Towards Fully-fledged Archiving for RDF Datasets

Olivier Pelgrin a,*, Luis Galárraga b and Katja Hose a

a Department of Computer Science, Aalborg University, Denmark
E-mails: olivier@cs.aau.dk, khose@cs.aau.dk
b Inria, France
E-mail: luis.galarraga@inria.fr

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.

Keywords: RDF Archiving, Knowledge Bases

1. Introduction

The amount of RDF data has steadily grown since the conception of the Semantic Web in 2001 [7], as more and more organizations opt for RDF [44] as the format to publish semantic data. For example, by July 2009 the Linked Open Data (LOD) cloud counted on a few more than 90 RDF datasets adding up to almost 6.7B triples [8]. By 2020, these numbers have catapulted to 1200+ datasets1 and at least 28B triples2, although estimates based on LODStats [15] suggest more than 10K datasets and 150B+ triples if we consider the datasets with errors omitted by the LOD Cloud [38]. This boom does not only owe credit to the increasing number of data providers, 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 [5], YAGO [51], or Wikidata [16], and also applies to automatically ever-growing projects such as NELL [11].

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 [4, 22, 26, 47]. They also allow data providers to study the evolution of the data [19] 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 [10, 29]. But archives are also of great value for consumer applications such as data analytics, e.g., mining correction patterns [40, 41] or historical trend analysis [30].

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 [2–4, 13, 24, 26, 35, 43, 47, 52, 54], to benchmarks to evaluate such engines [19, 32], as well as temporal extensions for SPARQL [6, 20, 25, 42]. Diverse in architecture and aim, all these works respond to

---

1Corresponding author. E-mail: olivier@cs.aau.dk.

1https://lod-cloud.net/

2http://lod-a-lot.lod.labs.vu.nl/
particular use cases. Examples are solutions such as R&WBases [47], R43ples [26], and Quit Store [4] that provide data maintainers with distributed version control management in the spirit of Git. Conversely, other works [24, 42] 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 functionality 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 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 [52] 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 \( t = (s, p, o) \), where \( s \in \mathcal{I} \cup \mathcal{B}, p \in \mathcal{L}, \) 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., strings, integers, dates), and blank nodes (anonymous entities) [44]. 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 [44] 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 1 provides the relevant notation related to RDF graphs.

<table>
<thead>
<tr>
<th>( {s, p, o} ), ( {s, p, o, \rho} )</th>
<th>triple and 4-tuple: subject, predicate, object, graph revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G )</td>
<td>an RDF graph</td>
</tr>
<tr>
<td>( g )</td>
<td>a graph label</td>
</tr>
<tr>
<td>( G_i )</td>
<td>the ( i )-th version or revision of graph ( G )</td>
</tr>
<tr>
<td>( {G_i, G_{i+1}, \ldots } )</td>
<td>an RDF graph archive</td>
</tr>
<tr>
<td>( u = {u^+, u^-} )</td>
<td>an update or changeset with sets of added and deleted triples.</td>
</tr>
<tr>
<td>( u_{ij} = {u^+<em>{ij}, u^-</em>{ij}} )</td>
<td>the changeset between graph revisions ( i ) and ( j (j &gt; i) )</td>
</tr>
<tr>
<td>( rv(\rho) )</td>
<td>revision number of graph revision ( \rho )</td>
</tr>
<tr>
<td>( ts(\rho) )</td>
<td>commit times of graph revision ( \rho )</td>
</tr>
<tr>
<td>( l(\rho), l(G) )</td>
<td>graph labels of graph revision ( \rho ) and graph ( G )</td>
</tr>
</tbody>
</table>

Table 1: Notation related to RDF Graphs.
of RDF graphs, where each \( G_i \) is a revision or version with revision number \( i \in \mathbb{N} \), 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 \( u_i = (u^+, u^-) \) 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 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 \( u_{i,j} = (u^+_{i,j}, u^-_{i,j}) \) 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 \( \rho \) to its commit time, i.e., the timestamp of application of changeset \( u \). Some solutions for RDF archiving [4, 24, 26, 35, 47, 52] implement this logical model in different ways and to different extents. For example, R43ples [26], R&WBase [47] and Quit Store [4] store changesets and/or their associated metadata in additional named graphs using PROV-O [14]. In contrast, x-RDF-3X [35] stores the temporal metadata in special indexes that optimize for concurrent updates at the expense of temporality 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 archive, we use the notation \( G^k \) for the \( k \)-th graph in a dataset, whereas \( G^k \) denotes the i-th revision of \( G^k \). The notation related to RDF datasets is detailed in Table 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 [44], 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 [23, 33] 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 \( A = \{D_0, D_1, \ldots, D_{i-1}\} \) as a temporally ordered collection of RDF datasets. The \( j \)-th revision of \( A (j > 1) \) can be obtained by applying a dataset update \( U_j = \{u^+_j, u^-_j\} \) to \( D_{j-1} = \{G^0_{j-1}, G^1_{j-1}, \ldots, G^m_{j-1}\} \). \( U_j \) consists of an update per graph plus a special changeset \( u_j = (u^+_j, u^-_j) \) that we call the graph changeset \( u^+_j \cap u^-_j = \emptyset \). The sets \( u^+_j, u^-_j \) store the labels of the graphs that should be added and deleted in revision \( j \) respectively. If a graph \( G^k \) is in \( u^-_j \) (i.e., \( k \) is scheduled for removal), then \( G^k \) as well as its corresponding changeset \( u^+_j \) of \( G^k \) must be empty. It follows that we can obtain revision \( D_j \) by (i) applying the individual changesets \( u^+_j(G^k_{j-1}) \) for each \( 0 \leq k \leq m \), (ii) removing the graphs in \( u^-_j \), and (iii) adding the graphs in \( u^+_j \).

Figure 2 illustrates an example of a dataset archive with two revisions \( D_0 \) and \( D_1 \), \( D_0 \) is a dataset with graphs \( \{G^0_0, G^1_0\} \) both at local revision 0. The dataset update \( U_1 \) generates a new global dataset revision \( D_1 \). \( U_1 \) consists of three changesets: \( u^0_1 \) that modifies the default graph \( G^0_0, u^1_1 \) that leaves \( G^1_0 \) untouched, and the
graph update \(u_2\) that adds graph \(G^2\) to the dataset and initializes it at revision \(s = 1\) (denoted by \(G^2_1\)).

As proposed by some RDF engines [21, 50], 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 \(\langle g^k, rdf:type, \eta; Graph \rangle\) in \(G^M\) for some namespace \(\eta\), then we can model a dataset archive as a set of 5-tuples \(\langle s, p, o, p, \zeta \rangle\). Here, \(p \in I\) is the RDF identifier of the local revision of the triple in an RDF graph with label \(g = l(p)\) (Table 2). Conversely, \(\zeta \in I\) identifies a (global) dataset revision \(j = rv(\zeta)\). Likewise, we overload the function \(ts(\zeta)\) (defined originally in Table 1) so that it returns the timestamp associated to the dataset revision identifier \(\zeta\). 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 [48]. 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 [48]. SPARQL is a graph-based language whose building blocks are triple patterns. A triple pattern \(i\) is a triple \((s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)\), where \(V\) is a set of variables such that \((I \cup B \cup L) \cap V = \emptyset\) (variables are always prefixed with \(?\) or \$). A basic graph pattern (BGP) \(\tilde{G}\) is the conjunction of a set of triple patterns \(\{ i_1, i_2, \ldots, i_m \}\), e.g.,

\[
\{ \text{?s a :Person. ?s :nationality :France} \}
\]

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 \(GRAPH \ \tilde{g}\ \{ \tilde{G} \}\), where \(\tilde{g} \in I \cup B \cup V\). In this paper we call this, a named BGP denoted by \(\tilde{G}_k\). A SPARQL select query \(Q\) on an RDF dataset has the basic form "SELECT \(V\) (FROM NAMED \(\tilde{g}_1\) FROM NAMED \(\tilde{g}_2\) \ldots) WHERE \(\tilde{G}^0 \ldots \tilde{G}_2 \ldots\) with projection variables \(V \subset V\). SPARQL supports named BGP with \(\tilde{g}\) variables \(g \in V\). In some implementations [21, 50] the bindings for those variables originate from the master graph \(G^M\). The BGP 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 [19, 52]. We illustrate them by means of our example graph archive from Figure 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?
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 quantify 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 [19], 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 RocksDB\(^3\) key-value store as storage and indexing backend. All metrics are originally defined for RDF graphs in the state of the art [19], and have been ported to RDF datasets in this paper. RDFev’s source code is available at https://relweb.cs.aau.dk/rdfev.

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 \(\Upsilon(D) \subseteq I \cup L \cup B\) 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 chosen 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. [19] for pairs of revisions \(i, j\) with \(j > i\).

Change ratio. The authors of BEAR [19] define the change ratio between two revisions \(i\) and \(j\) of an RDF graph \(G\) as

\[
\delta_{i,j}(G) = \frac{|u^+_i| + |u^-_i|}{|G_i \cup G_j|}, \tag{1}
\]

\(\delta_{i,j}\) 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 the revisions. For a more fine-grained analysis, Fernández et al. [19] also proposes the insertion and deletion ratios:

\(^3\)http://rocksdb.org
\[ \delta^+_i = \frac{|u^+|}{|G_i|} \]  \hspace{1cm} (2) \hspace{1cm} \delta^-_i = \frac{|u^-|}{|G_i|}. \hspace{1cm} (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 change-set \( U \) 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:

\[ \delta^+_i(D) = \frac{sz(U^+)}{\sum_{G \in D_i} |G\cup G_i| + \sum_{G \in D_i \triangle D_j} |G|} \] \hspace{1cm} (4) \]

\[ \delta^-_i(D) = \frac{sz(U^-)}{sz(D_i)} \] \hspace{1cm} (5) \hspace{1cm} \delta_{ij}(D) = \frac{sz(U^-)}{sz(D_i)} \] \hspace{1cm} (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 [19]:

\[ \text{vdyn}_{i,j} (G) = \frac{|\Upsilon(u_{i,j})|}{|\Upsilon(G_i) \cup \Upsilon(G_j)|} \] \hspace{1cm} (7)

\( \Upsilon(u_{i,j}) \) is the set of vocabulary terms – IRIs, literals, or blank nodes – in the changeset \( u_{i,j} \) (Table 2). The literature also defines the vocabulary dynamicity for insertions (\( \text{vdyn}^+_{i,j} \)) and deletions (\( \text{vdyn}^-_{i,j} \)):

\[ \text{vdyn}^+_{i,j} (G) = \frac{|\Upsilon(u_{i,j}^+)|}{|\Upsilon(G_i) \cup \Upsilon(G_j)|} \] \hspace{1cm} (8)

\[ \text{vdyn}^-_{i,j} (G) = \frac{|\Upsilon(u_{i,j}^-)|}{|\Upsilon(G_i) \cup \Upsilon(G_j)|}. \] \hspace{1cm} (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:

\[ \Gamma_{i,j}(G) = \frac{|G_j|}{|G_i|} \] \hspace{1cm} (10) \hspace{1cm} \Gamma_{i,j}(D) = \frac{sz(D_j)}{sz(D_i)}. \hspace{1cm} (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 [39, 46] 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 \( s \) is a subject for the sake of this analysis. We define the metric \( \text{entity changes} \) between revisions \( i, j \) in an RDF graph as:

\[ \text{ec}_{i,j} (G) = |\sigma_{i,j}(G)| = |\sigma_{i,j}^+(G) \cup \sigma_{i,j}^-(G)| \] \hspace{1cm} (12)

In the formula, \( \sigma^+_{i,j} \) is the set of added entities, i.e., the subjects present in \( \Upsilon(G_j) \) but not in \( \Upsilon(G_i) \) (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 \( \text{ec}(G) \) (with no subscripts) as the number of different subjects in a graph \( G \). It follows that,

\[ \text{ec}_{i,j} (D) = \sum_{G \in D_i \triangle D_j} \text{ec}_{i,j} (G) + \sum_{G \in D_i \cap D_j} \text{ec}(G). \] \hspace{1cm} (13)

We also propose the \textit{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:

\[ \text{ect}_{i,j} (G) = \frac{|\langle s, p, o \rangle \in u_{i,j}^+ \cup u_{i,j}^- : s \in \sigma_{i,j}(G)|}{\text{ec}_{i,j} (G)} \] \hspace{1cm} (14)

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

\[ \text{ect}_{i,j} (D) = \frac{|\langle s, p, o \rangle \in U_{i,j}^+ \cup U_{i,j}^- : s \in \sigma_{i,j}(D)|}{\text{ec}_{i,j} (D)}. \] \hspace{1cm} (15)
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 [51], DBpedia [5], and Wikidata [16] 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 [1] in the period from 2014-05 to 2016-08. These dumps provide a clean subset 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 4 maps revision numbers to releases for the sake of conciseness in the evolution analysis.

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

Table 4
Datasets revision mapping

4.2. Low-level Evolution Analysis

Change ratio. Figures 3a, 3d and 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 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 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 magni-
tude 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 3b, 3e, and 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 means that DBpedia 2015-14 contained more entities, but fewer – potentially 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 3c, 3f, and 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 DBpe-
dia 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

4.3.1. Entity changes

Figures 4a, 4d, and 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 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 4e and 4h), where values for this metric are significantly lower than for DBpedia, and remain almost constant. This shows that minor releases in YAGO improved the strategy to extract entities, but did not change much the amount of extracted triples per entity.

4.3.2. Object Updates and Orphan Object Additions/Deletions

We present the evolution of the number of object updates for our experimental datasets in Figures 4c, 4f, and 4i. For DBpedia, the curve is consistent with the change ratio (Figure 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 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 5a and 5b. Compared to the other two datasets, Wikidata’s number of object updates, shown in Figure 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 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
Fig. 4. Entity changes and object updates

(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. 5. Orphan object additions and deletions

(a) DBpedia
(b) YAGO
(c) Wikidata
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 [3, 4, 26, 54], most available systems support archiving of a single RDF graph. Ostrich [52], for instance, manages quads of the form \( \langle s, p, o, \text{rv}(\rho) \rangle \). Other solutions do not support revision numbers and use the \( \rho \)-component \( \rho \in \mathcal{I} \) to model temporal metadata such as insertion, deletion, and validity timestamps for triples [24]. 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 \( \rho \), 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 5 where we characterize the state-of-the-art approaches according to the following criteria:

- **Storage paradigm.** The storage architecture is probably the most important feature as it shapes the system’s architecture. We identify three main storage paradigms in the literature [19], namely independent copies (IC), change-based (CB), and timestamp-based (TB). Some systems [52] may fall within multiple categories, whereas Quit Store [4] 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.

5.1.1. Independent Copies Systems

In an IC-like approach, each revision \( D_j \) 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 incur 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 [37, 54]. SemVersion [54], 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 \( \langle s, p, o, l(\rho), \text{rv}(\rho) \rangle \), 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.

5.1.2. Change-based Systems

Solutions based on the CB paradigm store a subset \( \hat{A} \subset \mathcal{A} \) 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 [47] 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) [14] to model the metadata (e.g., timestamps, parent branches) about the updates of a single RDF graph. An update $u$ is a 5-tuple of the form

\[
\langle j \in N, o, p, l(p), \xi \rangle
\]

containing the added and deleted triples at revision $j$. 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 [26]. 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 \textit{REVISION $j$ ($j \in N$)} used in conjunction with the \textit{GRAPH} clause to match a BGP against a specific revision of a graph. Last, the approach presented by Dong-hyuk et al. [31] 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 [2] 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 $\langle s, p, o, l(p), \xi \rangle$ (i.e., metadata is stored at the dataset level). While the details of Stardog’s internal architecture are not public, the documentation suggests a CB paradigm with a relational database backend.

### 5.1.3. 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 [35] is a system based on the RDF-3X [34] engine. Logically x-RDF-3X supports quads of the form $\langle s, p, o, \rho \rangle$ where $\rho \in I$ 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 [3] is a TB archiving system that supports archiving of multi-graph datasets. Logically, Dydra stores 5-tuples of the form $\langle s, p, o, l(p), \xi \rangle$, that is, revised.
vision metadata lies at the dataset level. In its physical design, Dydra indexes quads \( \langle s, p, o, l(p) \rangle \) 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 – \( \text{gspo}, \text{gpos}, \text{gsp}, \text{spos}, \text{pgsp}, \text{pgos} \), and \( \text{osp} \) – to support arbitrary SPARQL queries \( (g = l(p)) \) is the graph label). Moreover, Dydra extends the query language with the clause \( \text{REVISION} \ x \), 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 \( \zeta \), or a dataset changeset \( U_{jk} \). A revision can be identified by its IRI \( \xi \), its revision number \( rv(\zeta) \) or by a timestamp \( \tau' \). The latter case matches the revision \( \zeta \) 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 [24] 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 stores the triples 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 \( g \) to BGs. This component can match only time objects \( \tau \) of type timestamp or time interval. The attributes of such objects can be queried via built-in functions, e.g., \( \text{year}(\tau) \). While RDF-TX offers interval semantics at the query level, it stores only timestamps.

v-RDFCSA [13] is a lightweight and storage-efficient TB approach that relies on suffix-array encoding [9] for efficient storage with basic retrieval capabilities (much in the spirit of HDT [17]). Each triple is associated to a bisquence of length equals the number of revisions in the archive. That is, v-RDFCSA logically stores quads of the form \( \langle s, p, o, rv(p) \rangle \). Its query functionalities are limited since it supports only VM, DM, and V queries on single triple patterns.

5.1.4. Hybrid and Fragment-based Systems

Some approaches can combine the strengths of the different storage paradigms. One example is Ostrich [52], which borrows inspirations from IC, CB, and TB systems. Logically, Ostrich supports quads of the form \( \langle s, p, o, rv(p) \rangle \). Physically, it stores snapshots of an RDF graph using HDT [17] as serialization format. Delta chains are stored as B+ trees time-stamped 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. [31]. 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 \( u_0 \).

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 BGs is possible via integration with the Comunica query engine.3

Like R43ples [26], Quit Store [4] 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 (e.g. N-quads files) and is accessible via a SPARQL endpoint on a set of virtual 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 [19] uses AnQL to express the query types described in Section 2.5. AnQL [55] is a SPARQL extension based on quad patterns \( (\hat{s}, \hat{p}, \hat{o}, \hat{g}) \). AnQL is more general than SPARQL-T (proposed by RDF-TX [24]) because the \( g \) 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 1 could be written as follows:

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

3https://github.com/rdfostrich/comunica-actor-init-sparql-ostrich
A recent approach [53] 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 Related Work

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 4. This evaluation is complementary to the Ostrich’s evaluation on BEAR (available at [52]), as it shows the performance of the system in three real-world large RDF datasets.

6.1. Functionality Analysis

As depicted in Table 5, existing RDF archiving solutions differ greatly in design and functionality. The first works [12, 35, 54] 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 [4, 26, 31, 47]. Such solutions are attractive for data maintainers in collaborative projects, however they still lack scalability, e.g., they cannot handle large datasets and change-sets, besides conflict management is still delegated to users. More recent works [13, 52] have therefore focused on improving storage and querying performance, alas, at the expense of features. For example, Ostrich [52] is limited to a single snapshot and delta chain. In addition to the limitations in functionality, Table 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 [52], Quit Store [4], R&WBase [47], R43ples [26] and x-RDF-3X as testable solutions, only [52] 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\(^6\). 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 R43plies, we had to modify its source code to handle large files\(^7\). 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 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.

**Storage space.** Figure 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 \cite{sohler2016} for snapshot storage, as well as compression for the delta chains.

**Ingestion time.** Figure 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 \(\sim 22\) hours to ingest revision 9 of DBpedia (2.43M added and 2.46M deleted triples) while it takes only \(\sim 14\) hours to ingest revision 5 (12.85M added and 5.95M deleted triples). This confirms the trends observed in \cite{sohler2016} 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 changest 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 \(\langle ?, \ p, \ ? \rangle\), \(\langle s, p, \ ? \rangle\), and \(\langle ?, \ p, \ o \rangle\) in equal numbers. We also considered queries of the form \(\langle ?, \ <\text{top}\ p>\ , \ o \rangle\), where \(<\text{top}\ p>\) corresponds to the top 5 most common predicates in the dataset. Revision numbers for all queries were also randomly generated. Table 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 mag-
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 [26] and Quit Store [4] 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 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 [24, 25], 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 $\rho$ and $\zeta$, 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 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 [14] 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 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 provenance history to a graph and use the revision identifiers to override or extend that default history for new triples. Moreover, the global revision identifier \( \zeta \) 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 [3, 13, 24, 43, 52], (ii) solutions inspired by version control systems such as Git [4, 26, 31, 47, 54], and (iii) approaches with full support for highly concurrent updates [35]. 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 [11] or data-intensive (e.g., streaming) applications may need the features of solutions such as x-RDF-3X [35]. 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 [4, 26, 47] 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*. Among those, RDF* [28] is the only one that natively support multiple levels of metadata (still in a very verbose fashion). For example RDF* could serialize the tuple \( \langle :\text{USA}, :dr, :\text{Cuba}, \rho, \zeta \rangle \) with graph label \( (;gl) l(\rho) = :\text{UN} \) and global timestamp \( (;ts) ts(\zeta) = 2020-07-09 \) as follows:

\[
\text{<<< :USA :dr :Cuba > :gl :UN > :ts "2020-07-09" xsd:date >}
\]

The authors of [28] propose this serialization as part of the Turtle* format. Moreover, they propose SPARQL* that allows for nested triple patterns. While SPARQL* enables the definition of metadata constraints at different levels, a fully archive-compliant language could offer further syntactic sugar such as the clauses \text{REVISION} [3, 26] or \text{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*.

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 [19, 52]. 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 [4, 52]. We endorse this design philosophy because it eases modularity. Existing applications in mining archives [30, 40, 41] 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. [52], 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 [52], for instance, chose to benefit faster version materialization via redundant deltas at the expense of larger ingestion times. Users requiring shorter ingestion times, 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 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 disk-based stores for faster query response time (as done by QuitStore [4]). 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 [45], singleton properties [36], and named graphs [44]. 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 $\rho$ and $\zeta$ 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. [36] 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 $\rho$ 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 $\zeta$. The same is true for a solution based on named graphs. A more recent solution is HDTQ [18], which extends HDT with support for quads. An additional extension could account for a fifth component. Systems such as Dydra [3] or v-RDFCSA [13] 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 consumption. Parameter tuning could rely on the metrics proposed in Section 3. Nonetheless, these desiderata translate into some de-

---

8 For single triple patterns on VM queries
sign 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 [35].

8. Conclusions

In this paper we have argued the importance of RDF archiving for both maintainers and consumers of RDF data. Besides, we have argued the importance of evolution patterns in the design of a fully-fledged RDF archiving solution. On those grounds, we have posessed 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 those 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 [26], R&WBBase [47], Quit Store [4], and x-RDF-3X [35] 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* combined with additional syntactic sugar as proposed by some approaches [3, 20, 26].

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, (ii) 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 a ultimate solution for this problem.
Acknowledgements

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.

References


[29] Ian Horrocks, Thomas Hubauer, Ernesto Jiménez-Ruiz, Evgeny Kharchaviev, Manolis Koubarakis, Ralf Möller, Konstantina Bereta, Christian Neuenstadt, Özgür Özczeń, Michael Roshchin, Panayiotis Smeros, and Dmitrij 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.


[31] Dong hyuk Im, Sang won Lee, and Hyoung joo kim. A Version Rudra Pratap Deb Nath, Katja Hose, Torben Bach Pedersen, Marios Meimaris. EvoGen: A Generator for Synthetic Ver-


[41] Mayesha Tasnim, Diego Collarana, Damien Graux, Fabrizio Orlandi, and Maria-Esther Vidal. Summarizing Entity Temporal Evolution in Knowledge Graphs. In Companion Proceed-


ference, 2005.