Evaluating Systems and Benchmarks for Archiving Evolving Linked Datasets

Vassilis Papakonstantinou [a,*], Giannis Roussakis [a], Kostas Stefanidis [b], Irini Fundulaki [a], and Giorgos Flouris [a]
[a] *Institute of Computer Science-FORTH, Greece*
E-mail: papv@ics.forth.gr, rousakis@ics.forth.gr, fundul@ics.forth.gr, fgeo@ics.forth.gr
[b] *University of Tampere, Finland*
E-mail: kostas.stefanidis@uta.fi

Abstract. As dynamicity is an indispensable part of Linked Data, which are constantly evolving at both schema and instance level, there is a clear need for archiving systems that are able to support the efficient storage and querying of such data. The purpose of this paper is to provide a framework for systematically studying the state-of-art RDF archiving systems and the different types of queries that such systems should support. Specifically, we describe the strategies that archiving systems follow for storing multiple versions of a dataset, and detail the characteristics of the archiving benchmarks. Moreover, we evaluate the archiving systems, and present results regarding their performance. Finally, we highlight difficulties and open issues arisen during experimentation in order to serve as a springboard for researchers in the Linked Data community.

Keywords: benchmarking archiving systems, archiving and storage, versioning, LOD, RDF

1. Introduction

With the growing complexity of the Web, we face a completely different way of creating, disseminating and consuming big volumes of information. The recent explosion of the Data Web and the associated Linked Open Data (LOD) initiative [5] has led several large-scale corporate, government, or even user-generated data from different domains (e.g., DBpedia [2], Freebase [6], YAGO [32]) to be published online and become available to a wide spectrum of users [8]. Most of these datasets are represented in RDF, the de facto standard for data representation on the Web.

Dynamicity is an indispensable part of LOD [15, 33]; both the data and the schema of LOD datasets are constantly evolving for several reasons, such as the inclusion of new experimental evidence or observations, or the correction of erroneous conceptualizations [37].

The open nature of the Web implies that these changes typically happen without any warning, centralized monitoring, or reliable notification mechanism; this raises the need to keep track of the different *versions* of the datasets and introduces new challenges related to assuring the quality and traceability of Web data over time. Indeed, for many applications, having access to the latest version of a dataset is not enough. For example, applications may require access to both the old and the new version(s) to allow synchronization and/or integration of autonomously developed (but interlinked) datasets [9,16,26]. Moreover, many applications focus on identifying evolution trends in the data, in which case features like visualizing the evolution history of a dataset [25,27], or supporting historical or cross-version queries [31] are necessary.

The challenge of managing different versions of an evolving dataset is handled by *archiving systems*. Archiving systems should not only store and provide access to the different versions, but should also be able to support various types of queries on the data, including queries that access multiple versions [31] (*cross-*

[*]Corresponding author. E-mail: papv@ics.forth.gr.

version queries), queries that access the evolution history (delta) itself [10], as well as combinations of the above. For example, traditional Web archives, such as the Internet Archive[1] are only going half-way, because, even though the different versions are appropriately archived, time-traversing query capabilities are not considered.

To support these advanced functionalities, various RDF archiving mechanisms and tools have been developed [37]. In their simplest form, archiving tools just store all the different snapshots (versions) of a dataset (*full materialization*); however, alternative proposals include *delta-based approaches* [7,12,14,26,28], the use of *temporal annotations* [24,35], as well as *hybrid approaches* that combine the above techniques [21,31, 35]. Also, even though "pure" SPARQL does not support cross-version or delta-based queries, recent extensions [12,20] are addressing this need.

All these archiving strategies can support the needs associated with versioning and archiving, but different approaches excel at different aspects or needs. For example, delta-based approaches may be able to quickly answer queries on the evolution history of the data, but may not be equally efficient at cross-version queries. On the other hand, delta-based approaches are generally (depending on the evolution intensity of the dataset) less demanding in terms of storage space than, e.g., full materialization approaches.

Given the complexity of the problem and the multitude of aspects that need to be considered, being able to objectively evaluate the pros and cons of each archiving system is a challenging task that requires appropriate *benchmarks*. Benchmarking is an important process that allows not only the evaluation of different systems across different dimensions, but also the identification of their weak and strong points. Thus, benchmarks play the role of a driver for improvement, and also allow users to take informed decisions regarding the quality of different systems for different problem types and settings.

The problem of benchmarking archiving systems has been considered only very recently, and, to the best of our knowledge, only two such benchmarks exist up to this day [10,19].

In this paper, we leverage on existing benchmarks and systems in order to provide the first complete evaluation of existing archiving systems using existing archiving benchmarks. Our work shows that the lack of mature benchmark systems and standard methodologies/languages for storing and querying multiple versions causes a lot of technical difficulties in evaluating archiving systems. The main contributions of our work are the following:

- We present the existing archiving benchmarks and describe their features and characteristics.
- We analyse the most popular archiving systems, revisiting the different strategies and approaches used for maintaining multiple versions.
- We evaluate the archiving systems using existing benchmarks and we report on the different identified technical difficulties as well as systems' performance.

The paper is organized as follows: Section 2 describes the basic strategies used for implementing archiving systems, and organizes the different query types that need to be supported by such systems. Section 3 describes the most popular archiving systems and frameworks in the literature, whereas Section 4 gives some basic requirements for archiving benchmarks, and describes in detail existing benchmarks. We present an experimental evaluation in Section 5 and conclude in Section 6.

2. About Versioning

2.1. Archiving Strategies

In the literature, three alternative RDF archiving strategies have been proposed: *full materialization*, *delta-based*, and *annotated triples* approaces, each with its own advantages and disadvantages. *Hybrid* strategies (that combine the above) have also been considered. A detailed description of those approaches follows.

2.1.1. Full Materialization

Full materialization was the first and most widely used approach for storing different versions of datasets. Using this strategy, all different versions of an evolving dataset are stored explicitly in the archive [36]. Although there is no processing cost for storing the archives, the main drawback of the full materialization approach concerns scalability issues with respect to storage space: since each version is stored in its entirety, unchanged information between versions is duplicated (possibly multiple times). In scenarios where we have large versions that change often (and no mat-

[1] https://archive.org/

ter how little), the space overhead may become enormous. On the other hand, query processing over versions is usually efficient as all the versions are already materialized in the archive.

2.1.2. Delta-based

The *delta-based approach* is an alternative proposal where one full version of the dataset needs to be stored, and, for each new version, only the set of changes with respect to the previous version (also known as the *delta*) has to be kept. This strategy has much more modest space requirements when compared to the full materialization approach, as deltas are (typically) much smaller than the dataset itself. However, the *delta-based* strategy imposes additional computational costs for computing and storing deltas. Also, an extra overhead at query time is introduced, as many queries would require the on-the-fly reconstruction of one or more full versions of the data. Various approaches try to ameliorate the situation, by storing the first version and computing the deltas according to it [7,12,35] or storing the latest (current) version and computing reverse deltas with respect to it [14,17].

2.1.3. Annotated Triples

The *annotated triples* approach is based on the idea of augmenting each triple with its temporal validity. Usually, temporal validity is composed of two timestamps that determine when the triple was *created* and *deleted*; for triples that exist in the dataset (thus, have not been deleted yet) the latter is *null* [24]. This annotation allows us to reconstruct the dataset version at any given time point *t*, by just returning all triples that have been created before *t* and were deleted after time point *t* (if at all). An alternative annotation model uses a single annotation value that is used to determine the version(s) in which each triple existed in the dataset [35].

2.1.4. Hybrid Approaches

Hybrid approaches aim at combining the above strategies in order to enjoy most of the advantages of each approach, while avoiding many of their respective drawbacks. This is usually implemented as a combination of the *full materialization* and *delta-based* strategies, where several (but not all, or just one) of the versions are materialized explicitly, whereas the rest are only stored implicitly through the corresponding deltas [21]. To determine how many, and which, versions must be materialized, a cost model (such as the one proposed in [31]) could be used to quantify the corresponding overheads (including space overhead for storage, time overhead at storage time, and time overhead at query time), so as to determine the optimal storage strategy. Another combination is the use of *delta-based* and *annotated triples* strategies as there are systems that store consecutive deltas, in which each triple is augmented with a value that determines its version [35].

2.2. Versioning Query Types

An important novel challenge imposed by the management of multiple versions is the generation of different types of queries (e.g., queries that access multiple versions and/or deltas). There have been some attempts in the literature [10,31] to identify and categorize these types of queries. Our suggestion, which is a combination of them, is shown in Figure 1.

Firstly, queries are distinguished when considering their focus, in *version* and *delta* queries. Version queries consider complete versions, whereas delta queries consider deltas. Version queries can be further classified to *modern* and *historical*, depending on whether they require access to the latest version (the most common case) or a previous one. Obviously, such a categorization cannot be applied to delta queries, as they refer to time changes between versions (i.e., intervals).

In addition, queries can be further classified according to their type, to *materialization*, *single-version* and *cross-version* queries. Materialization queries essentially request the entire respective data (a full version, or the full delta); single-version queries can be answered by imposing appropriate restrictions and filters over a single dataset version or a single delta; whereas cross-version queries request data related to multiple dataset versions (or deltas).

Of course, the above categories are not exhaustive; one could easily imagine queries that belong to multiple categories, e.g., a query requesting access to a delta, as well as multiple versions. These types of queries are called *hybrid* queries.

More specifically the types of queries that we consider are:

- *Modern version materialization* queries ask for a full current version to be retrieved. For instance, in a social network scenario, one may want to ask a query about the whole network graph at present time.
- *Modern single-version structured* queries are performed in the current version of the data. For in-

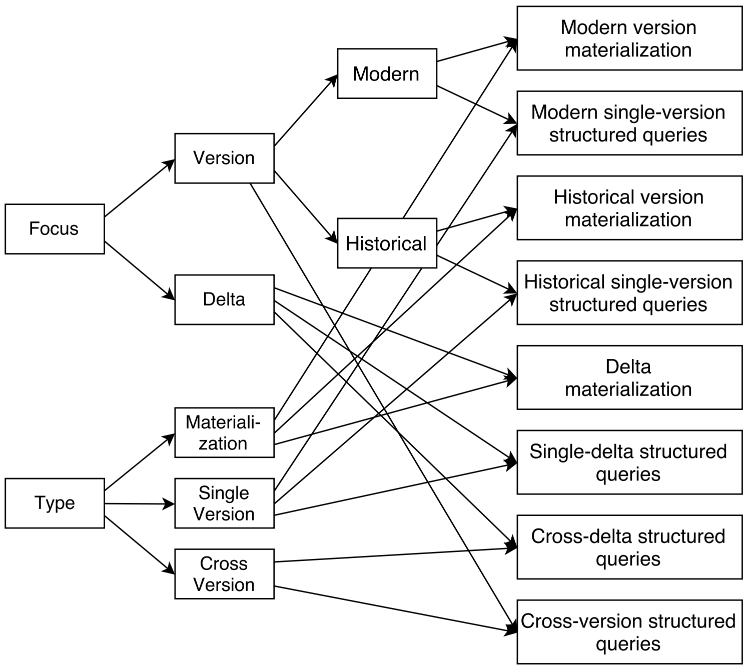
Focus
Version
Delta
Modern
Historical
Type
Materiali-zation
Single Version
Cross Version
Modern version materialization
Modern single-version structured queries
Historical version materialization
Historical single-version structured queries
Delta materialization
Single-delta structured queries
Cross-delta structured queries
Cross-version structured queries

Fig. 1. Different types of queries according to their focus and type.

stance, a query that asks for the number of friends that a certain person has at the present time.

- *Historical version materialization* queries on the other hand ask for a full past version. E.g., a query that asks for the whole network graph at a specific time in the past.
- *Historical single-version structured* queries are performed in a past version of the data. For example, when a query asks for the number of comments a post had at a specific time in the past.
- *Delta materialization* queries ask for a full delta to be retrieved from the repository. For instance, in the same social network scenario, one may want to pose a query about the total changes of the network graph that happened from some version to another.
- *Single-delta structured* queries are queries which are performed on the delta of two consecutive versions. One, for instance, could ask for the new friends that a person obtained between some version and its previous one.
- *Cross-delta structured* queries are evaluated on changes of several versions of the dataset. For example, a query that asks about how friends of a person change (e.g., friends added and/or deleted) belongs in this category.
- *Cross-version structured* queries must be evaluated on several versions of the dataset, thereby retrieving information common in many versions. For example, one may be interested in assessing all the status updates of a specific person through time.

3. RDF Archiving Systems

A variety of RDF archiving systems and frameworks have been proposed in recent years; details on these systems are discussed in the subsections below, whereas an overview of their characteristics appears in Table 1. Such characteristics are the archiving strategy that each system/framework implements, their ability to answer SPARQL queries and to identify equivalent blank nodes across versions and, finally, their ability to support versioning features such as committing, merging, branching etc.

3.1. x-RDF-3X

Neumann and Weikum [24] proposed an extension of the RDF-3X RDF engine [23]. Even though the original RDF-3X did not support archiving and versioning features, x-RDF-3X is essentially a timestamp-

System / Framework	**Archiving Strategy**	**SPARQL support**	**Blank nodes support**	**Versioning features**
x-RDF-3X [24]	Annotated Tuples	✓	-	-
SemVersion [36]	Full Materialization	-	✓	✓
Cassidy et al. [7]	Delta Based	-	-	✓
Memento [30]	Full Materialization	-	-	-
R&Wbase [35]	Annotated Tuples	✓	✓	✓
R43ples [12]	Delta Based	✓	-	✓
TailR [21]	Hybrid	-	-	-
Im et al. [14]	Delta Based	-	-	-
Dydra [1]	Full Materialization	✓	-	-

Table 1: An overview of RDF archiving systems and frameworks

based temporal RDF engine that supports versioning, time-travel access (i.e., temporal SPARQL queries) and transactions on RDF databases. To achieve such functionality they employ the *annotated triples* strategy, and augment triples with two timestamp fields referring to the *creation* and *deletion* time of each triple. Using these timestamps, the database state of a given point in time can be easily reconstructed.

Ideally, timestamps reflect the commit order of transactions, but unfortunately the commit order is not known when inserting new data. To overcome this problem, a write timestamp is assigned to each transaction once it starts updating the differential indexes (temporal small indexes that are periodically merged to the main ones), and this timestamp is then used for all subsequent operations.

To support cross-version queries, snapshot isolation and the efficient retrieval of transactions order, a *transaction inventory* is proposed. The transaction inventory (see Table 2) tracks transaction ids, their begin and commit times (BOT and EOT), the version number used for each transaction, and the largest version number of all committed transactions (highCV #) at the commit time of a transaction.

3.2. SemVersion

SemVersion [36] is inspired by the Concurrent Versioning System (CVS) [4] which was basically used in earlier years to allow collaborative development of source code during software development. SemVersion is a java library for providing versioning capabilities to RDF models and RDF-based ontology languages like RDFS. More specifically it supports *branch* and *merge* operations at the version level, as well as the reporting of conflicts.

In SemVersion, every version is annotated with metadata like its parent version, its branches, a label and a provenance URI. Versions are identified by a globally unique URI and they follow the approach of *full materialization* for storage, as they focus more on the management of the distributed engineering processes rather than the storage space necessary to store the different versions. Users can commit a new version either by providing the complete contents of the graph or by providing the delta with respect to the previous one. In both cases, every version of the RDF model is stored independently as a separate graph.

One of the main functionalities of SemVersion is the calculation of *diffs* in the structural or semantic level. A *structural* diff is the set of changes reported as sets of added/deleted triples (taking into account only the explicit triples), whereas a *semantic* diff considers also the semantically inferred triples while reporting the set of changes. One problem that may occur when building structural diffs is that the system cannot decide whether two blank nodes are equal or not, as they cannot be globally identified. This can be semantically wrong, if a blank node in one version represents the same resource as a blank node in another version. To overcome this problem, SemVersion introduces a technique called *blank node enrichment*. With this solution, an inverse functional property that leads to a unique URI is added to each blank node making it globally identifiable.

transId	version #	BOT	EOT	highCV#
T_{101}	100	2009-03-20 16:56:12	2009-03-20 17:00:01	300
T_{102}	200	2009-03-20 16:58:25	2009-03-20 16:59:15	200
T_{103}	300	2009-03-20 16:59:01	2009-03-20 16:59:42	300
...	...	...	...	...

Table 2: The transaction inventory that keeps information about all transactions [24]

3.3. Version Control for RDF Triple Stores

Cassidy and Ballantine [7] have proposed an archiving system for RDF triple stores that is based on Darcs[2] (a version control system built to manage software source code) and its theory of patches.

The system uses the delta-based strategy: each version is described as a sequence of patches (deltas) that are all applied sequentially to one version in order to construct the current one. Each of these patches is represented as a named graph consisting of a set of added and deleted triples and is stored in a different RDF store than the original data. Optionally, a dependency sub-graph may be included in the patch, which is a set of triples that have to exist in the dataset in order for a patch to be applicable to it.

A set of operations on patches is supported: the *commute* operation can revert the order of two patches; the *revert* operation reverts the most recent patch from the context; whereas the *merge* operation can be applied to parallel patches in order to combine them into one.

An implementation on MySQL[3] backend for the RedLand store [3] was evaluated and it was shown that their proposed approach of managing versions adds a significant overhead compared to the raw RDF store. More specifically, query answering becomes four to eight times slower and space consumption increased from two to four times from the raw RDF store of RedLand.

3.4. Memento

Memento [29] is an HTTP-based framework proposed by Van de Sompel et al. that connects Web archives with current resources by using datetime negotiations in HTTP. More specifically, each original resource (identified in memento terminology with URI-R) may have one or more *mementoes* (identified with URI-M$_i$, $i = 1, .., n$) which are the archived representations of the resource that summarize its state in the past. The time t_i that a memento was captured is called *Memento-datetime*.

Memento can also be adopted in the context of Linked Data [30] as it had been used for providing access to prior versions of DBpedia. To do so, versions of DBpedia are stored in a MySQL database as complete snapshots, so the full materialization approach is followed, and served through a Memento endpoint.

3.5. R&Wbase

R&Wbase [35] tracks changes and versions by following a *hybrid strategy*, as it uses the *delta-based* in conjunction with the *annotated triples* archiving strategies. In particular, triples are stored in a quad-store as consecutive deltas. Each altered triple is assigned a context value, which is a number from a continuous sequence. More specifically, every new delta obtains an even number $2 \cdot y$ that is larger than all preceding delta numbers. Then, each triple of said delta is assigned the value $2 \cdot y$ (even for added triples) and $2 \cdot y + 1$ (odd for deleted triples). Furthermore, the delta identifier $2 \cdot y$ is used in order to store the delta's provenance metadata in triple format, using the PROV-O vocabulary [18]. These metadata include a UID, the delta's parent, the responsible person of the changes, the delta's date etc. By following the above approach, it is possible to significantly reduce the required storage space, as the number of stored triples is relative to the delta size instead of the graph size, which is much smaller in most cases.

R&Wbase allows querying the data stored, using SPARQL queries that are translated in such a way that the quad-store is treated as a triple-store. In particular, when a query is applied in a specific version, all version's ancestors have to be identified, by traversing the metadata of such version, and then the query is applied to the set of returned versions. Finally, being a Git-like tool, R&Wbase supports versioning features

[2]`http://darcs.net/`
[3]`https://www.mysql.com/`

like *branching* and *merging* of previously *committed* graphs.

3.6. R43ples

R43ples [12] offers a central repository based on a Copy-Modify-Merge mechanism, where clients get the requested information via SPARQL (copy), work with it locally (modify) and commit their updates also via SPARQL (merge).

Much like R&Wbase, R43ples supports the basic versioning features like *tagging*, *branching* and *merging*. To do so, it introduces an enhanced, non-standard version of the SPARQL language that includes a set of new keywords (`REVISION`, `USER`, `MESSAGE`, `BRANCH` and `TAG`).

R43ples follows the *delta-based* approach for storing versions. In particular, each version is represented using a temporary graph which is connected to two additional named graphs corresponding to the delta's *ADD* and *DELETE* sets. The connection is based on an extended version of the PROV-O ontology [18], called Revision Management Ontology (RMO). Applying these delta sets to the prior revision will lead to the current one. The aforementioned approach of using temporary copies of graphs for storing versions and deltas tends to be rather costly when querying the data, as only medium sized data sets can be handled by R43ples. In fact, the authors of [12] noted that queries on datasets with more than a few thousand triples take longer than most users are willing to wait.

3.7. TailR

TailR [21] is a platform for preserving the history of arbitrary linked datasets over time, implemented as a Python web application. It follows the *hybrid approach* for storing the data: the history of each individual tracked resource is encoded as a series of deltas or deletes based on interspersed snapshots. More specifically, their storage model consists of *repositories*, *changesets* and *blobs*. A repository can be created by users and is actually the linked dataset along with its history. A changeset encodes the information about modifications that happen to the data at a particular time point. According to the archiving strategy they follow, there are three types of changesets: *snapshot*, *delta* and *delete* (a set of deleted triples). To decide which one must be stored when changes occurred in the data, a set of rules is followed, which are defined in such a way that try to minimize the storage and retrieval cost. Finally, blobs contain optional data that refer to changesets, as they are sometimes needed in order to answer some types of queries.

Their implementation consists of two HTTP APIs: a Push API for submitting changes according to a dataset, and a read-only Memento API for accessing the previously stored versions. All entities such as changesets and blobs are stored in the relational database system MariaDB[4].

In their experimental evaluation, the authors measured the response times of the Push and Memento API [29] as well as the growth of the required storage space for an increasing number of versions. For their experiments they used a random sample of 100K resources selected from each version of DBpedia [2] 3.2 to 3.9. Regarding the Push API response times, push requests for the first release took the longest time on average. Memento API response times tend to slightly increase for later revisions due to the longer base/delta chains that result to higher reconstruction costs. Finally, the storage overhead is directly related to the nature of the data and especially to the delta encoding.

3.8. A version management framework for RDF triple stores

Im et al. [14] propose a framework for managing RDF versions on top of relational databases (where all triples are stored in one large triple table). Their framework follows the *delta-based* approach as they store the last version and the deltas that led to it. To improve the performance of cross-delta queries, the authors introduce *aggregated deltas*, which associate the latest version with each of the previous ones (not only the last one); obviously, this comes at the cost of increasing space (storage) requirements. The delta of each version is separately stored in an *INSERT* and a *DELETE* relational table, so a version can be constructed on the fly using appropriate SQL statements.

For evaluating their approach, Im et al. used the Uniprot dataset [34] versions v1-v9 on top of an implementation in Oracle 11g Enterprise edition[5]. They evaluated their approach of aggregated deltas against the approaches of full materialization and sequential deltas. The authors conducted experiments related to storage overhead, version construction and delta computation times, compression ratio and query perfor-

[4]`https://mariadb.org/`

[5]`http://www.oracle.com/technetwork/database/enterprise-edition`

mance. As expected, their approach is less efficient than the sequential deltas, but outperforms the full materialization approach regarding storage space and deltas computation time. Moreover, their approach highly outperforms the sequential deltas approach regarding version re-construction. In particular, while construction time in the sequential delta approach is proportional to the number of past versions that must be considered, the aggregated delta can compute any version almost at constant time (with respect to the number of past versions). Regarding the query answering performance, the full materialization approach has the best performance for the types of queries that refer to specific versions, but the aggregated delta approach outperforms the sequential delta one in most cases.

3.9. Dydra

Dydra [1] is an RDF graph store service in the cloud that stores and retrieves RDF data through SPARQL, LDF and LDP interfaces. In order to have access to previous store states, in addition to the current one, includes a REVISION clause analogous to GRAPH, as each versioned dataset is stored in its own named graph in a quad store. They characterize the queries that are supported by Dydra according to two dimension: the dataset constitution, and algebra combination. *Constitution* determines the revisions that are included in the target dataset, and may be: none, single, multiple, a range or a difference of versions and *Combination* concerns how the query combines the later versions.

4. Benchmarking RDF Archiving Systems

A benchmark is a set of tests against which the performance of a system is evaluated. In particular, a benchmark helps computer systems to compare and assess their performance in order for them to become more efficient and competitive.

In order for the systems to be able to use the benchmark and report reliable results, a set of generic and more domain-specific requirements and characteristics must be satisfied. First, the benchmark should be *open* and easily *accessible* from all third parties that are interested to test their systems. Second, it has to be *unbiased*, which means that there should not exist a conflict of interest between the creators of the benchmark and the creators of the system(s) under test. These features guarantee a fair and reproducible evaluation of the systems under test.

To guarantee (additionally) that the benchmark will produce useful results, it should be highly *configurable* and *scalable*, in order to cope with the different characteristics and needs of each system. Pertaining to our focus on benchmarks for archiving systems, configurability and scalability may refer to the number of versions that a data generator can produce, the size of each version, the number of changes from version to version etc.

In addition, the benchmark should be *approach-agnostic* to different implementation techniques. For example, for benchmarks related to RDF archiving systems, one should also take into account the different strategies that are being employed, and should be agnostic with regards to the strategy that an RDF archiving system uses for its implementation. In particular, the benchmark should be fair with respect to the real expected use of such a system, and should not artificially boost or penalize specific strategies.

Finally, the benchmark should be *extensible*, to be able to test additional features or requirements for an archiving system that may appear in the future.

To our knowledge, there have been only two proposed benchmarks for RDF archiving systems in the literature, which are described in detail below.

4.1. BEAR

Fernandez et al. [11,10] have proposed a blueprint on benchmarking semantic web archiving systems by defining a set of operators that cover crucial aspects of querying and archiving semantic web data. To instantiate their blueprints in a real-world scenario, they introduced the *BEAR benchmark*, along with an implementation and evaluation of the three archiving strategies (*Full Materialization*, *Delta-Based* and *Annotated Triples*) described in Section 2.1.

Based on their analysis of these RDF archiving strategies, they provide a set of directions that must be followed when evaluating the efficiency of RDF archiving systems, many of which are similar to the requirements we outlined above. The first BEAR directive for benchmarks is that they should be agnostic with respect to the used archiving strategy in order for the comparison to be fair. Secondly, queries have to be simple and become more complex as the strategies and systems are better understood. And finally, the benchmark should be extensible as lessons learnt from previous work and new retrieval features arise.

As a basis for comparing the different archiving strategies, they introduced 4 features that describe the dataset configuration. The proposed features are the following:

- *Data dynamicity* measures the number of changes between versions, and is described via the *change ratio* and the *data growth*. The *change ratio* quantifies how much (what proportion) of the dataset changes from one version to another and the *data growth* determines how its size changes from one version to another.
- *Data static core* contains the triples that exist in all dataset versions.
- *Total version-oblivious triples* computes the total number of different triples in an archive, independently of their timestamp (i.e., the version in which they appear).
- *RDF vocabulary* represents the different subjects, predicates and objects in an RDF archive.

Regarding the generation of the queries of the benchmark, the *result cardinality* and *selectivity* of the query should be considered, keeping in mind that the results of a query can highly vary among different versions. For example, by selecting queries with similar result cardinality and selectivity, one could guarantee that potential retrieval differences in response times could be attributed to the archiving strategy. In order to be able to judge the different systems, authors introduced various categories of queries, which are similar to the ones we discussed in Section 2.2 and have been used as a source of inspiration for our categorization. In particular, the authors propose queries on versions (i.e., modern and historical version materialization queries), deltas (delta materialization and structured queries), as well as the so-called change materialization queries, which essentially check the version in which the answer to a query changes with respect to previous versions.

Even though BEAR provides a detailed theoretical analysis of the features that are useful in the process of designing a benchmark, it fails to satisfy one of the five requirements that we have previously set. In particular, its data workload is composed of a static dataset, so BEAR is not a benchmark generator; thus, the configurability and scalability requirements are not met.

4.2. EvoGen

Meimaris and Papastefanatos have proposed the *EvoGen Benchmark Suite* [19], a generator for evolving RDF data that is used for benchmarking archiving and change detection systems. EvoGen is based on the LUBM generator [13], extended with 10 new classes and 19 new properties in order to support schema evolution. Their benchmarking methodology is based on a set of requirements and parameters that affect the data generation process, the context of the tested application and the query workload, as required by the nature of the evolving data.

EvoGen is a *Benchmark Generator*, and is extensible and highly configurable in terms of the number of generated versions and the number of changes occurring from version to version. Similarly, the query workload is generated adaptively to the data generation process. EvoGen takes into account the archiving strategy of the system under test, by providing adequate input data formats (full versions, deltas, etc.) as appropriate.

In more details, EvoGen defines a set of parameters that are taken into account in the data and query workload generation processes. The first category of parameters refers to the evolution of instances and consists of the parameters *Shift* and *Monotonicity*. The *Shift* parameter shows how a dataset evolves with respect to its size and can be distinguished to a *positive* and a *negative shift* for versions of *increasing* or *decreasing* size respectively. The *monotonicity* property is a boolean value that determines whether the above shift is monotonic (i.e., only additions or only deletions happen); monotonic shifts can be used to simulate datasets where data strictly increased or decreased, such as sensor data.

The second category of parameters includes the parameters *ontology evolution* and *schema variation* that refer to the schema evolution of the dataset. The *ontology evolution* parameter is a number representing the amount of change to happen, computed as the ratio of the number of added classes to the number of total classes in the original dataset. The *schema variation* parameter ranges from 0 to 1 and quantifies the percentage of different characteristic sets, with respect to the total number of possible characteristic sets that will be created for each new class introduced in the schema. We recall the characteristic set definition from [22] where for each entity s occurring in an RDF data set, its characteristic set R is defined as follows:

$$S_c(s) = \{p \mid \exists o : (s, p, o) \in R\}.$$

In EvoGen, the user is able to choose the output format of the generated data by allowing him to request fully materialized versions or deltas; this allows supporting (and testing) systems employing different archiving strategies.

The query workload produced by EvoGen leverages the 14 LUBM queries, appropriately adapted to apply for evolving versions. In particular, EvoGen generates the following six types of queries, which are based on the previous generated data and their characteristics:

– *Retrieval of a diachronic dataset*: a query asking for all the triples in all versions of a dataset.
– *Retrieval of a specific version*: a query requesting all triples in a specific version (i.e., modern or historical version materialization queries).
– *Snapshot queries* on the data, i.e., queries accessing a single version (*single-version historical queries*).
– *Longitudinal (temporal) queries* that retrieve the timeline of particular subgraphs, through a subset of past versions (cross-version structured queries).
– *Queries on changes*, which access the deltas (delta materialization or single-delta structured queries).
– *Mixed queries* which use a mix of sub-queries from the above types (hybrid queries).

EvoGen is, in practice, a more complete benchmark, as it is a approach-agnostic, highly configurable and extensible benchmark generator. However, its query workload seems to exhibit some sort of approach-dependence, in the sense that the delta-based queries require that benchmarked systems store information about the low level deltas (additions/deletion of classes, addition/deletion of class instances etc.) in order to be answered. Moreover, to successfully answer the 14 original LUBM queries, the benchmarked systems are required to support reasoning (forward or backward). As a result, archiving systems that do not support reasoning functionalities fail to answer the majority (11 of 14) of generated queries (see Section 5.3 on how we handled this problem for R43ples).

5. Evaluation

Taking into consideration the characteristics of the previously described benchmarking approaches, BEAR and EvoGen (Subsections 4.1 and 4.2, respectively), we chose to use the latter as our baseline benchmark. The main reason behind this choice was that EvoGen is a *benchmark generator* which allows us to produce datasets of varying sizes or change granularity, thereby enabling us to meet the requirements for checking the limitations of benchmarked systems. On the contrary, BEAR's dataset uses a static set of data, composed of the first 58 weekly snapshots from the Dynamic Linked Data Observatory[6] corpus, which monitors weekly crawls from more than 650 different domains.

Out of the 9 RDF Archiving Systems appearing in Table 1, we succeeded to conduct experiments only in R43ples and TailR. For the rest, we encountered various difficulties related to the installation and use (e.g., no documentation on how to use a given API, no way to contact with developers for further instructions, no access to the versioning extensions, or no option for a local deployment to be tested).

All the experiments were conducted in a single machine, which uses an Intel Xeon E5-2630 at 2.30GHz, with a total of 384GB of RAM running Debian Linux wheezy version, with Linux kernel 3.16.4. We dedicated 64GB of memory for R43ples which is implemented in JAVA. TailR was free to allocate as much memory as it required to execute the benchmarks.

5.1. Data Generator

Keeping in mind the limitations of benchmarked systems, we chose to use EvoGen for producing three different datasets with initial size of around 1M, 5M and 10M triples. Each of these datasets evolved (producing new versions) until we had 3, 5 and 7 total versions for each of the above dataset sizes. Regarding the version-to-version growth, we used a realistic rate of around 3.5% per version, so the total growth rate for 3, 5 and 7 versions was around 6%, 14% and 23%, respectively, as shown in Tables 3, 4 and 5.

Despite the fact that EvoGen gives the ability to produce scalable datasets, it is not stable enough regarding the way that the initial dataset evolves. In particular, the *shift* parameter of EvoGen (ranging from 0 to 1, as only positive monotonic shifts are supported on its implementation), along with the *schema evolution* parameter, do not seem to evolve the initial dataset according to the specifications, leading to minor changes even for their maximum possible values. Also, the *shift* parameter does not have a deterministic effect; for example, in one experiment, the same *shift* (0.1) changed the 1M, 5M and 10M datasets by 987%, 304% and 187% through 30 versions, respectively.

In order to overcome the above undesirable behavior of EvoGen and reach the growth rates that we had set,

[6]`http://swse.deri.org/dyldo`

we produced a large number of versions and manually chose those that would meet our goals. More specifically, in order to produce the 1M dataset, we produced 5 universities (recall that EvoGen is based on the LUBM generator) and requested the production of 20 versions, using 0.05 as the value of the *shift* parameter and the value 0.7 for the *schema evolution* parameter; from the 20 produced versions, we selected the 1st, 6th, 9th, 11th, 14th, 15th and 17th, and labelled them with v0, v1, v2, v3, v4, v5 and v6, as shown in Table 3. Likewise, for creating the 5M and 10M datasets we produced 13 and 23 universities respectively, that evolved in 30 versions with 0.1 *shift* and 0.7 *schema evolution*, and selected the appropriate intermediate ones, to get the versions shown in Tables 4 and 5.

Table 3: Versions generation with about 1M triples.

	Triples	New (added) triples	Cumulative growth
v0	1,291,816	–	–
v1	1,335,681	43,865	3.40%
v2	1,379,150	43,469	6.76%
v3	1,439,175	60,025	11.41%
v4	1,499,579	60,404	16.08%
v5	1,565,415	65,836	21.18%
v6	1,593,791	28,376	23.38%

Table 4: Versions generation with about 5M triples.

	Triples	New (added) triples	Cumulative growth
v0	5,039,543	–	–
v1	5,165,491	125,948	2.50%
v2	5,314,252	148,761	5.45%
v3	5,536,769	222,517	9.87%
v4	5,687,347	150,578	12.85%
v5	5,959,572	272,225	18.26%
v6	6,253,809	294,237	24.09%

5.2. *Data Ingestion*

For benchmarking data ingestion, we measured the execution time and the space requirements of R43ples and TailR for storing the different versions of our 3 datasets.

Tables 6 and 7 show, respectively, the experimental results for the datasets with about 1M and 5M

Table 5: Versions generation with about 10M triples.

	Triples	New (added) triples	Cumulative growth
v0	10,438,730	–	–
v1	10,744,654	305,924	2.93%
v2	10,986,713	242,059	5.25%
v3	11,424,557	437,844	9.44%
v4	11,750,295	325,738	12.56%
v5	12,126,162	375,867	16.17%
v6	12,577,826	451,664	20.49%

triples, when R43ples is used[7]. It is clear that the execution time for the ingestion of the first version is much higher compared to the other versions, which is reasonable given that R43ples is a delta-based archiving system which keeps materialized the first version produced. In fact, R43ples stores v0, as-is (because it is the first version), whereas the other versions are stored through their (smaller) deltas. This is also reflected on the space requirements, where the first version requires much more space than the rest. Overall, for all versions, except the first one, the ingestion execution time and space requirement are proportional to the number of added triples.

Table 6: Data ingestion in R43ples - 1M.

	Duration (ms)	Size (KBs)	Size increase (KBs)
v0	265,000	753	–
v1	11,000	761	8
v2	12,000	777	16
v3	14,000	811	34
v4	14,000	849	38
v5	16,000	888	39
v6	7,000	912	24

Table 7: Data ingestion in R43ples - 5M.

	Duration (ms)	Size (KBs)	Size increase (KBs)
v0	1,201,000	2,778	–
v1	42,000	2,841	63
v2	36,000	2,930	89
v3	63,000	3,058	128
v4	43,000	3,144	86
v5	88,000	3,310	166
v6	101,000	3,480	170

[7]Due to memory limitations entailing from the implementation of R43ples, we could not conduct experiments with 10M triples.

Unlike R43triples, TailR allowed us to load all three datasets, leading to the results shown in Tables 8, 9 and 10. Unlike R43ples, the execution time for the data ingestion is not correlated to the version itself (first or other), but is roughly proportional to the size of the imported version. As for the space requirements, TailR uses a hybrid approach in which some versions are materialized and stored as they are within the system, whereas for others only the deltas from the previous versions are stored. This can be seen clearly in Table 10, where the number of triples in version v4 are more than those in version v3, while the space requirements are less in v4 compared to v3, which implies that v4 was not fully materialized during its ingestion.

Table 8: Data ingestion in TailR - 1M.

	Duration (ms)	Size (KBs)	Size increase (KBs)
v0	55,000	13,984	–
v1	58,000	27,296	13,312
v2	60,000	41,632	14,336
v3	63,000	58,016	16,384
v4	66,000	70,304	12,288
v5	68,000	86,688	16,384
v6	69,000	10,3072	16,384

Table 9: Data ingestion in TailR - 5M.

	Duration (ms)	Size (KBs)	Size increase (KBs)
v0	209,000	148,128	–
v1	220,000	197,280	49,152
v2	229,000	242,340	45,060
v3	239,000	295,588	53,248
v4	246,000	348,836	53,248
v5	257,000	406,180	57,344
v6	270,000	467,620	61,440

Table 10: Data ingestion in TailR - 10M.

	Duration (ms)	Size (KBs)	Size increase (KBs)
v0	418,000	565,924	–
v1	428,000	656,036	90,112
v2	455,000	754,340	98,304
v3	471,000	869,028	114,688
v4	488,000	963,236	94,208
v5	508,000	1,073,828	110,592
v6	526,000	1,188,516	114,688

Overall, our tables show that R43ples requires less space for the storage of the datasets versions due to its delta-based implementation and is also faster with respect to import time for all versions, except from the first one which has to be materialized. However, R43ples has some limitations regarding the size of a dataset that can be stored, due to its main memory implementation; this is not an issue for TailR.

5.3. Query performance

For studying query performance, we used the R43ples system only, because TailR supports data storing and archiving, but does not offer query facilities on the stored data.

As already mentioned, EvoGen's query workload leverages the 14 original LUBM queries, appropriately adapted as to be applicable on evolving data[8]. For our experiments on query performance, we ran these queries on the generated datasets and report on the query evaluation time and the number of the query results. Query evaluation time consists of the time required for version materialization, when a version reconstruction is needed, and the time required for the actual query execution. Given the fact that Evogen produces queries which (unrealistically) assume reasoning capabilities on behalf of the query engine in order to be answered properly (Section 4.2), we had to update them accordingly in order to get non-empty result sets by R43ples. These updates included the replacement of certain implicit triple patterns that were only implicitly present in the datasets, with similar ones that were explicitly present.

Our analysis is broken down into three types of queries, namely single-version queries, cross-version queries and materialization queries. For getting the cross-version queries, we converted the single-version ones by rewriting them so as to consider all versions (details in Appendix B). For materialization queries, according to their definition, we retrieve the contents of an entire version of a dataset, or even the whole set of versions of the dataset.

Note, that Evogen produces also queries that can be applied over the changes (i.e., deltas) between the produced versions. However, even though R43ples implements a delta-based approach for storing versions, it does not provide any functions for the access or management of each produced delta; this process remains

[8]Detailed descriptions on the queries can be found in Appendix A and B.

as a black box. As a result, we were unable to conduct any experiments over this type of queries.

5.3.1. *Single-version queries*

Single-version queries are answered by applying restrictions and filters over a single dataset version. In our study, we considered settings in which either 3, 5 or 7 versions participate. In all cases, only one version needs to be materialized for answering a query. When the query refers to a non-materialized version, R43ples had to build this version using its deltas from the materialized one, and run the query over it.

Tables 11 and 12 show the query performance evaluation when 3, 5 and 7 versions with about 1M and 5M triples are used (recall that the 10M-triple dataset could not be loaded in R43ples). The detailed SPARQL representation of the queries can be found in Appendix A.

The tables show that the materialization part, which is needed for query answering, is solely affected by the dataset version that is mentioned in the query. Overall, 5 queries refer to the first version, 5 queries refer to the last version and 4 queries refer to the version in the middle of the dataset (namely, to the second, third or fourth version, of the dataset with the 3, 5 or 7 versions, respectively). This explains the different amounts of time required for version materialization.

The query execution time is, in principle, affected by both the query form, i.e., query complexity, and by the number of results. Looking deeper in our results, we noticed that the query form is much more important than the number of results. For instance, in the 3 versions scenario, the queries 110 and 113 are two of the three queries with the highest execution times and fetch 1 and 0 results, respectively. This is because both queries contain the triples pattern of the generic form: ?x rdf:type ?y (Table 20), which is instantiated by many thousands of triples within the dataset versions. This pattern is also joined with another triple pattern in both cases, causing the queries to be "expensive".

5.3.2. *Cross-version queries*

Cross-version queries request data related to multiple dataset versions. Again, we considered cases in which either 3, 5 or 7 versions participate. When a query is submitted, R43ples constructs the needed versions in order to be able to identify the query results. The queries constructed by EvoGen and used in our benchmark need all dataset versions for identifying their results.

Tables 13 and 14 show the query performance evaluation when 3, 5 and 7 versions with about 1M triples and 5M triples, respectively, are used. Compared to single-version queries, cross-version queries demand many more version reconstructions, and thus, the materialization time is significantly larger.

Regarding query execution, we noticed that the query form is more important than the number of results, as in single-version queries. That is, given that queries are applied over the union of the versions of a dataset, the number of joins plays a critical role for query execution. For instance, query ll9 (Appendix B) is the most time-consuming query, as each part of its union consists of 6 triple patterns, which should be satisfied along with 3 joins. The same also holds for query ll2, which is the second most expensive query, consisting also of 6 query patterns per union part and 3 joins.

5.3.3. *Materialization queries*

Materialization queries request an entire version of a dataset, or even the whole set of versions of a dataset (diachronic queries). Intuitively, this type of queries examines how effectively each system can retrieve the contents of a specific version or the contents of the entire dataset (i.e., all versions). As in single- and cross-version queries, we considered datasets that consists of 3, 5 and 7 versions. Given the nature of these queries, both R43ples and TailR can be used for experimentation.

Running the EvoGen materialization queries (originally expressed in SPARQL) over R43ples and TailR required some preprocessing. For R43ples, this was relatively easy: we only had to change them syntactically, i.e., as explained above, replace certain implicit triple patterns that were only implicitly present in the datasets, with others that were explicitly present. On the other hand, TailR does not support SPARQL, and access to versions is managed using the Restful API provided by Memento [29]. As a result, the provided queries had to be "transformed" into the corresponding service calls which would be applied over the Memento API, and the query execution times reported below actually refer to the response times of the corresponding service calls. As a result of this "hack", the direct comparison of the performance of materialization queries between TailR and R43ples is somewhat problematic.

Tables 15 and 16 present the query evaluation performance, i.e., the version materialization time and the query execution time, in R43ples, in which only one version is materialized, when 3, 5 and 7 versions with about 1M and 5M triples are used. As expected, di-

Table 11: Query evaluation performance for single-version queries when 3, 5 and 7 versions with about 1M triples are used: version materialization time, query execution time, number of query results.

Queries	3 versions			5 versions			7 versions		
	Material. (ms)	Exec. (ms)	# results	Material. (ms)	Exec. (ms)	# results	Material. (ms)	Exec. (ms)	# results
l1	6	2	1	5	1	1	5	1	1
l2	47,206	185	9	86,991	189	9	61,634	250	9
l3	8	2	10	5	1	10	6	2	10
l4	49,858	7	49	63,413	17,398	49	85,960	11	49
l5	52,462	12,304	871	59,062	7	871	63,034	20,639	871
l6	67,481	78	38,151	76,708	74	41,725	71,662	83	46,771
l7	5	2	31	4	2	50	4	2	50
l8	4	110	11,065	3	124	11,403	4	162	12,227
l9	58,929	411	2,077	59,211	536	2,175	59,323	635	2,302
l10	53,113	13,000	1	61,208	3	1	83,163	4	1
l11	47,618	3	15	63,270	21,368	15	81,953	4	15
l12	47,547	7	661	81,641	7	661	86,855	9	661
l13	52,442	12,850	0	81,666	3	0	86,243	3	0
l14	9	54	41,725	8	42	51,229	8	44	58,550

Table 12: Query evaluation performance for single-version queries when 3, 5 and 7 versions with about 5M triples are used: version materialization time, query execution time, number of query results.

Queries	3 versions			5 versions			7 versions		
	Material. (ms)	Exec.(ms)	# results	Material.(ms)	Exec.(ms)	# results	Material.(ms)	Exec.(ms)	# results
l1	6	1	13	5	2	13	6	1	13
l2	219,647	1,498	36	381,304	524	36	310,699	659	36
l3	8	2	10	6	2	10	5	1	10
l4	240,145	10	49	280,493	73,794	83	427,223	9	83
l5	268,850	54,923	964	240,778	12	2,202	328,638	93,508	2,202
l6	187,771	194	250,046	258,127	134	121,996	300,788	153	137,062
l7	4	2	115	3	3	148	4	3	148
l8	4	146	13,109	4	196	17,395	3	278	19,391
l9	275,716	1,359	5,735	240,125	1,384	6,179	302,739	1,781	6,744
l10	255,133	55,875	1	253,774	3	1	400,907	3	1
l11	235,678	4	15	282,567	78,811	15	392,096	4	15
l12	218,862	9	661	326,273	8	661	437,186	12	661
l13	247,911	54,739	0	341,057	4	0	418,462	3	0
l14	9	87	121,996	9	81	147,309	17	202	184,600

achronic queries are the most time consuming both for version materialization and query execution, because the system has to reconstruct all the un-materialized versions at query time; also the result set is much larger compared to other query types. Similar conclusions can be reached for TailR, whose results are reported in Tables 17, 18 and 19, respectively.

6. Summary

The primary focus of this paper was on systematically studying the state-of-art approaches for managing and benchmarking evolving RDF data. We presented the basic strategies that archiving tools follow for storing multiple versions of a dataset, and described the existing archiving benchmarks along with their features and characteristics. Moreover, we evaluated the archiving systems using the EvoGen bench-

Table 13: Query evaluation performance for cross-version queries when 3, 5 and 7 versions with about 1M triples are used: version materialization time, query execution time, number of query results.

Queries	3 versions			5 versions			7 versions		
	Material. (ms)	Exec. (ms)	# results	Material. (ms)	Exec. (ms)	# results	Material. (ms)	Exec. (ms)	# results
ll1	116,707	2	3	257,981	2	5	477,153	2	7
ll2	115,874	688	30	259,569	1,310	52	464,257	1796	84
ll3	116,776	4	30	260,452	3	50	456,867	3	70
ll4	120,280	4	147	257,719	5	245	439,493	8	343
ll5	100,591	15	2,613	257,307	17	4,355	464,413	25	6,097
ll6	121,222	88	118,027	258,583	132	238,313	435,356	216	397,736
ll7	118,686	3	150	258,887	4	250	468,027	5	350
ll8	118,484	387	32,837	260,846	629	55,663	459,865	899	85,589
ll9	119,229	1,870	6,329	270,951	3,095	11,630	441,341	5,089	17,995
ll10	116,929	3	3	259,301	4	5	464,372	3	7
ll11	116,826	2	45	262,036	2	75	461,161	3	105
ll12	114,633	31	1,983	258,842	60	3,305	464,661	64	4,809
ll13	120,205	2	0	258,916	2	0	473,763	2	0
ll14	121,050	76	118,027	252,186	109	238,313	471,760	194	397,736

Table 14: Query evaluation performance for cross-version queries when 3, 5 and 7 versions with about 5M triples are used: version materialization time, query execution time, number of query results.

Queries	3 versions			5 versions			7 versions		
	Material. (ms)	Exec. (ms)	# results	Material. (ms)	Exec. (ms)	# results	Material. (ms)	Exec. (ms)	# results
ll1	590,115	1	15	1,247,531	3	65	2,343,355	3	91
ll2	586,536	2,062	108	1,248,570	3,514	185	2,256,628	4,913	288
ll3	564,908	2	30	1,229,323	2	50	2,223,244	4	70
ll4	607,541	6	181	1,252,369	9	415	2,132,360	12	581
ll5	505,789	24	4,130	1,248,326	45	11,010	2,262,908	60	15,414
ll6	595,201	297	342,826	1,242,965	313	695,557	2,096,159	580	1,178,254
ll7	600,583	4	264	1,271,684	8	740	2,283,217	10	1,036
ll8	580,881	450	34,885	1,223,805	766	69,831	2,207,608	1,384	135,737
ll9	533,036	5,096	17,649	1,245,004	7,368	34,066	2,160,584	13,946	53,908
ll10	603,552	3	15	1,227,828	3	65	2,198,790	5	91
ll11	586,861	2	45	1,234,594	3	75	2,254,713	3	105
ll12	582,098	35	2,070	1,243,615	49	3,815	2,237,536	99	6,713
ll13	578,978	6	0	1,267,584	3	0	2,307,237	2	0
ll14	606,849	167	342,826	1,238,183	294	695,557	2,306,815	542	1,178,254

mark, which allows producing datasets of varying sizes and change granularity, thereby enabling us to check the limitations of benchmarked systems. Our results showed that, even though EvoGen can produce datasets of varying sizes and characteristics, it is not stable or reliable enough regarding the production of the evolving datasets.

We succeeded to conduct experiments only in R43ples and TailR; for the other systems, we encountered a number of difficulties related to installation and use. R43ples requires less space than TailR for the storage of the datasets versions due to its delta-based implementation and it is also faster with respect to import time for all versions, except from the first one which has to be fully materialized. However, R43ples has some limitations regarding the size of a dataset that can be stored, due to its main memory implementation; this is not an issue in TailR.

Table 15: Query evaluation performance, in R43ples, for materialization queries when 3, 5 and 7 versions with about 1M triples are used: version materialization time and query execution time.

Queries	3 versions		5 versions		7 versions	
	Material. (ms)	Exec. (ms)	Material. (ms)	Exec. (ms)	Material. (ms)	Exec. (ms)
diachronic	84,269	2,680	218,994	3,525	412,871	5,710
version1	71,526	583	77,047	586	91,116	725
version2	65,026	728	76,570	592	84,383	689
version3	11,298	1,125	74,518	725	87,080	732
version4	-	-	84,832	771	96,754	626
version5	-	-	14,014	1,126	87,860	701
version6	-	-	-	-	88,206	857
version7	-	-	-	-	19,915	1,278

Table 16: Query evaluation performance, in R43ples, for materialization queries when 3, 5 and 7 versions with about 5M triples are used: version materialization time and query execution time.

Queries	3 versions		5 versions		7 versions	
	Material. (ms)	Exec. (ms)	Material. (ms)	Exec. (ms)	Material. (ms)	Exec. (ms)
diachronic	366,275	7,707	1,167,750	14,140	2,203,072	22,319
version1	281,348	2,061	299,671	2,131	409,601	2,966
version2	261,406	2,282	332,564	3,081	372,448	2,443
version3	40,337	3,290	301,379	2,216	389,982	2,259
version4	-	-	301,661	3,372	357,960	2,805
version5	-	-	53,522	3,086	354,783	3,221
version6	-	-	-	-	386,212	3,429
version7	-	-	-	-	75,942	3,374

Table 17: Query execution time, in TailR, for materialization queries when 3, 5 and 7 versions with about 1M triples are used.

Queries	3 versions	5 versions	7 versions
	Exec. (ms)	Exec. (ms)	Exec. (ms)
diachronic	7,010	12,046	19,183
version1	2,600	262	2,630
version2	2,770	269	2,710
version3	2,910	271	2,630
version4	-	289	2,330
version5	-	253	2,950
version6	-	-	2,700
version7	-	-	3,150

Table 18: Query execution time, in TailR, for materialization queries when 3, 5 and 7 versions with about 5M triples are used.

Queries	3 versions	5 versions	7 versions
	Exec. (ms)	Exec. (ms)	Exec. (ms)
diachronic	29,210	51,100	77,260
version1	8,930	8,210	9,050
version2	10,870	110,020	10,240
version3	9,710	9,350	10,010
version4	-	11,150	10,980
version5	-	11,010	11,100
version6	-	-	10,770
version7	-	-	11,680

For evaluating query performance, we used mostly the R43ples system, since TailR does not support query facilities on the stored data. For running our experiments, we had to update the queries produced by EvoGen, since it assumes impractical reasoning capabilities on behalf of the query engine in order to answer queries properly. For both single- and cross-version queries, we noticed that the execution time depends mostly on the query complexity than on the size of the result set. Given the nature of materialization queries, both R43ples and TailR can be used for experimentation. However, since TailR does not support SPARQL, we had to transform the queries, so as to access the versions via particular service calls. This way, the ex-

Table 19: Query execution time, in TailR, for materialization queries when 3, 5 and 7 versions with about 10M triples are used.

Queries	3 versions	5 versions	7 versions
	Exec. (ms)	Exec. (ms)	Exec. (ms)
diachronic	61,510	108,000	154,340
version1	19,830	19,560	20,100
version2	20,150	20,230	20,320
version3	19,910	19,120	18,950
version4	-	21,170	20,150
version5	-	22,520	21,220
version6	-	-	22,800
version7	-	-	25,180

ecution times of materialization queries from R43ples and TailR cannot be directly compared. Finally, even though R43ples implements a delta-based approach for storing versions, it does not provide functionalities for accessing the produced deltas, thus, we were unable to conduct any experiments over delta-based queries.

Overall, the technical difficulties that arose during experimentation, indicate the lack of mature benchmark systems and standard methodologies for storing and querying multiple datasets versions.

Acknowledgments

The work presented in this paper was funded by the H2020 project HOBBIT (#688227).

References

[1] Anderson, J., Bendiken, A.: Transaction-time queries in dydra. In: Joint Proceedings of the 2nd Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW 2016) and the 3rd Workshop on Linked Data Quality (LDQ 2016) co-located with 13th European Semantic Web Conference (ESWC 2016): MEPDaW-LDQ. vol. 1585, pp. 11–19 (2016)

[2] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A nucleus for a web of open data. In: The Semantic Web (2007)

[3] Beckett, D.: The design and implementation of the Redland RDF application framework. Computer Networks 39(5) (2002)

[4] Berliner, B.: CVS II: Parallelizing software development. In: USENIX Winter 1990 Technical Conference. vol. 341 (1990)

[5] Bizer, C., Heath, T., Berners-Lee, T.: Linked Data-the story so far. Semantic Services, Interoperability and Web Applications: Emerging Concepts (2009)

[6] Bollacker, K., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: a collaboratively created graph database for structuring human knowledge. In: SIGMOD (2008)

[7] Cassidy, S., Ballantine, J.: Version Control for RDF Triple Stores. ICSOFT (ISDM/EHST/DC) 7 (2007)

[8] Christophides, V., Efthymiou, V., Stefanidis, K.: Entity Resolution in the Web of Data. Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool Publishers (2015)

[9] Cloran, R., Irvin, B.: Transmitting RDF graph deltas for a cheaper semantic Web. In: SATNAC (2005)

[10] Fernandez Garcia, J.D., Umbrich, J., Polleres, A.: BEAR: Benchmarking the Efficiency of RDF Archiving. Tech. rep., Department für Informationsverarbeitung und Prozessmanagement, WU Vienna University of Economics and Business (2015)

[11] Fernandez Garcia, J.D., Umbrich, J., Polleres, A., Knuth, M.: Evaluating Query and Storage Strategies for RDF Archives. In: SEMANTiCS (2016)

[12] Graube, M., Hensel, S., Urbas, L.: R43ples: Revisions for triples. LDQ (2014)

[13] Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2) (2005)

[14] Im, D.H., Lee, S.W., Kim, H.J.: A version management framework for RDF triple stores. IJSEKE 22(01) (2012)

[15] Käfer, T., Abdelrahman, A., Umbrich, J., O'Byrne, P., Hogan, A.: Observing Linked Data Dynamics. In: ESWC (2013)

[16] Kondylakis, H., Plexousakis, D.: Ontology evolution without tears. JWS 19 (2013)

[17] Kondylakis, H., Plexousakis, D.: Ontology evolution without tears. Journal of Web Semantics 19 (2013)

[18] Lebo, T., Sahoo, S., McGuinness, D., Belhajjame, K., Cheney, J., Corsar, D., Garijo, D., Soiland-Reyes, S., Zednik, S., Zhao, J.: PROV-O: The PROV Ontology. W3C Recommendation 30 (2013)

[19] Meimaris, M., Papastefanatos, G.: The EvoGen Benchmark Suite for Evolving RDF Data. MeDAW (2016)

[20] Meimaris, M., Papastefanatos, G., Viglas, S., Stavrakas, Y., Pateritsas, C., Anagnostopoulos, I.: A Query Language for Multi-version Data Web Archives. In: arXiv:1504.01891 (2016)

[21] Meinhardt, P., Knuth, M., Sack, H.: TailR: a platform for preserving history on the web of data. In: SEMANTiCS. ACM (2015)

[22] Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: ICDE (2011)

[23] Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. VLDB Endowment 1(1) (2008)

[24] Neumann, T., Weikum, G.: x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. VLDB Endowment 3(1-2) (2010)

[25] Noy, N.F., Chugh, A., Liu, W., Musen, M.A.: A Framework for Ontology Evolution in Collaborative Environments. In: ISWC (2006)

[26] Papavasileiou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V.: High-level change detection in RDF(S) KBs. ACM TODS 38(1) (2013)

[27] Roussakis, Y., Chrysakis, I., Stefanidis, K., Flouris, G.: D2V: A tool for defining, detecting and visualizing changes on the data web. In: Proceedings of the 14^{th} International Semantic Web Conference, Posters and Demonstrations Track (ISWC-15) (2015)

[28] Roussakis, Y., Chrysakis, I., Stefanidis, K., Flouris, G.,

Stavrakas, Y.: A flexible framework for understanding the dynamics of evolving RDF datasets. In: Proceedings of the 14th International Semantic Web Conference (ISWC-15) (2015)

[29] Van de Sompel, H., Nelson, M.L., Sanderson, R., Balakireva, L.L., Ainsworth, S., Shankar, H.: Memento: Time travel for the web. arXiv preprint arXiv:0911.1112 (2009)

[30] Van de Sompel, H., Sanderson, R., Nelson, M.L., Balakireva, L.L., Shankar, H., Ainsworth, S.: An HTTP-based versioning mechanism for linked data. arXiv preprint arXiv:1003.3661 (2010)

[31] Stefanidis, K., Chrysakis, I., Flouris, G.: On designing archiving policies for evolving RDF datasets on the Web. In: ER (2014)

[32] Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a core of semantic knowledge. In: WWW (2007)

[33] Umbrich, J., Hausenblas, M., Hogan, A., Polleres, A., Decker, S.: Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources. In: LDOW (2010)

[34] UniProt, Consortium: The universal protein resource (UniProt). Nucleic acids research 36(suppl 1) (2008)

[35] Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., Van de Walle, R.: R&Wbase: git for triples. In: LDOW (2013)

[36] Völkel, M., Groza, T.: SemVersion: An RDF-based ontology versioning system. In: IADIS Int'l Conf. WWW/Internet. vol. 2006 (2006)

[37] Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Kondylakis, H., Motta, E., Plexousakis, D., Sabou, M.: Ontology evolution: a process-centric survey. KER 30(1) (2015)

Appendix

A. Single-version Queries

In this section, we present the SPARQL queries l1-l14, which were used for the evaluation of R43ples. For all queries, we used the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#` for rdf and `http://swat.cse.lehigh.edu/onto/univ-bench.owl\#` for ub. The SPARQL representation of the queries appears in Table 20. In order for the queries to be executed by R43ples, a specific named graph is used, i.e., GRAPH ?g, which is enriched with all the appropriate revision information collected by the R43ples API. The revision number is denoted in every query, i.e., REVISION ?rev. We should mention here that Evogen produces queries which are applied in the first, middle or the last version. For instance, in the 3 versions scenario, the first version is denoted by "2", the version in the middle is denoted by "3" and last version is denoted by "4". Number "1" denotes the initial "empty" version in R43ples.

B. Cross-version Queries

Each single-version query presented above can be converted into a cross-version query by rewriting it into a union which considers all the versions. For instance, consider the single version query l1:

```
SELECT ?X WHERE {
GRAPH ?g REVISION ?rev{
?X rdf:type ub:GraduateStudent.
?X ub:takesCourse ?X ub:takesCourse <http://www.
Department0.University0.edu/GraduateCourse0>.
} }
```

and a scenario in which we have three versions. Then, the cross-version query, l11, can be expressed as follows:

```
SELECT ?X WHERE {
{
GRAPH ?g REVISION "2"{
?X rdf:type ub:GraduateStudent.
?X ub:takesCourse ?X ub:takesCourse <http://www.
Department0.University0.edu/GraduateCourse0>.
} } UNION {
GRAPH ?g REVISION "3"{
?X rdf:type ub:GraduateStudent.
?X ub:takesCourse ?X ub:takesCourse <http://www.
Department0.University0.edu/GraduateCourse0>.
} } UNION {
GRAPH ?g REVISION "4"{
?X rdf:type ub:GraduateStudent.
?X ub:takesCourse ?X ub:takesCourse <http://www.
Department0.University0.edu/GraduateCourse0>.
}
}
}
```

In a similar manner, all single-version queries were transformed into their corresponding cross-version queries, taking into consideration the number of versions that appear in a dataset.

Table 20: Single Version Queries

Query	SPARQL

I1	SELECT ?X WHERE {
	GRAPH ?g REVISION ?rev{
	?X rdf:type ub:GraduateStudent.
	?X ub:takesCourse ?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>.
	}
	}

I2	SELECT ?X ?Y ?Z WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ub:GraduateStudent .
	?Y rdf:type ub:University .
	?Z rdf:type ub:Department .
	?X ub:memberOf ?Z .
	?Z ub:subOrganizationOf ?Y .
	?X ub:undergraduateDegreeFrom ?Y.
	}
	}

I3	SELECT ?X WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ub:Publication .
	?X ub:publicationAuthor <http://www.Department0.University0.edu/AssistantProfessor0>
	}
	}

I4	SELECT ?X ?Y1 ?Y2 ?Y3 WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?Professor.
	?X ub:worksFor <http://www.Department0.University0.edu> .
	?X ub:name ?Y1 .
	?X ub:emailAddress ?Y2 .
	?X ub:telephone ?Y3.
	}
	}

I5	SELECT ?X WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?Person.
	?X ub:memberOf <http://www.Department0.University0.edu>.
	}
	}

I6	SELECT ?X WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type [rdfs:subClassOf ?Student].
	}
	}

I7	SELECT ?X ?Y WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?Student.
	?Y rdf:type ub:Course .
	?X ub:takesCourse ?Y .
	<http://www.Department0.University0.edu/AssociateProfessor0> ub:teacherOf ?Y.
	}
	}

Query	SPARQL

l8	SELECT ?X ?Y ?Z WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?Student .
	?Y rdf:type ?Department .
	?X ub:memberOf ?Y .
	?Y ub:subOrganizationOf <http://www.University0.edu> .
	?X ub:emailAddress ?Z.
	} }

l9	SELECT ?X ?Y ?Z WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?Student .
	?Y rdf:type ?Faculty .
	?Z rdf:type ?Course .
	?X ub:advisor ?Y .
	?Y ub:teacherOf ?Z .
	?X ub:takesCourse ?Z.
	}
	}

l10	SELECT ?X WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?Student .
	?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>.
	}
	}

l11	SELECT ?X WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?ResearchGroup .
	?X ub:subOrganizationOf <http://www.University0.edu>.
	}
	}

l12	SELECT ?X ?Y WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?Chair .
	?Y rdf:type ?Department.
	?X ub:worksFor ?Y .
	?Y ub:subOrganizationOf <http://www.University0.edu>.
	}
	}

l13	SELECT ?X WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ?Person .
	<http://www.University0.edu> ub:hasAlumnus ?X.
	}
	}

l14	SELECT ?X WHERE {
	GRAPH ?g REVISION ?rev {
	?X rdf:type ub:UndergraduateStudent.
	}
	}