Versioning for Linked Data: Archiving Systems and Benchmarks

Vassilis Papakonstantinou[1], Giorgos Flouris[1], Irini Fundulaki[1], Kostas Stefanidis[2], and Giannis Roussakis[1]

[1] Institute of Computer Science-FORTH, Greece
[2] University of Tampere, Finland

Abstract. As LOD datasets are constantly evolving, both at schema and instance level, there is a need for systems that support efficiently storing and querying such evolving data. The aim of this paper is to describe the way that such RDF archiving systems could be evaluated by presenting the different benchmarks in the literature, as long as the state-of-the-art archiving systems that currently exist. In addition, the weak points of such benchmarks are mentioned, and a blueprint is provided on how we are willing to deal with them.

Keywords: RDF, Linked Data, Versioning, Archiving, SPARQL, Benchmarking

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 [4] has led several large-scale corporate, government, or even user-generated data from different domains (e.g., DBpedia [1], Freebase [5], YAGO [30]) 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 [16, 31]; 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 [34].

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, 17, 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], or supporting historical or cross-version queries [29] are necessary.

Thus, archiving systems not only need to 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 [29] (*cross-version queries*), queries that access the evolution history (delta) itself [12], as well as combinations of the above. Even though "pure" SPARQL does not support these types of queries, recent extensions [20, 13] are addressing this need.

To support these functionalities, various RDF archiving mechanisms and tools have been developed [34]. 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* [26, 7, 13, 15], the use of *temporal annotations* [24, 32], as well as *hybrid approaches* that combine the above techniques [29, 21, 32].

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 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 the weak and strong points of each one. 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 [12, 19].

This paper aims at:

- presenting these benchmarks including their features and characteristics,
- analyzing the most popular archiving systems, and
- revisiting the different strategies and approaches that are used for maintaining multiple versions.

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 the existing benchmarks along with their weaknesses. Finally, Section 5 concludes the paper.

2 About Versioning

In this section, we discuss the different archiving strategies implemented in the existing archiving systems for Linked Data and the different types of queries that have to be supported by such systems.

2.1 Archiving Strategies

In the literature, three alternative RDF archiving strategies have been proposed: *full materialization*, *delta-based* and approaches based on *annotated triples*, each with its own pros and cons. *Hybrid* strategies (that combine the above) have also been considered. Details on these strategies appear below.

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 [33]. Although there is no processing cost in order to store 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 matter 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.

Delta-based is an alternative approach 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, 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, 13, 32] or storing the latest (current) version and computing reverse deltas with respect to it [15].

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 determines the version(s) in which each triple existed in the dataset [32].

Hybrid Strategies 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 [29]) 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 encountered combination is the usage 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 determine its version [32].

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 [12, 29] to identify and categorize these types of queries. Our suggestion, which is a combination of them, is shown in Figure 1.

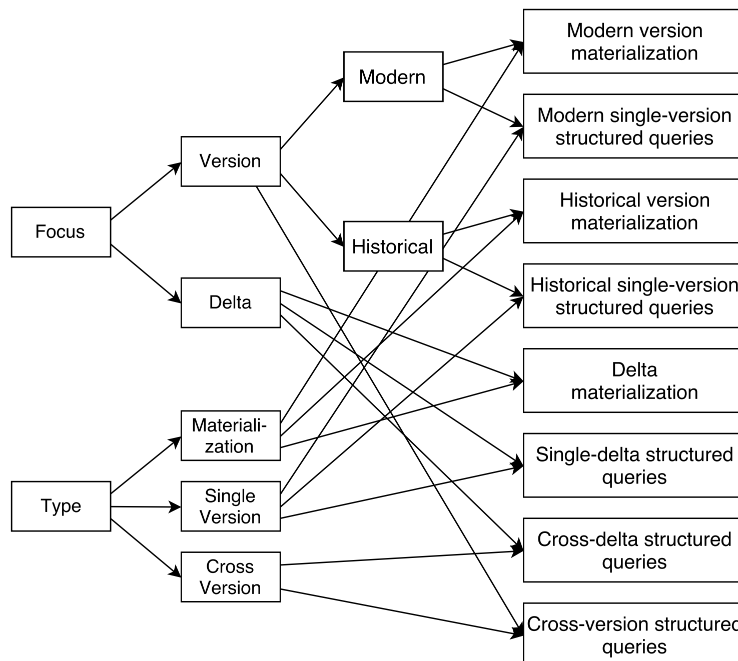

Fig. 1: Different types of queries according to their focus and type.

Firstly, queries are distinguished based on their focus, into *version* and *delta* queries. Version queries require data on the versions themselves, whereas delta queries require evolution data (delta). Version queries in their turn, can be further classified to *modern* and *historical*, depending on whether they request 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 intervals. In addition, queries can be further classified according to their type, to *materialization*, *single-version structured* and *cross-version structured* 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:

- *Modern version materialization* queries ask for a full current version to be retrieved. For instance, in a social network scenario, one may want to pose a query about the whole network graph at the current time.
- *Modern single-version structured* queries are performed in the current version of the data. For instance, a query that asks for the number of friends that a certain person has.
- *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 previous 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 one version to another.
- *Single-delta structured* queries are queries which are performed in two consecutive versions. One, for instance, could ask for the new friends that a person obtained between the last and the current version.
- *Cross-delta structured* queries must be satisfied on changes of several versions of the dataset. For example, a query that asks for the evolution of added/deleted friends of a person across versions.
- Finally, *Cross-version structured* queries must be evaluated on several versions of the dataset, thereby retrieving information residing in multiple 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 with their characteristics appears in Table 1. Such characteristics are the archiving strategy that each system/framework implement, their ability to answer SPARQL queries and to identify equivalent blank nodes across versions and finally their ability to support versioning concepts as committing, merging, branching etc.

System / Framework	**Archiving Policy**	**SPARQL support**	**Blank Nodes support**	**Versioning Concepts**
x-RDF-3X [24]	Annotated Tuples	✓	-	-
SemVersion [33]	Full Materialization	-	✓	✓
Cassidy et al. [7]	Delta Based	-	-	✓
R&Wbase [32]	Annotated Tuples	✓	✓	✓
R43ples [13]	Delta Based	✓	-	✓
TailR [21]	Hybrid Approach	-	-	-
Im et al. [15]	Delta Based	-	-	-
Memento [28]	Full Materialization	-	-	-

Table 1: An overview of RDF archiving systems and frameworks

3.1 x-RDF-3X

Neumann and Weikum [24] proposed an extension of RDF-3X system [23] that supports versioning, time-travel access 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. In order to get through this problem a write timestamp is assigned to each transaction once it starts updating the differential indexes (temporal small indexes that periodically merged to the main ones), and this timestamp is then used for all subsequent operations.

To support cross-version queries, snapshot isolation as long as the efficient retrieval of transactions order, a *transaction inventory* (as shown in Table 2) is proposed that 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.

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 all transactions information [24]

3.2 SemVersion

SemVersion [33] is inspired by the Concurrent Versioning System (CVS) [3] which was basically used in earlier years in software development to allow collaborative development of source code. 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 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* in order to be stored, 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.

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[3] (a version control system built to manage software source code) and its theory of patches.

[3] http://darcs.net/

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[4] backend for the RedLand store [2] was evaluated and it was shown that the current approach of managing versions adds a significant overhead compared to the raw RDF store. More specifically, query answering becomes four to six times slower and space consumption increased from two to four times.

3.4 R&Wbase

R&Wbase [32] 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, a number from a continuous sequence. More specifically, every new delta obtains an even number $2y$ that is larger than all preceding delta numbers. Even number $2y$ and odd number $2y+1$ are assigned as context values to the added and deleted triples respectively of the delta. Furthermore, the delta identifier $2y$ is used in order to store the delta's provenance metadata in triple format, using the PROV-O vocabulary [18]. This 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 through SPARQL queries, 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 apply the query to the set of returned versions. Finally, as a Git-like tool their approach supports versioning concepts like *branching* and *merging* of previously *committed* graphs.

3.5 R43ples

R43ples [13] 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).

[4] `https://www.mysql.com/`

Much like R&Wbase, R43ples supports the basic versioning concepts like *tagging*, *branching* and *merging*. To do so, it introduces an enhanced non-standard version of the SPARQL language by defining a set of new keywords (`REVISION`, `USER`, `MESSAGE`, `BRANCH` and `TAG`) to the reserved SPARQL keywords. So, the user is able to commit new changes such as the one below:

```
INSERT DATA INTO <graph> REVISION "X"
```

or query a specific version (revision) of the data

```
SELECT ?x FROM <graph> REVISION "X"
```

R43ples follows the *delta-based* approach for storing versions. In particular, each version is represented by a temporary graph which is connected, by using an extended version of PROV-O ontology [18] called Revision Management Ontology (RMO), to two additional named graphs corresponding to the delta's *ADD* and *DELETE* sets. 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. Queries on datasets with more than a few thousand triples take longer than most users are willing to wait.

3.6 TailR

TailR [21] is a platform implemented as a Python web application, for preserving the history of arbitrary linked datasets over time. It follows the *hybrid approach* for storing the data while 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, that 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[5].

To evaluate their platform they ran experiments for quantifying the Push and Memento API [27] response times as well as the growth of the required

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

storage space while storing more versions. For their experiments they used a random sample of 100K resources selected from each version of DBpedia [1] 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.

3.7 A version management framework for RDF triple stores

Im et al. [15] 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. In order to improve the performance of cross-delta queries, at the cost of increasing space overheads, they introduce *aggregated deltas*, which associate the latest version with each of the previous ones (not only the last one). 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 Uniprot dataset [10] versions v1-v9 on top of an implementation in Oracle 11g Enterprise edition[6]. 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 performance. As expected, their approach is (a) less efficient than the sequential deltas but outperforms the full materialization approach regarding storage space and deltas computation time and (b) highly outperforms the sequential deltas regarding version re-construction. In particular, while construction time in the sequential delta is proportional to the number of past versions that must be considered, the aggregated delta can compute any version almost at constant time. 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.

3.8 Memento

Memento [27] 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 the memento was captured is called *Memento-datetime*.

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

Memento can also be adopted in the context of linked data [28] as it had been used for providing access to prior versions of DBpedia. To do so, versions of DBpedia stored in a MySQL database as complete snapshots, so the full materialization approach is followed, and served through a Memento endpoint.

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 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, the configurability and scalability may refer to the number of versions that a data generator will produce, the size of each version, the number of changes from version to version etc.

In addition, the benchmark should be *portable* 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 as 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 the best of our knowledge, there have been only two proposed benchmarks for RDF archiving systems in the literature. These are described in detail below.

4.1 BEAR

Fernandez et al. [12, 11] 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 such RDF archiving strategies, they provide a set of directions that must be followed when evaluating the efficiency of RDF

archiving systems. First, the benchmark should be agnostic with respect to the used archiving strategy in order for the comparison to be fair. Next, 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 basis for comparing the different archiving strategies, they introduce the four following features that describe the dataset configuration. Although such features could serve in the process of automatic generation of synthetic data, this is not addressed in their approach.

- *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 *data growth* determines how its size changes from one version to another.
- *Data static core* contains the triples that exist in all dataset's versions.
- *Total version-oblivious triples* computes the total number of different triples in an archive independently of the timestamp and finally
- *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, as the results of a query can highly vary in different versions. For example by selecting queries with similar result cardinality and selectivity, should 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 the following six different categories of queries (note that these categories are similar to the ones we discussed previously (Section 2.2) and have been used as a source of inspiration for our categorization):

- *Version materialization*: refers to the retrieval of a specific version (*modern* or *historical version* materialization queries).
- *Delta materialization*: provides the different results of a query between two versions.
- *Version query*: provides the results of a query annotated with the version label (*modern* or *historical single version* structured queries).
- *Change checking*: it answers with a boolean value to state if there is a change between two versions (*single delta structured* queries).
- *Cross-version join*: it serves the join between two different queries in two different versions (*cross-version structured* queries).
- *Change materialization*: reports the point in which a query evaluated differently.

4.2 EvoGen

Meimaris and Papastefanatos have proposed the *EvoGen Benchmark Suite* [19], a generator for evolving RDF data, used for benchmarking versioning and change

detection approaches. EvoGen is based on the LUBM [14] generator, 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: (a) the data generation process; (b) the context of the tested application; and, (c) the query workload, as required by the nature of the evolving data.

EvoGen, acts like a *Benchmark Generator*, is extensible and highly configurable, in terms of the number of generated versions, or the number of changes occurring from version to version. Similarly, the query workload is generated adaptively to such configurable 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 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 positive or negative shifts change monotonically a dataset, and is used in order 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 represents the change of the ontology with respect to the total number of classes and is actually the ratio of added classes to the total classes in the original dataset. The *schema variation* parameter ranges from 0 to 1 and quantifies the percentage of different characteristic sets [22], with respect to the total number of possible characteristic sets that will be created for each inserted class. In EvoGen, the user is able to choose the output format of generated data by allowing him to request fully materialized versions or deltas; this allows supporting (and testing) systems employing different archiving strategies.

Finally, regarding the query workload generation, based on the previous generated data and their characteristics, the following six types of queries are generated:

- *Retrieval of a diachronic dataset*: a query asking for all the versions of a dataset.
- *Retrieval of a specific version* (*modern or historical version materialization queries*).
- *Snapshot queries* on the data i.e., queries affecting 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* i.e., queries accessing the deltas (*delta materialization or single-delta structured queries*).

– *Mixed queries* which use sub-queries from previously described type of queries (*hybrid queries*).

4.3 Discussion

The previously described benchmarks (Sections 4.1 and 4.2) although they provide a detailed theoretical analysis of the features that are useful in the process of designing a benchmark (especially BEAR), they both lack a detailed study and definition of the queries that the query workload is composed of. To do so, first, all problems that archiving systems are asked to solve, have to be specified. For example, the existence of blank nodes in the data makes the problem of finding the delta between two versions more complicated. According to the RDF semantics blank nodes can only be identified in the context of one dataset, hence comparing blank nodes to choose what to store in the delta based approach is a difficult task.

In addition, a more detailed analysis must be done, in order to identify the technical difficulties when answering queries on top of more than one versions. More specifically, we have to define the *choke points* [6], the technical challenges whose resolution will significantly improve the performance of a versioning system. One such a choke point would be the parallel execution of some types of queries. For instance, an archiving system would benefit from a query optimizer that decides to parallel reconstruct the versions required to answer a cross-version query.

5 Summary

In this paper, we sketched the current state-of-the-art for the problem of managing and benchmarking evolving RDF data. More specifically, we presented the most popular archiving tools that currently exist, and described the different types of queries that such tools should ideally support. In addition, we described the basic strategies that archiving tools follow for storing (and/or providing access to) the different versions (*full materialization, delta-based, annotated triples, hybrid strategies*). Regarding archiving benchmarks, we described some basic requirements and presented the only two archiving benchmarks that currently exist (to the best of our knowledge). Finally, we reported, what is missing from them and how such a gap can be filled, which is under construction from our side.

Acknowledgments

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

References

1. 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, pp. 722–735. Springer (2007)
2. Beckett, D.: The design and implementation of the redland rdf application framework. Computer Networks 39(5), 577–588 (2002)
3. Berliner, B., et al.: CVS II: Parallelizing software development. In: USENIX Winter 1990 Technical Conference. vol. 341, p. 352 (1990)
4. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. Semantic Services, Interoperability and Web Applications: Emerging Concepts pp. 205–227 (2009)
5. Bollacker, K., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: a collaboratively created graph database for structuring human knowledge. In: ACM-SIGMOD. pp. 1247–1250. ACM (2008)
6. Boncz, P., Neumann, T., Erling, O.: TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In: TPC-TC. pp. 61–76. Springer (2013)
7. Cassidy, S., Ballantine, J.: Version Control for RDF Triple Stores. ICSOFT (ISDM/EHST/DC) 7, 5–12 (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. Consortium, U., et al.: The universal protein resource (uniprot). Nucleic acids research 36(suppl 1), D190–D195 (2008)
11. Fernandez Garcia, J.D., Umbrich, J., Knuth, M., Polleres, A.: Evaluating Query and Storage Strategies for RDF Archives. In: SEMANTiCS (2016, forthcoming)
12. 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)
13. Graube, M., Hensel, S., Urbas, L.: R43ples: Revisions for triples. LDQ (2014)
14. 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), 158–182 (2005)
15. Im, D.H., Lee, S.W., Kim, H.J.: A version management framework for RDF triple stores. Int'l Journal of Software Engineering and Knowledge Engineering 22(01), 85–106 (2012)
16. Käfer, T., Abdelrahman, A., Umbrich, J., O'Byrne, P., Hogan, A.: Observing linked data dynamics. In: ESWC. pp. 213–227 (2013)
17. Kondylakis, H., Plexousakis, D.: Ontology evolution without tears. J. Web Sem. 19, 42–58 (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: Int'l Conf. on Semantic Systems. pp. 57–64. ACM (2015)
22. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: ICDE. pp. 984–994. IEEE (2011)
23. Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for rdf. Proceedings of the VLDB Endowment 1(1), 647–659 (2008)
24. Neumann, T., Weikum, G.: x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. VLDB Endowment 3(1-2), 256–263 (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), 1 (2013)
27. 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)
28. 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)
29. Stefanidis, K., Chrysakis, I., Flouris, G.: On designing archiving policies for evolving RDF datasets on the Web. In: Int'l Conf. on Conceptual Modeling. pp. 43–56. Springer (2014)
30. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a core of semantic knowledge. In: WWW. pp. 697–706. ACM (2007)
31. Umbrich, J., Hausenblas, M., Hogan, A., Polleres, A., Decker, S.: Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources. In: LDOW (2010)
32. Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., Van de Walle, R.: R&Wbase: git for triples. In: LDOW (2013)
33. Völkel, M., Groza, T.: SemVersion: An RDF-based ontology versioning system. In: IADIS Int'l Conf. WWW/Internet. vol. 2006, p. 44 (2006)
34. Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Kondylakis, H., Motta, E., Plexousakis, D., Sabou, M.: Ontology evolution: a process-centric survey. Knowledge Eng. Review 30(1), 45–75 (2015)