SPBv: Benchmarking Linked Data Archiving Systems

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 Linked Open Data (LOD) datasets are constantly evolving, both at schema and instance level, there is a need for systems that efficiently support storing and querying of such evolving data. However, there is a limited number of such systems and even fewer benchmarks that test their performance. In this paper, we describe in detail the first version of the *SPBv* benchmark developed in the context of the HOBBIT EU H2020 project. *SPBv* aims to test the ability of archiving systems to efficiently manage evolving Linked Data datasets and queries evaluated across multiple versions of these datasets. We discuss the benchmark data generator and the query workload, and we describe a set of experiments we conducted with Virtuoso and R43ples systems.

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

1 Introduction

A key step towards abolishing the barriers to the adoption and deployment of Big Data is to provide companies with open benchmarking reports that allow them to assess the fitness of existing solutions for their purposes.

There exist a number of storage benchmarks that test the ability of Linked Data systems to store and query data in an efficient way without addressing the management of data versions. To the best of our knowledge, only a limited number of systems (mostly academic) and benchmarks exist for handling evolving data, and testing the proposed solutions respectively.

However, the existence of such systems and benchmarks is of utmost importance, as dynamicity is an indispensable part of the Linked Open Data (LOD) initiative [1, 2]. In particular, 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 [3]. 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.

In this paper, we discuss the *SPBv* benchmark developed in the context of HOBBIT project[3] for testing the ability of archiving systems to efficiently manage evolving datasets and queries, evaluated across the multiple versions of said datasets. The benchmark is based on Linked Data Benchmark Council's (LDBC)[4] Semantic Publishing Benchmark (SPB). It leverages the scenario of the BBC media organisation, which makes heavy use of Linked Data Technologies, such as RDF and SPARQL. We extend SPB to produce *SPBv*, a versioning benchmark that is not tailored to any strategy or system. We followed a choke point-based design [4] for the benchmark, where we extend the SPB queries with features that stress the systems under test.

The outline of the paper is the following. In Section 2, we discuss the state of the art of archiving strategies, benchmarks and query types. We present HOBBIT's versioning benchmark *SPBv* in Section 3, and experiments are provided in Section 4. Finally, Section 5 concludes and outlines future work.

2 State of the Art

This section presents the state of the art of archiving (a) strategies (Section 2.1), (b) different types of queries (Section 2.2), and (c) benchmarks (Section 2.3). A detailed presentaton is provided in [5].

2.1 Archiving Strategies

Three alternative RDF archiving strategies have been proposed in the literature: *full materialization*, *delta-based*, and *annotated triples* approaches, each with its own advantages and disadvantages. *Hybrid* strategies (combining the above) have also been considered. Next, we provide a description of those approaches.

Full Materialization was the first and most widely used approach for storing different versions of datasets. In this strategy, all different versions of an evolving dataset are stored explicitly in the archive [6].

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/next version (also known as the *delta*) has to be kept. There are various alternatives in the literature, such as storing the first version and computing the deltas according to it [7–9] or storing the latest (current) version and computing reverse deltas with respect to it [10, 11].

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* [12]. 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 [9].

[3] `https://project-hobbit.eu/`

[4] `ldbc.council.org`

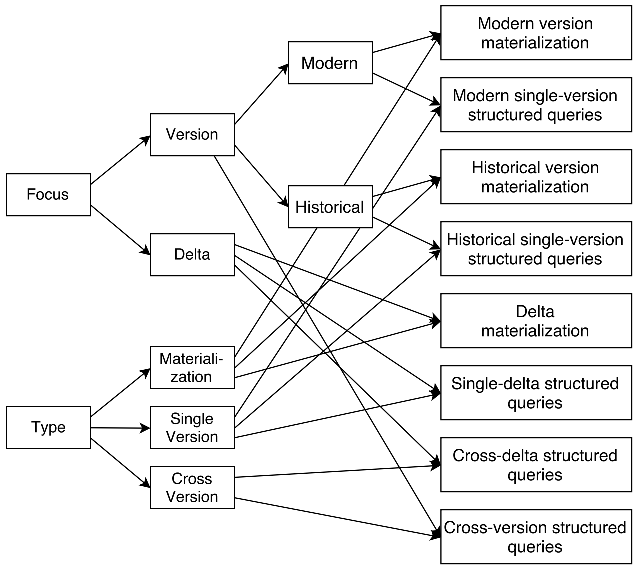

Focus
Version
Delta
Modern
Historical
Type
Materiali-zation
Single Version
Cross Version
Modern version materialization
Modern single-version structured queries
Historical version materialization
Historical single-version structured queries
Delta materialization
Single-delta structured queries
Cross-delta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

Fig. 1: Different queries organized by focus and type.

Hybrid Approaches [13] aim at combining these strategies 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 [14]. 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 [9].

2.2 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 [15, 13, 16, 17] to identify and categorize these types of queries. Our suggestion, which is a combination of such efforts and was presented in detail by Papakonstantinou et al. [5], is shown in Figure 1.

Firstly, queries are distinguished by focus (i.e., target), 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, the latter categorization cannot be applied to delta queries, as they refer to time changes between versions (i.e., intervals), which have no specific characteristics that are related to time.

In addition, queries can be further classified according to type, to *materialization*, *single-version* and *cross-version* queries. Materialization queries essentially request the entire respective data (a full version, or a 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:

– **QT1 - 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.
– **QT2 - 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 at the present time.
– **QT3 - 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.
– **QT4 - 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.
– **QT5 - 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.
– **QT6 - Single-delta structured** queries 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.
– **QT7 - 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.
– **QT8 - 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.

2.3 Benchmarks for Evolving RDF Data

A benchmark is a set of tests against which the performance of a system is evaluated. A benchmark helps computer systems to compare and assess their performance in order to become more efficient and competitive. To our knowledge, there have been only two proposed benchmarks for systems handling evolving RDF data in the literature, which are described below (see [5] for more details).

BEAR [18, 15] benchmark is an implementation and evaluation of a set of operators that cover crucial aspects of querying and versioning Semantic Web data for the three archiving strategies (*Full Materialization*, *Delta-Based* and *Annotated*

Triples) described in Section 2.1. As a basis for comparing the different strategies, the BEAR benchmark introduces some features that describe the dataset configuration. Such features are I) the *data dynamicity* that measures the number of changes between versions, II) the *data static core* that contains the triples that exist in all dataset versions, III) the *total version-oblivious triples* that compute the total number of different triples in an archive and finally IV) the *RDF vocabulary* that 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 are considered to guarantee that potential retrieval differences in response times are attributed to the archiving strategy. In order to be able to judge the different systems, BEAR introduces various categories of queries, which are similar to the ones we discuss in Section 2.2. 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 for designing a benchmark, it lacks configurability and scalability as its data workload is composed of a static, non configurable dataset.

EVOGEN [17] is a generator for evolving RDF data that is used for benchmarking archiving and change detection systems. EVOGEN is based on the LUBM generator [19], by extending its schema with 10 RDF classes and 19 properties to support schema evolution. Its 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. The query workload produced by EVOGEN leverages the 14 LUBM queries, appropriately adapted to apply for evolving versions. In particular, the following six types of queries are generated: I) *Retrieval of a diachronic dataset*, II) *Retrieval of a specific version* (QT1, QT3 from our categorization), III) *Snapshot queries* (QT2, QT4), IV) *Longitudinal (temporal) queries* (QT8), V) *Queries on changes* (QT5, QT6), VI) *Mixed queries*. Regarding the data generation, in EVOGEN, the user is able to choose the output format of the generated data (e.g., fully materialized versions or deltas); this allows supporting (and testing) systems employing different archiving strategies.

EVOGEN is a more complete benchmark, as it is a strategy-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 the benchmarked systems store meta data about underlying deltas (addition/deletion of classes, addition/deletion of class instances etc.) in order to be answered.

Moreover, to successfully answer 11 of the 14 original LUBM queries, the benchmarked systems must support RDFS reasoning (forward or backward).

3 Versioning Benchmark

In this section, we present the versioning benchmark *SPBv*, that we developed in the context of the HOBBIT project. The full source code of the benchmark can be found in the HOBBIT github page[5]. Figure 2 presents an overview of the HOBBIT platform components. The orange ones are those which *SPBv* is built on, and are described in the following sections.

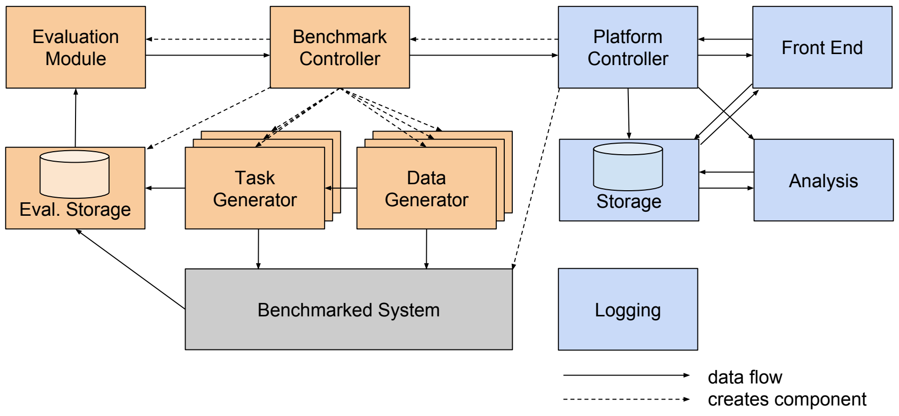


Fig. 2: Overview of the HOBBIT platform components

As mentioned in Section 1, the benchmark is based upon LDBC's Semantic Publishing Benchmark for RDF database engines inspired by the Media/Publishing industry, particularly by the BBC's Dynamic Semantic Publishing approach. The application scenario considers a media or a publishing organization that deals with a large volume of streaming content, namely news, articles or "media assets". This content is enriched with metadata that describes it and is linked to reference knowledge, such as taxonomies and databases that include relevant concepts, entities and factual information. This metadata allows publishers to efficiently retrieve relevant content, according to their business models. For instance, some news publishers, like BBC, can use it to maintain rich and interactive web presence for their content, while others, e.g. news agencies, would be able to provide better defined content feeds.

3.1 Choke Point-based benchmark design

"Choke points" are those technological challenges underlying a benchmark, whose resolution will significantly improve the performance of a product [4]. So, a benchmark can be characterized as valuable if its workload stresses those choke

[5] `https://github.com/hobbit-project/versioning-benchmark`

points that systems should manage. In *SPBv*, the following choke points are considered:

- **CP1: Storage Space** tests the ability of the systems to efficiently handle the storage space growth as new versions are stored.
- **CP2: Partial Version Reconstruction** tests the ability of the systems to only reconstruct the part of the version that is required from the targeted query in order to be answered, instead of the whole version.
- **CP3: Parallel Version Reconstruction** tests the ability of the systems following the delta-based or hybrid archiving strategies to reconstruct in parallel multiple versions when a query asks for information from more than one versions.
- **CP4: Parallel Delta Computation** tests the ability to compute in parallel multiple deltas when a query asks for information from more than one delta.
- **CP5: On Delta Evaluation** tests the ability of the systems that follow the delta-based or hybrid archiving strategies, to evaluate queries on top of deltas when requested by the query (delta-based queries).

3.2 Data Generation

The data generator of *SPBv* extends the data generator of SPB that was described by Kotsev et al. [20]. SPB's data generator uses seven *core* and three *domain* RDF ontologies (see Table 1) for the data production. Also, a set of reference datasets are employed by the data generator to produce the data of interest.

Domain			Core
`creativework 0.9`	`company 1.4`	`tagging 1.0`	`cnews-1.2`
`coreconcepts 0.6`	`CMS 1.2`	`provenance 1.1`	`sport 2.3`
`person 0.2`			`curriculum 4.0`

Table 1: BBC Core & Domain Ontologies

The SPB data generator produces RDF descriptions of *creative works* that are valid instances of the BBC creative work core ontology. A creative work can be defined as metadata about a real entity (or entities) that exist in reference datasets. A creative work collects all RDF descriptions of creative works created by the publisher's editorial team. A creative work has a number of properties such as *title, shortTitle, description, dateCreated, audience* and *format*; it has a *category* and can be *about* or *mention* any entity from the reference datasets. That way a creative work provides metadata (facts) about one or several entities and defines relations between them. SPB's data generator models three types of relations in the data, as described later and shown in Figure 3.

Clustering of data. The clustering effect is produced by generating *creative works* about a *single entity from reference* datasets and for a *fixed period of time.*

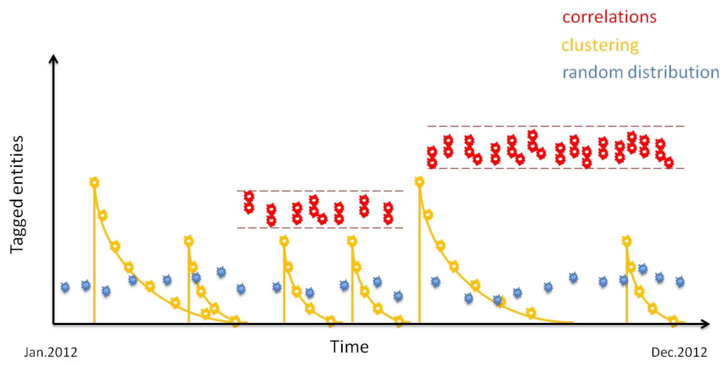


Fig. 3: Data generator and types of produced models in generated data.

The number of creative works, referencing an entity, starts with a high peak at the beginning of the clustering period and follows a smooth decay towards its end. The data generator produces major and minor clusters with sizes (i.e., number of creative works) of different magnitude.

Correlations of entities. The correlation effect is produced by generating *creative works about two or three entities from reference data* in a *fixed period of time*. Each entity is *tagged by creative works* solely at the beginning and end of the correlation period, whereas in the middle of this period the same creative work tags both of them.

Random tagging of entities. Random data distributions are defined with a *bias towards popular entities* created when the *tagging* is performed, that is when values are assigned to *about* and *mentions* creative work properties. This is achieved by *randomly* selecting 5% of all the resources from reference data and marking them as *popular* while the remaining ones are marked as *regular*. When creating creative works, 30% of them are tagged with *randomly selected* popular resources and the remaining 70% are linked to the *regular* ones.

Creative works, as journalistic assets, are highly dynamic, since the world of online journalism is constantly evolving through time. Every day plenty of new "creative works" are published, while the already published ones, often change. As a result, editors need to keep track of changes occurred as times goes by. This is the behaviour that the data generator of *SPBv* tries to simulate, by extending the generator of SPB in such a way that generated data is stored in different versions according to their creation date (creative work's creation date).

The following parameters can be set to configure *SPBv*'s data generator.

1. **Generator seed**: used to set the random seed for the data generator. This seed is used to control all random data generation happening in *SPBv*.
2. **Generated data format**: serialization format for generated synthetic data. Available options are: TriG, TriX, N-Triples, N-Quads, N3, RDF/XML, RDF/JSON and Turtle.

3. **Seed year**: defines a seed year that will be used as starting point for generating the creative works date properties.
4. **Substitution parameters amount**: The amount of queries that will be produced for each query type.
5. **Generation period**: the period of generated data in years.
6. **Size**: defines the size of generated synthetic data in triples produced by the data generator.
7. **Number of versions**: defines the total number of versions in which generated data will be stored.

In order for the data generator to be able to *tag* creative works with *entities* it is necessary for such entities to be extracted from the previously described reference datasets. To do so, all instances from the different domain ontologies that exist in the reference datasets are identified. Such identification process consists of the execution of queries that collect data about the stored entities. As this extraction would burden the benchmarking procedure we did it offline once and stored the result entities in files, so that they can be used as input by the data generator when tagging a generated creative work.

By having such exported entities as input, the data generator proceeds as follows:

– Retrieves entities, DBpedia locations and Geonames locations from the appropriate files.
– Selects the popular and regular entities from the previous set of retrieved instances.
– Adjusts the number of major/minor events and number of correlations, in order to let the ratio of the three types of modelled data (clusterings, correlations, random) to be 33%, 33% and 33%, respectively.
– Major and minor events and correlations are distributed to all available data generator instances. This is an indispensable step, as each instance has to produce the whole event/correlation in order for the event to be valid.
– Each data generator instance produces the creative works according to the three strategies previously discussed and sends the generated data to the system that will be benchmarked, as shown in Figure 2.
– One of the data generator instances generates the SPARQL queries based on the already generated data, and sends them to the Task Generator component, as shown in Figure 2.
– The same instance that previously generated the SPARQL queries, computes the Gold Standard, for comparison against the results of the benchmarked system. In particular, the generated data loaded into VIRTUOSO triplestore and the SPARQL queries evaluated on top of them. The results that such queries return, compose the gold standard which sent to the Task Generator component.

3.3 Task Generation

As shown in Figure 2, the Task Generator of *SPBv* (which may consist of several instances running in parallel) is responsible for sending the gold standard, previously received from the Data Generator, to the Evaluation Storage component. Moreover, its main job is to provide all the tasks (that should be solved by the system) to the benchmarked system which, in turn, sends the results to the Evaluation Storage. In detail, there are three types of tasks:

- **Ingestion tasks**, which trigger the system to report the time required for loading a new version.
- **Storage Space task** prompts the system to report the total storage space overhead for storing the different versioned datasets.
- **Query Performance tasks** are used to test query performance. For each versioning query type (QT1-QT8 in Section 2.2), a set of SPARQL queries is generated. The generated SPARQL query uses templates for its parameterization, so by using parameter substitution, a set of similar queries of the same type is generated. The amount of the different substitution parameters, determines the amount of different queries of the same type that are provided, and is given in the configuration of the benchmark. Given that there is neither a standard language, nor an official SPARQL extension for querying RDF evolving data, the definition of our query templates assumed that each version was stored in its own named graph. Each benchmarked system should rewrite such queries in order to be compatible with the query language it implements. [21] provides a detailed description of all query tasks that were produced, including query type, text representation and related choke points.

3.4 Evaluation Module

Finally, as we can see in Figure 2, the Evaluation Storage sends the gold standard and the results reported by the benchmarked system to the Evaluation Module that is responsible for evaluating the performance of the system under test. Analogous to task types, there are three performance metrics that can be used to evaluate such performance:

1. The *space* required to *store* the different versioned datasets. Such a metric is essential to understand whether a system can choose the best archiving strategy (as explained in Section 2.1) for storing the versions or to identify the benefits of systems using compression techniques for storing their data.
2. The *time* that a system needs for *storing* a new version is measured. By doing so, the possible overhead of complex computations, such as delta computation, during data ingestion can be quantified.
3. The *time required to answer* a query is measured. In particular, we measure the average execution time of all queries of each different query type as described in Section 2.2.

In order to evaluate the success of systems to cope with the previously described metrics, we define the following *Key Performance Indicators (KPIs)*:

- **Initial version ingestion speed** (in triples per second): the total triples that can be loaded per second for the dataset's initial version. We distinguish this from the ingestion speed of the other versions because the loading of the initial version greatly differs in relation to the loading of the following ones, where different underlying procedures as, computing deltas, reconstructing versions, storing duplicated information between versions, may take place.

- **Applied changes speed** (in changes per second): tries to quantify the overhead of such underlying procedures that take place when a set of changes are applied to a previous version. To do so, this KPI measures the average number of changes that could be stored by the benchmarked systems per second after the loading of all new versions.
- **Storage cost** (in KBs): This KPI measures the total storage space required to store all versions.
- **Average Query Execution Time** (in ms): The average execution time, in milliseconds for all different query types, as those described in Section 2.2.

4 Experiments

In order to test the benchmark's implementation on top of the HOBBIT platform, from the different archiving systems described by Papakonstantinou et al. [5], we managed to conduct experiments only for R43ples (Revision for triples) [8], which uses Jena TDB as an underlying storage/querying layer. Also, for having a baseline system we decided to implement the *full materialization* archiving strategy (see Section 2.1), by assuming that each version is represented in its own named graph, to a triple store that cannot handle evolving data. Such triplestore was the OpenLink Virtuoso Opensource[6]

For our experiments we produced four datasets of different sizes that correspond to around 100K, 500K, 1M and 5M triples. The generated data follows the three models described in Section 3.2 starting from January 1st 2016 and for a duration of 1 year. According to their creation date, they were divided in 5 different versions of equal time intervals of around 2 and half months. 5 different queries were produced per query type, and the average execution time of these queries was computed. For fairness, we run three experiments per dataset size and computed the average values for all reported results. The experiment timeout was set to 30 minutes for both R43ples and Virtuoso systems.

Regarding the *full materialization* strategy that we implemented on top of Virtuoso we report the following results:

In the left histogram of Figure 4 we can see for all datasets the *initial version ingestion speed* for Virtuoso triple store. For the ingestion of new triples we used the bulk loading process offered, with 12 RDF loaders so that we can parallelize the data load and hence maximize loading speed.

As we can see, the speed ranges from 35K to 185K triples per second and increases as the dataset size, and consequently the size of its initial version, increases. This is an expected result, as Virtuoso bulk loads files containing much more triples, as the dataset size increases. The same holds for the *applied changes speed*, shown in the right side of the same figure, which increases from 10K to 115K changes per second. We can observe here that the time required to

[6] `https://virtuoso.openlinksw.com/`

perform the changes is larger than the time required to insert initially the triples in the archive. This is an overhead of the chosen archiving strategy i.e., *full materialization* (Section 2.1). Recall that the unchanged information between versions is duplicated when a new version is coming, so the time required for applying the changes of a new version is significantly increased as it includes the loading of data from previous versions.

In Figure 5 we can see the storage space required for storing the data for all different datasets. For measuring such space we measured the size of ”virtuoso.db” file before and after the loading of all versions' triples. The space requirements expectantly increase as the total number of triples increases, from 30 MB to 1450 MB. This significant overhead on storage space is due to the archiving strategy used (i.e., Full Materialization).

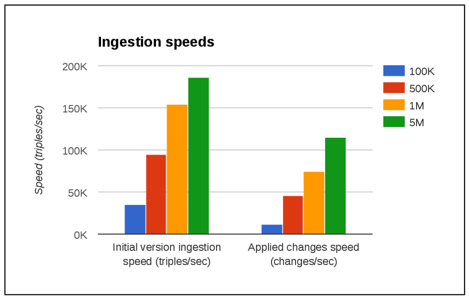


Fig. 4: Ingestion speeds

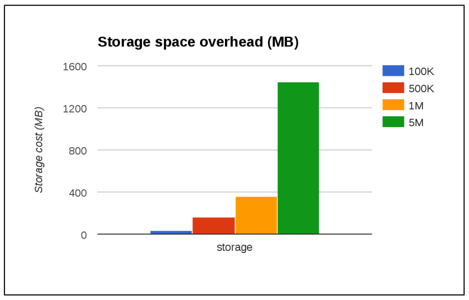


Fig. 5: Storage space overhead

In Figures 6, 7, 8, 9 and 10 we present the average execution time (in ms) for the five queries of each versioning query type, and for each dataset size.

In Figure 6 we can see the time required for materializing I) the modern (current) version; II) an historical (past) one; or III) the difference between two versions (delta). In the left and middle histograms the times required for materializing the modern and a historical version are presented respectively. As expected, the execution time increases as the dataset size increases and the time required for materializing a historical version is much shorter than the modern one, as it contains less triples. In both cases, although we do not have a system implementing the delta-based approach, we observe that execution times are short enough, as all the versions are already materialized in the triple store. For the 5M triples dataset VIRTUOSO failed to execute the modern and historical materialization queries, as it has hard-coded limits for the result size of the queries – upper limit 1.048.576 results. In the right side of the same Figure we can see the time required for materializing a delta. Since deltas have to be computed on the fly when the queries are evaluated, we see a significant increase in the time required for evaluation.

In Figures 7, 8 , 9 and 10 we can see the execution times for all types of structured queries. In most of the cases, similarly to materialization queries, the

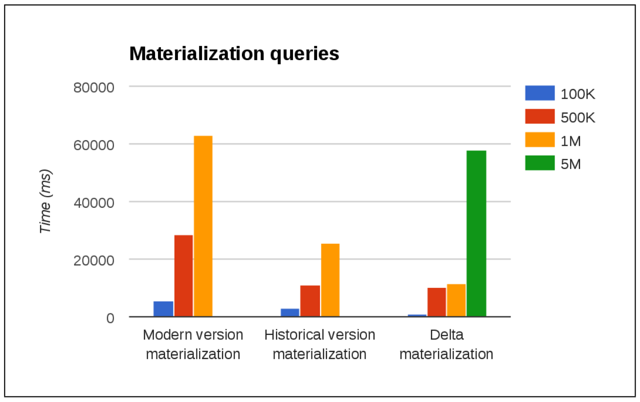


Fig. 6: Execution times for materialization queries

execution time increases as the number of triples increases. Although someone would expect that delta-based queries were to be slower than the version-based ones, as deltas have to be computed on the fly, this does not seem to be the case. This is happening as the version-based queries are much harder regarding query evaluation than the delta-based ones. According to the performance of version-based structured queries which, as shown in the Appendix of [21], are all of the same form, we observe that the oldest the version queried, the shorter execution time we have. This is an expected result, as the number of triples from which a version is composed of, is decreased as the version becomes older.

Regarding the R43ples system, we only managed to run experiments for the first dataset, composed of 100K triples, so we do not report the results graphically. For the remaining datasets the experiment time exceed the timeout of 30 minutes.

In Figure 11 we can see the results after running the three experiments for the dataset of 100K triples, as shown in the GUI of the HOBBIT platform. At first, we can see that R43ples failed to execute all queries on deltas (delta materialization, single/cross-delta structured queries). Such queries are composed of MINUS operations between revisions and this does not seem to be supported by the system.

Regarding the speeds for ingesting the initial version and applying new changes, we can see that changes were applied slower than the initial version is loaded. This is an expected result as the version that is kept materialized is the current one, so for every new delta, the current version has to be computed. Compared to Virtuoso, R43ples is 1 order of magnitude slower, but the changes' speed is much closer to the ingestion speed than the corresponding speeds for Virtuoso, as in R43ples the unchanged information between versions is not duplicated (Delta-based archiving strategy).

To quantify the storage space overhead, we measured the size of the directory (ptriplestore.url property in the config file) where Jena TDB stores all data,

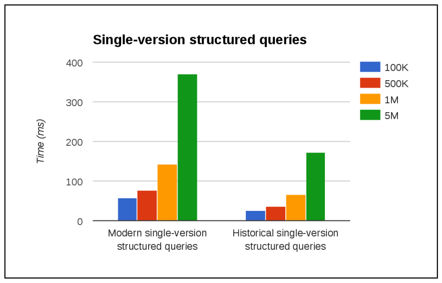


Fig. 7: Execution times for single version structured queries

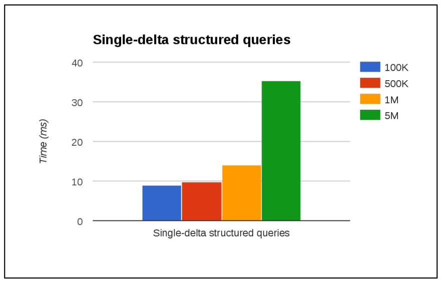


Fig. 8: Execution times for single delta structured queries

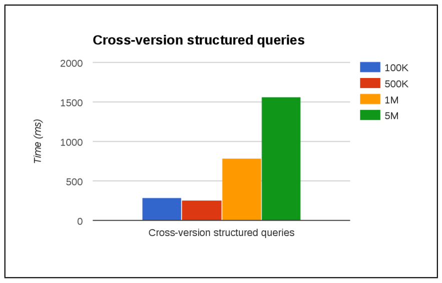


Fig. 9: Execution times for cross-version structured queries

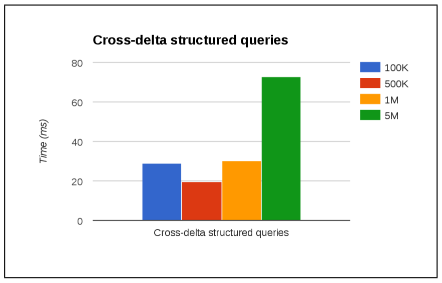
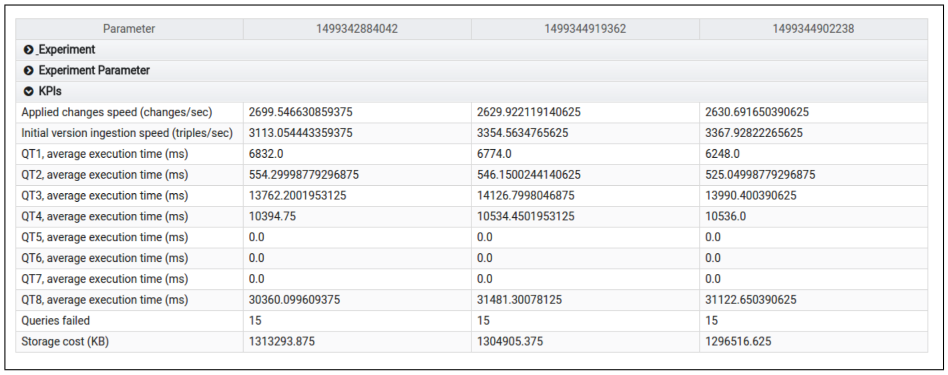


Fig. 10: Execution times for cross-delta structured queries

before and after the loading of all versions. As we can see the overhead is extremely high, even for the small dataset of 100K triples. Someone would expect R43ples to outperform Virtuoso as in Virtuoso we implemented the Full Materialization strategy, but that seems not to be the case, as the underlying storage strategies of Virtuoso and Jena TDB (that is used as triplestore from R43ples) seem to be very different.

Next, regarding the query execution times, we can see that in most cases R43ples queries take too much time to be executed. More specifically, for materializing the current version (QT1) R43ples requires similar time compared to Virtuoso and this is something that we expected, as the current version is kept materialized just like in Virtuoso, where all versions are materialized. This is not happening when a historical version was retrieved (QT3), as R43ples is 1 order of magnitude slower than Virtuoso. This is also an expected result as R43ples needs to reconstruct the queried version on-the-fly.

Concerning the single-version structured queries, R43ples also answers them much slower than Virtuoso, as it requires 1 and 3 orders of magnitude more time for answering the QT2 (modern) and QT4 (historical) query types respectively. The reason why the historical single-version structured queries were answered much slower (20 times) than the corresponding modern version ones is also the need for on-the-fly queried version reconstruction. The same holds for the answering of cross-version queries (QT8) by R43ples where the time that is required is 2 orders of magnitude higher than the corresponding one for Virtuoso.



Parameter	1499342884042	1499344919362	1499344902238
_Experiment			
Experiment Parameter			
KPIs			
Applied changes speed (changes/sec)	2699.546630859375	2629.922119140625	2630.691650390625
Initial version ingestion speed (triples/sec)	3113.054443359375	3354.5634765625	3367.92822265625
QT1, average execution time (ms)	6832.0	6774.0	6248.0
QT2, average execution time (ms)	554.29998779296875	546.1500244140625	525.04998779296875
QT3, average execution time (ms)	13762.2001953125	14126.7998046875	13990.400390625
QT4, average execution time (ms)	10394.75	10534.4501953125	10536.0
QT5, average execution time (ms)	0.0	0.0	0.0
QT6, average execution time (ms)	0.0	0.0	0.0
QT7, average execution time (ms)	0.0	0.0	0.0
QT8, average execution time (ms)	30360.099609375	31481.30078125	31122.650390625
Queries failed	15	15	15
Storage cost (KB)	1313293.875	1304905.375	1296516.625

Fig. 11: Experiment results for R43ples system from the HOBBIT platform

5 Conclusions and Future work

In this paper we first described the state-of-the-art approaches for managing and benchmarking evolving RDF data. We presented the basic strategies that archiving systems follow for storing multiple versions of a dataset, and described the existing versioning benchmarks along with their features and characteristics. Subsequently, we described in detail a first version of the versioning Benchmark *SPBv*, along with a set of preliminary experimental results.

In the future we will extend the data generator in order to produce more realistic evolving data. In particular, not only additions will be supported, but deletions or modifications of existing data as well. Furthermore, we will let the benchmarked systems decide the generated data format, according to the archiving strategy they implement. So, if a system implements the full materialization archiving strategy, it will receive the generated data as separate versions. On the other hand, if a system implements the delta-based strategy, it will get the data as expected: the initial version and the subsequent sets of added/deleted triples. Also, we will try to re-design some types of queries (e.g., delta-based), in order to be comparable with the corresponding version-based. Therefore, we will be able to identify benefits or pitfalls of systems according to the archiving strategy they implement.

Acknowledgments

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

References

1. Tobias Käfer, Ahmed Abdelrahman, et al. Observing linked data dynamics. In *ESWC*, 2013.
2. Jurgen Umbrich, Michael Hausenblas, et al. Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources. In *LDOW*, 2010.
3. Fouad Zablith, Grigoris Antoniou, et al. Ontology evolution: a process-centric survey. *Knowledge Eng. Review*, 30(1):45–75, 2015.
4. Peter Boncz, Thomas Neumann, et al. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*, 2013.
5. Vassilis Papakonstantinou, Giorgos Flouris, et al. Versioning for linked data: Archiving systems and benchmarks. In *BLINK*, 2016.
6. Max Völkel and Tudor Groza. SemVersion: An RDF-based ontology versioning system. In *IADIS*, volume 2006, page 44, 2006.
7. Steve Cassidy and James Ballantine. Version Control for RDF Triple Stores. *ICSOFT*, 7:5–12, 2007.
8. Markus Graube, Stephan Hensel, et al. R43ples: Revisions for triples. *LDQ*, 2014.
9. Miel Vander Sande, Pieter Colpaert, et al. R&Wbase: git for triples. In *LDOW*, 2013.
10. Dong-Hyuk Im, Sang-Won Lee, et al. A version management framework for RDF triple stores. *IJSEKE*, 22(01):85–106, 2012.
11. Haridimos Kondylakis and Dimitris Plexousakis. Ontology evolution without tears. *Journal of Web Semantics*, 19, 2013.
12. Thomas Neumann and Gerhard Weikum. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *VLDB Endowment*, 3(1-2):256–263, 2010.
13. Kostas Stefanidis, Ioannis Chrysakis, et al. On designing archiving policies for evolving RDF datasets on the Web. In *ER*, pages 43–56. Springer, 2014.
14. Paul Meinhardt, Magnus Knuth, et al. TailR: a platform for preserving history on the web of data. In *SEMANTICS*, pages 57–64, 2015.
15. Javier David Fernandez Garcia, Jürgen Umbrich, et al. BEAR: Benchmarking the Efficiency of RDF Archiving. Technical report, Department für Informationsverarbeitung und Prozessmanagement, WU Vienna University of Economics and Business, 2015.
16. Marios Meimaris, George Papastefanatos, et al. A query language for multi-version data web archives. *Expert Systems*, 33(4):383–404, 2016.
17. Marios Meimaris and George Papastefanatos. The EvoGen Benchmark Suite for Evolving RDF Data. *MeDAW*, 2016.
18. Javier David Fernandez Garcia, Jurgen Umbrich, et al. Evaluating Query and Storage Strategies for RDF Archives. In *SEMANTiCS*, 2016, forthcoming.
19. Yuanbo Guo, Zhengxiang Pan, et al. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.
20. Venelin Kotsev, Nikos Minadakis, et al. Benchmarking RDF Query Engines: The LDBC Semantic Publishing Benchmark. In *BLINK*, 2016.
21. Vassilis Papakonstantinou, Irini Fundulaki, et al. Deliverable 5.2.1: First version of the versioning benchmark. `https://project-hobbit.eu/wp-content/uploads/2017/06/D5.2.1_First_Version_Versioning_Benchmark.pdf`.