mendez, and D. Vrandečić, "Introducing Wikidata to the Linked Data Web," in *International Semantic Web Conference*, 2014, pp. 50–65.

[27] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF Representation for Publication and Exchange (HDT)," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 19, pp. 22–41, 2013.

[28] J. D. Fernández, M. A. Martínez-Prieto, A. Polleres, and J. Reindorf, "HDTQ: Managing RDF Datasets in Compressed Space," in *The Semantic Web*, 2018, pp. 191–208.

[29] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating Query and Storage Strategies for RDF Archives," in *Proceedings of the 12th International Conference on Semantic Systems*, 2016, pp. 41–48.

[30] V. Fionda, M. W. Chekol, and G. Pirrò, "Gize: A Time Warp in the Web of Data," in *International Semantic Web Conference (Posters & Demos)*, vol. 1690, 2016.

[31] T. A. S. Foundation, "Apache Jena Semantic Web Framework," jena.apache.org.

[32] J. Frey, M. Hofer, D. Obraczka, J. Lehmann, and S. Hellmann, "DBpedia FlexiFusion the Best of Wikipedia > Wikidata > Your Data," in *The Semantic Web – ISWC 2019*, 2019, pp. 96–112.

[33] L. Galárraga, K. Ahlstrøm, K. Hose, and T. B. Pedersen, "Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets," in *The Semantic Web – ISWC 2018*, 2018, pp. 547–565.

[34] S. Gao, J. Gu, and C. Zaniolo, "RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases." in *Proceedings of the Extended Semantic Web Conference*, 2016, pp. 269–280.

[35] F. Grandi, "T-SPARQL: A TSQL2-like Temporal Query Language for RDF," in *Local Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems*, 2010, pp. 21–30.

[36] M. Graube, S. Hensel, and L. Urbas, "R43ples: Revisions for Triples," in *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS)*, 2014.

[37] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Journal of Web Semantics*, vol. 3, no. 2, pp. 158–182, 2005.

[38] N. Gür, J. Nielsen, K. Hose, and T. B. Pedersen, "GeoSemOLAP: Geospatial OLAP on the Semantic Web Made Easy," in *Proceedings of the 26th International Conference on World Wide Web Companion*. ACM, 2017, pp. 213–217. [Online]. Available: https://doi.org/10.1145/3041021.3054731

[39] N. Gür, T. B. Pedersen, E. Zimányi, and K. Hose, "A foundation for spatial data warehouses on the semantic web," *Semantic Web*, vol. 9, no. 5, pp. 557–587, 2018.

[40] O. Hartig, "Foundations of RDF⋆ and SPARQL⋆: An Alternative Approach to Statement-Level Metadata in RDF," in *Proc. of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management*, ser. AMW, 2017.

[41] O. Hartig and C. B. Aranda, "Bindings-restricted triple pattern fragments," in *On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016*, 2016, pp. 762–779. [Online]. Available: https://doi.org/10.1007/978-3-319-48472-3_48

[42] O. Hartig, K. Hose, and J. F. Sequeda, "Linked Data Management," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Y. Zomaya, Eds. Springer, 2019.

[43] I. Horrocks, T. Hubauer, E. Jiménez-Ruiz, E. Kharlamov, M. Koubarakis, R. Möller, K. Bereta, C. Neuenstadt, O. Özçep, M. Roshchin, P. Smeros, and D. Zheleznyakov, "Addressing Streaming and Historical Data in OBDA Systems: Optique's Approach (Statement of Interest)," in *Workshop on Knowledge Discovery and Data Mining Meets Linked Open Data (Know@LOD)*, 2013.

[44] K. Hose and R. Schenkel, "RDF stores," in *Encyclopedia of Database Systems, Second Edition*, L. Liu and M. T. Özsu, Eds. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_80676

[45] T. Huet, J. Biega, and F. M. Suchanek, "Mining History with Le Monde," in *Proceedings of the Workshop on Automated Knowledge Base Construction*, 2013, pp. 49–54.

[46] D. hyuk Im, S. won Lee, and H. joo kim, "A Version Management Framework for RDF Triple Stores," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, 04 2012.

[47] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi, "Towards Exploratory OLAP Over Linked Open Data - A Case Study," in *Enabling Real-Time Business Intelligence - International Workshops, BIRTE 2013*, vol. 206. Springer, 2014, pp. 114–132. [Online]. Available: https://doi.org/10.1007/978-3-662-46839-5_8

[48] ——, "Processing Aggregate Queries in a Federation of SPARQL Endpoints," in *The Semantic Web. Latest Advances and New Domains - 12th European*

Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings, vol. 9088. Springer, 2015, pp. 269–285. [Online]. Available: https://doi.org/10.1007/978-3-319-18818-8_17

[49] ——, "Optimizing Aggregate SPARQL Queries Using Materialized RDF Views," in *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I*, 2016, pp. 341–359. [Online]. Available: https://doi.org/10.1007/978-3-319-46523-4_21

[50] K. A. Jakobsen, A. B. Andersen, K. Hose, and T. B. Pedersen, "Optimizing RDF Data Cubes for Efficient Processing of Analytical Queries," in *International Workshop on Consuming Linked Data (COLD) co-located with 14th International Semantic Web Conference (ISWC 2105)*, vol. 1426. CEUR-WS.org, 2015. [Online]. Available: http://ceur-ws.org/Vol-1426/paper-02.pdf

[51] M. Meimaris, "EvoGen: A Generator for Synthetic Versioned RDF," in *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016*, vol. 1558, 2016. [Online]. Available: http://ceur-ws.org/Vol-1558/paper9.pdf

[52] S. Metzger, K. Hose, and R. Schenkel, "Colledge: a vision of collaborative knowledge networks," in *Proceedings of the 2nd International Workshop on Semantic Search over the Web*. ACM, 2012, pp. 1–8. [Online]. Available: https://doi.org/10.1145/2494068.2494069

[53] T. Minier, H. Skaf-Molli, and P. Molli, "SaGe: Web Preemption for Public SPARQL Query Services," in *The World Wide Web Conference, WWW 2019*. ACM, 2019, pp. 1268–1278. [Online]. Available: https://doi.org/10.1145/3308558.3313652

[54] G. Montoya, C. Aebeloe, and K. Hose, "Towards Efficient Query Processing over Heterogeneous RDF Interfaces," in *Proceedings of the 2nd Workshop on Decentralizing the Semantic Web co-located with the 17th International Semantic Web Conference, DeSemWeb@ISWC*, ser. CEUR Workshop Proceedings, vol. 2165. CEUR-WS.org, 2018. [Online]. Available: http://ceur-ws.org/Vol-2165/paper4.pdf

[55] G. Montoya, H. Skaf-Molli, and K. Hose, "The Odyssey Approach for Optimizing Federated SPARQL Queries," in *International Semantic Web Conference*, vol. 10587. Springer, 2017, pp. 471–489. [Online]. Available: https://doi.org/10.1007/978-3-319-68288-4_28

[56] R. P. D. Nath, K. Hose, T. B. Pedersen, and O. Romero, "SETL: A programmable semantic extract-transform-load framework for semantic data warehouses," *Inf. Syst.*, vol. 68, pp. 17–43, 2017. [Online]. Available: https://doi.org/10.1016/j.is.2017.01.005

[57] R. P. D. Nath, K. Hose, T. B. Pedersen, O. Romero, and A. Bhattacharjee, "SET$_{LBI}$: An Integrated Platform for Semantic Business Intelligence," in *Proceedings of The 2020 World Wide Web Conference*, 2020.

[58] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 647–659, 2008.

[59] ——, "x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 256–263, 2010.

[60] V. Nguyen, O. Bodenreider, and A. Sheth, "Don't Like RDF Reification?: Making Statements About Statements Using Singleton Property," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014, pp. 759–770.

[61] N. Noy and M. Musen, "Ontology Versioning in an Ontology Management Framework," *Intelligent Systems, IEEE*, vol. 19, pp. 6–13, 08 2004.

[62] G. Papadakis, K. Bereta, T. Palpanas, and M. Koubarakis, "Multi-core Meta-blocking for Big Linked Data," in *Proceedings of the 13th International Conference on Semantic Systems*, 2017, pp. 33–40.

[63] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides, "On Detecting High-level Changes in RDF/S KBs," in *International Semantic Web Conference (ISWC)*, 2009, pp. 473–488.

[64] T. Pellissier Tanon, C. Bourgaux, and F. Suchanek, "Learning How to Correct a Knowledge Base from the Edit History," in *The World Wide Web Conference*, 2019, pp. 1465–1475.

[65] T. Pellissier Tanon and F. M. Suchanek, "Querying the Edit History of Wikidata," in *Extended Semantic Web Conference*, 2019.

[66] M. Perry, P. Jain, and A. P. Sheth, "SPARQL-ST: Extending SPARQL to Support Spatio-temporal Queries," in *Geospatial Semantics and the Semantic Web*, vol. 12, 2011, pp. 61–86.

[67] M. Psaraki and Y. Tzitzikas, "CPOI: A Compact Method to Archive Versioned RDF Triple-Sets," 2019.

[68] B. Quilitz and U. Leser, "Querying Distributed RDF Data Sources with SPARQL," in *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, 2008, pp. 524–538. [Online]. Available: https://doi.org/10.1007/978-3-540-68234-9_39

[69] Y. Raimond and G. Schreiber, "RDF 1.1 primer," W3C Recommendation, 2014, http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[70] ——, "RDF 1.1 Semantics," W3C Recommendation, 2014, http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/.

[71] Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris, and Y. Stavrakas, "A Flexible Framework for Understanding the Dynamics of Evolving RDF Datasets," in *International Semantic Web Conference (ISWC)*, 2015.

[72] M. V. Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens, and R. V. de Walle, "R&Wbase: Git for triples," *Linked Data on the Web Workshop*, 2013.

[73] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "FedX: Optimization Techniques for Federated Query Processing on Linked Data," in *International Semantic Web Conference*, vol. 7031. Springer, 2011, pp. 601–616. [Online]. Available: https://doi.org/10.1007/978-3-642-25073-6_38

[74] A. Seaborne and S. Harris, "SPARQL 1.1 query language," W3C, W3C Recommendation, 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[75] R. T. Snodgrass, I. Ahn, G. Ariav, D. S. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. G. Kulkarni, T. Y. C. Leung, N. A. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada, "TSQL2 Language Specification," *SIGMOD Record*, vol. 23, no. 1, pp. 65–86, 1994.

[76] O. Software, "OpenLink Virtuoso," http://virtuoso.openlinksw.com.

[77] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A Large Ontology from Wikipedia and Wordnet," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 3, pp. 203–217, 2008.

[78] R. Taelman, M. V. Sande, and R. Verborgh, "OSTRICH: Versioned Random-Access Triple Store," in *Companion of the The Web Conference 2018, WWW*, 2018, pp. 127–130.

[79] M. Tasnim, D. Collarana, D. Graux, F. Orlandi, and M.-E. Vidal, "Summarizing Entity Temporal Evolution in Knowledge Graphs," in *Companion Proceedings of The 2019 World Wide Web Conference*, 2019, pp. 961–965.

[80] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert, "Triple Pattern Fragments: A low-cost knowledge graph interface for the Web," *J. Web Semant.*, vol. 37-38, pp. 184–206, 2016. [Online]. Available: https://doi.org/10.1016/j.websem.2016.03.003

[81] M. Volkel, W. Winkler, Y. Sure, S. R. Kruk, and M. Synak, "SemVersion: A Versioning System for RDF and Ontologies," *Second European Semantic Web Conference*, 2005.

[82] A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia, "A General Framework for Representing, Reasoning and Querying with Annotated Semantic Web Data," *Web Semantics*, vol. 11, pp. 72–95, 2012.

Paper B

TrieDF: Efficient In-memory Indexing for Metadata-augmented RDF

Olivier Pelgrin, Luis Galárraga, Katja Hose

The paper has been published in
Managing the Evolution and Preservation of the Data Web (MEPDaW), Vol 3225,
pp. 20-29, 2021.

Abstract

Metadata, such as provenance, versioning, temporal annotations, etc., is vital for the maintenance of RDF data. Despite its importance in the RDF ecosystem, support for metadata-augmented RDF remains limited. Some solutions focus on particular annotation types but no approach so far implements arbitrary levels of metadata in an application-agnostic way. We take a step to tackle this limitation and propose an in-memory tuple store architecture that can handle RDF data augmented with any type of metadata. Our approach, called TrieDF, builds upon the notion of tries to store the indexes and the dictionary of a metadata-augmented RDF dataset. Our experimental evaluation on three use cases shows that TrieDF outperforms state-of-the-art in-memory solutions for RDF in terms of main memory usage and retrieval time, while remaining application-agnostic.

Pelgrin, O., Galárraga, L., Hose, K. (2021). TrieDF: Efficient In-memory Indexing for Metadata-augmented RDF. In: Managing the Evolution and Preservation of the Data Web, MEPDaW 2021, Volume 3225, Pages 20-29, CEUR-WS Proceedings.
The layout has been revised.

1 Introduction

During the last 20 years, the Web has seen a proliferation of large collections of RDF data, i.e., triples ⟨ `subject`, `predicate`, `object` ⟩ describing real-world concepts ranging from common-sense to specialized domains. The triples are structured in what we call an *RDF graph* or *knowledge graph* (KG). KGs find applications in multiple AI-related tasks, such as question answering, information retrieval, smart assistants, etc.

Building and maintaining a large-scale KG is a titanic effort. It does not only require sophisticated RDF stores and collaborative tools, but also procedures and protocols to extract, cleanse, and integrate data from potentially heterogeneous sources. This is true regardless of whether the KG is manually or (semi-)automatically populated. A central aspect in KG construction is *metadata management*. RDF metadata includes, but is not limited to, provenance, validity intervals, spatial annotations, confidence statements, and versioning. As existing KGs grow and new initiatives come to existence, the need to manage statements about triples, i.e., RDF tuples, becomes more and more crucial.

There exist multiple solutions to represent metadata about RDF triples. Popular solutions are named graphs and reification. That said, these approaches are not free of limitations. A large number of fine-grained RDF graphs can be a challenge for quad stores [5]. Reification, on the other hand, quintuples the number of statements in a dataset; not to mention the fact that it also complexifies the queries. For these reasons, RDF engines support at most one level of metadata in an out-of-the-box fashion. This means that current stores can model statements about triples, but not statements about quads, i.e., 5-tuples. They can neither model a versioned collection of graphs nor RDF statements originating from different sources with multiple validity intervals. Support for higher levels of metadata – equivalent to n-ary relationships – remains limited to very specific scenarios such as archiving [17].

This work takes a step to tackle the aforementioned limitations and proposes TrieDF, an in-memory RDF tuple store. TrieDF stores tuples of arbitrary length in a *trie*, an in-memory prefix-based tree originally used for compact storage and efficient retrieval of strings. TrieDF models *everything* as a trie, namely all indexes and the dictionary. Our evaluation shows that such an architecture yields a significant speed-up in retrieval with little penalty in memory consumption w.r.t. existing triple/quad stores. We also illustrate the utility of TrieDF at handling 5-tuples in the context of provenance and version management.

2 Preliminaries

RDF Graphs. An *RDF graph* $G \subseteq (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times \mathcal{T}$ is a set of triples $\langle$ s, p, o $\rangle$, with subject s, predicate p, and object o that model binary assertions about *entities*, for example, $\langle$:Denmark, :locatedIn, :Europe $\rangle$. The sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{T}$ are countably infinite sets of IRIs, blank nodes, and RDF terms respectively, with $\mathcal{T} = \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ and $\mathcal{L}$ the set of literals. IRIs are web-scoped identifiers for entities such as http://dbpedia.org/resource/Denmark (abbreviated *:Denmark* for default prefix http://dbpedia.org/resource/); blank nodes are anonymous file-scoped identifiers; literals are non-referenceable data such as strings, numbers, and dates.
Metadata-augmented RDF Graphs. Given an RDF graph G, a metadata-augmented graph $\Gamma : G \rightarrow \mathcal{T}^n$ is an injective function that annotates each triple of the graph with a k-tuple of RDF terms. We can also see Γ as a set of n-tuples $q = \langle$ s, p, o, ... $\rangle$ with $n = k + 3$. Metadata-augmented RDF graphs can model n-ary relationships in contrast to standard RDF graphs that can only model binary relationships.

3 Related Work

The need to store and manage metadata for RDF triples has given rise to a large literature body that we survey in two stages. First, we survey different techniques to encode metadata-augmented triples and store them in triple stores. In a second stage we discuss existing solutions to manage additional components in triples.

3.1 Encoding Metadata-augmented Triples

Reification is the process of encoding an n-ary statement through a set of binary relationships. For instance, consider the versioned triple $\langle$:Aalborg, :cityIn, :Denmark, 3 $\rangle$ – that states that the triple is present in revision 3 of the graph. Under the standard reification, the triple is assigned a surrogate IRI (or blank node) u that is linked to all the components of the quad, resulting in 4 new triples: $\langle$ u, :subject, :Aalborg $\rangle$, $\langle$ u, :predicate, :cityIn $\rangle$, $\langle$ u, :object, :Denmark $\rangle$, and $\langle$ u, :version, 3 $\rangle$. Since reification incurs a significant overhead both for storage and querying, other approaches have proposed more compact encoding strategies. The authors of [15] propose singleton properties as unique keys for statements in a context. In our example, such a context could be the graph revision where a triple occurs, e.g., $\langle$:Aalborg, cityIn#3, :Denmark $\rangle$. A more flexible scheme, called companion properties [10], proposes singleton properties per subject, for example, $\langle$

`:Aalborg`, `cityIn#3`.*si*, `:Denmark` ⟩, where *si* is a local identifier that can be used to model subject-level metadata.

Reification and single properties have been used to encode one level of metadata, e.g., versioning, probabilities, provenance, temporal validity, etc. However, porting these strategies to scenarios with arbitrary levels of statement-centered metadata, e.g., provenance plus versioning, requires a careful application-dependent combination of the different schemes. This need has motivated the development of RDF-star [9], a data model that treats triples as first-class citizens and allows for nested statements such as ⟨ ⟨ `:Aalborg, :cityIn, :Denmark` ⟩, `:version, 3` ⟩. Support for RDF-star is gaining traction in current RDF engines. Some commercial solutions such as RDFox, GraphDB, and Stardog can parse TriG, an RDF-star serialization text format. That said, none of those solutions can so far handle arbitrary levels of nestedness.

3.2 Beyond RDF Triples

RDF Named Graphs. Even though the RDF graph data model can be used to store metadata for RDF statements "natively", it was rather conceived as an analogy to documents and database tables. A named RDF graph is associated to an IRI *g*, and stores a presumably large collection of triples within a well-defined context, e.g., a particular data source. Nevertheless, named graphs have been used to store more fine-grained metadata such as validity intervals, revision numbers, and changesets [5, 17]. This can represent a challenge for classical RDF graph stores, such as Jena, Virtuoso, RDF4J, etc., that are not optimized for a large number of small RDF graphs. Since RDF named graphs cannot model metadata for quads, a few approaches have proposed highly specific solutions in the context of RDF/S inference [16].

Property Graphs. In this data model, both nodes (entities) and edges (relationships) can be assigned attributes, such as labels, timestamps, probabilities, sources, etc. Despite this flexibility, property graphs cannot store arbitrary levels of metadata for triples out of the box. Like named graphs, they rather provide a generic solution to store metadata about triples. This agnosticism has propelled adaptations of the graph model and existing engines (e.g., Neo4J, GraphDB) to particular applications, such as version and history management [7] and workflow provenance [6, 8, 12]. As for named graphs, solutions are application-dependent and not trivially portable to arbitrary settings.

Relational Databases. Notable designs to store RDF in tables are the three-column table and the entity-relationship model (a relation per class, a column per predicate). Storing metadata about RDF in a relational setting does not require any extension to the original data model, and standard engines provide efficient support for the most common metadata types such as temporal

metadata [13], version control [11], and validity intervals. That said, the relational model is not optimized for the schema-free nature of RDF, which in the first place motivated the design of triple stores.

4 TrieDF

We now elaborate on TrieDF, our in-memory architecture to manage metadata-augmented RDF. TrieDF stores RDF tuples of arbitrary length in different indexes. The indexes do not store actual RDF terms but rather references to those terms in a space efficient dictionary (Figure B.1b).

TrieDF borrows inspiration from tries – prefix-based trees for string storage. Nodes in a trie store single characters and each string is associated to a path in the tree. Strings with the same prefix, e.g., *web*, *weave*, *weasel*, share common nodes. TrieDF treats tuples as "strings" of items. Those items are either references to RDF terms (for indexes) or IRI chunks (for the dictionary). We elaborate on these use cases in the following sections.

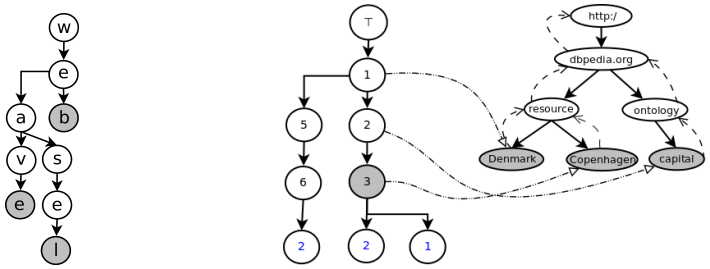
4.1 Trie-based Indexes

Consider a version annotated RDF graph with quads $\langle$ `s`, `p`, `o`, `v` $\rangle$ such that v models the versions where the triple $\langle$ `s`, `p`, `o` $\rangle$ is present. Figure B.1b depicts an SPOV index for the quads $\langle$ `1`, `2`, `3`, `1` $\rangle$, $\langle$ `1`, `2`, `3`, `2` $\rangle$, and $\langle$ `1`, `5`, `6`, `2` $\rangle$ – the triples are encoded using a dictionary. We can infer that the triple $\langle$ `1`, `5`, `6` $\rangle$ was added in the second revision of the graph. We now describe the implementation of TrieDF and the operations it supports.

Implementation. Logically, each element of a tuple is associated to a node in the tree. Physically, nodes store only references to their children. Those references are organized in a red-black tree keyed by the value of the child node. If $|T|$ is the number of nodes in a trie T, an index in TrieDF has a space complexity of $O(|T|c_{max})$, where c_{max} is the highest out-degree of a node in T. This happens because red-black trees exhibit $O(n)$ space complexity in the number of keys.

Tries are optimized for tuple lookup and prefix-based retrieval. These operations, as well as additions and deletions, are implemented as in standard tries. Therefore, looking up a tuple q incurs a time complexity of $O(|q|log(c_{max}))$.

Users of TrieDF can define indexes of arbitrary depth for all permutations of the elements of a tuple. Moreover, TrieDF can also operate as a standard triple store. In that case, the system stores triples in indexes SPO, POS, and OSP in line with standard engines. The nodes in the indexes store integers, precisely the memory addresses of the RDF terms in the dictionary [2], which is also implemented as a trie as explained next.



(a) A standard trie

(b) Trie-based index and dictionary

Fig. B.1: Example tries

4.2 Trie-based Dictionary Encoding

Dictionary encoding maps the terms of an RDF dataset to an integer space for the sake of efficient space consumption and query processing. Dictionary encoding is often implemented using two hash tables, i.e., one for encoding (string to integer) and one for decoding (integer to string).

Even though dictionary encoding reduces memory consumption drastically for RDF engines, it does not tackle the inherent redundancy of RDF terms. For instance, common prefixes in IRIs are still stored multiple times. Similarly to the work of Bazoobandi et al. [2], the dictionary for IRIs is stored in an trie. In [2], a node is usually associated to a single character, although multiple nodes can be compressed (fusioned) into a single node if they form a single branch subtree. A drawback of such an approach is that updates may cause fusioned nodes to split. In that case the tree must be rearranged, which increases update time. To make updates simpler – at the expense of some redundancy – TrieDF compresses IRIs by automatically coalescing all the nodes that lie between occurrences of the "/" character [21]. In other words, each node is logically associated to a chunk of an IRI as depicted in Figure B.1b. If a node marks the end of an IRI, the memory address of the node serves as the integer identifier used by the tuple indexes described in the previous section.

In practice dictionary tries are bidirectional. That is, nodes store references to their children and parent. That way, a dictionary can retrieve the identifier associated to an IRI (lookup) and vice versa (reverse lookup). (Figure B.1b). Because the benefit of a prefix-based representation is significantly less pronounced for literals [2, 20], they are currently stored in one-node tries.

5 Experiments

We evaluate TrieDF along three dimensions: loading time, main-memory consumption, and retrieval time (i.e., the time to return all the tuples that match a prefix). For this purpose, we test it in three use cases and compare it to other relevant in-memory solutions. Our scenarios cover a standard RDF graph, a versioned RDF graph (requiring quads), and a collection of 5-tuples.

TrieDF was written in C++. All experiments were run in a server with 256 GB of RAM, a 16-core CPU (AMD EPYC 7281), and an 8 TB HDD. The source code and experimental data is available at `https://relweb.cs.aau.dk/triedf`.

5.1 TrieDF for Triples

We first assess the performance of TrieDF when used as a standard in-memory triple store on DBpedia 2016-10 [1], YAGO 3.1 [18], YAGO 4 [19], and Wikidata [3]. For DBpedia we use the themes *mapping-based objects* and *mapping-based literals*. For YAGO 3.1 we consider the knowledge base's core, namely, the themes *facts*, *meta facts*, *literal facts*, *date facts*, and *labels*. For YAGO 4, we use the *facts* theme from the *English only* distribution. In regards to Wikidata we chose the *simple-statements* file of the 2016-08 RDF export[1]. Details about the dataset sizes are available in Table B.1. in which we compare TrieDF with Jena[2] and RDFLib[3], two fully-fledged in-memory platforms for RDF/SPARQL management. They are widely used in production environments and offer mature and well-tested implementations.

	DBpedia	*YAGO 3.1*	*YAGO 4*	*Wikidata*
Triples	38M	85M	22M	138M
Size	4.9GB	12GB	3.0GB	17GB
RDF terms	11M	59M	8.5M	54M

Table B.1: Details about the experimental datasets for the triples evaluation.

Loading time. Table B.2 shows the loading times of Jena, RDFLib, and TrieDF for the experimental datasets. Jena is consistently faster than the others systems, which can be explained by (i) a fast dictionary, (ii) a highly optimized RDF parser, and (iii) batching of insertions. Jena stores the dictionary in classical hash tables, which allows for constant time lookup and insertion of terms, in contrast to TrieDF that incurs a logarithmic lookup complexity.

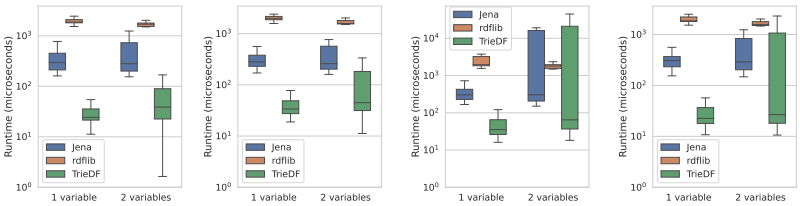
[1] `http://tools.wmflabs.org/wikidata-exports/rdf/index.html`
[2] `https://jena.apache.org/`
[3] `https://rdflib.readthedocs.org`

This, in contrast, optimizes for compactness (see paragraph on memory consumption). That said, TrieDF ranks second close to Jena, and outperforms RDFLib by a large margin.

	DBpedia	*YAGO 3.1*	*YAGO 4*	*Wikidata*
Jena	**587.14**	**1281.65**	**289.42**	**1665.48**
RDFLib	2816.16	7102.30	1626.85	9587.51
TrieDF	727.20	2105.37	358.20	2800.77

Table B.2: Loading time of the triples evaluation in seconds.

Retrieval time. We measure the average runtime of the different systems on single triple pattern queries of the following shapes: (i) ⟨ `s`, `p`, `?o` ⟩, (ii) ⟨ `?s`, `p`, `o` ⟩, (iii) ⟨ `s`, `?p`, `?o` ⟩, (iv) ⟨ `?s`, `?p`, `o` ⟩, and (v) ⟨ `?s`, `p`, `?o` ⟩ (`?` denotes a variable). For each query shape, we generated 200 queries by drawing the bound components randomly from the datasets, and then adding variables to the unbounded components as in [17]. The queries are implemented as classical retrieval operations, e.g., ⟨ `:Denmark`, `:capital`, `?o` ⟩ retrieves all tuples prefixed with ⟨ *:Denmark*, *:capital* ⟩ in the SPO trie index.

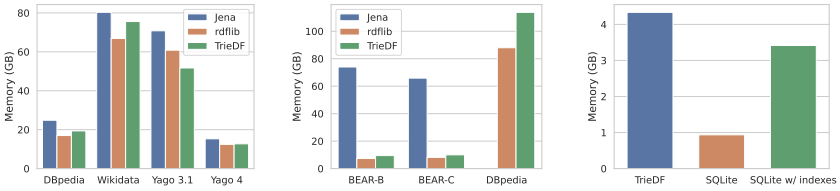


(a) DBpedia **(b)** Wikidata **(c)** YAGO 3.1 **(d)** YAGO 4

Fig. B.2: Query runtime (microseconds) for triples queries (log scale)

Figure B.2 depicts the query runtime of the tested systems on the different datasets. The results show that TrieDF is at least one order of magnitude faster than the competitors for queries with one variable. When there are two variables, TrieDF still exhibits lower median runtimes, but its variance is large, specially for YAGO. This can be explained by the fact that those queries may sometimes have a large number of results. This phenomenon also affects Jena. While RDFLib is mostly insensitive to large result sets, it lags behind the other systems in terms of average retrieval time.

Memory consumption. Figure B.3a shows the peek memory consumption of the different systems during loading and query execution. We observe that TrieDF outperforms Jena in all datasets, and all the competitors in YAGO 3. For the other datasets, our approach uses at most 13% more memory than RDFLib. This happens for two reasons. First, RDFLib's indexes are



(a) Triples **(b)** Quads **(c)** 5-tuples

Fig. B.3: Peek memory usage in gigabytes (GB)

of fixed depth, which allows for some space savings: leaves do not need to accommodate for an additional pointer to its potential children. Second, RDFLib does not implement explicit dictionary encoding but rather relies on Python's internal variable handling (Python variables are actually keys pointing to their actual value in a hash table) to store references to RDF terms in the indexes. Relying on Python incurs important memory savings for RDFLib, however, the system remains two orders of magnitude slower than TrieDF at retrieval.

5.2 TrieDF for Quads

In this section we evaluate TrieDF at managing RDF quads $\langle$ `s`, `p`, `o`, `v` $\rangle$, that represent triples annotated with a revision number v. Our evaluation is based on the BEAR [4] benchmark datasets, and an archive consisting of the DBpedia versions from the 3.5 to the 2016-10 release, where one release is equivalent to one revision [17]. For each release, we use the same DBpedia themes as in our experiments with triples. As for BEAR, we use both the *BEAR-B* and *BEAR-C* datasets. We omitted *BEAR-A* because our competitors could not parse the input files due to formatting issues.

In order to provide versioning capabilities to TrieDF, we store RDF quads in SPOV, POSV, and OSPV indexes. We also compare TrieDF with the Jena in-memory models and RDFLib. For both competitors, we use named graphs to store each revision. This storage strategy, called *independent copies*, optimizes for data retrieval [4, 17] at the expense of high memory consumption.

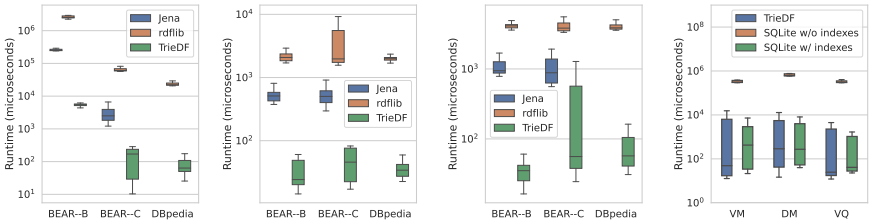
Loading time. Table B.3 shows the loading times of the evaluated systems. No results are provided for Jena in DBpedia, since the system runs out of memory. RDFLib lags behind Jena and TrieDF, with TrieDF being the fastest at loading the bulky revisions of BEAR-C, and Jena being the fastest at loading the more granular revisions of BEAR-B.

Retrieval time. We measure the average runtime of the different systems on 100 single triple pattern queries of different types on versioned RDF graphs, namely version materialization (VM), delta materialization (DM), and ver-

	DBpedia	*BEAR-B*	*BEAR-C*
Jena	-	**1094.83**	418.74
RDFLib	26433.76	4851.09	1663.40
TrieDF	**16074.17**	3743.55	**387.68**

Table B.3: Loading time of the quads evaluation in seconds.

sion queries (VQ). The queries were randomly generated according to the experimental setup in [17].



(a) VQ queries **(b)** VM queries **(c)** DM queries **(d)** 5-tuples

Fig. B.4: Query runtime for quads and 5-tuples (log scale)

Figure B.4 shows the runtime of the evaluated systems for each type of query. We observe that TrieDF outperforms all the competitors by far, although the performance gap can vary drastically. In particular, TrieDF's indexes are optimized for VQ queries, for which they are 3 orders of magnitude faster than the competitors. Even though the independent copies approach used in Jena and RDFLib is optimal for VM queries (and to a lesser extent for DM queries), TrieDF is still one order of magnitude faster than the competitors.

Memory consumption. Figure B.3b shows the peek memory consumption of the different systems during loading and query execution. We first highlight that Jena uses much more memory than the other systems. The reason is that Jena implements a classical independent copies approach, where each revision is entirety stored in a different graph. This leads to a lot of duplicated data. In contrast, RDFLib stores graphs (called contexts) in separate hash tables that map triples to graphs and graphs to triples. This mitigates redundancy. In the same vibe, TrieDF stores version identifiers in the last component of each index, which reduces redundancy. While RDFLib showcases the lowest memory consumption, TrieDF strikes the best trade-off with at most 28% more peek memory usage than RDFLib in exchange for a speed-up of 3 orders of magnitude for retrieval.

5.3 TrieDF for 5-tuples

In this section we evaluate TrieDF on a 5-tuples setup where triples are annotated with provenance and version identifiers q, v, i.e., we store tuples $\langle$ s, p, o, q, v $\rangle$. We use a dump[4] of 27M tuples of the NELL [14] dataset. The NELL extractors collect metadata-augmented knowledge iteratively from the Web. This metadata includes, among other fields, confidence scores for the extracted triples, extraction sources, extraction methods, and the iteration of promotion, i.e., the iteration at which a triple is considered true and "officially" added to the knowledge base. We use the two latter fields in this evaluation. Since RDF storage systems do not support 5-tuples, we compare TrieDF against relational database systems with support for in-memory tables. After a comparison between SQLite and MariaDB, we chose the former due to its good performance in our setting. 5-tuples in TrieDF are represented via SPOQV, POSQV, and OSPQV indexes. We test SQLite with and without those indexes.

Loading time. We report loading times of 36.55s, 16.98s, and 47.27s for TrieDF, SQLite, and SQLite with indexes respectively. We observe that SQLite loads data significantly faster when no indexes are built, however when indexing is enabled, TrieDF is faster.

Retrieval time. We tested both systems on 100 queries of the same types defined for the quads evaluation with randomly bounded q and v. As suggested by Figure B.4d, TrieDF achieves similar retrieval performance than an indexed 5-column SQLite in-memory table. The median runtime of TrieDF is better for VM and VQ queries.

Memory consumption. Figure B.3c shows the peak memory usage of both SQLite and TrieDF when loading and querying the NELL dataset. We observe that indexing multiplies memory consumption by a factor of 3 in SQLite. TrieDF still uses 26% more memory than indexed SQLite, however TrieDF cannot leverage its trie-based dictionary to its full capacity. This happens because NELL does not use prefixed IRIs. Despite this rather suboptimal setting, TrieDF still shows comparable performance to SQLite.

6 Conclusion

We have presented an in-memory architecture based on tries to index and access annotated RDF triples efficiently. Our solution provides the user with a flexible architecture to manage arbitrary RDF metadata in main memory. Our experimental evaluation has shown that such as an approach strikes an interesting trade-off between retrieval time and memory footprint: it can yield a speed-up of up to 3 orders of magnitude in retrieval time in return to

[4] http://rtw.ml.cmu.edu/rtw/resources

little (and sometimes no penalty) in memory usage. We believe that TrieDF is a first step towards a holistic solution to manage knowledge beyond RDF triples. As future work we envision to explore different strategies to reduce TrieDF's memory footprint. We also envision to couple our architecture with suitable in-disk storage and provide SPARQL query support.

References

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives, "DBpedia: A Nucleus for a Web of Open Data," in *ISWC/ASWC*, 2007, pp. 722–735.

[2] H. R. Bazoobandi, S. de Rooij, J. Urbani, A. ten Teije, F. van Harmelen, and H. E. Bal, "A Compact In-Memory Dictionary for RDF Data," in *ESWC*, 2015.

[3] F. Erxleben, M. Günther, M. Krötzsch, J. Mendez, and D. Vrandečić, "Introducing Wikidata to the Linked Data Web," in *ISWC*, 2014, pp. 50–65.

[4] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for RDF archives," in *SEMANTICS*, 2016, pp. 41–48.

[5] L. Galárraga, K. A. Jakobsen, K. Hose, and T. B. Pedersen, "Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets," in *ISWC*, 2018.

[6] L. Galárraga, K. A. M. Mathiassen, and K. Hose, "QBOAirbase: The European Air Quality Database as an RDF Cube," in *ISWC (Posters, Demos & Industry Tracks)*, ser. CEUR Workshop Proceedings, vol. 1963. CEUR-WS.org, 2017.

[7] M. Haeusler, T. Trojer, J. Kessler, M. Farwick, E. Nowakowski, and R. Breu, "ChronoGraph: A Versioned TinkerPop Graph Database," in *DATA*, 2017.

[8] E. R. Hansen, M. Lissandrini, A. Ghose, S. Løkke, C. Thomsen, and K. Hose, "Transparent Integration and Sharing of Life Cycle Sustainability Data with Provenance," in *ISWC*, 2020, pp. 378–394.

[9] O. Hartig, "Foundations of RDF⋆ and SPARQL⋆: An Alternative Approach to Statement-Level Metadata in RDF," in *AMW*, 2017.

[10] D. Hernández, A. Hogan, C. Riveros, C. Rojas, and E. Zerega, "Querying Wikidata: Comparing SPARQL, Relational and Graph Databases," in *ISWC*, 2016.

[11] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. G. Parameswaran, "OrpheusDB: Bolt-on Versioning for Relational Databases," *PVLDB*, vol. 10, no. 10, pp. 1130–1141, 2017.

[12] A. Kashliev, "Storage and Querying of Large Provenance Graphs Using NoSQL DSE," in *BigDataSecurity/HPSC/IDS*, 2020, pp. 260–262.

[13] M. Kaufmann, P. M. Fischer, N. May, and D. Kossmann, "Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past?" in *EDBT*, 2014.

[14] T. M. Mitchell and et al., "Never-Ending Learning," in *AAAI*, 2015, pp. 2302–2310.

[15] V. Nguyen, O. Bodenreider, and A. P. Sheth, "Don't like RDF reification?: making statements about statements using singleton property," in *WWW*, 2014.

[16] P. Pediaditis, G. Flouris, I. Fundulaki, and V. Christophides, "On Explicit Provenance Management in RDF/S Graphs," in *TAPP*, 2009.

[17] O. Pelgrin, L. Galárraga, and K. Hose, "Towards Fully-fledged Archiving for RDF Datasets," *Semantic Web Journal*, 2021.

[18] F. M. Suchanek, G. Kasneci, and G. Weikum, "YAGO: A Large Ontology from Wikipedia and WordNet," *J. Web Semant.*, vol. 6, no. 3, pp. 203–217, 2008.

[19] T. P. Tanon, G. Weikum, and F. M. Suchanek, "YAGO 4: A Reason-able Knowledge Base," in *ESWC*, 2020, pp. 583–596.

[20] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over Linked Data," *World Wide Web*, vol. 14, no. 5-6, pp. 495–544, 2011.

[21] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "TripleBit: a Fast and Compact System for Large Scale RDF Data," *PVLDB*, vol. 6, no. 7, pp. 517–528, 2013.

Paper C

Scaling Large RDF Archives To Very Long Histories

Olivier Pelgrin, Ruben Taelman, Luis Galárraga, Katja Hose

The paper has been published in the
Proceedings of 17th IEEE International Conference on Semantic Computing (ICSC 2023), pp. 41–48, 2023
DOI: 10.1109/ICSC56153.2023.00013

Abstract

In recent years, research in RDF archiving has gained traction due to the ever-growing nature of semantic data and the emergence of community-maintained knowledge bases. Several solutions have been proposed to manage the history of large RDF graphs, including approaches based on independent copies, time-based indexes, and change-based schemes. In particular, aggregated changesets have been shown to be relatively efficient at handling very large datasets. However, ingestion time can still become prohibitive as the revision history increases. To tackle this challenge, we propose a hybrid storage approach based on aggregated changesets, snapshots, and multiple delta chains. We evaluate different snapshot creation strategies on the BEAR benchmark for RDF archives, and show that our techniques can speed up ingestion time up to two orders of magnitude while keeping competitive performance for version materialization and delta queries. This allows us to support revision histories of lengths that are beyond reach with existing approaches.

Pelgrin, O., Taelman, R., Galárraga, L., Hose, K. (2023). Scaling Large RDF Archives To Very Long Histories. In: Proceedings of 17th IEEE International Conference on Semantic Computing (ICSC 2023). Pages 41–48, 2023
https://doi.org/10.1109/ICSC56153.2023.00013
The layout has been revised.

1 Introduction

The ever-growing nature of RDF data and the emergence of large collaborative knowledge graphs have propelled research in efficient techniques for *RDF archiving* [5, 9], which is the task of keeping track of an RDF graph's change history. RDF archiving is of great value to both maintainers and consumers of RDF data. To the former, archives are the basis of version control [2], which opens the door not only to novel data processing tasks, e.g., mining of temporal and correction patterns [10], but also to temporal data analytics [3, 12]. For data consumers, archives are a way to query the past and study the evolution of a given domain of knowledge [1, 7, 17].

From a technical point of view, building and maintaining RDF archives is a very challenging endeavor, primarily due to the massive size of current knowledge graphs. As of April 2022, DBpedia accounts for 220M entities and 1.45B facts[1], and changes from one release to the next one can be in the order of millions [9]. However, large changesets are not the only issue that challenges state-of-the-art RDF archive systems. For instance, DBpedia Live[2] receives continuous updates with changes made by the Wikipedia community. This dynamicity makes DBpedia's revision history extremely long, and exacerbates the challenges of managing an archive for a dataset of that nature. As shown in our experimental section, existing approaches for RDF archiving cannot ingest long histories on large datasets, even when the changes between revisions are small.

We therefore propose an approach to ingest, store, and query long revision histories on very large RDF graphs. Our techniques rely on a combination of dataset snapshots and sequences of aggregated changesets – called *delta chains* [16] in the literature. We evaluate our approaches on the BEAR benchmark [5] and show that our techniques can ingest the BEAR-B instant dataset in no more than 2 hours – something that so far has been beyond reach. Moreover, our techniques exhibit competitive runtimes for most types of queries on RDF archives. We implemented our approach on top of OSTRICH [16], a state-of-the-art engine for archiving large RDF graphs.

The remainder of this paper is structured as follows. Section 2 elaborates on the background concepts and the state of the art in RDF archiving. Our storage techniques are detailed in Section 3, while Section 4 explains how to query archives with multiple snapshots and delta chains. Section 5 provides details of our implementation. The viability of our techniques is evaluated in Section 6. Finally, Section 7 concludes the paper and discusses future work.

[1] https://www.dbpedia.org/resources/knowledge-graphs/
[2] https://www.dbpedia.org/resources/live/

2 Background and Related Work

2.1 RDF Graphs and RDF Archives

An *RDF graph* G (also called a *knowledge graph*) consists of a set of triples $\langle$ `s`, `p`, `o` $\rangle$ with subject $s \in \mathcal{I} \cup \mathcal{B}$, predicate $p \in \mathcal{I}$, and object $o \in \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$, where $\mathcal{I}$ is a set of IRIs, $\mathcal{L}$ is a set of literals, and $\mathcal{B}$ is a set of blank nodes [11]. RDF graphs are queried using SPARQL [14], whose building blocks are *triple patterns*, i.e., triples that allow variables (prefixed with a '?') in any position, e.g., $\langle$ `?x`, `cityIn`, `USA` $\rangle$ matches all American cities in G.

An *RDF archive* $\mathcal{A}$ is a temporally ordered collection of RDF graphs that represents all the states of the graph throughout its history of updates. This can be formalized as $\mathcal{A} = \{G_0, \ldots, G_k\}$, with G_i being the graph at version (or revision) $i \in \mathbb{Z}_{\geq 0}$. The transition from G_{i-1} to version G_i is implemented via an update operation $G_i = (G_{i-1} \setminus u_i^-) \cup u_i^+$, where u_i^+ and u_i^- are disjoint sets of added and deleted triples. We call the pair $u_i = \langle u_i^+, u_i^- \rangle$ a *changeset* or *delta*. We can generalize changesets to any pair of versions, i.e., $u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$ defines the changes between versions i and j. When a triple $\langle$ `s`, `p`, `o` $\rangle$ is present in a version i of the archive, we write it as a quad $\langle$ `s`, `p`, `o`, `i` $\rangle$.

2.2 Querying RDF Archives

The literature identifies five types of queries over RDF archives [5, 16]. We explain them next and provide examples with a single triple pattern for simplicity.

- **Version Materialization (VM).** These are standard SPARQL queries run against a single version i, e.g., $\langle$ `?s`, `type`, `Country`, `5` $\rangle$ returns the countries present in version $i = 5$.
- **Delta Materialization (DM).** These are queries defined on changesets $u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$, e.g., the query asking for the countries added between versions $i = 3$ and $j = 5$, which implies to run $\langle$ `?s`, `type`, `Country` $\rangle$ on $u_{3,5}^+$.
- **Version Query (V).** These are standard SPARQL queries that provide results annotated with the versions where those results hold. An example is $\langle$ `?s`, `type`, `Country`, `?v` $\rangle$, which returns pairs $\langle$ `country`, `version` $\rangle$.
- **Cross-version (CV).** CV queries combine results from multiple versions, for example: *which of the countries in the latest version have diplomatic relationships with the countries in revision 0?*
- **Cross-delta (CD).** CD queries combine results from multiple changesets, for example: *in which versions were the most countries added?*

Both CV and CD queries build upon the other types of queries, i.e., V and

DM queries. Therefore, full support for VM, DM, and V queries suffices for applications relying on RDF archives.

2.3 Solutions for RDF Archive Management

Several solutions have been proposed for managing the history of RDF graphs efficiently. We review the most prominent approaches in this section and refer the reader to [9] for a detailed survey.

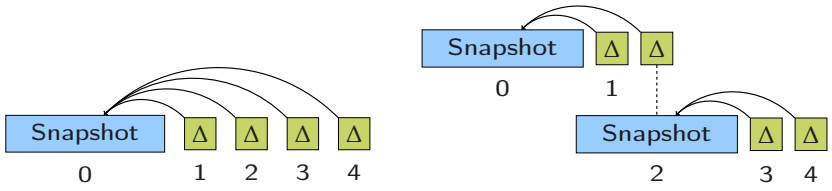
In the literature, RDF archive approaches are typically categorized according to their storage architecture. We distinguish three major design paradigms: independent copies (IC), change-based solutions (CB), and timestamp-based systems (TB). IC approaches, such as [18], implement full redundancy: all triples present in a version i are stored as an independent RDF graph G_i. While IC approaches excel at executing VM queries, they are impractical for today's knowledge graphs due to their prohibitive storage footprint. This fact has shifted the research trend towards CB and TB systems. In a CB solution, some versions are stored as *changesets* (also called deltas) w.r.t. a previous reference version stored as a snapshot. We call a sequence of changesets – representing an arbitrary sequence of versions – and its corresponding reference revision, a *delta chain*. CB approaches require less disk space than IC architectures and are optimal for DM queries – at the expense of efficiency for VM queries. This makes them particularly attractive for version-control systems, e.g., [2, 6], where changesets are rather small and frequent. TB solutions, on the other hand, optimize for V queries as they store temporal metadata, such as validity intervals or insertion/deletion timestamps [8] in specialized indexes.

Recent approaches borrow inspiration from more than one paradigm. QuitStore [2], for instance, stores the data in fragments, for which it implements a selective IC approach. This means that only modified fragments generate new copies, whereas the latest version is always materialized in main memory. OSTRICH [16] combines the advantages of CB and TB approaches in a *single* delta chain; an initial snapshot stores revision 0, whereas a new revision i is built from a changeset of the form $u_{i-1,i}$ and stored as an aggregated changeset $u_{0,i}$, i.e. the changes between the snapshot to i. This storage architecture is depicted in Figure C.1a. OSTRICH supports VM, DM, and V queries on single triple patterns natively. CV, CD, and arbitrary SPARQL queries can be executed by connecting OSTRICH to a query engine. Aggregated changesets have been shown to speed up VM and DM queries significantly w.r.t. a standard CB approach. As shown in [9, 16], OSTRICH is the only solution that can handle histories for large RDF graphs, such as DBpedia. That said, scalability still remains a challenge for OSTRICH because aggregated changesets grow monotonically. This leads to prohibitive ingestion times for large histories [9, 15] – even when the original changesets

are small. In this paper, we build upon OSTRICH and propose a solution to this problem.

3 Storing Archives with Multiple Delta Chains

As discussed in Section 2, ingesting new revisions as aggregate changesets can quickly become prohibitive for long revision histories when the RDF archive is stored in a single delta chain (see Figure C.1a). In such cases, we propose the creation of a fresh snapshot that becomes the new reference for subsequent deltas. Those new deltas will be smaller, and thus easier to build and maintain. They will also constitute a new delta chain as depicted in Figure C.1b.



(a) Single delta chain **(b)** Multiple delta chains

Fig. C.1: Delta chain architectures

3.1 Delta Chains

While creating a fresh snapshot with a new delta chain should presumably reduce ingestion time for subsequent revisions, its impact on query efficiency seems mixed. V queries, for instance, will have to be evaluated on multiple delta chains, becoming more challenging to answer. In contrast, VM queries defined on revisions already materialized as snapshots should be executed much faster. Storage size and DM response time may be highly dependent on the actual evolution of the data. If a new version includes many deletions, fresh snapshots may be smaller than aggregated deltas. We highlight that in our proposed architecture, revisions stored as snapshots also exist as aggregated deltas w.r.t. the previous snapshot – as shown for revision 2 in Figure C.1b. Such a design decision allows us to speed up DM queries as explained later.

It follows from the previous discussion that introducing multiple snapshots and delta chains raises a natural question: *"When is the right moment to create a snapshot?"* We elaborate on this question from the perspectives of storage, ingestion time, and query efficiency next. We then explain how to query archives in a multi-snapshot setting in Section 4.

3.2 Strategies for Snapshot Creation

A key aspect of our proposed design is to determine the right moment to place a snapshot, as this decision is subject to a trade-off among ingestion speed, storage size, and query performance. We formalize this decision via an abstract *snapshot oracle* $f : \mathbb{A} \times \mathbb{U} \to \{0, 1\}$ that, given an archive $\mathcal{A} \in \mathbb{A}$ with k revisions and a changeset $u_{k-1,k} \in \mathbb{U}$, decides whether revision k should (1) or should not (0) be materialized as a snapshot – otherwise the revision is stored as an aggregated delta. The oracle can rely on properties of the archive and the input changeset to make a decision. In the following, we describe some natural alternatives for our snapshot oracle f and illustrate them with a running example (Table C.1). All strategies start with a snapshot at revision 0. Note that we do not provide an exhaustive list of all possible strategies one could implement.

Baseline. The baseline oracle never creates snapshots, except for the very first revision, i.e., $f(\mathcal{A}, u) \equiv (\mathcal{A} = \emptyset)$. This is akin to OSTRICH's snapshot policy [16].

Periodic. A new snapshot is created when a fixed number d of versions has been ingested as aggregated deltas, i.e., $f(\mathcal{A}, u) \equiv (|\mathcal{A}| \ mod \ (d + 1) = 0)$. We call d the period.

Change-ratio. Long delta chains do not only incur longer ingestion times but also higher disk consumption due to redundancy in the aggregated changesets. When low disk usage is desired, the snapshot strategy may take into account the editing dynamics of the RDF graph. This notion has been quantified in the literature via the change ratio score [5]:

$$\delta_{i,j}(\mathcal{A}) = \frac{|u^+_{i,j}| + |u^-_{i,j}|}{|G_i \cup G_j|} \tag{C.1}$$

Given two revisions i and j, the change ratio normalizes the number of changes (additions and deletions) between the revisions by their joint size. If we aggregate the change ratios of all the revisions coming after a snapshot revision s, we can estimate the level of redundancy in the current delta chain. A reasonable snapshot strategy would therefore bound $\sum_{i=s+1}^{k} \delta_{s,i}$, put differently: $f(\mathcal{A}, u) \equiv (\gamma \leq \sum_{i=s+1}^{k} \delta_{s,i})$ for some user-defined budget threshold $\gamma \in \mathbb{R}_{>0}$.

Time. If we denote by t_k the time required to ingest revision k as an aggregated changeset in an archive $\mathcal{A}$, this oracle is implemented as $f(\mathcal{A}, u) \equiv (\frac{t_k}{t_{s+1}} > \theta)$, where $s + 1$ is the first revision stored as an aggregated changeset in the current delta chain. This strategy therefore creates a new snapshot as soon as ingestion takes θ times longer than the ingestion of version $s + 1$.

Version (k)	0	1	2	3	4	5
$\lvert u^+_{i,j} \rvert$	100	20	20	20	20	20
$\lvert u^-_{i,j} \rvert$	0	10	10	10	10	10
$\sum_{i=s+1}^{k} \delta_{s,i}$	-	0.23	0.61	1.08	0.19	0.51
t_k	-	1.00	1.50	2.25	3.38	1.0
Baseline	**S**	Δ	Δ	Δ	Δ	Δ
Periodic ($d = 2$)	**S**	Δ	**S**	Δ	**S**	Δ
Change ratio ($\gamma = 1.0$)	**S**	Δ	Δ	**S**	Δ	Δ
Time ($\theta = 3.0$)	**S**	Δ	Δ	Δ	**S**	Δ

Table C.1: Creation of snapshots according to the different strategies on a toy RDF graph comprised of 100 triples and 5 revisions defined by changesets. An **S** denotes a snapshot whereas a Δ denotes an aggregated changeset.

4 Querying Archives with Multiple Delta Chains

In the following, we detail our algorithms to compute version materialization (VM), delta materialization (DM), and V (version) queries on RDF archives with multiple delta chains. As is common in other RDF archiving approaches [16], we focus our algorithms on answering single triple patterns queries, since they constitute the building blocks for more complex query answering, which is outside the scope of this work. All the routines described next are defined w.r.t. to an implicit RDF archive $\mathcal{A}$.

4.1 VM Queries

In a single delta chain with aggregated deltas and reference snapshot s, executing a VM query with triple pattern p on a revision i requires us to materialize the target revision as $G_i = (G_s \cup u^+_{s,i}) \setminus u^-_{s,i}$ and then execute p on G_i. In our baseline, $s = 0$; in the presence of multiple delta chains $s = \mathit{snapshot}(i)$ corresponds to i's reference snapshot in the archive's history. Our implementation relies on OSTRICH, which can efficiently compute G_i and run queries on top of it.

4.2 DM Queries

The procedure *queryDM* in Algorithm 4 describes how to answer a DM query on two revisions i and j ($i < j$) with triple pattern p on an RDF archive with multiple delta chains. The algorithm relies on two important sub-routines. The first one, denoted *deltaDiff*, executes standard DM queries on single triple patterns over a single delta chain as proposed by OSTRICH [16]. The second routine, called *snapshotDiff*, computes the difference between the results of

Algorithm 4 DM query algorithm

```
1:  function snapshotDiff(S_i, S_j, p)
                                          ▷ snapshots S_i, S_j, triple pattern p
2:      d ← distance(i, j)
3:      if d > 1 then
4:          q_i ← query(S_i, p); q_j ← query(S_j, p)
5:          delta ← (q_j \ q_i) ∪ (q_i \ q_j)
6:      else
7:          delta = deltaDiff(i, j, p)
8:      return delta
9:
10: function queryDM(i, j, p)   ▷ versions i, j, triple pattern p
11:     sid_i ← snapshot(i); sid_j ← snapshot(j)
12:     if sid_i = sid_j then   ▷ i and j in the same delta-chain
13:         delta ← deltaDiff(i, j, p)
14:     else ▷ i and j not in the same delta-chain
15:         u_{si,sj} ← snapshotDiff(sid_i, sid_j, p)
16:         u_{si,i}, u_{sj,j} ← ∅
17:         if i ≠ sid_i then   ▷ test if version i is a delta
18:             u_{si,i} ← deltaDiff(sid_i, i, p)
19:         if j ≠ sid_j then   ▷ test if version j is a delta
20:             u_{sj,j} ← deltaDiff(sid_j, j, p)
21:         u_{i,sj} ← mergeBackwards(u_{si,i}, u_{si,sj})
22:         delta ← mergeForward(u_{i,sj}, u_{sj,j})
23:     return delta
```

p on two reference snapshots S_i, S_j. It works by first testing if the delta chains of S_i and S_j are not consecutive (line 2). If they are not, *snapshotDiff* implements a set-difference between p's results on S_i and S_j (lines 4–5). In case the snapshots define consecutive delta chains, we leverage the fact that S_j also exists as an aggregated delta w.r.t. S_i (see Section 3.1). We can therefore treat this case efficiently as a standard DM query via *deltaDiff* (line 7).

We now have the elements to explain the main DM query procedure (*queryDM*). First, it checks whether both revisions are in the same delta chain, i.e., if they have the same reference snapshot (line 14). If so, the problem boils down to a single delta chain DM query that can be answered with *deltaDiff* (line 15). Otherwise, we invoke the routine *snapshotDiff* on the reference snapshots (line 17) to compute the results' difference between the delta chains. This is denoted by *ds*.

If revisions i and j are not snapshots themselves, lines 20 and 23 compute

the changes between the target versions and their corresponding reference snapshots – denoted by $u_{si,i}$ and $u_{sj,j}$. The last steps, i.e., lines 25 and 26, merges the intermediate results to produce the final output. First, the routine *mergeBackwards* merges $u_{si,sj}$, i.e., the changes between the two delta chains, with $u_{si,i}$, i.e., the changes within the first delta chain. This routine is designed as a regular sorted merge because triples are already sorted in the OSTRICH indexes. Unlike a classical merge routine, *mergeBackwards* inverts the flags of the changes present in $u_{si,i}$ but not in $u_{si,sj}$. Indeed, if a change in $u_{si,i}$ did not survive to the next delta chain, it means it was later reverted in revision sid_j. The result of this operation are therefore the changes between revisions i and sid_j, which we denote by $u_{i,sj}$. The final merge step, *mergeForward*, combines $u_{i,sj}$ with the changes in the second delta chain, i.e., $u_{sj,sj}$. The routine *mergeForward* runs also a sorted merge, but now triples with opposite change flag present in both changesets are filtered from the final output as they indicate revertion operations.

4.3 V Queries

Algorithm 5 V query algorithm

```
1: function queryV(p)   ▷ p a triple pattern
2:     r ← ∅
3:     for c ∈ C do   ▷ C the list of delta chains
4:         v ← singleQueryV(c, p)
5:         r ← merge(r, v)   ▷ merge intermediate results
6:     return r
```

Algorithm 5 describes the process of executing a V query p over multiple delta chains. This relies on the capability to execute V queries on individual delta chains implemented in OSTRICH [16] via the function *singleQueryV*. The routine iterates over the list of delta chains (line 3), and runs *singleQueryV* on each delta chain (line 4). This gives us triples annotated with lists of versions within the range of the delta chain. At each iteration we carry out a merge step (line 5) that consists of a set union of the triples from the current delta chain and the results seen so far. When a triple is present in both sets, we merge their lists of versions.

5 Implementation

We implemented the proposed snapshot creation strategies and query algorithms for RDF archives on top of OSTRICH [16]. We briefly explain the most important aspects of our implementation.

Storage. An RDF archive consists of a snapshot for revision 0 and a single delta chain of aggregated changesets for the upcoming revisions (Fig. C.1a). The snapshot is stored as an HDT [4] file, whereas the delta chain is materialized in two stores: one for additions and one for deletions. Each store consists of 3 indexes in different triple component orders, namely SPO, OSP, and POS, implemented as B+trees. Keys in those indexes are individual triples linked to version metadata, i.e., the revisions where the triple is present and absent. Besides the change stores, there is an index with addition and deletion counts for all possible triple patterns, e.g., $\langle$ `?s`, `?p`, `?o` $\rangle$ or $\langle$ `?s`, `cityIn`, `?o` $\rangle$, which can be used to efficiently compute cardinality estimations – particularly useful for SPARQL engines.

Dictionary. As common in RDF stores [8, 19], RDF terms are mapped to an integer space to achieve efficient storage and retrieval. Two disjoint dictionaries are used in each delta chain: the snapshot dictionary (using HDT) and the delta chain dictionary. Hence, our multi-snapshot approach uses $D \times 2$ (potentially non-disjoint) dictionaries, where D is the number of delta chains in the archive.

Ingestion. The ingestion routine depends on whether a revision will be stored as an aggregated delta or as a snapshot. For revision 0, our ingestion routine takes as input a full RDF graph to build the initial snapshot. For subsequent revisions, we take as input a standard changeset $u_{k-1,k}$ ($|\mathcal{A}| = k$), and use OSTRICH to construct an aggregated changeset of the form $u_{s,k}$, where revision $s = \mathit{snapshot}(k)$ is the latest snapshot in the history. When the snapshot policy decides to materialize a revision s' as a snapshot, we use the aggregated changeset $u_{s,s'}$ to compute the snapshot efficiently as $G_{s'} = (G_s \setminus u^-_{s,s'}) \cup u^+_{s,s'}$.

Change-ratio estimations. The change-ratio snapshot strategy computes the cumulative change ratio of the current delta chain w.r.t. a reference snapshot s to decide whether to create a new snapshot or not. We therefore store the approximated change ratios $\delta_{s,k}$ of each revision in a key-value store. To approximate each $\delta_{s,k}$ according to Equation C.1, we rely on OSTRICH's count indexes. The terms $|u^+_{s,k}|$ and $|u^-_{s,k}|$ can be obtained from the count indexes of the fully unbounded triple pattern $\langle$ `?s`, `?p`, `?o` $\rangle$ in $O(1)$ time. We estimate $|G_s \cup G_j|$ as $|G_s| + |u^+_{s,j}|$, where $|G_s|$ is efficiently provided by HDT.

The source code of our implementation as well as the experimental scripts to reproduce this paper are available in a Zenodo archive[3].

[3] `https://doi.org/10.5281/zenodo.7256988`

6 Experiments

To determine the effectiveness of our multi-snapshot approach for RDF archiving, we evaluate the four proposed snapshot creation strategies along three dimensions: ingestion time (Section 6.2), disk usage (Section 6.3), and query runtime for VM, DM, and V queries (Section 6.4).

6.1 Experimental Setup

We resort to the BEAR benchmark for RDF archives [5] for our evaluation. BEAR comes in three flavors: BEAR-A, BEAR-B, and BEAR-C, which comprise a representative selection of different RDF graphs and query loads. We omit BEAR-C from our experiments because its query load consists of full SPARQL queries and diverse constructs, which are not supported by our implementation, nor by any other RDF archiving approaches. Table C.2 summarizes the characteristics of the experimental datasets and query loads. Due to the very long history of BEAR-B instant, OSTRICH could only ingest one third of the archive's history (7063 out 21046 revisions) after one month of execution. In a similar vibe, OSTRICH took one month to ingest the first 18 revisions (out of 58) of BEAR-A. Despite the dataset's short history, changesets in BEAR-A are in the order of millions of changes, which also makes ingestion intractable in practice. On these grounds, the original OSTRICH paper [16] omitted BEAR-B instant and included only the first 10 versions of BEAR-A. Multi-snapshot solutions, on the other hand, allow us to manage these datasets. All our experiments were run on a Linux server with a 16-core CPU (AMD EPYC 7281), 256 GB of RAM, and 8TB hard disk drive.

		BEAR-B		
	BEAR-A	Daily	Hourly	Instant
# versions	58	89	1299	21046
$\lvert G_i \rvert$'s range	30M - 66M	33K - 44K	33K - 44K	33K - 44K
$\overline{\lvert \Delta \rvert}$	22M	942	198	23
# queries	368	62 (49 ?P? and 13 ?PO)		

Table C.2: Dataset characteristics. $|G_i|$ is the size of the individual revisions, $\overline{|\Delta|}$ denotes the average size of the individual changesets $u_{k-1,k}$.

We evaluate the different strategies for snapshot creation detailed in Section 3.2 along ingestion speed, storage size, and query runtime. Except for our baseline (OSTRICH), all our strategies are defined by parameters that we adjust according to the dataset:

Periodic. This strategy is defined by the period d. We set $d \in \{2, 5\}$ for BEAR-A, $d \in \{5, 10\}$ for BEAR-B daily, $d \in \{50, 100\}$ for BEAR-B hourly, and

$d \in \{100, 500\}$ for BEAR-B instant. Values of d were adjusted per dataset experimentally w.r.t. the length of the revision history and the baseline ingestion time. High periodicity, i.e., smaller values for d, lead to more and shorter delta chains.

Change-ratio (CR). This strategy depends on a cumulative change-ratio budget threshold γ. We set $\gamma \in \{2.0, 4.0\}$ for all the tested datasets. $\gamma = 2.0$ yields 10 delta chains for BEAR-A, as well as 5, 23, and 151 delta chains for BEAR-B daily, hourly, and instant, respectively. For $\gamma = 4.0$, we obtain instead 6 delta chains for BEAR-A, and 3, 16, and 98 for the BEAR-B alternatives.

Time. This strategy depends on the ratio θ between the ingestion time of the new revision and the ingestion time of the first delta in the current delta chain. We set $\theta = 20$ for all datasets. This produces 3, 26, and 293 delta chains for the daily, hourly, and instant variants of BEAR-B respectively, and 2 delta chains for BEAR-A.

We omit the reference systems included with the BEAR benchmark since they are outperformed by OSTRICH [16].

		BEAR-B		
	BEAR-A	Daily	Hourly	Instant
High Periodicity	**13472.16**	**0.67**	**12.95**	57.89
Low Periodicity	14499.45	0.98	23.05	298.36
CR $\gamma = 2.0$	20505.93	1.88	13.79	77.01
CR $\gamma = 4.0$	21588.25	2.34	19.47	114.83
Time $\theta = 20$	49506.15	2.64	15.83	**43.53**
Baseline	-	6.89	1514.85	-

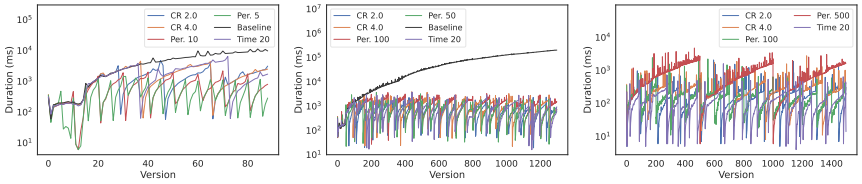
Table C.3: Ingestion times in minutes

		BEAR-B		
	BEAR-A	Daily	Hourly	Instant
High Periodicity	72417.47	199.17	322.34	2283.43
Low Periodicity	49995.00	102.96	**185.33**	**787.75**
CR $\gamma = 2.0$	47335.74	51.49	284.47	1690.38
CR $\gamma = 4.0$	**42203.04**	37.91	211.71	1175.15
Time $\theta = 20$	46614.98	38.33	325.13	3972.32
Baseline	-	**19.82**	644.50	-

Table C.4: Disk usage in MB

6.2 Ingestion Time

Table C.3 depicts the total time to ingest the experimental datasets. Since we always test two different values of d for the periodic strategy on each



(a) BEAR-B Daily **(b)** BEAR-B Hourly **(c)** BEAR-B Instant

Fig. C.2: Detailed ingestion times (log scale) per revision. We include the first 1500 revisions for BEAR-B instant since the runtime pattern is recurrent along the entire history.

dataset, in both Table C.3 and C.4, we refer to them as "high" and "low" periodicity. This is meant to abstract away the exact parameters, which vary for each dataset, so that we can focus instead on the effects of higher/lower periodicity. We remind the reader that the baseline (OSTRICH) cannot ingest BEAR-A and BEAR-B instant in a reasonable amount of time. This explains their absence in Table C.3. But even when OSTRICH can ingest the entire history (in less than 26 hours), a multi-snapshot strategy still incurs a significant speed-up. This becomes more significant for long histories as observed for BEAR-B hourly, where the speed-up can reach two orders of magnitude. The good performance of the high periodicity strategy and change-ratio with the smaller budget threshold $\gamma = 2.0$ suggests that shorter delta chains are beneficial for ingestion time. This is confirmed by Fig. C.2, where we also notice that ingestion time reaches a minimum for the revisions following a snapshot.

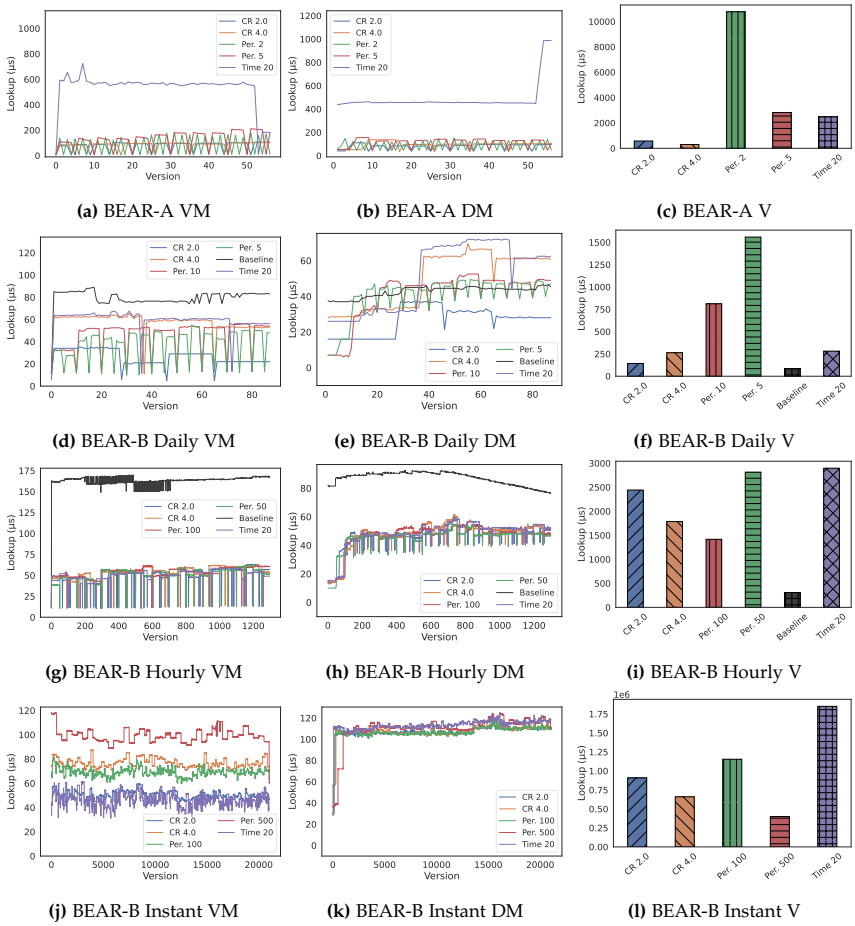
6.3 Disk Usage

Unlike ingestion time where shorter delta changes are clearly beneficial, the gains in terms of disk usage need fine-grained tuning because they depend on the dataset as shown in Table C.4. Overall, more delta chains tend to increase disk usage. For BEAR-B daily, frequent snapshots (high periodicity $d = 5$) incur a large overhead w.r.t. the baseline because the changesets are small and the revision history is short. Similar results are observed for BEAR-A and BEAR-B instant, even though we still need multiple snapshots to be able to ingest the data. BEAR-B hourly is interesting because it shows that for long histories, a single delta chain can be inefficient in terms of disk usage. Interestingly for BEAR-A, the change-ratio $\gamma = 4.0$ uses less storage than the time strategy with $\theta = 20$, despite using more delta chains. This hints that very large aggregated deltas are also inefficient compared to multiple delta chains with smaller aggregated deltas. For BEAR-B instant, the good performance of the change-ratio strategies and the low periodicity strategy ($d = 500$) suggests that a few delta chains can provide significant space sav-

ings. On the other hand, the time strategy with $\theta = 20$ performs slight worse because it creates too many delta chains.

6.4 Query Runtime

In this section we evaluate the impact of our snapshot creation strategies on query runtime. We use the queries provided with the BEAR benchmark for BEAR-A and BEAR-B. These are DM, VM, and V queries on single triple patterns. Each individual query was executed 5 times and the runtimes averaged. All the query results are depicted in Figure C.3.



(a) BEAR-A VM **(b)** BEAR-A DM **(c)** BEAR-A V

(d) BEAR-B Daily VM **(e)** BEAR-B Daily DM **(f)** BEAR-B Daily V

(g) BEAR-B Hourly VM **(h)** BEAR-B Hourly DM **(i)** BEAR-B Hourly V

(j) BEAR-B Instant VM **(k)** BEAR-B Instant DM **(l)** BEAR-B Instant V

Fig. C.3: Query results for the BEAR benchmark

VM queries

We report the average runtime of the benchmark VM queries for each version i in the archive. The results are depicted in Figures C.3a, C.3d, C.3g, and C.3j. We report runtimes in micro-seconds for all strategies.

Using multiple delta chains is consistently beneficial for VM query runtime, which is best when the target revision was materialized as a snapshot. When it is not the case, runtime is proportional to the *size* of the delta chain, which depends on its length and the volume of changes that must be applied to the snapshot before running the query. This is obvious for BEAR-A with the time $\theta = 20$ strategy, which splits the history into two imbalanced delta-chains, where one of them contains the first 53 revisions (out of 58).

DM Queries

We report for each revision i in the archive the average runtime of the benchmark DM queries between revisions $\langle$ `0, i` $\rangle$ and $\langle$ `1, i` $\rangle$. Such a setup tests the query routine in all possible scenarios: between two snapshots, between a snapshot and a delta (and vice versa), and between two deltas. The results are depicted in Figures C.3b, C.3e, C.3h, and C.3k. The results shows a rather mixed benefit of multiple delta chains in query runtime: highly positive for the long history of BEAR-B hourly and negligible for BEAR-B daily. Overall, DM queries benefit from short delta chains as illustrated by Figure C.3b and to a lesser degree by the periodic strategy with $d = 5$ in Figure C.3e. All our strategies beat the baseline by a large margin on BEAR-B hourly because delta operations become very expensive as the single delta chain grows. That said, the baseline runtime tends to decrease slightly with i because the data from two distant versions tends to diverge more, which requires the engine to filter fewer results from the aggregated deltas. For BEAR-B daily, multiple delta chains may perform comparably or slightly worse – by no more than 20% – than the baseline. This happens because BEAR daily's history is short, and hence efficiently manageable with a single delta chain. In this case the overhead of multiple snapshots and delta chains does not bring any advantage for DM queries.

V Queries

Figure C.3c, C.3f, C.3i, and C.3l show the total runtime of the benchmark V queries on the different datasets. V queries are the most challenging queries for the multi-snapshot archiving strategies as suggested by Figures C.3f and C.3i. As described in Algorithm 5, answering V queries requires us to query each delta chain individually, buffer the intermediate results, and then merge them. It follows that runtime scales proportionally to the number of delta chains, which means that contrary to DM and VM queries, many short

delta chains are detrimental to V query performance. Nonetheless, querying datasets such as BEAR-A and BEAR-B instant is only possible with a multi-snapshot solution.

6.5 Discussion

We now summarize our findings and draw a few design lessons for RDF archives.

- For small datasets, small changesets, or relatively short histories, the overhead of multi-snapshot strategies does not pay off in terms of query runtime and disk usage. This observation is particularly striking for V queries for which runtime increases with the number of delta chains.
- While many short delta chains are detrimental to V queries and often to storage consumption, they are mostly benefitial for VM and DM queries because these query types require us to iterate over changes within delta chains (two in the worst case of DM queries). Moreover, short delta chains reduce ingestion time systematically.
- Disk usage usually benefit from less numerous delta chains, except for long change history and large aggregated deltas.
- Change-ratio strategies strike an interesting trade-off because they take into account the amount of data stored in the delta chain as criterion to create a snapshot. This ultimately has a direct positive effect on ingestion time, VM/DM querying, and storage size.

The bottom line is that the snapshot creation strategy for RDF archives is subject to a trade-off among ingestion time, disk consumption, and query runtime for VM, DM, and V queries. As shown in our experimental section, there is no one-size-fits-all strategy. The suitability of a strategy depends on the application, namely the users' priorities or constraints, the characteristics of the archive (snapshot size, history length, and changeset size), and the query load. For example, implementing version control for a collaborative RDF graph will likely yield an archive like BEAR-B instant, i.e., a very long history with many small changes and VM/DM queries mostly executed on the latest revisions. Depending on the server's capabilities and the frequency of the changes, the storage strategy could therefore rely on the change ratio or the ingestion time ratio and be tuned to offer arbitrary latency guarantees for ingestion. On a different note, a user doing data analytics on the published versions of DBpedia (as done in [9]) may be confronted to a dataset like BEAR-A and therefore resort to numerous snapshots, unless their query load includes many real-time V queries.

7 Conclusion

In this paper we have presented a hybrid storage approach for RDF archiving based on multiple snapshots and chains of aggregated deltas. We studied different snapshot creation strategies and discussed the trade-offs in terms of ingestion time, storage size, and query runtime. Our experimental evaluation shows that our techniques allow us to handle very long revision histories that could not be managed by previous approaches. Moreover, we drew a set of design lessons for RDF archive design that can help users decide the best strategy based on the application scenario. As future work, we plan to develop more complex snapshot strategies, e.g., based on machine learning. Moreover, the development of more efficient encoding and serialization techniques for timestamped deltas is a promising research avenue to further lower storage size. We also plan to study the impact of our techniques on the performance of SPARQL query execution and consider improvements within the landscape of alternative RDF representations and indexing approaches [13].

References

[1] C. Aebeloe, G. Montoya, and K. Hose, "Colchain: Collaborative linked data networks," in *WWW*, 2021, pp. 1385–1396.

[2] N. Arndt, P. Naumann, N. Radtke, M. Martin, and E. Marx, "Decentralized collaborative knowledge management using git," *J. Web Semant.*, vol. 54, pp. 29–47, 2019.

[3] J. Brunsmann, "Archiving Pushed Inferences from Sensor Data Streams," in *International Workshop on Semantic Sensor Web*, 2010, pp. 38–46.

[4] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange (HDT)," *J. Web Semant.*, vol. 19, pp. 22–41, 2013.

[5] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for RDF archives," *J. Web Semant.*, vol. 10, no. 2, pp. 247–291, 2019.

[6] M. Graube, S. Hensel, and L. Urbas, "R43ples: Revisions for triples - an approach for version control in the semantic web," in *LDQ@SEMANTICS*, 2014.

[7] T. Huet, J. Biega, and F. M. Suchanek, "Mining History with Le Monde," in *Workshop on Automated Knowledge Base Construction*, 2013, pp. 49–54.

[8] T. Neumann and G. Weikum, "x-rdf-3x: Fast querying, high update rates, and consistency for RDF databases," *PVLDB*, vol. 3, no. 1, pp. 256–263, 2010.

[9] O. Pelgrin, L. Galárraga, and K. Hose, "Towards fully-fledged archiving for RDF datasets," *Semantic Web Journal*, vol. 12, no. 6, pp. 903–925, 2021.

[10] T. Pellissier Tanon, C. Bourgaux, and F. Suchanek, "Learning How to Correct a Knowledge Base from the Edit History," in *WWW*, 2019, pp. 1465–1475.

[11] Y. Raimond and G. Schreiber, "RDF 1.1 primer," W3C Recommendation, 2014, http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[12] Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris, and Y. Stavrakas, "A flexible framework for understanding the dynamics of evolving RDF datasets," in *ISWC*, vol. 9366, 2015, pp. 495–512.

[13] T. Sagi, M. Lissandrini, T. B. Pedersen, and K. Hose, "A design space for RDF data representations," *VLDB J.*, vol. 31, no. 2, pp. 347–373, 2022.

[14] A. Seaborne and S. Harris, "SPARQL 1.1 query language," W3C, W3C Recommendation, 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[15] R. Taelman, T. Mahieu, M. Vanbrabant, and R. Verborgh, "Optimizing storage of RDF archives using bidirectional delta chains," *J. Web Semant.*, vol. 13, pp. 705–734, 2022.

[16] R. Taelman, M. V. Sande, J. V. Herwegen, E. Mannens, and R. Verborgh, "Triple Storage for Random-access Versioned Querying of RDF Archives," *J. Web Semant.*, vol. 54, pp. 4–28, 2019.

[17] T. P. Tanon and F. M. Suchanek, "Querying the edit history of wikidata," in *ESWC*, vol. 11762, 2019, pp. 161–166.

[18] M. Volkel, W. Winkler, Y. Sure, S. R. Kruk, and M. Synak, "SemVersion: A Versioning System for RDF and Ontologies," *ESWC*, 2005.

[19] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

Paper D

Expressive Querying and Scalable Management of Large RDF Archives

Olivier Pelgrin, Ruben Taelman, Luis Galárraga, Katja Hose

This paper is under review at the
Semantic Web Journal

Abstract

The proliferation of large and ever-growing RDF datasets has sparked a need for robust and performant RDF archiving systems. In order to tackle this challenge, several solutions have been proposed throughout the years, including archiving systems based on independent copies, time-based indexes, and change-based approaches. In recent years, modern solutions combine several of the above mentioned paradigms. In particular, aggregated changesets of time-annotated triples have showcased a noteworthy ability to handle and query relatively large RDF archives. However, such approaches still suffer from scalability issues, notably at ingestion time. This makes the use of these solutions prohibitive for large revision histories. Furthermore, applications for such systems remain often constrained by their limited querying abilities, where SPARQL is often left out in favour of single triple-pattern queries. In this paper, we propose a hybrid storage approach based on aggregated changesets, snapshots, and multiple delta chains that additionally provides full querying SPARQL on RDF archives. This is done by interfacing our system with a modified SPARQL query engine. We evaluate our system with different snapshot creation strategies on the BEAR benchmark for RDF archives and showcase improvements of up to one order of magnitude in ingestion speed compared to state-of-the-art approaches, while keeping competitive querying performance. Furthermore, we demonstrate our SPARQL query processing capabilities on the BEAR-C variant of BEAR. This is, to the best of our knowledge, the first openly-available endeavour that provides full SPARQL querying on RDF archives.

The layout has been revised.

1 Introduction

The exponential growth of RDF data and the emergence of large collaborative knowledge graphs have driven research in the field of efficient RDF archiving [9, 19], the task of managing the change history of RDF graphs. This offers invaluable benefits to both data maintainers and consumers. For data maintainers, RDF archives serve as the foundation for version control [4]. This not only enables data mining tasks, such as identifying temporal and corrections patterns [23], but in general opens the door to advanced temporal data analytics of evolving graphs [6, 14, 26, 28]. For data consumers, RDF archives provide a valuable means to access historical data and delve into the evolution of specific knowledge domains [2, 15, 35]. In essence, these archives offer a way to query past versions of RDF data, allowing for a deeper understanding of how knowledge has developed over time.

However, building and maintaining RDF archives presents substantial technical challenges, primarily due to the large size of contemporary knowledge graphs. For instance, DBpedia, as of April 2022, comprises 220 million entities and 1.45 billion triples[1]. The number of changes between consecutive releases can reach millions [19]. Yet, dealing with large changesets is not the sole obstacle faced by state-of-the-art RDF archive systems. Efficient querying also remains an open challenge since support for full SPARQL is rare among existing systems [9, 19].

To address these challenges, we propose an approach for ingesting, storing, and querying long revision histories on large RDF archives. Our approach, which combines multiples snapshots and delta chains, has been previously detailed in our prior work [22] and outperforms existing state-of-the-art systems in terms of ingestion time and query runtime for archive queries on single triple patterns. This paper builds on top of this prior work and extends it with the following contributions:

- The design and implementation of a full SPARQL querying middleware on top of our multi-snapshot RDF archiving engine.
- A novel representation for the versioning metadata stored in our indexes. This representation is designed to improve ingestion time and disk usage without increasing query runtime.
- An evaluation of the two aforementioned contributions in addition to an extended evaluation of our prior work with additional baselines.

In general, we evaluate the effectiveness of our enhanced approach using the BEAR benchmark [9], our results demonstrate remarkable improvements, namely, up to several order of magnitude faster ingestion times, reduced disk

[1] https://www.dbpedia.org/resources/knowledge-graphs/

usage, and overall improved querying speed compared to existing baselines. Additionally, we showcase our new SPARQL querying capabilities on the BEAR-C variant of the BEAR benchmark. This is the first time, to the best of our knowledge, that a system complete this benchmark suite and publish its results.

The remainder of this paper is organized as follows: Section 2 explains the background concepts used throughout the paper. In Section 3 we discuss the state of the art in RDF archiving, in particular existing approaches to store and query RDF archives. In Section 4, we detail our storage architecture that builds upon multiple delta chains, and proposes several strategies to handle the materialization of new delta chains. Section 5 details the algorithms employed for processing single triple patterns over our multiple-delta-chain-architecture, while Section 6 describes our new versioning metadata serialization method, and showcases how it improves ingestion times and disk usage. In Section 7, we explain our solution to support full SPARQL 1.1 archives queries on top of our storage architecture. Section 8 describes our extensive experimental evaluation. The paper concludes with Section 9, which summarizes our contributions and discusses future research directions.

2 Preliminaries

An *RDF graph* G (also called a *knowledge graph*) consists of a set of triples $\langle$ `s`, `p`, `o` $\rangle$ with subject $s \in \mathcal{I} \cup \mathcal{B}$, predicate $p \in \mathcal{I}$, and object $o \in \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$, where $\mathcal{I}$ is a set of IRIs, $\mathcal{L}$ is a set of literals, and $\mathcal{B}$ is a set of blank nodes [27]. RDF graphs are queried using SPARQL [30], whose building blocks are *triple patterns*, i.e., triples that allow variables (prefixed with a '?') in any position, e.g., $\langle$ `?x`, `cityIn`, `USA` $\rangle$ matches all American cities in G.

An *RDF archive* $\mathcal{A}$ is a temporally ordered collection of RDF graphs that represents all the states of the graph throughout its history of updates. This can be formalized as $\mathcal{A} = \{G_0, \ldots, G_k\}$, with G_i being the graph at version (or revision) $i \in \mathbb{Z}_{\geq 0}$. The transition from G_{i-1} to version G_i is implemented via an update operation $G_i = (G_{i-1} \setminus u_i^-) \cup u_i^+$, where u_i^+ and u_i^- are disjoint sets of added and deleted triples. We call the pair $u_i = \langle u_i^+, u_i^- \rangle$ a *changeset* or *delta*. We can generalize changesets to any pair of versions, i.e., $u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$ defines the changes between versions i and j. When a triple $\langle$ `s`, `p`, `o` $\rangle$ is present in a version i of the archive, we write it as a quad $\langle$ `s`, `p`, `o`, `i` $\rangle$. We summarize the notations used throughout the paper in Table D.1.

3 Related Work

$\langle$ s, p, o $\rangle$	an RDF triple
$\langle$ s, p, o, i $\rangle$	a versioned triple, i.e., an RDF quad
G	an RDF graph
G_i	the i-th version or revision of graph G
$\mathcal{A}$	an RDF graph archive
$u_i = \langle u_i^+, u_i^- \rangle$	a changeset with sets of added and deleted triples for version i
$u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$	the changeset between graph versions i and j ($j > i$)

Table D.1: Notations summary.

In this section, we discuss the current state of RDF archiving in the literature. We will first present how RDF archives are usually queried. We then discuss the existing storage paradigms for RDF archives and how they perform on the different query types. We conclude this section by detailing the functioning of OSTRICH, a prominent solution for managing RDF archives, which we use as baseline for our proposed design.

3.1 Querying RDF Archives

In contrast to conventional RDF, the presence of multiple versions within an RDF archive requires the definition of novel query categories. Some categorizations for versioning queries over RDF Archives has been proposed in the literature [9, 18, 26]. In this work we build upon the proposal of Fernández et al. [9] due to its greater adoption by the community. They identify five query types, which we explain in the following through a hypothetical RDF archive that stores information about countries and their diplomatic relationships:

- **Version Materialization (VM).** These are standard SPARQL queries run against a single version i, e.g., $\langle$ `?s, type, Country, 5` $\rangle$ returns the countries present in version $i = 5$.
- **Delta Materialization (DM).** These are queries defined on changesets $u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$, e.g., the query asking for the countries added between versions $i = 3$ and $j = 5$, which implies to run $\langle$ `?s, type, Country` $\rangle$ on $u_{3,5}^+$.
- **Version Query (V).** These are standard SPARQL queries that provide results annotated with the versions where those results hold. An example is $\langle$ `?s, type, Country, ?v` $\rangle$, which returns pairs $\langle$ `country, version` $\rangle$.

- **Cross-version (CV).** CV queries combine results from multiple versions, for example: *which of the countries in the latest version have diplomatic relationships with the countries in revision 0?*
- **Cross-delta (CD).** CD queries combine results from multiple changesets, for example: *in which versions were the most countries added?*

Both CV and CD queries build upon the other types of queries, i.e., V and DM queries. Therefore, full support for VM, DM, and V queries is the minimum requirement for applications relying on RDF archives. Papakonstantinou et al. [18], on the other hand, propose a categorisation into two main categories, *version* and *delta* queries, which can be of any of three types: *Materialization*, *Single version*, or *Cross Version*. As such, *materialization* queries request the full set of triples present in a given version, while *single version* queries are answered by applying restrictions or filters on the triples of that version. *Cross version* queries instead needs access to multiple versions of the data. In practice, the categorizations of [18] and [9] are equally expressive. Polleres et al. [26] propose two categories of versioned queries: *Version Materialization* and *Delta Materialization*. Those are identical to the categories used by Fernández et al. [9] and described above. Queries applied to multiple versions are categorized as *Cross Version*, which includes the *Version Queries (V)* from Fernández et al. [9]'s classification.

SPARQL is the recommended W3C standard to query RDF data, however adapting versioned query types to standard SPARQL remains one of the main challenges in RDF archiving. Indeed, current RDF archiving systems are often limited to queries on single triple patterns [19, 32]. This puts the burden of combining the results of single triple pattern queries onto the user, further raising the barrier for the adoption of RDF archiving systems. While support for standard SPARQL on RDF archives is nonexistent, multiple endeavors have proposed either novel query languages or temporal extensions for SPARQL. Fernández et al. [9] for example, formulates their categories of versioning queries using the AnQL [38] query language. AnQL is a SPARQL extension operating on quad patterns instead of triples pattern. The additional component can be mapped to any term $u \in \mathcal{I} \cup \mathcal{L}$, and is used to represent time objects such as timestamps or version identifiers. Other works have focused on expressing SPARQL queries with temporal constraints [5, 11, 25]. T-SPARQL for example, takes inspiration from the TSQL2 language and can match triples annotated with validity timestamps. SPARQL-LTL [10] on the other hand, supports triples annotated with versions numbers, which are implemented as named graphs.

All in all, there is currently no widely accepted standard for representation of versioned queries over RDF archives within the community. Instead current proposals are often tailored to specific use cases and applications, and no standardization effort has been proposed.

In this work we formulate complex queries on RDF archives as standard SPARQL queries, but we assume that revisions in the archive are modeled logically as RDF graphs named according to a particular convention (explained in Section 7). This design decision makes our solution suitable to any standard RDF/SPARQL engine with support for named graphs.

3.2 Main Storage Paradigms for Storing RDF Archives

Several solutions have been proposed for storing the history of RDF graphs efficiently. We review the most prominent approaches in this section and refer the reader to [19] for a detailed survey. We distinguish three major design paradigms: independent copies (IC), change-based solutions (CB), and timestamp-based systems (TB).

Independent copies (IC) systems, such as SemVersion [36], implement full redundancy: all triples present in a version i are stored as an independent RDF graph G_i. While IC approaches excel at executing VM queries, DM and V queries suffer from the need to execute queries independently across multiple versions, requiring subsequent result set integration and filtering. Similarly, IC approaches are impractical for today's knowledge graphs due to their prohibitive storage footprint. This fact has shifted the research trend towards CB and TB systems.

Change-based (CB) solutions store their initial version as a full snapshot and subsequent versions G_j as *changesets* $u_{i,j}$ (also called deltas) where $j > i$. We call a sequence of changesets – representing an arbitrary sequence of versions – and its corresponding reference revision i, a *delta chain*. CB approaches usually require less disk space than IC architectures and are optimal for DM queries – at the expense of efficiency for VM queries. R43ples [12] is a prominent example of a system employing a CB storage paradigm.

Timestamp-based (TB) systems store triples annotated with versioning metadata such as temporal validity intervals, addition/deletion timestamps, and list of valid versions among others. This makes TB solutions usually well suited to efficiently answer V queries, while VM and DM queries still necessitate further processing. The storage efficiency of TB solutions depends on the representation chosen to serialize the versioning metadata. TB systems notably include x-RDF-3X [17], Dydra [3], and v-RDFCSA/v-HDT [7]. The latter has been showed to provide excellent storage efficiency and query performance. However, this is achieved at the cost of flexibility, by limiting itself to storing and indexing existing full archives, and leaving out the possibility of subsequent updates. Moreover, no implementation of their method has been made publicly available at the time of writing. Dydra is only available through their cloud service and is otherwise closed source, which makes a fair comparative evaluation in an independent and controlled setup impossible.

Finally, recent approaches borrow inspiration from more than one paradigm. QuitStore [4], for instance, stores the data in fragments, for which it implements a selective IC approach. This means that only modified fragments generate new copies, whereas the latest version is always materialized in main memory. OSTRICH [32] proposes a customized CB based approach based on aggregated changesets with version-annotated triples. This approach has showed great potential both in terms of scalability and query performance [19, 32]. For this reason our solutions use this system as underlying architecture. We describe OSTRICH in detail in the following section.

3.3 OSTRICH's Architecture and Storage Paradigm

Change-based (CB) approaches work by storing the first revision of an archive as a fully materialized snapshot, and subsequent versions as deltas $u_{i,j}$ with $i = j - 1$ (see Figure D.2a for an illustration). This approach provides better storage efficiency than approaches based on independent copies (IC), as long as the deltas between versions are not larger than the materialized revisions. Also, CB approaches provide good query performance for delta-materialization (DM) queries. However, some queries can become increasingly expensive as the delta chain grows, because each delta is relative to the previous one. For instance version-materialization (VM) queries require the iteration of the full delta chain, up to the target version, in order to reconstruct the needed data on which the query will be executed. Similarly, version queries (V) need to iterate over the entire delta chain to provide a complete list of the valid version numbers for each solution of a query. On long delta chains this process can become prohibitive.

As such, OSTRICH [32] proposes instead the use of *aggregated delta chains*, as illustrated in Figure D.2b. An aggregated delta chain works by storing an initial reference snapshot, as conventional delta chains do, and then storing subsequent versions as deltas $u_{0,j}$ with 0 being the reference version. Such an approach allows for a more efficient version materialization process compared to conventional delta chains, since only one delta needs to be considered to reconstruct any version.

OSTRICH stores the initial snapshot in a HDT file [8] , which provides a dictionary and a compressed, yet easy to read, serialization for the triples. As standard for RDF engines, the dictionary maps RDF terms to integer identifiers, which are used in all data structures for efficiency.

In contrast to the initial snapshot, the delta chain is stored in two separate triple stores, one for additions and one for deletions. Each triple store is comprised of three differently ordered indexes (SPO, POS, and OSP), as illustrated in Figure D.1. Each triple is annotated with versioning metadata, which is used for several purposes: First, this reduces data redundancy in the delta chain, allowing each triple to be stored only once. Secondly, the meta-

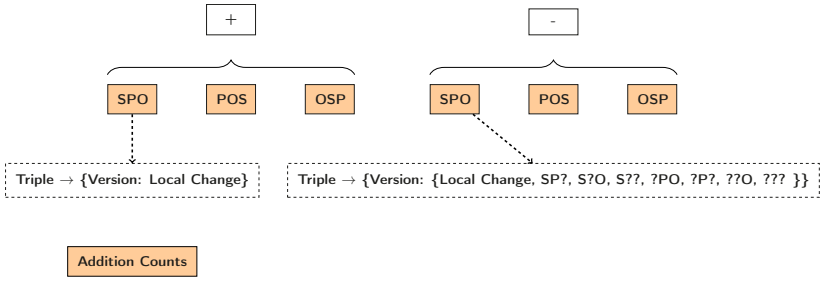


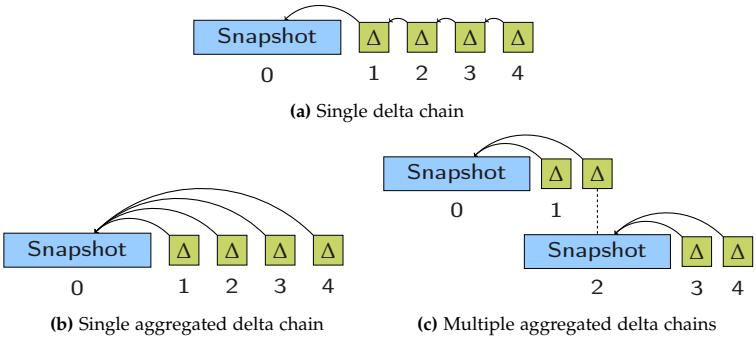
Fig. D.1: OSTRICH Delta Chain Storage Overview

data can be used to accelerate query processing. As seen in Figure D.1, this metadata differs between additions and deletions triples. All triples feature a collection of mappings from version to a local change flag, which indicates whether the triple reverts a previous change in the delta chain. For instance, consider the quad $\langle$ s, p, o, 0 $\rangle$ and a changeset in revision 2 that removes the triple $\langle$ s, p, o $\rangle$. If a subsequent change adds this triple again, let us say in revision 4, then the entry for triple $\langle$ s, p, o $\rangle$ will contain the entries $\{4 : \textit{true}\}$ and $\{2 : \textit{false}\}$ for the addition and deletion indexes respectively. This flag can be used to filter triples early during querying. Since deltas in OSTRICH are aggregated, entries in the versioning metadata are copied for each version where a change is relevant. From the previous example, the entry $\{2 : \textit{false}\}$ in the deletion index will also exist for revision 3 as $\{3 : \textit{false}\}$, since the triple is deleted in both $u_{0,2}$ and $u_{0,3}$. This can create redundancies, especially in long delta chains.

We notice that deleted triples are associated to an additional vector that stores the triple's relative position in the delta for every possible triple pattern order. This allows OSTRICH to optimize for offset queries, and enables fast computation of deletion counts for any triple pattern and version. Since HDT files cannot be edited, the delta chain has also its own writable dictionary for the RDF terms that were added after the first snapshot. More details about OSTRICH's storage system can be found in the original paper [32].

OSTRICH supports VM, DM, and V queries on single triple patterns natively. Aggregated changesets have been shown to speed up VM and DM queries significantly w.r.t. a standard CB approach. As shown in [19, 32], OSTRICH is the only available solution that can handle histories for large RDF graphs, such as DBpedia. That said, scalability still remains a challenge for OSTRICH because aggregated changesets grow monotonically. This leads to prohibitive ingestion times for large histories [19, 31] – even when the original changesets are small. In this paper, we build upon OSTRICH and propose a solution to this problem.

4 Storing Archives with Multiple Delta Chains



(a) Single delta chain

(b) Single aggregated delta chain

(c) Multiple aggregated delta chains

Fig. D.2: Delta chain architectures

4.1 Multiple Delta Chains

As discussed in Section 3.3, ingesting new revisions as aggregate changesets can quickly become prohibitive for long revision histories when the RDF archive is stored in a single delta chain (see Figure D.2b). In such cases, we propose the creation of a fresh snapshot that becomes the new reference for subsequent deltas. Those new deltas will be smaller, and thus easier to build and maintain. They will also constitute a new delta chain as depicted in Figure D.2c.

While creating a fresh snapshot with a new delta chain should presumably reduce ingestion time for subsequent revisions, its impact on query efficiency remains unclear. V queries, for instance, will have to be evaluated on multiple delta chains, becoming more challenging to answer. In contrast, VM queries defined on revisions already materialized as snapshots should be executed much faster. Storage size and DM response time may be highly dependent on the actual evolution of the data. If a new version includes many deletions, fresh snapshots may be smaller than aggregated deltas. We highlight that in our proposed architecture, revisions stored as snapshots also exist as aggregated deltas w.r.t. the previous snapshot – as shown for revision 2 in Figure D.2c. Such a design decision allows us to speed up DM queries as explained later.

It follows from the previous discussion that introducing multiple snapshots and delta chains raises a natural question: *When is the right moment to create a snapshot?* We elaborate on this question from the perspectives of storage, ingestion time, and query efficiency next. We then explain how to query archives in a multi-snapshot setting in Section 5.

4.2 Strategies for Snapshot Creation

A key aspect of our proposed design is to determine the right moment to place a snapshot, as this decision is subject to a trade-off among ingestion speed, storage size, and query performance. We formalize this decision via an abstract *snapshot oracle* $f : \mathbb{A} \times \mathbb{U} \to \{0, 1\}$ that, given an archive $\mathcal{A} \in \mathbb{A}$ with k revisions and a changeset $u_{k-1,k} \in \mathbb{U}$, decides whether revision k should (1) or should not (0) be materialized as a snapshot – otherwise the revision is stored as an aggregated delta. The oracle can rely on properties of the archive and the input changeset to make a decision. In the following, we describe some natural alternatives for our snapshot oracle f and illustrate them with a running example (Table D.2). All strategies start with a snapshot at revision 0. Note that we do not provide an exhaustive list of all possible strategies one could implement.

Baseline. The baseline oracle never creates snapshots, except for the very first revision, i.e., $f(\mathcal{A}, u) \equiv (\mathcal{A} = \emptyset)$. This is akin to OSTRICH's snapshot policy [32].

Periodic. A new snapshot is created when a fixed number d of versions has been ingested as aggregated deltas, i.e., $f(\mathcal{A}, u) \equiv (|\mathcal{A}| \ mod \ (d+1) = 0)$. We call d the period.

Change-ratio. Long delta chains do not only incur longer ingestion times but also higher disk consumption due to redundancy in the aggregated changesets. When low disk usage is desired, the snapshot strategy may take into account the editing dynamics of the RDF graph. This notion has been quantified in the literature via the change ratio score [9]:

$$\delta_{i,j}(\mathcal{A}) = \frac{|u^+_{i,j}| + |u^-_{i,j}|}{|G_i \cup G_j|} \tag{D.1}$$

Given two revisions i and j, the change ratio normalizes the number of changes (additions and deletions) between the revisions by the joint size of the revisions. If we aggregate the change ratios of all the revisions coming after a snapshot revision s, we can estimate the amount of data not materialized in the snapshot for the current delta chain. A reasonable snapshot strategy would therefore bound the aggregated change ratios $\sum_{i=s+1}^{k} \delta_{s,i}$, put differently: $f(\mathcal{A}, u) \equiv (\sum_{i=s+1}^{k} \delta_{s,i}) \geq \gamma$ for some user-defined budget threshold $\gamma \in \mathbb{R}_{>0}$.

Time. If we denote by t_k the time required to ingest revision k as an aggregated changeset in an archive $\mathcal{A}$, this oracle is implemented as $f(\mathcal{A}, u) \equiv (\frac{t_k}{t_{s+1}} > \theta)$, where $s+1$ is the first revision stored as an aggregated changeset in the current delta chain. This strategy therefore creates a new snapshot as soon as ingestion time exceeds θ times the ingestion time of version $s+1$.

Version (k)	0	1	2	3	4	5
$\lvert u^{+}_{i,j} \rvert$	100	20	20	20	20	20
$\lvert u^{-}_{i,j} \rvert$	0	10	10	10	10	10
$\sum_{i=s+1}^{k} \delta_{s,i}$	-	0.25	0.68	1.24	0.20	0.55
t_k	-	1.00	1.50	2.25	3.38	1.00
Baseline	**S**	Δ	Δ	Δ	Δ	Δ
Periodic ($d = 2$)	**S**	Δ	**S**	Δ	**S**	Δ
Change ratio ($\gamma = 1.0$)	**S**	Δ	Δ	**S**	Δ	Δ
Time ($\theta = 3.0$)	**S**	Δ	Δ	Δ	**S**	Δ

Table D.2: Creation of snapshots according to the different strategies on a toy RDF graph comprised of 100 triples and 5 revisions defined by changesets. An **S** denotes a snapshot whereas a Δ denotes an aggregated changeset.

4.3 Implementation

We implemented the proposed snapshot creation strategies and query algorithms for RDF archives on top of OSTRICH [32]. We briefly explain the most important aspects of our implementation.

Storage. In OSTRICH, an RDF archive consists of a snapshot for revision 0 and a single delta chain of aggregated changesets for the upcoming revisions (Fig. D.2b). The snapshot is stored as an HDT [8] file, whereas the delta chain is materialized in two stores: one for additions and one for deletions. Each store consists of 3 indexes in different triple component orders, namely SPO, OSP, and POS, implemented as B+trees. Keys in those indexes are individual triples linked to version metadata, i.e., the revisions where the triple is present and absent. Besides the change stores, there is an index with addition and deletion counts for all possible triple patterns, e.g., ⟨ `?s`, `?p`, `?o` ⟩ or ⟨ `?s`, `cityIn`, `?o` ⟩, which can be used to efficiently compute cardinality estimations – particularly useful for SPARQL engines.

Dictionary. As common in RDF stores [17, 37], RDF terms are mapped to an integer space to achieve efficient storage and retrieval. Two disjoint dictionaries are used in each delta chain: the snapshot dictionary (using HDT) and the delta chain dictionary. Hence, our multi-snapshot approach uses $D \times 2$ (potentially non-disjoint) dictionaries, where D is the number of delta chains in the archive.

Ingestion. The ingestion routine depends on whether a revision will be stored as an aggregated delta or as a snapshot. For revision 0, our ingestion routine takes as input a full RDF graph to build the initial snapshot. For subsequent revisions, we take as input a standard changeset $u_{k-1,k}$ ($|\mathcal{A}| = k$), and use OSTRICH to construct an aggregated changeset of the form $u_{s,k}$, where revision $s = \mathit{snapshot}(k)$ is the latest snapshot in the history. When the snapshot policy decides to materialize a revision s' as a snapshot, we

use the aggregated changeset $u_{s,s'}$ to compute the snapshot efficiently as $G_{s'} = (G_s \setminus u^-_{s,s'}) \cup u^+_{s,s'}$.

Change-ratio estimations. The change-ratio snapshot strategy computes the cumulative change ratio of the current delta chain w.r.t. a reference snapshot s to decide whether to create a new snapshot or not. We therefore store the approximated change ratios $\delta_{s,k}$ of each revision in a key-value store. To approximate each $\delta_{s,k}$ according to Equation D.1, we rely on OSTRICH's count indexes. The terms $|u^+_{s,k}|$ and $|u^-_{s,k}|$ can be obtained from the count indexes of the fully unbounded triple pattern $\langle$ `?s`, `?p`, `?o` $\rangle$ in $O(1)$ time. We estimate $|G_s \cup G_j|$ as $|G_s| + |u^+_{s,j}|$, where $|G_s|$ is efficiently provided by HDT.

The source code of our implementation as well as the experimental scripts to reproduce the results of this paper are available in a Zenodo archive[2].

5 Single Queries on Archives with Multiple Delta Chains

In the following, we detail our algorithms to compute version materialization (VM), delta materialization (DM), and V (version) queries on RDF archives with multiple delta chains. Our algorithms focus on answering single triple patterns queries, since they constitute the building blocks for answering arbitrary SPARQL queries – which we address in Section 7. All the routines described next are defined w.r.t. to an implicit RDF archive $\mathcal{A}$.

5.1 VM Queries

In a single delta chain with aggregated deltas and reference snapshot s, executing a VM query with triple pattern p on a revision i requires us to materialize the target revision as $G_i = (G_s \cup u^+_{s,i}) \setminus u^-_{s,i}$ and then execute p on G_i. In our baseline OSTRICH, $s = 0$. In the presence of multiple delta chains, we define $s = \mathit{snapshot}(i)$ as the revision number of i's reference snapshot in the archive's history.

Algorithm 1 provides a high level description of the query algorithm used for version materialization queries (VM). Our baseline, OSTRICH, uses a similar algorithm where $\mathit{sid}_i = 0$. The algorithm starts by getting the corresponding snapshot of the target version (line 2), and retrieving the matches of the query triple pattern (line 3) on the snapshot – as a stream of results. If the target version correspond to the snapshot, the query stops there and we return the results stream (line 5). Otherwise, the algorithm retrieves, from the delta chain indexes, those added and deleted triples of the target version that match the given query pattern (line 7 and 8). The deleted triples are then

[2] https://doi.org/10.5281/zenodo.7256988

Algorithm 6 VM query algorithm

```
1:  function queryVM(i, p)  ▷ version i, triple pattern p
2:      sid_i ← snapshot(i)
3:      q_i ← query(sid_i, p)  ▷ we query the triple pattern on the snapshot
4:      if sid_i = i then  ▷ the target version correspond to a snapshot
5:          return q_i
6:      u^+ ← getAdditions(i, p)
7:      u^− ← getDeletions(i, p)
8:      vm_i ← q_i \ u^−  ▷ filter out the deleted triples
9:      vm_i ← vm_i ∪ u^+  ▷ add the added triples
10:     return vm_i
```

filtered out from the snapshot results (line 9), which are extended with the added triples (line 10). It is important to note that this process is implemented in a streaming way, and is therefore computed lazily, as needed by the query consumer.

5.2 DM Queries

Algorithm 7 DM query algorithm on a single delta chain

```
1:  function singleDCQueryDM(i, j, p)  ▷ versions i, j, triple pattern p
2:      sid_i ← snapshot(i)
3:      if sid_i = i then  ▷ the start version correspond to the snapshot
4:          u_j^+ ← getAdditions(j, p)
5:          u_j^− ← getDeletions(j, p)
6:          return u_j^+, u_j^−
7:      else
8:          u_i^+ ← getAdditions(i, p); u_j^+ ← getAdditions(j, p)
9:          u_i^− ← getDeletions(i, p); u_j^+ ← getDeletions(j, p)
10:         u_{i,j}^+ ← u_j^+ \ u_i^+
11:         u_{i,j}^− ← u_j^− \ u_j^−
12:         return u_{i,j}^+, u_{i,j}^−
```

Algorithm 7 describes the procedure *singleDCQueryDM* that answers DM queries with start and end versions i, j on a single delta chain for triple pattern p. This procedure is crucial for handling DM query algorithms on multiple delta chains. This algorithm consists of two cases: The first case, described on lines 3–6, is met when the start version i corresponds to the

snapshot of the delta chain. When this is the case, the execution of the query is trivial as triple pattern p can be directly evaluated on the corresponding aggregated delta $u_{i,j}$. The second case, starting from line 7, deals with a start version stored as a delta. We get the changes (additions and deletions) for both the start and end versions (lines 8–9), and filter the additions and deletions so that the ones from the end version j prevail (lines 10–11). The results consist of the combination of the newly computed addition and deletion sets. In practice, this can be efficiently implemented as a sort-merge join operation where triples are emitted only when present for version j, or when their addition flag for i and j is different (in which case the flag for version j is kept).

Algorithm 8 DM query algorithm on multiple delta chains

```
1:  function snapshotDiff(S_i, S_j, p)  ▷ snapshots S_i, S_j, triple pattern p
2:      d ← j − i
3:      if d > 1 then
4:          q_i ← query(S_i, p); q_j ← query(S_j, p)
5:          delta ← (q_j \ q_i) ∪ (q_i \ q_j)
6:      else
7:          delta = singleDCQueryDM(i, j, p)
8:      return delta
9:
10: function queryDM(i, j, p)  ▷ versions i, j, triple pattern p
11:     sid_i ← snapshot(i); sid_j ← snapshot(j)
12:     if sid_i = sid_j then  ▷ i and j are in the same delta-chain
13:         delta ← singleDCQueryDM(i, j, p)
14:     else ▷ i and j are not in the same delta-chain
15:         u_{si,sj} ← snapshotDiff(sid_i, sid_j, p)
16:         u_{si,i}, u_{sj,j} ← ∅
17:         if i ≠ sid_i then  ▷ test if version i is a delta
18:             u_{si,i} ← singleDCQueryDM(sid_i, i, p)
19:         if j ≠ sid_j then  ▷ test if version j is a delta
20:             u_{sj,j} ← singleDCQueryDM(sid_j, j, p)
21:         u_{i,sj} ← mergeBackwards(u_{si,i}, u_{si,sj})
22:         (u_i, u_j) ← mergeForward(u_{i,sj}, u_{sj,j})
23:     return u_i, u_j
```

We now turn our attention to archives with multiple delta chains. The procedure *queryDM* in Algorithm 8 describes how to answer a DM query on two revisions i and j ($i < j$) with triple pattern p on an RDF archive with multiple delta chains. The algorithm relies on two important sub-routines, which

we now explain. The first one, *singleDCQueryDM*, was already described in Algorithm 7 and executes standard DM queries on single triple patterns over a single delta chain. The second routine, called *snapshotDiff*, computes the difference between the results of p on two reference snapshots S_i and S_j. It works by first testing if the delta chains of S_i and S_j are not consecutive (line 2 in Algorithm 8). If they are not, *snapshotDiff* implements a set-difference between p's results on S_i and S_j (lines 4–5). In case the snapshots define consecutive delta chains, we leverage the fact that S_j also exists as an aggregated delta w.r.t. S_i (see Section 4.1). We can therefore treat this case efficiently as a standard DM query via *singleDCQueryDM* (line 7).

We now have the elements to explain the main DM query procedure (*queryDM*). First, the procedure checks whether both revisions are in the same delta chain, i.e., if they have the same reference snapshot (line 14). If so, the problem boils down to a single delta chain DM query that can be answered with *singleDCQueryDM* (line 15). Otherwise, we invoke the routine *snapshotDiff* on the reference snapshots (line 17) to compute the results' difference between the delta chains. This is denoted by $u_{si,sj}$.

If revisions i and j are not snapshots themselves, lines 20 and 23 compute the changes between the target versions and their corresponding reference snapshots – denoted by $u_{si,i}$ and $u_{sj,j}$. The last steps, i.e., lines 25 and 26, merge the intermediate results to produce the final output. First, the routine *mergeBackwards* merges $u_{si,sj}$, i.e., the changes between the two delta chains, with $u_{si,i}$, i.e., the changes within the first delta chain. This routine is designed as a regular sorted merge because triples are already sorted in the OSTRICH indexes. Unlike a classical merge routine, *mergeBackwards* inverts the flags of the changes present in $u_{si,i}$ but not in $u_{si,sj}$. Indeed, if a change in $u_{si,i}$ did not survive to the next delta chain, it means it was later reverted in revision sid_j. The result of this operation are therefore the changes between revisions i and sid_j, which we denote by $u_{i,sj}$. The final merge step, *mergeForward*, combines $u_{i,sj}$ with the changes in the second delta chain, i.e., $u_{sj,j}$. The routine *mergeForward* runs also a sorted merge, but now triples with opposite change flag present in both changesets are filtered from the final output as they indicate reversion operations.

5.3 V Queries

Algorithm 9 describe how V queries are executed in a single delta chain setup. This is akin to how our baseline, OSTRICH, processes queries, and is used by our multiple snapshot query algorithm. We assume that each triple is annotated with its version validity, i.e. a vector of versions in which the triple exists. In OSTRICH, this is stored directly in the delta chain as versioning metadata, and therefore does not need additional computation. For the deletion triples, this metadata contains the list of versions where the

Algorithm 9 V query algorithm for single delta chains

```
1:  function singleDCQueryV(c, p)  ▷ c is a delta chain, p is a triple pattern
2:      v ← ∅
3:      s ← getSnapshot(c)  ▷ we get the snapshot of the delta chain
4:      q_s ← query(s, p)  ▷ query the snapshot with p
5:      q_a ← queryAdditions(c, p)  ▷ query the delta chain additions with p
6:      for t ∈ q_s ∪ q_a do  ▷ we iterate the triples from the queries
7:          t_del ← getDeletionTriple(t, c)  ▷ get the triple from the deletion set
    of the delta chain
8:          if t_del ≠ ∅ then
9:              t.versions ← t.versions \ t_del.versions  ▷ filter the deleted ver-
    sions from the triple valid versions
10:         v.add(t)
11:     return v
```

triple is absent instead. The core of the algorithm iterates over the triples (line 6) that match triple pattern p in the snapshot. Each triple is queried for its existence in the deletion delta chain (line 7). If the triple exists there, then it has been deleted in a subsequent revision. We remove the versions where the triple is absent from the version validity set of the triple (line 9). Finally, we add the triple to the result set in line 11. Like the other algorithms, this routine can also be implemented in a streaming way, where each loop iteration is triggered on demand.

Algorithm 10 V query algorithm for multiple delta chains

```
1:  function queryV(p)  ▷ p a triple pattern
2:      r ← ∅
3:      for c ∈ C do  ▷ C the list of delta chains
4:          v ← singleDCQueryV(c, p)
5:          r ← merge(r, v)  ▷ merge intermediate results
6:      return r
```

Algorithm 10 describes the process of executing a V query p over multiple delta chains. This relies on the capability to execute V queries on individual delta chains via the function *singleQueryV* described above. The routine iterates over the list of delta chains (line 3), and runs *singleQueryV* on each delta chain (line 4). This gives us triples annotated with lists of versions within the range of the delta chain. At each iteration we carry out a merge step (line 5) that consists of a set union of the triples from the current delta chain and the results seen so far. When a triple is present in both sets, we merge their lists of versions.

6 Optimization of Versioning Metadata Serialization

The versioning metadata stored in the delta chain indexes is paramount to functioning of our solution, and influences the multiple aspects of the system's performance. One of the current limitations of our architecture based on aggregated deltas, is that it does not scale well in terms of ingestion speed and disk usage, when the number of versions grows. As we show in this section, this happens because the delta chain indexes still contain a lot of redundancy that could be removed with a proper compression scheme. In this section, we therefore discuss the limitations of the current serialization of the versioning metadata, and propose an alternative serialization scheme that brings significant improvements in terms of ingestion speed and disk usage.

6.1 Versioning Metadata Encoding

As discussed in Section 3.3, OSTRICH indexes additions and deletions in separate triple stores for each delta chain. Because aggregated deltas introduce redundancy, OSTRICH annotates triples with additional version metadata that prevents the system from storing the same triple multiple times. This versioning metadata is in turn leveraged during querying to filter triples based on their version validity.

Version	2	3	4	6
LC	T	T	T	T

(a) Original addition metadata in OSTRICH

Version	[2,4)	-	-	[5,∞)
LC	[2,∞)	-	-	-

(b) Compressed addition metadata

Version	2	3	4	6
LC	F	F	F	T
SP?	0	0	0	0
S?O	0	0	0	0
S??	4	6	6	0
?PO	0	0	0	1
?P?	6	8	8	0
??O	0	0	0	0
???	8	8	8	0

(c) Original deletion metadata in OSTRICH

Version	[2,5)	-	-	[6,∞)
LC	-	-	-	[6,∞)
SP?	0	-	-	-
S?O	0	-	-	-
S??	4	+2	-	-6
?PO	0	-	-	+1
?P?	6	+2	-	-8
??O	0	-	-	-
???	8	-	-	-8

(d) Compressed deletion metadata

Table D.3: Representation of the versioning metadata in the indexes for arbitrary example triples in OSTRICH and compressed in our new implementation. *LC* denotes the local change flag.

In Table D.3, we show side-by-side the versioning metadata of an arbitrary triple as stored by OSTRICH (see Section 3.3), and as stored using our pro-

posed representation. Even though triples are stored only one, we observe that the index entries still store a lot of repeated information. Consider the example in Table D.3a where we can see that the local change flag is always set to true for each version. Our proposed representation compresses this information by storing intervals of versions where the value does not change. In our example, in Table D.3b, the local change flag is stored as an interval [2,∞), meaning that the flag is true starting from revision 2. We highlight that the version numbers are also stored as intervals. Similarly to the uncompressed metadata, the logical model is one of a mapping from version to a local change flag. In practice, this means that if a version is not present in any of the intervals, then there is no corresponding valid local change flag, regardless of the content of the local change intervals.

Deletion indexes contain more metadata than the addition indexes. Indeed, they also store the relative position of the triple within its respective delta for all triple pattern combinations, as illustrated in Table D.3c. This data can be large, especially for long delta chains, and can be both costly to create during ingestion and to deserialize during querying. OSTRICH alleviates these issues by restricting this metadata to the SPO index. We propose to replace this representation by a delta-compressed vector list, as illustrated in Table D.3d. This first position vector in the list is stored plainly, as before, but we only store deltas for subsequent changes. In case where no changes occur in a given revision, like in version 4 of our example, the vector is empty. In next section we elaborate on the implementation details of this serialization scheme.

6.2 Implementation Considerations

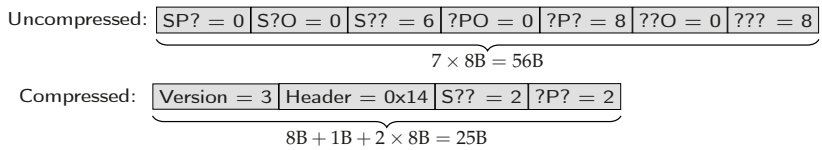


Fig. D.3: Representation of the positions vectors for version 3 of our example, without and with compression.

As depicted in Figure D.3d, some of the entries in the position vectors of the deleted triples can be empty. We handle those empty entries by means of a 8-bit header mask that precedes the position vector. Consider the second column vector (version 3) in our example Table D.3d. This vector contains two values: +2 for the triple pattern S?? in the the third position and +2 for the triple pattern ?P? in the fifth position. As such, the 8-bits header would be the string "0010100" (or 14 in hexadecimal), with 1s indicating the

positions where valid values exists. Notice that this header mask is preceded by the version identifier, since this number cannot be easily inferred from the intervals. Figure D.3 offers a visual representation of the serialization of the position vectors for our example D.3. This compressed representation uses only 25 bytes, versus the 56 bytes required by the original serialization.

Furthermore, we highlight a key advantage of delta encoding: since most vector entries consist of small values, our representation can benefit from further compression via variable size integer encoding. The compression is ultimately dependant on how often the value of the positions change between versions. Our experiments Section 8.4 demonstrate the efficacy of our approach in practice.

7 SPARQL 1.1 support for RDF Archives

Section 5 described the algorithms to answer versioned queries on single triple patterns on top of our multi-snapshot storage engine. In this section we describe our solution to support SPARQL queries over RDF Archives. We will first discuss how to formulate versioned queries using SPARQL. We then provide details of the proposed architecture and query engine.

7.1 SPARQL Versioned Queries

As discussed in Section 2, there have been a few efforts to write versioned queries comprising multiple triple patterns. All those endeavors rely on ad-hoc extensions to the SPARQL language. For this reason, none of those extensions has reached broad community acceptance. As consequence, we have opted for a query middle-ware based on native SPARQL that models revisions as named graphs [12]. Versioning requirements are therefore expressed using the SPARQL *GRAPH* keyword on named graphs with URIs of the form *<version:i>*, e.g., *<version:0>* denotes the initial revision. Our SPARQL engine interprets the provided graph URI and translates it into a proper retrieval operation within the physical data model described earlier in this paper. Our approach supports the base versioned query types, namely Version Materialization (VM), Delta Materialization (DM), and Version (V) queries.

We illustrate the different queries with an example RDF Archive $\mathcal{A}$ describing information about countries. For the sake of simplicity, we assume that each version of the graph G_i represents a specific year, e.g., G_{2003} contains information about countries in the year 2003. Table D.4 illustrates different versioned queries asking for country membership in the European Union (EU). First, VM queries are similar to standard SPARQL queries where the *GRAPH* clause is used to limit query evaluation to the target version. DM queries require the use of *FILTER* sub-queries to select the changes between

Version Materialization (VM) Which countries were part of the EU in 2003?	**Delta Materialization (DM)** Which countries joined the EU in 2004?	**Version Query (V)** Which countries were part of the EU in each year?

SELECT * WHERE {
GRAPH <version:2003> {
?country rdf:type ex:country .
?country ex:member ex:EU .
}
}
	SELECT * WHERE {
	GRAPH <version:2004> {
	?country rdf:type ex:country .
	?country ex:member ex:EU .
	} FILTER (NOT EXISTS {
	GRAPH <version:2003> {
	?country rdf:type ex:country .
	?country ex:member ex:EU .
	}
	})
	}
		SELECT * WHERE {
		GRAPH ?version {
		?country rdf:type ex:country .
		?country ex:member ex:EU .
		}
		}

<ex:Austria> <ex:Belgium> <ex:Denmark> <ex:Finland> ...	<ex:Cyprus> <ex:Czech Republic> <ex:Estonia> <ex:Hungary> ...	<ex:Austria> <version:1995> <ex:Austria> <version:1996> ... <ex:Belgium> <version:1958> ...

Table D.4: Example of SPARQL representation and results for VM, DM, and V queries using the *GRAPH* keyword.

versions as exemplified in Table D.4. The example DM query retrieves the countries that joined in revision 2004, i.e., EU members in $u^{+}_{2003,2004}$. In our design, a query on $u^{-}_{2003,2004}$ (deletions) has the same form but the version numbers are swapped. Finally, V queries can be expressed with the *GRAPH* keyword followed by a variable.

7.2 Architecture and Implementation

In order to support full SPARQL query processing over RDF archives, we make use of the *Comunica* [34] query engine deployed on top of our multi-snapshot storage layer and our algorithms for processing single triple patterns (described in Section 5). Comunica is a scalable and extensible query engine written in TypeScript with full support for the SPARQL 1.1 language. Due to its modularity, it is a natural choice for extending our system, and initial work was already done by Taelman et al. [33] to support archives queries (although without V queries support). We have adapted this implementation to our multi-snapshot storage architecture, and extended it to support V queries. We have also implemented several optimizations, notably in regards to the communication between the query engine and the storage layer. Previously, results from a triple pattern query would be buffered in OSTRICH until all results have been gathered. Because Comunica is designed to work with streams of triples, this create locking in the query processing while waiting for the availability of the triples. Instead, we now buffer triples into smaller buffers of configurable size. When a buffer is filled, the triples it contains can be sent to Comunica without waiting the remaining triples. This permit shorter locking time in Comunica and allows us to take better advantage of its asynchronous query processing capabilities.

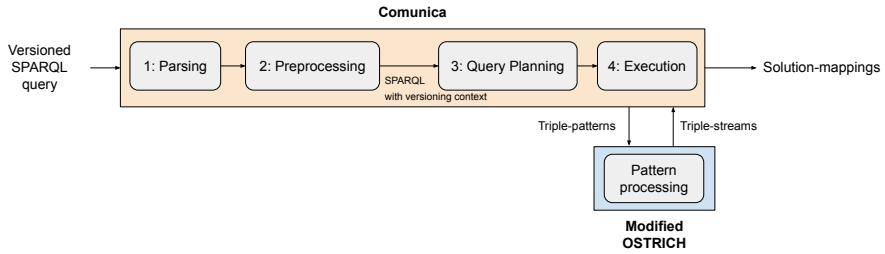


Fig. D.4: SPARQL query processing pipeline

Figure D.4 illustrates the query processing pipeline of our solution for SPARQL queries on RDF archives. There are two main software components interacting with each other: the first is the storage layer consisting of our multi-snapshot version of OSTRICH (see Sections 4 and 5), and the second is the Comunica [34] query engine, which includes several modules.

A versioned SPARQL query like the ones in Table D.4 (Section 7.1), is first transformed back to a graph- and filter-free SPARQL query, i.e. without the special GRAPH URIs and/or FILTER clauses, and annotated with a versioning context. This versioning context depends on the query type (e.g., revisions for VM queries, changesets for DM queries) and the target version(s) when relevant, and is used to select the type of triple pattern queries to send for execution by OSTRICH. The communication between Comunica and OSTRICH is done through a NodeJS native addon. Our implementation is open source[3][4] and a demonstration system is available at [20].

8 Experiments

To determine the effectiveness of our multi-snapshot approach for RDF archiving, we evaluate the four proposed snapshot creation strategies described in Section 4 along three dimensions: ingestion time (Section 8.2), disk usage (Section 8.2), and query runtime for VM, DM, and V queries (Section 8.3). Thereafter, in Section 8.4, we delve into the effectiveness of our versioning metadata representation described in Section 6. This is done by comparing its performance against the original representation – across the three aforementioned evaluation dimensions. Section 8.5 concludes our experiments with an evaluation of our full SPARQL query capabilities.

[3] https://github.com/dkw-aau/ostrich-node
[4] https://github.com/dkw-aau/comunica-feature-versioning

8.1 Experimental Setup

We resort to the BEAR benchmark for RDF archives [9] for our evaluation. BEAR comes in three flavors: BEAR-A, BEAR-B, and BEAR-C, which comprise a representative selection of different RDF graphs and query loads. Table D.5 summarizes the characteristics of the experimental datasets and query loads. Due to the very long history of BEAR-B instant, OSTRICH could only ingest one third of the archive's history (7063 out 21046 revisions) after one month of execution – before crashing. In a similar vibe, OSTRICH took one month to ingest the first 18 revisions (out of 58) of BEAR-A. Despite the dataset's short history, changesets in BEAR-A are in the order of millions of changes, which also makes ingestion intractable in practice. On these grounds, the original OSTRICH paper [32] excluded BEAR-B instant from the evaluation, and considered only the first 10 versions of BEAR-A. Multi-snapshot solutions, on the other hand, allow us to manage these datasets. We provide, nevertheless, the results of the baseline strategy (OSTRICH) for entire history of BEAR-A. We emphasize however that ingesting this archive was beyond what would be considered reasonable for any use case: it took more than five months of execution. We provide those results as a baseline (although an easy one) to highlight the challenge of scaling to large datasets. All our experiments were run on a Linux server with a 16-core CPU (AMD EPYC 7281), 256 GB of RAM, and 8TB hard disk drive.

		BEAR-B			
	BEAR-A	Daily	Hourly	Instant	BEAR-C
# versions	58	89	1299	21046	32
$\|G_i\|$'s range	30M - 66M	33K - 44K	33K - 44K	33K - 44K	485K - 563K
$\overline{\|\Delta\|}$	22M	942	198	23	568K
# queries	368	62 (49 ?P? and 13 ?PO)			11 (SPARQL)

Table D.5: Dataset characteristics. $|G_i|$ is the size of the individual revisions, $\overline{|\Delta|}$ denotes the average size of the individual changesets $u_{k-1,k}$.

We evaluate the different strategies for snapshot creation detailed in Section 4.2 along ingestion speed, storage size, and query runtime. Except for our baseline (OSTRICH), all our strategies are defined by parameters that we adjust according to the dataset:

Periodic. This strategy is defined by the period d. We set $d \in \{2, 5\}$ for BEAR-A and BEAR-C, $d \in \{5, 10\}$ for BEAR-B daily, $d \in \{50, 100\}$ for BEAR-B hourly, and $d \in \{100, 500\}$ for BEAR-B instant. Values of d were adjusted per dataset experimentally w.r.t. the length of the revision history and the baseline ingestion time. High periodicity, i.e., smaller values for d, lead to more and shorter delta chains.

Change-ratio (CR). This strategy depends on a cumulative change-ratio bud-

get threshold γ. We set $\gamma \in \{2.0, 4.0\}$ for all the tested datasets. $\gamma = 2.0$ yields 10 delta chains for BEAR-A, 9 for BEAR-C, as well as 5, 23, and 151 delta chains for BEAR-B daily, hourly, and instant, respectively. For $\gamma = 4.0$, we obtain instead 6 delta chains for BEAR-A, 6 for BEAR-C, and 3, 16, and 98 for the BEAR-B datasets.

Time. This strategy depends on the ratio θ between the ingestion time of the new revision and the ingestion time of the first delta in the current delta chain. We set $\theta = 20$ for all datasets. This produces 3, 26, and 293 delta chains for the daily, hourly, and instant variants of BEAR-B respectively, and 2 delta chains for BEAR-A. As for BEAR-C, no new delta chains are created with $\theta = 20$, and so it is equivalent to the baseline.

We omit the reference systems included with the BEAR benchmark since they are outperformed by OSTRICH [32].

8.2 Results on Resource Consumption

		BEAR-B			
	BEAR-A	Daily	Hourly	Instant	BEAR-C
High Periodicity	**13472.16**	**0.67**	**12.95**	57.89	**43.97**
Low Periodicity	14499.45	0.98	23.05	298.36	96.38
CR $\gamma = 2.0$	20505.93	1.88	13.79	77.01	75.61
CR $\gamma = 4.0$	21588.25	2.34	19.47	114.83	111.78
Time $\theta = 20$	49506.15	2.64	15.83	**43.53**	543.82
Baseline	253676.98	6.89	1514.85	-	550.90

(a) Ingestion times in minutes

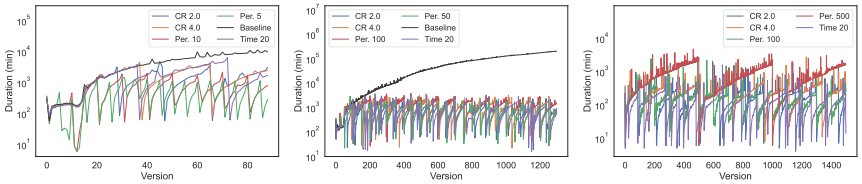
		BEAR-B			
	BEAR-A	Daily	Hourly	Instant	BEAR-C
High Periodicity	72417.47	199.17	322.34	2283.43	1149.63
Low Periodicity	49995.00	102.96	**185.33**	**787.75**	**890.44**
CR $\gamma = 2.0$	47335.74	51.49	284.47	1690.38	920.46
CR $\gamma = 4.0$	**42203.04**	37.91	211.71	1175.15	939.30
Time $\theta = 20$	46614.98	38.33	325.13	3972.32	1365.10
Baseline	45965.40	**19.82**	644.50	-	1365.10

(b) Disk usage in MB

Table D.6: Time and disk usage used by our different strategies to ingest the data of the BEAR benchmark datasets.

Ingestion Time

Table D.6a depicts the total time to ingest the experimental datasets. Since we always test two different values of d for the periodic strategy on each



(a) BEAR-B Daily **(b)** BEAR-B Hourly **(c)** BEAR-B Instant

Fig. D.5: Detailed ingestion times (log scale) per revision. We include the first 1500 revisions for BEAR-B instant since the runtime pattern is recurrent along the entire history.

dataset, in both Table D.6a and D.6b, we refer to them as "high" and "low" periodicity. This is meant to abstract away the exact parameters, which vary for each dataset, so that we can focus instead on the effects of higher/lower periodicity. We remind the reader that the baseline (OSTRICH) cannot ingest BEAR-B instant, which explains its absence in Table D.6a. But even when OSTRICH can ingest the entire history (in around 26 hours), a multi-snapshot strategy still incurs a significant speed-up. This becomes more significant for long histories as observed for BEAR-B hourly, where the speed-up can reach two orders of magnitude. The good performance of the high periodicity strategy and change-ratio with the smaller budget threshold $\gamma = 2.0$ suggests that shorter delta chains are beneficial for ingestion time. This is confirmed by Fig. D.5, where we also notice that ingestion time reaches a minimum for the revisions following a snapshot.

Disk Usage

Unlike ingestion time where shorter delta changes are clearly beneficial, the gains in terms of disk usage depend on the dataset as shown in Table D.6b. Overall, more delta chains tend to increase disk usage. For BEAR-B daily, frequent snapshots (high periodicity $d = 5$) incur a large overhead w.r.t. the baseline because the changesets are small and the revision history is short. Similar results are observed for BEAR-A and BEAR-B instant, even though we still need multiple snapshots to be able to ingest the data. BEAR-B hourly is interesting because it shows that for long histories, a single delta chain can be inefficient in terms of disk usage. Interestingly for BEAR-A, the change-ratio $\gamma = 4.0$ uses less storage than the both the time strategy with $\theta = 20$ and the baseline, despite using more delta chains. This hints that very large aggregated deltas can be less efficient than multiple delta chains with smaller aggregated deltas. For BEAR-B instant, the good performance of the change-ratio strategies and the low periodicity strategy ($d = 500$) suggests that a few delta chains can provide significant space savings. On the other hand, the time strategy with $\theta = 20$ performs slight worse because it creates too

many delta chains. The bottom line is that redundancies in the delta chains explain the storage overhead in archives, can be caused either by very long delta chains (BEAR-B Hourly and Instant), or by large delta chains (BEAR-A), i.e., delta chains with voluminous changesets. Multiple snapshots tackle the redundancy of long delta chains naturally, but can also be beneficial for bulky delta chains, as demonstrated by the BEAR-A results with change-ratio $\gamma = 4.0$.

8.3 Query Runtime Evaluation

In this section we evaluate the impact of our snapshot creation strategies on query runtime. We use the queries provided with the BEAR benchmark for BEAR-A and BEAR-B. These are DM, VM, and V queries on single triple patterns. Each individual query was executed 5 times and the runtimes averaged. All the query results are depicted in Figure D.6.

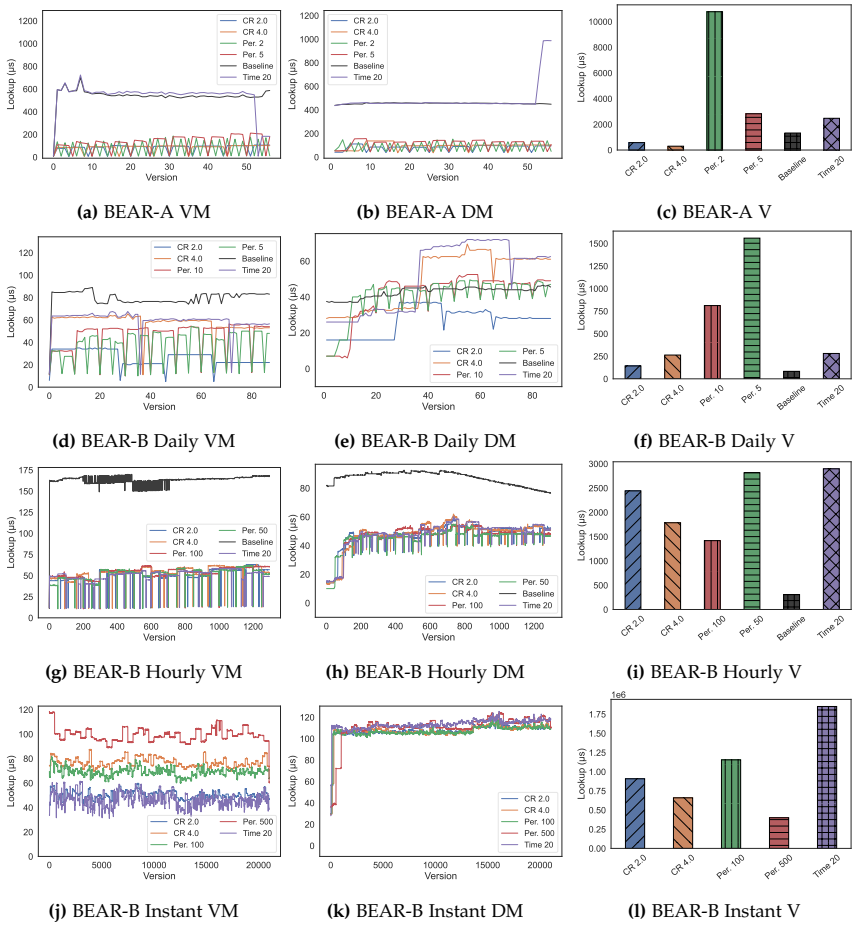
VM queries

We report the average runtime of the benchmark VM queries for each version i in the archive. The results are depicted in Figures D.6a, D.6d, D.6g, and D.6j. We report runtimes in micro-seconds for all strategies.

Using multiple delta chains is consistently beneficial for VM query runtime, which is best when the target revision is materialized as a snapshot. When it is not the case, runtime is proportional to the *size* of the delta chain, which depends on its length and the volume of changes that must be applied to the snapshot before running the query. This is obvious for BEAR-A with the baseline strategy or with the time $\theta = 20$ strategy. The latter strategy splits the history into two imbalanced delta-chains, where one of them contains the first 53 revisions (out of 58). Both strategies are significantly outperformed by the other multi-snapshot strategies. Similar results can be observed on the BEAR-B variants, particularly for BEAR-B Hourly, where the baseline strategy is outperformed by all the strategies with more than one snapshot.

DM Queries

We report for each revision i in the archive the average runtime of the benchmark DM queries on changesets $u_{0,i}$ and $u_{1,i}$. As described in Section 5.2, DM queries are executed on both the additions and deletions indexes in order to retrieve the full set of results for the given query pattern. Such a setup tests the query routine in all possible scenarios: between two snapshots, between a snapshot and a delta (and vice versa), and between two deltas. The results are depicted in Figures D.6b, D.6e, D.6h, and D.6k. The results shows a rather



(a) BEAR-A VM **(b)** BEAR-A DM **(c)** BEAR-A V

(d) BEAR-B Daily VM **(e)** BEAR-B Daily DM **(f)** BEAR-B Daily V

(g) BEAR-B Hourly VM **(h)** BEAR-B Hourly DM **(i)** BEAR-B Hourly V

(j) BEAR-B Instant VM **(k)** BEAR-B Instant DM **(l)** BEAR-B Instant V

Fig. D.6: Query results for the BEAR benchmark

mixed benefit of multiple delta chains in query runtime: highly positive for the long history of BEAR-B hourly and modest for BEAR-B daily. Overall, DM queries benefit from short delta chains as illustrated by Figure D.6b and to a lesser degree by the periodic strategy with $d = 5$ in Figure D.6e. All our strategies beat the baseline by a large margin on BEAR-B hourly because delta operations become very expensive as the single delta chain grows. That said, the baseline runtime tends to decrease slightly with i because the data from two distant versions tends to diverge more, which requires the engine to filter fewer results from the aggregated deltas. For BEAR-B daily, multiple delta chains may perform comparably or slightly worse – by no more than 20% – than the baseline. This happens because BEAR daily's history is short, and hence efficiently manageable with a single delta chain. In this

case the overhead of multiple snapshots and delta chains does not bring any advantage for DM queries.

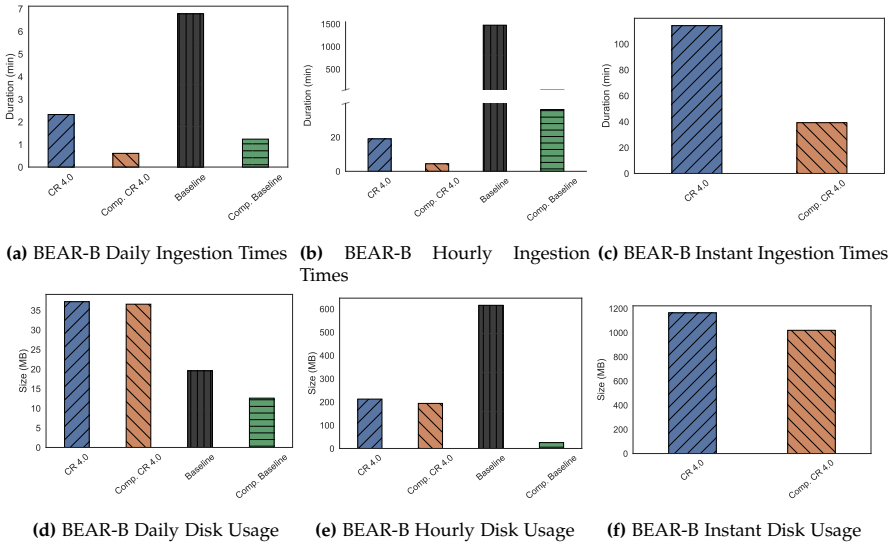
V Queries

Figure D.6c, D.6f, D.6i, and D.6l show the total runtime of the benchmark V queries on the different datasets. V queries are the most challenging queries for the multi-snapshot archiving strategies as suggested by Figures D.6f and D.6i. As described in Algorithm 10, answering V queries requires us to query each delta chain individually, buffer the intermediate results, and then merge them. It follows that runtime scales proportionally to the number of delta chains, which means that contrary to DM and VM queries, many short delta chains are detrimental to V query performance. The only exception is BEAR-A, where the change-ratio strategies can outperform the baseline strategy. This indicates that delta chains with very large aggregated deltas can also be detrimental to V query performance. However, BEAR-A is the only dataset showcasing such a behavior in our experiments. Nonetheless, due to their prohibitive ingestion cost, querying datasets such as BEAR-A and BEAR-B instant is only possible with a multi-snapshot solution.

8.4 Experiments on the Metadata Representation

We now evaluate our proposed encoding for versioning metadata, described in Section 6. We conduct the evaluation across the dimensions of ingestion time, disk usage, and query performance on the BEAR-B variants of the BEAR benchmark. For the sake of legibility, we apply our new encoding on archives stored using a single-snapshot strategy, i.e., the baseline OSTRICH, and one multi-snapshot strategy. We chose the change-ratio strategy with $\gamma = 4.0$ in this case since it exhibited overall good performance across the different evaluation criteria in Section 8.2. We omit the baseline strategy for BEAR-B Instant since we could not ingest it using a single snapshot.

Figure D.7 shows the ingestion time and disk usage of the BEAR-B variants with the original versioning metadata encoding as used in OSTRICH and our proposed compressed encoding – denoted with the "*Comp.*" prefix in the graphs. The new encoding incurs a drastic decrease in ingestion time. This is particularly notable for the baseline strategy where ingestion times are reduced by as much as a factor of 40 on the BEAR-B hourly dataset, as illustrated by Figure D.7b. Here, the ingestion time drops from 1473 minutes to just 36 minutes. Disk usage is also notably improved with the new encoding. This is particularly obvious for the baseline strategy, because larger delta chains imply more redundancy, which in turn means more room for compression. In the most remarkable case, disk usage is reduced from 615 MB to just 25 MB for BEAR-B hourly when using the baseline strategy. The



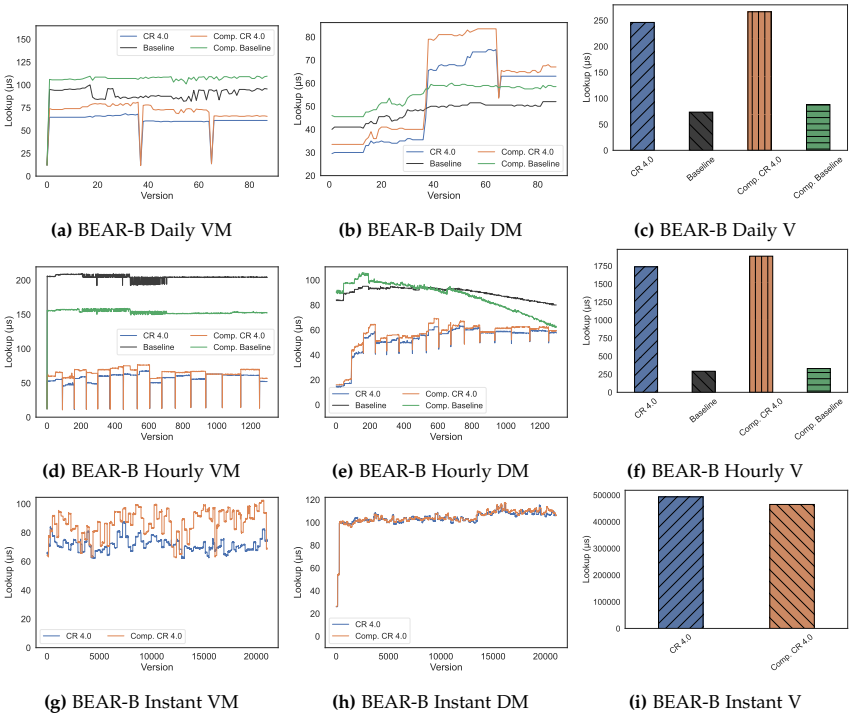
(a) BEAR-B Daily Ingestion Times **(b)** BEAR-B Hourly Ingestion Times **(c)** BEAR-B Instant Ingestion Times

(d) BEAR-B Daily Disk Usage **(e)** BEAR-B Hourly Disk Usage **(f)** BEAR-B Instant Disk Usage

Fig. D.7: Ingestion times (top row) and disk usage (bottom row) for OSTRICH and a multi-snapshot storage strategy applied on the different BEAR-B flavours with the original version metadata representation and our new compressed representation.

improvements in disk usage are more modest on multiple snapshot strategies, as expected from the smaller delta chains. In those cases, the snapshots account for more of the disk usage of the whole archive. For example, on BEAR-B hourly, disk usage is only reduced from 212 MB to 193 MB.

In Figure D.8, we show the performance of queries with the two encodings of versioning metadata. Contrary to ingestion times and disk usage, the picture for query runtime is more nuanced. Overall, our new encoding scheme does not provide a clear advantage over the previous encoding in terms of query performance. For BEAR-B Daily, as shown in Figures D.8a, D.8b and D.8c, the archives using the new encoding are systematically slower at resolving queries than the archives with the original encoding. As for BEAR-B Hourly, Figures D.8d, D.8e, and D.8c, we note that queries are faster with the new encoding on the baseline strategy, but slightly slower with the change-ratio strategy. We can explain this by the large amounts of data that must be retrieved from long delta chains. In such cases, the gains obtained by reading less data – thanks to compression – outweight the costs of decompression, which translates into overall faster retrieval times. Finally, for BEAR-B Instant, the compressed representation is slower for VM queries, similar for DM queries, and slightly faster for V queries when compared to the original representation. All in all, compressing the versioned metadata reduces the redundancy in the delta chains, which goes in the same direction as using shorter delta chains. This explains why the compressed representation



(a) BEAR-B Daily VM **(b)** BEAR-B Daily DM **(c)** BEAR-B Daily V

(d) BEAR-B Hourly VM **(e)** BEAR-B Hourly DM **(f)** BEAR-B Hourly V

(g) BEAR-B Instant VM **(h)** BEAR-B Instant DM **(i)** BEAR-B Instant V

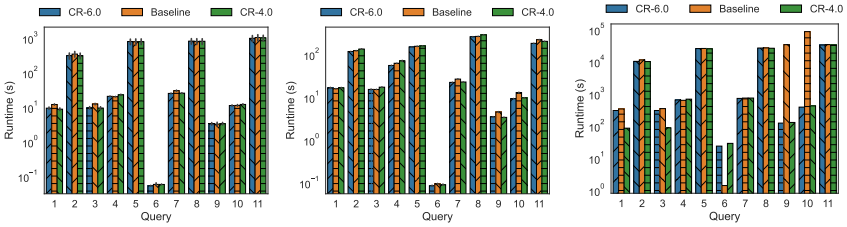
Fig. D.8: Query results for the BEAR benchmark with the original version metadata encoding and the new compressed encoding.

performs worst for querying on multiple delta chains: redundancy has been already (or partially) reduced by the use of multiple snapshots. This diminishes the gains of further compression with our approach, which comes with a performance penalty due to decompression.

8.5 SPARQL Performance Evaluation on BEAR-C

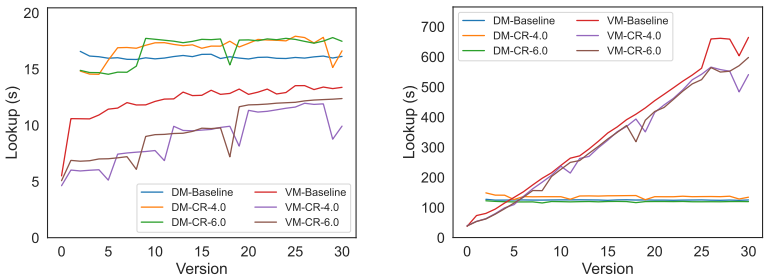
We evaluate our solution for full SPARQL support on the BEAR-C dataset. BEAR-C is based on 32 weekly snapshots of the European Open Data portal taken from the Open Data Portal Watch project [16]. Each version contains between 485K and 563K triples, which puts BEAR-C between BEAR-B Daily and BEAR-A in terms of size. Table D.5 in Section 8.1 summarizes the characteristics of the datasets. BEAR-C's query workload consists of 11 full SPARQL queries. The queries contain between 2 and 12 triple patterns and include the operators `FILTER`, `OPTIONAL` , `UNION`, `LIMIT` and `OFFSET`. In compliance with our experimental setup, we run each query 5 times and report the average runtime. Since there are no other publicly available SPARQL-

compliant RDF archiving systems [19], we compare our change-ratio (CR) multi-snapshot strategy ($\gamma = 4.0$ and $\gamma = 6.0$) to the baseline. We chose the CR multi-snapshot strategy due to its good overall performance in our evaluations in Sections 8.2 and 8.3. We include the strategy CR $\gamma = 6.0$ that generates snapshots less often than CR $\gamma = 4.0$ (used in the previous experiments). This is due to the smaller size of BEAR-C when compared to more challenging datasets such as BEAR-A or BEAR-B instant. Finally, we make use of the compressed representation for the versioning metadata presented in Section 6, and evaluated earlier in this section.



(a) BEAR-C VM **(b)** BEAR-C DM **(c)** BEAR-C V

Fig. D.9: BEAR-C average query execution time in seconds for VM, DM, and V queries. (log scale)



(a) DM and VM runtime for BEAR-C query #1 **(b)** DM and VM runtime for BEAR-C query #2

Fig. D.10: BEAR-C average query execution time in seconds for VM, DM, and V queries.

Figure D.9 illustrates the average execution time for each category of versioned query (VM, DM, V) on BEAR-C. The results are displayed per individual query and averaged across all revisions for VM queries, and pairs of revisions $\langle$ `1, i` $\rangle$ and $\langle$ `0, i` $\rangle$ for DM queries – in concordance with our experimental protocol. We note that the results are consistent with our single-triple patterns evaluation on the other BEAR datasets. That is, BEAR-C's relatively short history (32 versions) puts our CR stategies in disadvantage w.r.t. to the baseline strategy. The query runtime of the change-ratio strategies and the

baseline are almost identical for VM queries, with the change-ratio strategies having a slight advantage for some queries (notably, queries 1, 3, and 7). For DM queries, runtimes are also closely matched between the different strategies. Overall, the CR $\gamma = 6.0$ strategy performs best on average. Finally, we can observe large differences in runtimes between the different strategies on the V queries. While all strategies are closely matched overall, we can notice that the baseline strategy gets significantly outperformed on query 9 and 10, whereas it outperforms the CR strategies on query 6. The CR $\gamma = 4.0$ is notably faster than the alternatives on query 1 and 3. The overall good performance of the change-ratio strategies seems to contradict our previous findings, as V queries tend to become more expensive with more delta chains. However, we observed a similar behavior on BEAR-A in our previous experiments (Section 8.3), so to say, that the baseline strategy was outperformed by multi-snapshot strategies on V queries. This confirms the hypothesis that bulky deltas – common for BEAR-A and to a lesser extent for BEAR-C – are also detrimental to V query performance, justifying the use of multiple less voluminous delta chains.

In Figure D.10 we plot the runtime of VM and DM queries across revisions for queries #1 and query #2 of BEAR-C. The figures for all the other queries can be found in Section 10. We selected those queries due to their representative runtime behaviour. Query #1 has a relatively stable runtime for VM queries, with a slight increase in later revisions. Oppositely, query #2 sees a linear increase in runtime for VM queries as the target revision increases. In contrast, the runtime of DM queries is stable. We can observe that the CR strategies consistently outperform the baseline strategy on VM queries, while the baseline is faster on average for DM queries on query #1, and on par with CR $\gamma = 6.0$ for query #2. Overall, the differences between strategies are small, and vary depending on the query, as seen on Figure D.9.

8.6 Discussion

We now summarize our findings in previous sections and draw a few design lessons for efficient RDF archiving.

- The disk usage and overall performance in querying of a storage approach based on aggregated delta chains depends on the amount of redundancy present in the delta chain. This redundancy can be caused by various factors, such as large changesets, long change histories, but also by the nature of the changes, e.g., changes that revert previous changes.
- It follows that for small datasets, small changesets, or relatively short histories, the overhead of multi-snapshot strategies does not pay off in

terms of query runtime and disk usage. This observation is particularly striking for V queries for which runtime increases with the number of delta chains.
- Short delta chains are mostly beneficial for VM and DM queries because these query types require us to iterate over changes within two delta chains in the worst case (for DM queries). They also translate systematically into faster ingestion times. In contrast, numerous short delta chains are detrimental to storage consumption and V query performance.
- That said, when individual deltas are very bulky, as with BEAR-A and BEAR-C, multiple delta chains can be beneficial to V query performance, and can use less disk space than a single-snapshot storage strategy.
- Change-ratio strategies strike an interesting trade-off because they take into account the amount of data stored in the delta chain as criterion to create a snapshot. This ultimately has a direct positive effect on ingestion time, VM/DM querying, and storage size.
- In general, compressing the version metadata stored in the delta chain is a sensible alternative: compression increases ingestion speed and reduces disk storage. While it can increase query runtime, its impact is usually minimal and depends on the amount of data that needs to be fetched from disk. For very large delta chains (e.g., delta chains with big deltas), compression can even be beneficial for query performance because the overhead of decompression is insignificant compared to the savings in terms of retrieved data. This observation holds promise for distributed settings.
- The performance of full SPARQL queries on RDF archives is subject to same performance trade-off as queries on single triple patterns.

The bottom line is that the snapshot creation strategy for RDF archives is subject to a trade-off among ingestion time, disk consumption, and query runtime for VM, DM, and V queries. As shown in our experimental section, there is no one-size-fits-all strategy. The suitability of a strategy depends on the application, namely the users' priorities or constraints, the characteristics of the archive (snapshot size, history length, and changeset size), and the query load. For example, implementing version control for a collaborative RDF graph will likely yield an archive like BEAR-B instant, i.e., a very long history with many small changes and VM/DM queries mostly executed on the latest revisions. Depending on the server's capabilities and the frequency of the changes, the storage strategy could therefore rely on the change ratio or the ingestion time ratio and be tuned to offer arbitrary latency guarantees for ingestion. On a different note, a user doing data analytics on the published versions of DBpedia (as done in [19]) may be confronted to a dataset like

BEAR-A and therefore resort to numerous snapshots, unless their query load includes many real-time V queries.

Furthermore, we showcased our results for full SPARQL processing over RDF archives on the BEAR-C benchmark. To the best of our knowledge, this is the first approach that provides a solution for BEAR-C. Nonetheless, there are still several opportunities for future work in field of SPARQL querying over RDF archives. First, we highlight the lack of standardization for SPARQL querying on RDF archives. This has encouraged solution providers to come up with their own language extensions and ad-hoc implementations. None of them, however, has attained wide acceptance within the research and developer communities. Second, we note that the number and diversity of benchmarks for SPARQL query workloads on RDF archives is limited [21]. In BEAR, for example, only the BEAR-C dataset offers full SPARQL queries. Those 11 queries, are alas, insufficient to provide a comprehensive evaluation of the capabilities of novel systems. Alternatives, such as SPBv [18] have not seen similar adoption by the community, probably because they are not easy to deploy[5]. We expect this work to prepare the ground for the emergence of more efficient, standardized, and expressive solutions for managing RDF archives.

9 Conclusion

In this paper, we have presented a hybrid storage architecture for RDF archiving based on multiple snapshots and chains of aggregated deltas with support for full SPARQL versioned queries. We have evaluated this architecture with several snapshot creation strategies on ingestion times, disk usage, and query performance using the BEAR benchmark. The benefits of this architecture are bolstered thanks to a novel and efficient compression scheme for versioning metadata, which has yielded impressive improvements over the original serialization scheme. This has further improved the scalability of our system when handling large datasets with long version histories. All these building blocks cleared the way to introduce a new SPARQL processing system on top of our storage architecture. We are now capable of answering full SPARQL VM, DM or V queries over RDF archives.

Our evaluation shows that our architecture can handle very long version histories, at a scale not possible before with previous techniques. We used our experimental results on the different snapshot creation strategies to draw a set of design lessons that can help users choose the best storage policy based on their data and application needs. We showcased our ability to handle the BEAR-C variant of the BEAR benchmark – the first evaluation on this dataset

[5]We could not use the benchmark because it relies on outdated and unsatisfiable software dependencies.

to the best of our knowledge. This is a first step towards the support of more sophisticated applications on top of large RDF archives, and we hope it will expedite research and development in this area.

As future work, we plan to further explore different snapshot creation strategies, e.g., using machine learning, to further improve the management of complex and large RDF archives. Furthermore, we plan to investigate novel approaches in the compact representation of semantic data [24, 29], which could lead to a promising alternative to the use of B+ trees. We envision further efforts towards the practical implementation of versioning use cases of RDF, such as the implementation of version control features, like branching and tagging, into our system. Such features are paramount to real world usages of versioning software, and can benefit RDF dataset maintainers [4, 12]. With the recent popularity of RDF-star [1, 13], which can be used to capture versioning in the form of metadata, we also plan to look into recent advances in this area. Finally, the lack of an accepted standard for expressing versioning queries with SPARQL limits the wider adoption of RDF archiving systems. We aim to work towards a standardization effort, notably on a novel syntax and a formal definition of the semantics of versioned queries.

References

[1] G. Abuoda, C. Aebeloe, D. Dell'Aglio, A. Keen, and K. Hose, "Starbench: Benchmarking rdf-star triplestores," in *QuWeDa/MEPDaW@ISWC*, ser. CEUR Workshop Proceedings, vol. 3565. CEUR-WS.org, 2023, pp. 34–49.

[2] C. Aebeloe, G. Montoya, and K. Hose, "Colchain: Collaborative linked data networks," in *WWW*, 2021, pp. 1385–1396.

[3] J. Anderson and A. Bendiken, "Transaction-time queries in dydra," in *MEPDaW/LDQ@ESWC*, ser. CEUR Workshop Proceedings, vol. 1585. CEUR-WS.org, 2016, pp. 11–19.

[4] N. Arndt, P. Naumann, N. Radtke, M. Martin, and E. Marx, "Decentralized collaborative knowledge management using git," *J. Web Semant.*, vol. 54, pp. 29–47, 2019.

[5] K. Bereta, P. Smeros, and M. Koubarakis, "Representation and querying of valid time of triples in linked geospatial data," in *ESWC*, vol. 7882, 2013, pp. 259–274.

[6] J. Brunsmann, "Archiving Pushed Inferences from Sensor Data Streams," in *International Workshop on Semantic Sensor Web*, 2010, pp. 38–46.

[7] A. Cerdeira-Pena, G. de Bernardo, A. Fariña, J. D. Fernández, and M. A. Martínez-Prieto, "Compressed and queryable self-indexes for rdf archives," *Knowledge and Information Systems*, pp. 1–37, 2023.

[8] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange (HDT)," *J. Web Semant.*, vol. 19, pp. 22–41, 2013.

[9] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for RDF archives," *J. Web Semant.*, vol. 10, no. 2, pp. 247–291, 2019.

[10] V. Fionda, M. W. Chekol, and G. Pirrò, "Gize: A time warp in the web of data," in *ISWC*, vol. 1690, 2016.

[11] F. Grandi, "T-SPARQL: A tsql2-like temporal query language for RDF," in *ADBIS (Local Proceedings)*, 2010, pp. 21–30.

[12] M. Graube, S. Hensel, and L. Urbas, "R43ples: Revisions for triples - an approach for version control in the semantic web," in *LDQ@SEMANTICS*, 2014.

[13] O. Hartig, "Foundations of rdf⋆ and sparql⋆ (an alternative approach to statement-level metadata in RDF)," in *AMW*, ser. CEUR Workshop Proceedings, vol. 1912. CEUR-WS.org, 2017.

[14] K. Hose, "Knowledge graph (r)evolution and the web of data," in *MEPDaW@ISWC*, ser. CEUR Workshop Proceedings, vol. 3225. CEUR-WS.org, 2021, pp. 1–7.

[15] T. Huet, J. Biega, and F. M. Suchanek, "Mining History with Le Monde," in *Workshop on Automated Knowledge Base Construction*, 2013, pp. 49–54.

[16] S. Neumaier, J. Umbrich, and A. Polleres, "Automated quality assessment of metadata across open data portals," *ACM J. Data Inf. Qual.*, vol. 8, no. 1, pp. 2:1–2:29, 2016.

[17] T. Neumann and G. Weikum, "x-rdf-3x: Fast querying, high update rates, and consistency for RDF databases," *PVLDB*, vol. 3, no. 1, pp. 256–263, 2010.

[18] V. Papakonstantinou, G. Flouris, I. Fundulaki, K. Stefanidis, and Y. Roussakis, "Spbv: Benchmarking linked data archiving systems," in *BLINK/NLIWoD3@ISWC*, vol. 1932, 2017.

[19] O. Pelgrin, L. Galárraga, and K. Hose, "Towards fully-fledged archiving for RDF datasets," *Semantic Web Journal*, vol. 12, no. 6, pp. 903–925, 2021.

[20] O. Pelgrin, R. Taelman, L. Galárraga, and K. Hose, "GLENDA: querying RDF archives with full SPARQL," in *ESWC (Satellite Events)*, ser. Lecture Notes in Computer Science, vol. 13998. Springer, 2023, pp. 75–80.

[21] ——, "The need for better rdf archiving benchmarks," in *MEPDaW@ISWC*, ser. CEUR Workshop Proceedings. CEUR-WS.org, 2023.

[22] ——, "Scaling large RDF archives to very long histories," in *ICSC*, 2023, pp. 41–48.

[23] T. Pellissier Tanon, C. Bourgaux, and F. Suchanek, "Learning How to Correct a Knowledge Base from the Edit History," in *WWW*, 2019, pp. 1465–1475.

[24] R. Perego, G. E. Pibiri, and R. Venturini, "Compressed indexes for fast search of semantic data," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 9, pp. 3187–3198, 2021.

[25] M. Perry, P. Jain, and A. P. Sheth, "SPARQL-ST: extending SPARQL to support spatiotemporal queries," in *Geospatial Semantics and the Semantic Web*, vol. 12, 2011, pp. 61–86.

[26] A. Polleres, R. Pernisch, A. Bonifati, D. Dell'Aglio, D. Dobriy, S. Dumbrava, L. Etcheverry, N. Ferranti, K. Hose, E. Jiménez-Ruiz, M. Lissandrini, A. Scherp, R. Tommasini, and J. Wachs, "How does knowledge evolve in open knowledge graphs?" *TGDK*, vol. 1, no. 1, pp. 11:1–11:59, 2023.

[27] Y. Raimond and G. Schreiber, "RDF 1.1 primer," W3C Recommendation, 2014, http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[28] Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris, and Y. Stavrakas, "A flexible framework for understanding the dynamics of evolving RDF datasets," in *ISWC*, vol. 9366, 2015, pp. 495–512.

[29] T. Sagi, M. Lissandrini, T. B. Pedersen, and K. Hose, "A design space for RDF data representations," *VLDB J.*, vol. 31, no. 2, pp. 347–373, 2022.

[30] A. Seaborne and S. Harris, "SPARQL 1.1 query language," W3C, W3C Recommendation, 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[31] R. Taelman, T. Mahieu, M. Vanbrabant, and R. Verborgh, "Optimizing storage of RDF archives using bidirectional delta chains," *J. Web Semant.*, vol. 13, pp. 705–734, 2022.

[32] R. Taelman, M. V. Sande, J. V. Herwegen, E. Mannens, and R. Verborgh, "Triple Storage for Random-access Versioned Querying of RDF Archives," *J. Web Semant.*, vol. 54, pp. 4–28, 2019.

[33] R. Taelman, M. V. Sande, and R. Verborgh, "Versioned querying with OSTRICH and comunica in MOCHA 2018," in *SemWebEval@ESWC*, vol. 927, 2018, pp. 17–23.

[34] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, "Comunica: a modular sparql query engine for the web," in *ISWC*, 2018.

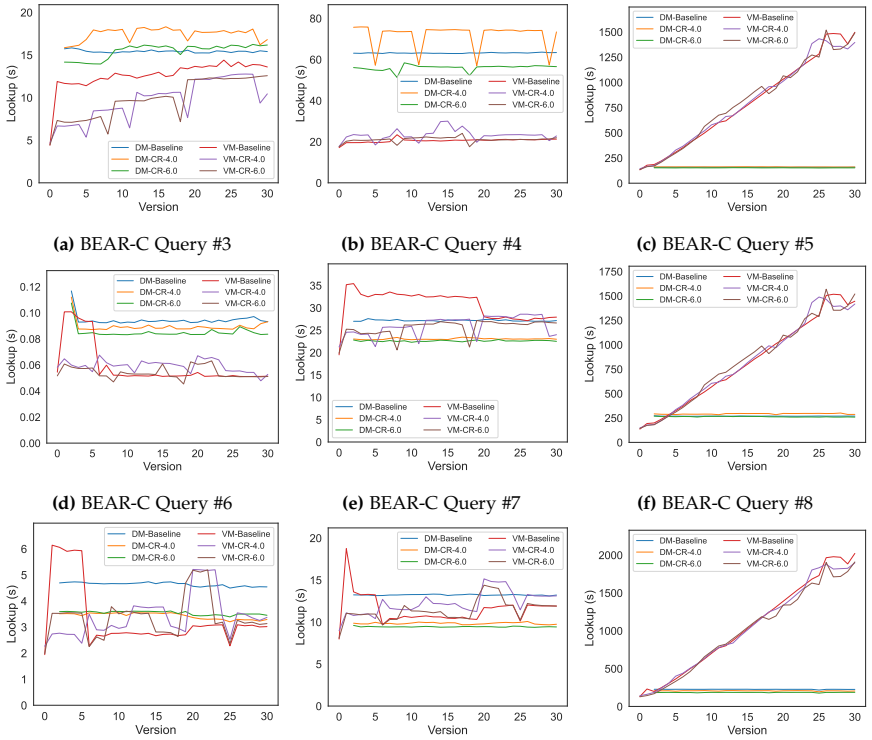
[35] T. P. Tanon and F. M. Suchanek, "Querying the edit history of wikidata," in *ESWC*, vol. 11762, 2019, pp. 161–166.

[36] M. Volkel, W. Winkler, Y. Sure, S. R. Kruk, and M. Synak, "SemVersion: A Versioning System for RDF and Ontologies," *ESWC*, 2005.

[37] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[38] A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia, "A general framework for representing, reasoning and querying with annotated semantic web data," *J. Web Semant.*, vol. 11, pp. 72–95, 2012.

10 Additional SPARQL Results



(a) BEAR-C Query #3 **(b)** BEAR-C Query #4 **(c)** BEAR-C Query #5

(d) BEAR-C Query #6 **(e)** BEAR-C Query #7 **(f)** BEAR-C Query #8

(g) BEAR-C Query #9 **(h)** BEAR-C Query #10 **(i)** BEAR-C Query #11

Fig. D.11: Individual runtime of DM and VM SPARQL queries for BEAR-C.

Paper E

GLENDA: Querying RDF Archives with full SPARQL

Olivier Pelgrin, Ruben Taelman, Luis Galárraga, Katja Hose

This paper has been published in the
20th Extended Semantic Web Conference (ESWC 2023), Vol. 13998, pp. 75–80, 2023.
DOI: 10.1007/978-3-031-43458-7_14

Abstract

The dynamicity of semantic data has propelled the research on RDF Archiving*, i.e., the task of storing and making the full history of large RDF datasets accessible. However, existing archiving techniques fail to scale when confronted with very large RDF datasets and support only simple SPARQL queries. In this demonstration, we therefore showcase GLENDA, a system that can run full SPARQL 1.1 compliant queries over large RDF archives. We achieve this through a multi-snapshot change-based storage architecture that we interface using the Comunica query engine. Thanks to this integration we demonstrate that fast SPARQL query processing over multiple versions of a knowledge graph is possible. Moreover, our demonstration provides different statistics about the history of RDF datasets that can be useful for tasks beyond querying and by providing insights about the evolution dynamics of the data.*

Pelgrin, O., Taelman, R., Galárraga, L., Hose, K. (2023). GLENDA: Querying RDF Archives with full SPARQL. In: 20th Extended Semantic Web Conference (ESWC 2023). Volume 13998, Pages 75–80, 2023.
https://doi.org/10.1007/978-3-031-43458-7_14
The layout has been revised.

1 Introduction

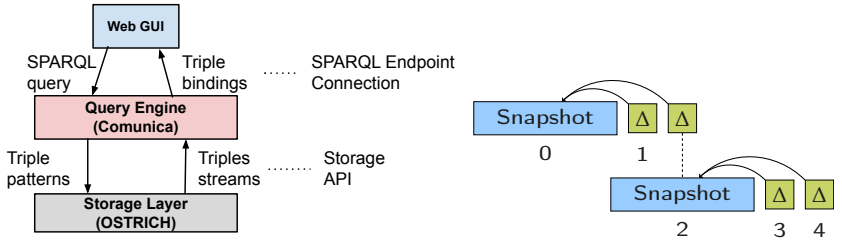
Despite most approaches assuming RDF datasets on the Web to be static and providing optimizations for this case, in reality most RDF datasets are consistently evolving [3, 5]. Although there has been some work on archiving, where the focus has been more on storing previous versions, research has not yet been paid much attention to efficiently querying past versions of a knowledge graph without depending on specific system setups [1].

A straightforward way to keep track of the history of RDF data is to store each revision of the dataset as an independent copy. Intuitively, this does not scale well and can become prohibitive for large RDF datasets with long histories. While few efficient solutions for RDF archiving have been proposed [6, 9], they support queries on single triple patterns only. This means that executing full SPARQL queries on RDF archives still requires additional post-processing.

In this demo paper, we therefore present GLENDA, a system for executing full SPARQL queries over RDF archives. GLENDA is built on top of a multi-snapshot change-based storage system for RDF archives [6] that has been integrated with the Comunica [11] SPARQL engine. In the remainder of this paper, we first detail the technical architecture of GLENDA in Section 2. Then, we describe and illustrate GLENDA's main functionalities in Section 3. Finally, we conclude and discuss future work in Section 4.

2 The GLENDA system

Overview. At its core, GLENDA is composed of three distinct and independents components, namely (i) a storage layer composed and an RDF archive store, (ii) a query engine that communicates with the storage layer via an API, and (iii) a user interface in the form of a web application. The query engine is accessible by the client through a SPARQL endpoint.



(a) GLENDA components. **(b)** Storage layer's architecture.

Fig. E.1: GLENDA architecture and components

Figure E.1a illustrates the high level architecture of GLENDA. The user interacts with a web-based GUI, where they can write SPARQL 1.1 [8] compliant queries. The query engine is exposed through a SPARQL endpoint with support for versioned queries. The query engine decomposes the full SPARQL query written by the user into versioned triple pattern queries that can be executed natively by the storage layer, which returns answers as triple streams.

Storage layer. We make use of an extension of the OSTRICH [6] system as storage layer. OSTRICH is a scalable engine for RDF archiving that stores the history of an RDF dataset in a single delta chain. A delta chain is comprised of an initial snapshot followed by a sequence of aggregated changesets (Figure E.1b). OSTRICH supports versioned queries on single triple patterns with optional offsets. It also provides efficient cardinality estimations for triple patterns. We resort to an extension of OSTRICH, presented in [6], that models revision histories using multiple delta chains. As shown in [6], this improves the ingestion time of new revisions drastically – in particular for very long histories.

Query engine. We chose the *Comunica* [11] query engine to build our SPARQL endpoint. Comunica is a modular, high-performance RDF query engine with full support for the SPARQL 1.1 standard. Building on top of the work from Taelman et al. [10], we opted for a minimal change to the SPARQL language, as a full extension is outside the scope of this demonstration. The semantic of the GRAPH keywork is changed so that it references versions instead of graphs. We implemented support for three standard types of versioned SPARQL queries [2] described in the following.

- **Version Materialization (VM).** These are queries over a specific version of the RDF Archive. These queries use the notation *GRAPH <version:k>* for $k \in \{0, 1, \dots\}$.
- **Delta Materialization (DM).** These are SPARQL queries over the changeset between two versions. This is achieved by using the notation for VM queries in combination with the *FILTER (NOT EXISTS)* construct.
- **Version Queries (VQ).** These are SPARQL queries that yield version-annotated query results. They resort to the notation *GRAPH ?version*.

User Interface. We build our GUI as a regular web-page using HTML, CSS, and Javascript. We resort to the Yasgui[1] library for the SPARQL query interface, and the Plotly[2] library for our graphics and visualizations. More details about the user interface and its functionalities can be found next in Section 3.

[1] https://triply.cc/docs/yasgui-api
[2] https://plotly.com/javascript/

3 Demonstration of GLENDA

We now demonstrate the capabilities of GLENDA on the BEAR-C dataset [2], which provides 32 snapshots from the Open Data Portal Watch project [4] together with ten full SPARQL queries . To the best of our knowledge, no publicly available system is currently capable of running the queries of this benchmark.

Figure E.2a depicts GLENDA's query interface, where the user can write and execute SPARQL 1.1 queries, optionally using our versioning constructs. The queries from the BEAR-C benchmark can be chosen from the dropdown menu on top. The query type can be chosen among VM, DM and VQ queries, and the provided sliders can help the user chose the versions to query.

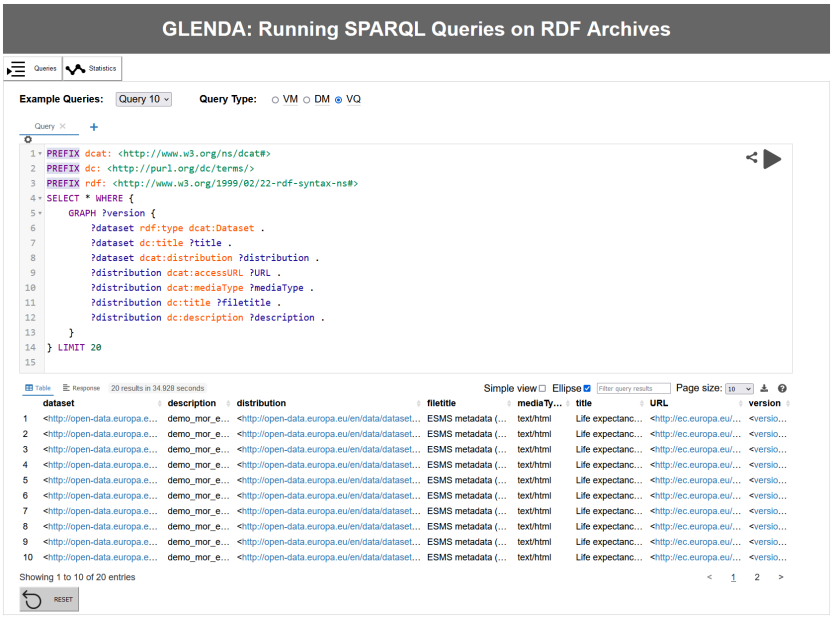
By selecting the tab "Statistics", the user can have access to various statistics about the underlying dataset (Figure E.2b). These are state-of-the-art metrics that describe the dynamics of an RDF archive [5]. Explanations for the metrics are available as tooltips triggered by hovering the mouse over the metric's name. A video showing all the capabilities of GLENDA can be found on YouTube[3]. The system is publicly available at `https://glenda.cs.aau.dk` and more information can be found on our project webpage[4].

4 Conclusion

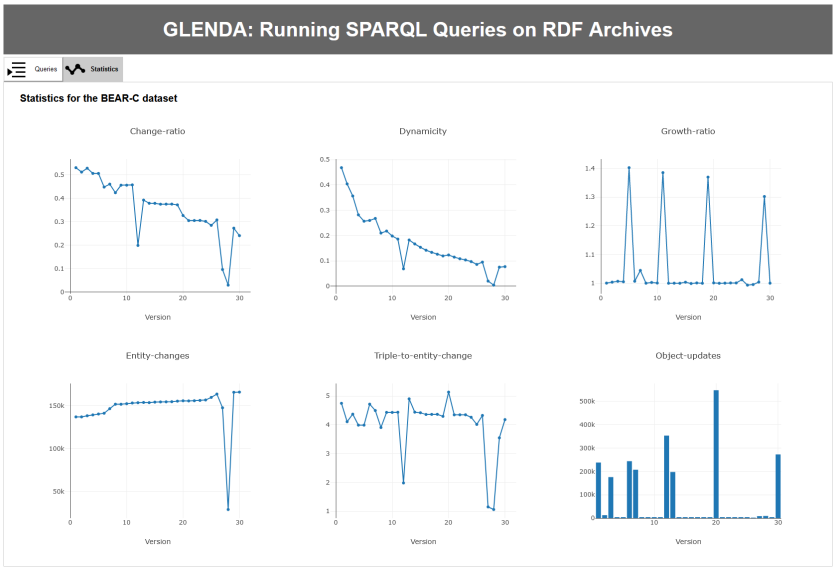
We have presented GLENDA, a system to execute full SPARQL queries on RDF archives. We detailed the technical makeup of the system and how its different components interact with each other. We explained how queries over archives can be executed with full SPARQL 1.1 via the use of special URIs for named graphs. GLENDA presents itself as a web interface to the user, with user-friendly tools to build and execute queries over RDF archives. We have demonstrated, GLENDA's capabilities on the BEAR-C dataset and queries, which no other system can currently fully support.

In our future work we have planned to consider the development and study of alternative snapshot strategies. Moreover, we envision to reduce the required storage space via more efficient serialization techniques for timestamped deltas. We also expect to improve query processing with advanced RDF representations and novel indexing approaches [7]. Similarly, we envision to study the use of dedicated extensions to the SPARQL language for versioned queries, which would allow for greater flexibility in the querying process, while enabling the simultaneous use of graphs and versions. Finally, we plan to improve the performance of the system further by implementing

[3] `https://youtu.be/DoNjw3V6oSo`
[4] `https://relweb.cs.aau.dk/glenda/`



(a) GLENDA main page and query interface.



(b) GLENDA statistics page.

Fig. E.2: GLENDA's user interface

a more efficient streaming of the results from the storage layer to the query engine.

References

[1] C. Aebeloe, G. Montoya, and K. Hose, "ColChain: Collaborative Linked Data Networks," in *WWW*. ACM / IW3C2, 2021, pp. 1385–1396.

[2] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for RDF archives," *JWS*, vol. 10, no. 2, pp. 247–291, 2019.

[3] K. Hose, "Knowledge Graph (R)Evolution and the Web of Data," in *MEPDaW@ISWC*, 2021, pp. 1–7.

[4] S. Neumaier, J. Umbrich, and A. Polleres, "Automated quality assessment of metadata across open data portals," *JDIQ*, vol. 8, no. 1, pp. 2:1–2:29, 2016.

[5] O. Pelgrin, L. Galárraga, and K. Hose, "Towards fully-fledged archiving for RDF datasets," *SWJ*, vol. 12, no. 6, pp. 903–925, 2021.

[6] O. Pelgrin, R. Taelman, L. Galárraga, and K. Hose, "Scaling Large RDF Archives To Very Long Histories," in *ICSC*, 2023.

[7] T. Sagi, M. Lissandrini, T. B. Pedersen, and K. Hose, "A design space for RDF data representations," *VLDB Journal*, vol. 31, no. 2, pp. 347–373, 2022.

[8] A. Seaborne and S. Harris, "SPARQL 1.1 query language," W3C, W3C Recommendation, 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[9] R. Taelman, M. V. Sande, J. V. Herwegen, E. Mannens, and R. Verborgh, "Triple Storage for Random-access Versioned Querying of RDF Archives," *JWS*, vol. 54, pp. 4–28, 2019.

[10] R. Taelman, M. V. Sande, and R. Verborgh, "Versioned querying with OSTRICH and comunica in MOCHA 2018," in *SemWebEval@ESWC*, vol. 927, 2018, pp. 17–23.

[11] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, "Comunica: a modular sparql query engine for the web," in *ISWC*, 2018.

Paper F

The Need for Better RDF Archiving Benchmarks

Olivier Pelgrin, Ruben Taelman, Luis Galárraga, Katja Hose

This paper has been published in
Managing the Evolution and Preservation of the Data Web (MEPDaW 2023), vol. 3565, pp. 50–54, 2023.

Abstract

Advancements and popularity of Semantic Web technologies in the last decades have led to an exponential adoption and availability of Web-accessible datasets. While most solutions consider such datasets to be static, they often evolve over time. Hence, efficient archiving solutions are needed to meet the users' and maintainers' needs. While some solutions to these challenges are being provided, standardized benchmarks are needed that systematically test the different capabilities of existing solutions and identify their limitations. Unfortunately, the development of new benchmarks has not kept pace with the evolution of RDF archiving systems. In this paper, we therefore identify the current state of the art in RDF archiving benchmarks and discuss to what degree such benchmarks reflect the current needs of real-world use cases and their requirements. Through this empirical assessment, we highlight the need for the development of more advanced and comprehensive benchmarks that align with the evolving landscape of RDF archiving.

Pelgrin, O., Taelman, R., Galárraga, L., Hose, K. (2023). The Need for Better RDF Archiving Benchmarks. In: Managing the Evolution and Preservation of the Data Web, MEPDaW 2023, Volume 3565, Pages 50–54, 2023.
The layout has been revised.

1 Introduction

The continuous advancement and widespread adoption of Semantic Web technologies have generated a growing demand for robust systems for managing knowledge graphs. This demand is particularly pronounced for RDF, the Semantic Web's most prevalent and accessible data model. Along with the rest of the Web, Semantic Web data is continuously evolving [5, 9, 13]. This has raised the need to keep track of the revision history of those datasets for the sake of multiple applications, such as version control or historical data analytics. This, in turn introduces new challenges for both data maintainers and users, sparking the development of dedicated techniques and systems for RDF archiving [13].

The availability of widely adopted benchmarks is of crucial importance for the development of RDF archiving systems. Standardized benchmarks enable the impartial evaluation of new indexing and storage techniques, as well as the performance of query engines. Although numerous benchmarks have been designed specifically for evaluating RDF stores [2, 4, 8], the number of benchmarking options for RDF archiving systems remains limited [12].

In this paper, we present an analysis of the current state of RDF archiving benchmarks, evaluating their strengths and limitations. We show that despite advancements in the field, current benchmarks do not sufficiently capture emerging challenges faced by archiving systems. We use this finding to derive a set of requirements, that we believe, are essential for benchmarks to advance research and development of RDF archives.

The remainder of this paper is organized as follows. First, we discuss the current state of RDF archiving research and relevant benchmarks in Section 2. Second, in Section 3, we discuss the shortcomings of current RDF archiving benchmarks and our recommendations and requirements for the future. Finally, Section 4 concludes the paper.

2 Related Work

In this section, we provide a brief survey of the available systems as well as existing languages and SPARQL extensions specifically designed for RDF archives. Furthermore, this section delves into existing benchmarks tailored for assessing the performance of RDF archive systems.

2.1 RDF Archiving

RDF archiving, at its core, consists of storing and querying the entire evolution history of an *RDF graph*. Efficient indexing and querying of RDF archives has proven to be a challenging task due to the additional temporal

dimension compared to traditional RDF stores. While the design of efficient indexing and querying systems for RDF archives is still an ongoing effort, multiple approaches have been proposed throughout the years [13]. Existing works can generally be categorized into three main paradigms [5], Independent Copies (IC), Change-based (CB), and Timestamp-based (TB), with some modern approaches proposing the use of a combination of those [1, 14–16]. Some approaches are now able to scale to much larger RDF archives compared to early proposals [15], however querying capabilities still remain very limited. Efficient processing of complex archive queries remains one of the key areas of development for the future of RDF archiving systems.

Relative to conventional RDF, the existence of multiple versions within an RDF archive introduces the need for novel query types that can be hardly expressed in the standard SPARQL language. Some approaches propose the extension of SPARQL to support temporal queries, i.e. specifying a timestamp or interval in which the query results should hold [6, 7]. Other works attempt to formally categorize the different types of queries possible on RDF archives [5, 12], but do not address the implementation of these categorizations via formal SPARQL extensions.

2.2 Benchmarks for RDF Archives

Benchmarks play a crucial role in guiding the development of systems by facilitating their evaluation and enabling comparisons with existing systems in terms of implementation and design. Due to being a relatively new area in RDF data management, we only account for three benchmarks tailored for RDF archiving in the literature: EvoGen [11], BEAR [5], and SPBv [12].

EvoGen [11] is a benchmark based on the LUBM [8] data generator extended to support evolving RDF scenarios. The benchmark data can be configured on the desired number of versions and the magnitude of changes. The querying workload is derived from the 14 LUBM queries and includes variations of materialization, delta, and mixed queries. Due to the nature of the LUBM queries, support for RDFS reasoning is needed to resolve the complete result sets.

BEAR [5] is a benchmark for RDF archives consisting of three different RDF archives. Those different flavours, namely *BEAR-A*, *BEAR-B* and *BEAR-C*, are extracted from real-world datasets, and are characterised by their various sizes and change behaviour. BEAR comes with predefined query workloads, based on single triple-patterns queries for both BEAR-A and BEAR-B, while for BEAR-C, a set of 10 different full SPARQL queries are proposed.

SPBv [12] is a benchmark for RDF archives that consists of a data generator based on the Semantic Publishing Benchmark (SPB) [10] from the Linked Data Benchmark Council (LDBC) [2]. The number of versions and the size of the data can be configured, as well as the number of generated queries. The

generated data comes as full versions, changesets, or both. The query workload consists of SPARQL queries where versions are represented as named graphs.

3 Benchmarking RDF Archives

In this section, we examine the qualities and features that a benchmark for RDF Archiving should strive to possess. We propose that benchmarks for RDF Archives should strive for three main overarching qualities, namely *reproducibility*, *realism*, and *configurability*. *Reproducibility* represents the ease at which the benchmark results can be shared and reproduced by others. *Realism* is about how the benchmark setting, both in the choice of dataset and query loads, models or emulates the real world. *Configurability* represents the ability of the benchmark to propose workloads of various sizes, relevant for a wide range of system configurations and use cases. We further detail our recommendations of a concrete implementation of the aforementioned qualities by first detailing the choice of data. We then will discuss the design of query workloads, and finally, we discuss whether existing benchmarks fulfil those requirements.

3.1 Dataset

The choice of data is an important aspect when designing a benchmark. Current benchmarks use either configurable generator for synthetic data [11, 12], or directly provide data based on existing real world datasets [5]. As discussed by Duan et al. [4], many data generators produce data that is not necessarily representative of real-world RDF datasets. However, they also demonstrate the possibility to make generators truer to the real world by taking into account their proposed *coherence* metric in the generation process. We are although not aware of any other generator-based benchmark for RDF archiving making use of this metric, which would improve the *realism* of the generated data.

Most importantly, a benchmark should cover different, realistic, scaling options. In the RDF archiving world, the scaling options do not only cover different data sizes, but also the temporal size, i.e. the number of versions and the magnitude of changes within each version. Generator-based benchmarks should provide users with all the necessary scaling parameters, while real-world-based benchmarks should offer different datasets scaling along those axes.

3.2 Query Workload

Early RDF archiving systems could be adequately tested with single triple pattern queries, but contemporary archiving benchmarks should prioritize comprehensive SPARQL query workloads. We believe that efficient support for full SPARQL in RDF Archiving represents a major challenge that RDF archiving systems currently need to solve. Consequently, in order to fullfil our *realism* requirement, benchmarks should provide comprehensive assessment of those capabilities, guiding the development of existing and new systems. Benchmark query workloads should be meticulously designed to align with real-world use cases. Following recommendations from the LDBC [2], a "choke-point" approach to the design of the benchmark should be considered through a comprehensive evaluation of real-world RDF archives usages.

Finally, the lack of a widely accepted standard to formulate archiving queries into SPARQL is a major brake for the design of benchmark queries. Addressing this issue necessitates a dedicated standardization effort, potentially drawing inspiration from the RDF stream community, and the RSP-QL standardization effort [3]. Such a solution would however require a more extensive study of the overlap between RDF streams processing and RDF archiving, notably on the relation between streams' temporal graphs and archives.

3.3 Comparison of Existing RDF Archiving Benchmarks

	Dataset	Reproducibility	Realism (data)	Realism (queries)	Configurability
EvoGen [11]	Synthetic	-/+	-	+	+
BEAR [5]	Real-world	+	+	-	-
SPBv [12]	Synthetic	-/+	-	+	+

Table F.1: Comparison table of existing RDF Archiving benchmarks.

Table F.1 summarizes the characteristics of the existing RDF archiving benchmarks. Among the available benchmarks, two of them rely on synthetic data generated through a data generator. Generator-based systems fulfil the *configurability* criteria easily due to their nature, but may fall short of also proving their *realism*, while their *reproducibility* is dependent on the sharing of the exact parameters and random seed. Both EvoGen [11] and SPBv [12] provide SPARQL queries of varied nature, but only focus on the generation of one restrictive type of datasets, which have not been evaluated for their realism, for example via the coherence metric. BEAR [5] on the other hand provides datasets of various size and based on real-world data. This increase the *reproducibility* and *relevance* of the benchmark compared to generator-based ones. The number of scalability options is however necessarily more lim-

ited, but BEAR still offers five different alternative datasets. However, full SPARQL queries are only provided for one of the options, the others being limited to single triple-patterns queries. As discussed in Section 3.2, it limits BEAR's *realism*, and makes the evaluation of SPARQL-capable archiving systems greatly limited.

4 Conclusion

In this paper, we presented the current state in RDF archives systems and benchmarks. We have proposed a set of requirements that benchmarks should have in order to contribute to the advancement of the field. We showed that among the only three available benchmarks for RDF archiving systems, none of them proposes a satisfactory set of features. This ranges from a general lack of realism w.r.t. the real world, lack of SPARQL support, or concerns with reproducibility. We see several areas open for future work. First, precisely defining the semantics and syntax of SPARQL archive queries would benefit greatly to the wider RDF community. This would open the door for standardized support across various RDF stores and research systems. Secondly, benchmarks relevant to the modern challenges faced by RDF archiving applications and systems are needed to guide and evaluate efforts in that area. We believe that this is paramount to current development efforts of fully-fledged RDF archiving systems.

References

[1] N. Arndt, P. Naumann, N. Radtke, M. Martin, and E. Marx, "Decentralized collaborative knowledge management using git," *J. Web Semant.*, vol. 54, pp. 29–47, 2019.

[2] P. A. Boncz, I. Fundulaki, A. Gubichev, J. L. Larriba-Pey, and T. Neumann, "The linked data benchmark council project," *Datenbank-Spektrum*, vol. 13, no. 2, pp. 121–129, 2013.

[3] D. Dell'Aglio, E. D. Valle, J. Calbimonte, and Ó. Corcho, "RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems," *Int. J. Semantic Web Inf. Syst.*, vol. 10, no. 4, pp. 17–44, 2014.

[4] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea, "Apples and oranges: a comparison of RDF benchmarks and real RDF datasets," in *SIGMOD*, 2011, pp. 145–156.

[5] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "Evaluating query and storage strategies for RDF archives," *Semantic Web Journal*, vol. 10, no. 2, pp. 247–291, 2019.

[6] S. Gao, J. Gu, and C. Zaniolo, "RDF-TX: A fast, user-friendly system for querying the history of RDF knowledge bases," in *EDBT*, 2016, pp. 269–280.

[7] F. Grandi, "T-SPARQL: A tsql2-like temporal query language for RDF," in *ADBIS (Local Proceedings)*, 2010, pp. 21–30.

[8] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *J. Web Semant.*, vol. 3, no. 2-3, pp. 158–182, 2005.

[9] K. Hose, "Knowledge graph (r)evolution and the web of data," in *MEPDaW@ISWC*, 2021, pp. 1–7.

[10] V. Kotsev, N. Minadakis, V. Papakonstantinou, O. Erling, I. Fundulaki, and A. Kiryakov, "Benchmarking RDF query engines: The LDBC semantic publishing benchmark," in *BLINK@ISWC*, 2016.

[11] M. Meimaris and G. Papastefanatos, "The EvoGen Benchmark Suite for Evolving RDF Data," in *MEPDaW/LDQ@ESWC*, 2016, pp. 20–35.

[12] V. Papakonstantinou, G. Flouris, I. Fundulaki, K. Stefanidis, and Y. Roussakis, "Spbv: Benchmarking linked data archiving systems," in *BLINK/NLIWoD3@ISWC*, vol. 1932, 2017.

[13] O. Pelgrin, L. Galárraga, and K. Hose, "Towards fully-fledged archiving for RDF datasets," *Semantic Web Journal*, vol. 12, no. 6, pp. 903–925, 2021.

[14] O. Pelgrin, R. Taelman, L. Galárraga, and K. Hose, "GLENDA: Querying RDF Archives with full SPARQL," in *ESWC*, 2023.

[15] O. Pelgrin, R. Taelman, L. Galárraga, and K. Hose, "Scaling large RDF archives to very long histories," in *ICSC*, 2023, pp. 41–48.

[16] R. Taelman, M. Vander Sande, J. Van Herwegen, E. Mannens, and R. Verborgh, "Triple storage for random-access versioned querying of rdf archives," *J. Web Semant.*, 2018.