# Assessing Linked Data Versioning Systems: The Semantic Publishing Versioning Benchmark*

Vassilis Papakonstantinou, Irini Fundulaki, and Giorgos Flouris

Institute of Computer Science-FORTH, Greece

**Abstract.** As the Linked Open Data Cloud is constantly evolving, both at schema and instance level, there is a need for systems that efficiently support storing and querying of such 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 *Semantic Publishing Versioning Benchmark* (SPVB) that aims to test the ability of versioning systems to efficiently manage versioned Linked Data datasets and queries evaluated on top of these datasets. We discuss the benchmark data and SPARQL query generation process, as well as the evaluation methodology we followed for assessing the performance of a benchmarked system. Finally, we describe a set of experiments conducted with the R43PLES and VIRTUOSO systems using SPVB.

**Keywords:** RDF, Linked Data, Versioning, 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. For this reason, there exist a number of benchmarks that test the ability of Linked Data systems to store and query data in an efficient way. However, to the best of our knowledge, only a limited number of systems (mostly academic) and benchmarks exist for handling *versioned* datasets, 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

* *This work will be published as part of the book "Emerging Topics in Semantic Technologies. ISWC 2018 Satellite Events. E. Demidova, A.J. Zaveri, E. Simperl (Eds.), ISBN: 978-3-89838-736-1, 2018, AKA Verlag Berlin"*

over time. The tracking of frequent changes is called *versioning*, and the systems that can handle such versioned data, *versioning systems*. Note here, that data versioning slightly differs from *data archiving*, since the later refers to *inactive* and *rarely* modified data that needs to be retained for long periods of time.

In this paper, we discuss the **S**emantic **P**ublishing **V**ersioning **B**enchmark (SPVB) developed in the context of the H2020 European HOBBIT project[1] for testing the ability of versioning systems to efficiently manage versioned datasets. SPVB acts like a *Benchmark Generator*, as it generates both the data and the queries needed to test the performance of the versioning systems. The main component of the benchmark is the *Data Generator* that is highly configurable. SPVB is not tailored to any versioning strategy (see Section 2.1) and can produce data of different sizes that can be altered in order to create arbitrary numbers of versions using configurable insertion and deletion ratios. It uses the data generator of Linked Data Benchmark Council's (LDBC)[2] Semantic Publishing Benchmark (SPB) as well as DBpedia [4] data. LDBC–SPB leverages the scenario of the BBC media organization, which makes extensive use of Linked Data Technologies, such as RDF and SPARQL. SPVB's Data Generator, is also responsible for producing the SPARQL queries (the so-called *tasks*) that have to be executed by the system under test. Such queries are of different types (see Section 2.2) and are partially based on a subset of the 25 query templates defined in the context of DBpedia SPARQL Benchmark (DBPSB) [5]. SPVB evaluates the correctness and performance of the tested system via the following Key Performance Indicators (KPIs): i) *Query failures* ii) *Initial version ingestion speed* iii) *Applied changes speed* iv) *Storage space cost* v) *Average Query Execution Time* and vi) *Throughput*.

The outline of the paper is the following. In Section 2, we discuss the background information regarding the versioning strategies and query types. We present the already proposed benchmarks for systems handling versioned RDF data in Section 3. In Sections 4 and 5 we describe the versioning benchmark SPVB and the experiments conducted with it, respectively. Finally, Section 6 concludes and outlines future work.

## 2 Background

### 2.1 Versioning Strategies

Three alternative RDF versioning strategies have been proposed in the literature. The first one is *full materialization*, where all different versions are explicitly stored [6]. Next, there is the *delta-based* strategy, 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 [7–11]. Finally, the *annotated triples* strategy, is based on the idea of augmenting each triple with its temporal validity, which is usually composed of two

[1] `https://project-hobbit.eu/`
[2] `ldbc.council.org`

timestamps that determine when the triple was *created* and *deleted* [12]. *Hybrid* strategies [13] that combine the above, have also been considered. Such strategies try to enjoy most of the advantages of each approach, while avoiding many of their respective drawbacks.

### 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 [14, 13, 15, 16] to identify and categorize these types of queries. Our suggestion, which is a combination of such efforts and was presented in [17], is shown in Figure 1.

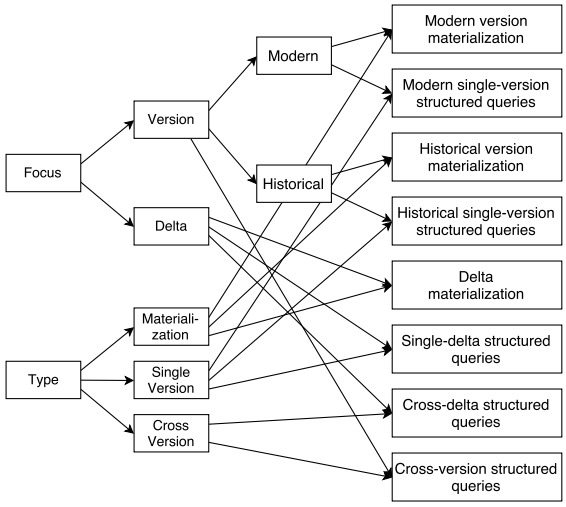


Fig. 1: Different queries organized by focus and type (see also [17])

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 in SPVB are:

- **QT1 - Modern version materialization** queries ask for a full current version to be retrieved.
- **QT2 - Modern single-version structured** queries are performed in the current version of the data.
- **QT3 - Historical version materialization** queries on the other hand ask for a full past version.
- **QT4 - Historical single-version structured** queries are performed in a past version of the data.
- **QT5 - Delta materialization** queries ask for a full delta to be retrieved from the repository.
- **QT6 - Single-delta structured** queries are performed on the delta of two consecutive versions.
- **QT7 - Cross-delta structured** queries are evaluated on changes of several versions of the dataset.
- **QT8 - Cross-version structured** queries are evaluated on several versions of the dataset.

## 3 Related Work

A benchmark is a set of tests against which the performance of a system is evaluated. A benchmark helps computer systems compare and assess their performance in order to become more efficient and competitive. To the best of our knowledge, there have been only two proposed benchmarks for versioning systems in the literature, which are described below (see [17] for more details).

The BEAR [18, 14] 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 versioning 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 a version, and, finally iv) the *RDF vocabulary* that describes the different subjects, predicates and objects in a single version. Regarding the generation of the queries of the benchmark, the *result cardinality* and *selectivity* of the queries are considered to guarantee that retrieval differences in query response times are attributed to the versioning strategy. In order to be able to judge the different strategies, BEAR introduces five foundational query atoms to cover the broad spectrum of emerging retrieval demands in RDF archiving. In particular, the authors propose i) queries on versions (QT2, QT4 from our categorization), ii) on deltas (QT5, QT6, QT7), iii)

the *Version queries* that return the results of the query $Q$ annotated with the version label in which each of them exists, iv) the *Cross-version join* queries that serves the join between the results of some $Q_1$ in $V_i$, and some $Q_2$ in $V_j$ and v) the *Change materialisation* queries that provide those consecutive versions in which the given query $Q$ produces different results.

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. Also, it is focusing on the evaluation of the versioning strategies instead of the systems that implement them, which is the main objective of SPVB.

EVOGEN[16] is a generator for versioned RDF data used for benchmarking versioning 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 an extensible and configurable *Benchmark Generator* 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 on 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) and vi) *Mixed queries*.

EVOGEN is a more complete benchmark, as it is a highly configurable and extensible benchmark generator. However, its query workload seems to be approach-dependent, in the sense that the delta-based queries require that the benchmarked systems store metadata about underlying deltas (addition/deletion of classes, 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).

## 4 Semantic Publishing Versioning Benchmark

Here, we present the second version of SPVB (the first version was presented in [20]), developed in the context of the HOBBIT H2020 project. The benchmark is built on top of the HOBBIT platform which is available online[3], but it can be locally deployed as well[4]. The source code of the benchmark can be found in the project's github repository[5]. SPVB is the first benchmark for versioned RDF data that uses both realistic synthetic data and real DBpedia data, while at the same time its query workload is mainly based on real DBpedia queries.

[3] `https://master.project-hobbit.eu/`
[4] `https://github.com/hobbit-project/platform`
[5] `https://github.com/hobbit-project/versioning-benchmark`

The new version of SPVB, when compared to its previous version, has been improved in various aspects. First, its older version supported only additions from one version to another, and the generated data was split into different versions of equal size according to their creation date. As a result, the benchmark was not useful for systems that had to be tested for versioned datasets of increasing or decreasing size. On the contrary, the new version of SPVB allows one to configure the number of additions and deletions for the different versions. his is a critical feature, that allows one to produce benchmarks that can be used to test a broad spectrum of situations regarding versioned datasets, a feature that reveals the benefits or pitfalls of systems under test.

A second improvement is related to the form of the data. In the previous version the benchmark, each data version was sent to the system only as an independent copy whereas now, the data generator can send data in different forms (independent copies, change-sets or both). As a result, systems do not need to post-process the data in order to load them, since they can choose the appropriate form to receive the data, according to the versioning strategy they implement.

Finally, the generated queries were exclusively synthetic. Conversely, in the current version, the use of DBpedia data allows us to use real DBpedia queries that arise from real world situations making the benchmark more realistic.

SPVB consists of the following four main components: the *Data Generator*, the *Task Provider*, the *Evaluation Storage* and the *Evaluation Module*. Each of these components is described in detail in the following Sections and their architecture is shown graphically in Figure 2.

### 4.1 Data Generation

The Data Generator, as shown in Figure 2, is the main component of SPVB. It is responsible for creating both the versions and the SPARQL queries, as well as to compute the expected results (gold standard) for the benchmark queries. SPVB's data generator is highly configurable, since it allows the user to generate data with different characteristics and of different forms. In particular, the following parameters can be set to configure the data generation process:

1. **Number of versions**: defines the number of versions to produce.
2. **Size**: defines the size of the initial version of the dataset in terms of triples.
3. **Version insertion ratio**: defines the proportion of added triples between two consecutive versions (originally proposed in [18]). In particular, given two versions $V_i$ and $V_{i+i}$, the version insertion ratio $\delta^+_{i,i+1}$ is computed by the formula $\delta^+_{i,i+1} = |\Delta^+_{i,i+1}|/|V_i|$, where $|\Delta^+_{i,i+i}|$ is the number of added triples from version $i$ to version $i+1$, and $|V_i|$ is the total number of triples of version $i$ (in which triples are added).
4. **Version deletion ratio**: defines the proportion of deleted triples between two consecutive versions (also proposed in [18]). Given two versions $V_i$ and $V_{i+i}$, the version deletion ratio $\delta^-_{i,i+i}$ is computed by the formula $\delta^-_{i,i+i} = |\Delta^-_{i,i+i}|/|V_i|$, where $|\Delta^-_{i,i+i}|$ is the number of deleted triples from version $i$ to

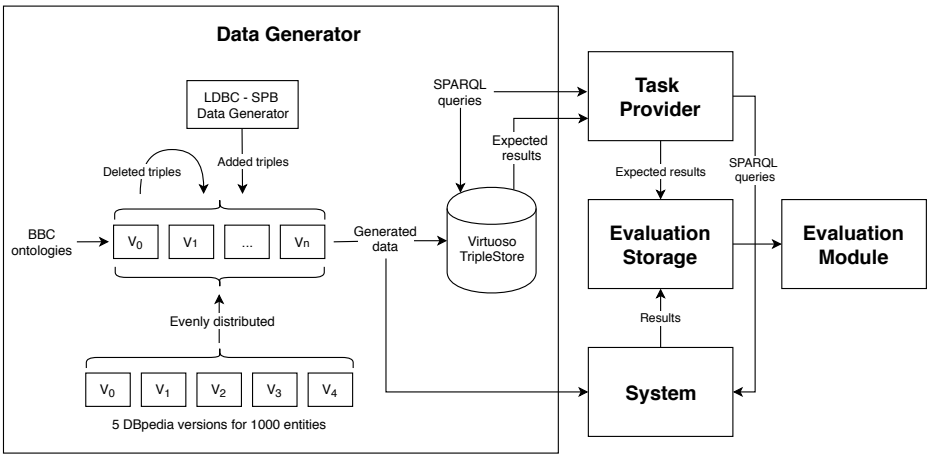


Fig. 2: Architecutre of SPVB

version $i + i$, and $|V_i|$ is the total number of triples of version $i$ (from which the triples are deleted).

5. **Generated data form**: each system implements a different versioning strategy, so it requires the generated data in a specific form. SPVB's data generator can output the data of each version as i) an *Independent Copy* (suitable for systems implementing the full materialization strategy), ii) as a *Changeset* – set of added and deleted triples (suitable for systems implementing the delta-based or annotated triples strategies) or iii) both as an independent copy and changeset (suitable for systems implementing a hybrid strategy).
6. **Generator seed**: used to set the random seed for the data generator. This seed is used to control all random data generation happening in SPVB.

Based on these parameters the generator can produce a version of the original dataset that contains both realistic synthetic data and real DBpedia data. Regarding the generation of synthetic data, the data generator of SPVB uses the data generator of LDBC–SPB [21] for producing the initial version of the dataset as well as the triples that will be added from one version to another. To do so, it uses seven *core* and three *domain* RDF ontologies (as those described in [21]) and a reference dataset of DBpedia for producing the data.

The data generator of LDBC–SPB 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 the reference dataset of DBpedia. 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 DBpedia reference dataset. Thus, a creative work provides metadata about one or more entities and

defines relations between them. The data generator of LDBC–SPB models three types of relations in the data, and for each one produces 1/3 of the number of creative works. Such relations are the following ones:

- **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*.
- **Clustering of data.** The clustering effect is produced by generating *creative works* about a *single entity from the reference* dataset and for a *fixed period of time*. 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.
- **Random tagging of entities.** Random data distributions are defined with a *bias towards popular entities* created when the *tagging* is performed.

For producing the *initial version* of the dataset the data generator of LDBC–SPB runs with the *Size* configuration parameter as input. For producing the triples to add, we first compute the number of triples to be added using the *version insertion ratio*, and then run the data generator with this number as input. As a result, the set of creative works that have to be added are produced. At this point we let the generator *produce random data distributions*, as we do not want to "break" the existing relations among creative works and entities (clustering or the correlation of entities) of the initial version.

For producing the triples that will be deleted, we first compute the number of triples to be deleted based on the version deletion ratio. Then, we take as input the triples that were produced randomly in the previous versions, and we choose creative works in a random manner until we reach the targeted number of triples to delete. The reason we only choose creative works that were previously produced randomly, is the same as in case of additions (we do not want to "break" the clustering or correlations for the already existing entities). Recall that in the initial version of the dataset, the maximum number of triples that can be produced randomly equals the 1/3 of the total number of triples. So, if the generator is configured to delete more than 1/3 of the number of triples for each version, that is, the triples to be deleted exceeds the triples that were randomly produced, we impose the threshold of 33% for $\delta^{-}_{i,i+1}$.

As we mentioned earlier, except for creating the versioned dataset for creative works, SPVB's data generator supports the versioning of the reference dataset of DBpedia employed by LDBC–SPB to annotate the creative works. In particular, we maintain 5 different versions of DBpedia, from year 2012 to year 2016 (one for each year). Such versions contain the subgraph of each entity used by the data generator of LDBC–SPB to annotate creative works through the *about* and *mentions* properties. In practice, all DBpedia triples in which the entity URI is in the subject position are maintained. However, LDBC–SPB is using about 1 million entities for the annotation of creative works. Obviously, it is not possible to burden the generated creative works with such volume of data. So, from those 1 million entities we keep the 1000 most popular ones, based on the score

provided by LDBC–SPB, and we extract their RDF graph from the 5 different versions of DBpedia. By doing so, we ended up with 5 DBpedia subgraphs (one for each version) containing 40K, 35K, 65K, 60K and 71K triples respectively. These versions enhance the generated creative works and are *evenly distributed* to the total number of versions that the data generator is configured to produce. E.g., assuming that 10 versions are produced by the data generator, the triples of the 5 versions of DBpedia will be added to versions 0, 2, 5, 7 and 9 respectively. In cases that the data generator is configured to produce less than 5 versions (say $N$), we only keep the first $N$ DBpedia versions.

After the generation of both creative works and DBpedia data has finished, they are loaded into a Virtuoso triplestore[6]. This way, we can later evaluate the produced SPARQL queries and compute the expected results that are required by the *Evaluation Module* (Section 4.3) to assess the correctness of the results reported by the benchmarked system.

### 4.2 Tasks Generation

As shown in Figure 2, the generation of the SPARQL queries that have to be executed by the systems under test is a process that is also taking place in the *Data Generator* component. Given that there is neither a standard language, nor an official SPARQL extension for querying RDF versioned data, the generated queries of SPVB assume that each version is stored in its own named graph. Each benchmarked system should express these queries in the query language that it supports in order to be able to execute them.

The queries produced by the Data Generator are based on a set of *query templates*. In particular, for each one of the eight versioning query types (Section 2.2), we have defined one or more query templates. We show an example in Listing 1.1 that retrieves creative works that are about different topics, along with the topics type from a past version. The full list of the query templates can be found online[7]. Such query templates contain *placeholders* of the form {{{placeholder}}} which may refer either to the queried version ({{{historicalVersion}}}) or an IRI from the reference dataset of DBpedia ({{{cwAboutUri}}}). The placeholders are replaced with concrete values, in order to produce a set of similar queries.

```
1 SELECT DISTINCT ?creativeWork ?v1
2 FROM {{{historicalVersion}}}
3 WHERE {
4   ?creativeWork cwork:about {{{cwAboutUri}}} .
5   {{{cwAboutUri}}} rdf:type ?v1 .
6 }
```

Listing 1.1: Historical single-version structured query template

[6] http://vos.openlinksw.com

[7] https://hobbitdata.informatik.uni-leipzig.de/MOCHA_ESWC2018/Task3/query_templates/

For the query types that refer to structured queries on one or more versions (QT2, QT4 and QT8), we use 6 of the 25 query templates proposed by the DBpedia SPARQL Benchmark (DBPSB) [5] which are based on real DBpedia queries. The reason we chose those 6 templates was that if they are evaluated on top of the 5 versions of the reference dataset of DBpedia, with their corresponding placeholder replaced with a variable, will always return results. At the same time, they comprise most of the SPARQL features (FILTERs, OPTIONALs, UNIONs, etc.). Note here that the DBPSB query templates were generated to be executed on top of DBpedia data only. On the other hand, SPVB generates data that combine DBpedia and creative works. So, in order for the 6 DBPSB query templates to be applicable to the data generated by SPVB we added an extra triple pattern to them that "connects" the creative works with DBpedia through the *about* or *mentions* properties, as shown in line 4 of Listing 1.1.

As we mentioned earlier, the placeholders may refer either to the queried version or an IRI from the reference dataset of DBpedia. The ones that refer to the queried version, are replaced in such a way that a wide range of available versions is covered. For example, assume that we have the query template shown in Listing 1.1 and the generator is configured to produce $n$ versions in total. The {{{historicalVersion}}} placeholder will be replaced with the graph names denoting i) the initial version, ii) an intermediate version and iii) the $n-1$ version. The placeholders that refer to an IRI, are the same placeholders used in the DBPSB query templates. To replace them with concrete values we use a technique similar to the one used by DBPSB. We run offline on top of the 5 different versions of DBpedia each of the 6 DBPSB query templates, having replaced their placeholder with a variable, and keep at most 1000 possible concrete values that each placeholder may be replaced with. So, according to the queried version we can randomly pick one of those values. Such a technique guarantees that the produced query will always return results.

After replacing all the placeholders of each query template, a set of similar queries is generated (a complete set of such queries for a specific configuration can be found online[8]). As shown in Figure 2, such queries are evaluated on top of Virtuoso[9], where the already generated versions have been loaded into, in order to calculate the expected results. After the expected results have been computed, the *Data Generator* sends the queries along with the expected results to the *Task Provider* component and the generated data to the system under test. The job of the *Task Provider* is to *sequentially* send the SPARQL queries to the system under test and the expected results to the *Evaluation Storage* component. So, the system under test evaluates the queries on top of the data (after the appropriate rewritings, if necessary) and reports the results to the *Evaluation Storage*.

[8] `https://hobbitdata.informatik.uni-leipzig.de/MOCHA_ESWC2018/Task3/queries`

[9] `http://vos.openlinksw.com`

### 4.3 Evaluation Module

The final component of the benchmark is the *Evaluation Module*. The Evaluation Module receives from the *Evaluation Storage* component the results that are sent by the system under test, as well as the expected results sent by the *Task Provider* and evaluates the performance of the system under test. To do so, it calculates the following *Key Performance Indicators (KPIs)*, covering a high spectrum of aspects that one needs for assessing the performance of a versioning system:

- **Query failures**: The number of queries that failed to execute. Failure refers to the fact that the system under test returns a result set ($RS_{sys}$) that is not equal to the expected one ($RS_{exp}$). This means that i) $RS_{sys}$ has equal size to $RS_{exp}$ and ii) every row in $RS_{sys}$ has one matching row in $RS_{exp}$, and vice versa (a row is only matched once). If the size of the result set is larger than 50.000 rows, only condition i) is checked for performance reasons.
- **Initial version ingestion speed** (triples/second): the total triples of the initial version that can be loaded per second. 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 underlying processes such as, computing deltas, reconstructing versions, storing duplicate information between versions etc., may take place.
- **Applied changes speed** (changes/second): tries to quantify the overhead of the underlying processes that take place when a set of changes is 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 space cost** (MB): This KPI measures the total storage space required to store all versions measured in MB.
- **Average Query Execution Time** (ms): The average execution time, in milliseconds for each one of the eight versioning query types, as those described in Section 2.2.
- **Throughput** (queries/second): The execution rate per second for all queries.

## 5 Experiments

In order to test the implementation of SPVB on top of the HOBBIT platform for the different versioning systems described in [17], we were able to conduct experiments only for R43PLES (Revision for triples) [8], which uses JENA TDB as an underlying storage/querying layer. For the rest, since most of them are academic products, 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). In order to have a baseline system we decided to implement the *full materialization* versioning strategy, by assuming that each version is stored in its own named graph, on top of VIRTUOSO[10] triplestore that does not handle versioned data.

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

The online instance of the HOBBIT platform is deployed on a server cluster. Each of the Data Generator, Task Provider, Evaluation Module and System components is created and runs on its own node having 32 threads in total (2 sockets, each with 8 cores with hyper-threading) and a total of 256GB of RAM. For our experiments, we produced 3 datasets of different initial sizes that correspond to 100K, 500K and 1M triples. For each dataset, we produced 5 different versions following a version insertion and deletion ratio of 15% and 10% respectively. As a result, for each one of the initial datasets of 100K, 500K and 1M triples, we produced a final version of 141K, 627K and 1.235M triples respectively. The number of triples in the final versions includes the versioned data of the reference dataset of DBpedia as well. For fairness, we ran three experiments per dataset size and computed the average values for all reported results. The experiment timeout was set to 1 hour for both R43PLES and VIRTUOSO systems. This was necessary, in order to have a good use of the HOBBIT platform, since the submitted experiments run sequentially for more than twenty benchmarks.

Regarding the *full materialization* strategy that we implemented on top of VIRTUOSO, for the datasets of 100K, 500K and 1M triples, the queries were executed at a rate of 1.4, 0.3 and 0.16 queries per second respectively and all of them returned the expected results. For the rest of the KPIs we graphically report the results.

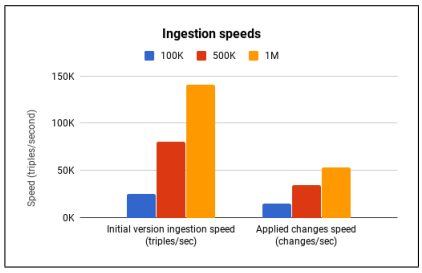


Fig. 3: Ingestion speeds

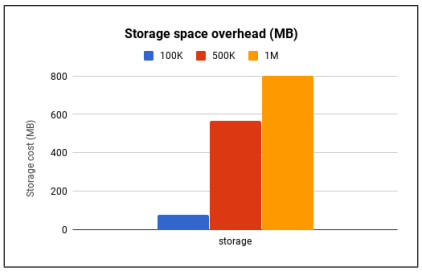


Fig. 4: Storage space overhead

In the left bar chart of Figure 3 we can see for all datasets the *initial version ingestion speed*. 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 24K to 140K triples per second and increases with the dataset size, and consequently with the size of its initial version. 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 14K to 53K changes per second. We can observe here that the initial version ingestion speed outperforms the applied changes speed. This is an overhead of the chosen versioning strategy i.e., *full materialization*. 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 4 we can see the storage space required for storing the data for all different datasets. The space requirements as expected increase as the total number of triples increases from 80MB to 800MB. This significant overhead on storage space is due to the archiving strategy used (i.e., Full Materialization).

In Figures 5, 6, 7, 8 and 9 we present the average execution time (in ms) for the queries of each versioning query type, and for each dataset size.

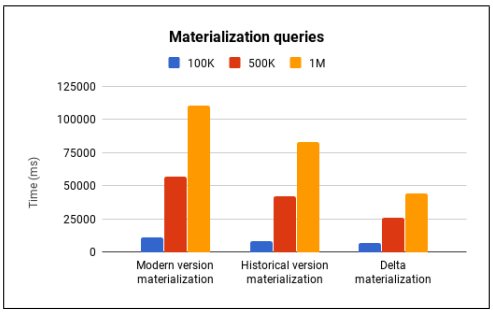


Fig. 5: Execution times for materialization queries

In Figure 5 we can see the time required for materializing i) the modern (current) version; ii) a historical (past) one; and iii) the difference between two versions (delta). In the left and middle bars 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 quite smaller than the modern one, as it contains less triples. In both cases, we observe that execution times are small enough, as all the versions are already materialized in the triple store. 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 overhead in the time required for evaluation.

In Figures 6, 7 , 8 and 9 we can see the execution times for all types of structured queries. In all of the cases, similarly to materialization queries, the 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 (that are based on DBPSB query templates) are much harder regarding query evaluation than the delta-based ones.

Regarding the R43PLES system, we only managed to run experiments for the first dataset, of 100K triples, since for the remaining ones the experiment time exceed the timeout of 1 hour. In most of the cases the response times are order(s)

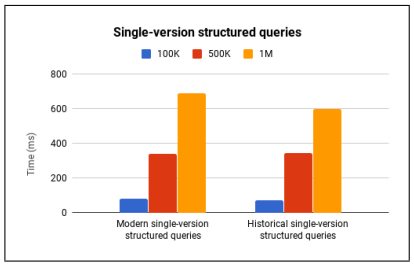


Fig. 6: Execution times for single version structured queries

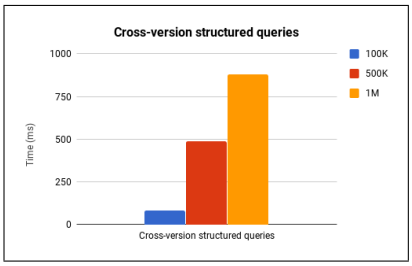


Fig. 7: Execution times for cross-version structured queries

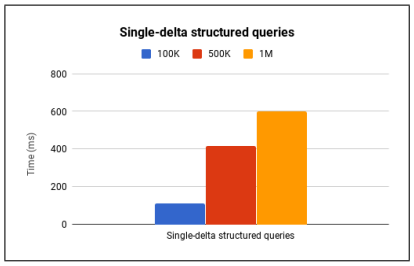


Fig. 8: Execution times for single delta structured queries

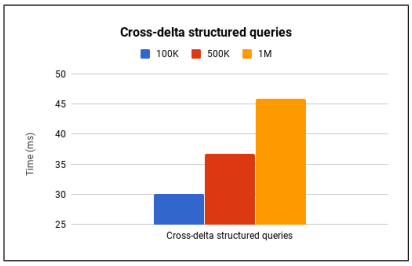


Fig. 9: Execution times for cross-delta structured queries

of magnitude slower, so we do not report and compare the results graphically with the corresponding ones of Virtuoso. At first, in the left table of Table 1 we can see the initial version ingestion speed and applied changes speed. The changes are applied in a slower rate than the triples of the initial version are loaded, as the version that is materialized is always 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 even in the case of applied changes speed where we would expect it to be faster. The storage space overhead, as shown in the same Tables, is double the storage space for Virtuoso. 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 seem to be very different.

Next, regarding the execution of queries, with a quick glance, we can see that queries are executed at a much lower rate and many of them failed to return the correct results. Such failures are possibly due to the fact that R43ples failed to correctly load all the versioned data. E.g. in the final version R43ples maintained a total of 140.447 triples instead of 141.783. Concerning the queries execution time, for materializing the current version (QT1) or executing a structured query on it (QT2), R43ples (although in some cases returned slightly less results) required similar time compared to Virtuoso. This is something that we expected, as the current version is kept materialized just like in Virtuoso. This is not happening for the rest of the query types as R43ples is 1 to 3 orders of

magnitude slower than VIRTUOSO. This is also an expected result as R43PLES needs to reconstruct the queried version on-the-fly.

| Metric | Result |
|---|---|
| $V_0$ Ingestion speed (triples/sec) | 3502.39 |
| Changes speed (changes/sec) | 2767.56 |
| Storage cost (MB) | 197378 |
| Throughput (queries/second) | 0.09 |
| Queries failed | 25 |

| Metric | Result | Succeeded queries |
|---|---|---|
| QT1 (ms) | 13887.33 | 0/1 |
| QT2 (ms) | 146.28 | 25/30 |
| QT3 (ms) | 18265.78 | 0/3 |
| QT4 (ms) | 11681.49 | 13/18 |
| QT5 (ms) | 31294.00 | 0/4 |
| QT6 (ms) | 12299.58 | 4/4 |
| QT7 (ms) | 35294.33 | 2/3 |
| QT8 (ms) | 19177.33 | 30/36 |

Table 1: Results for R43PLES system

## 6 Conclusions and Future work

In this paper we described the state-of-the-art approaches for managing and benchmarking evolving RDF data, and described in detail SPVB, a novel benchmark for versioning systems, along with a set of experimental results.

We plan to keep SPVB an evolving benchmark, so that we can attract more versioning systems to use it, as well as to try to asses the performance of more versioning systems on ourselves. To achieve this, we plan to add some extra functionalities or improve the existing ones. In particular, we want to move the expected results computation out of the Data Generator component. By doing so, SPVB will be able to generate data in parallel through multiple Data Generators. Also, we want to make the query workload more configurable, by giving the ability to the benchmarked system to include or exclude specific query types. Moreover, we want to optimize the responses evaluation taking place in the Evaluation Module component, as for dozens of thousands of results the evaluation may become very costly. Finally, we plan to add functionalities that the second version of the HOBBIT platform offers, such as graphical visualisation of the KPIs.

## Acknowledgments

This work was supported by grants from the EU H2020 Framework Programme provided for the project HOBBIT (GA no. 688227).

## References

1. T. Käfer, A. Abdelrahman, et al. Observing linked data dynamics. In *ESWC-13*.

2. J. Umbrich, S. Decker, et al. Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources. In *LDOW*, 2010.
3. F. Zablith, G. Antoniou, et al. Ontology evolution: a process-centric survey. *Knowledge Eng. Review*, 30(1):45–75, 2015.
4. C. Bizer, T. Heath, et al. Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pages 205–227, 2009.
5. M. Morsey, J. Lehmann, et al. Dbpedia sparql benchmark–performance assessment with real queries on real data. In *ISWC*, pages 454–469. Springer, 2011.
6. M. Völkel and T. Groza. SemVersion: An RDF-based ontology versioning system. In *IADIS*, volume 2006, page 44, 2006.
7. S. Cassidy and J. Ballantine. Version Control for RDF Triple Stores. *ICSOFT*, 2007.
8. M. Graube, S. Hensel, et al. R43ples: Revisions for triples. *LDQ*, 2014.
9. M. Vander Sande, P. Colpaert, et al. R&Wbase: git for triples. In *LDOW*, 2013.
10. D. H. Im, S. W. Lee, et al. A version management framework for RDF triple stores. *IJSEKE*, 22(01):85–106, 2012.
11. H. Kondylakis and D. Plexousakis. Ontology evolution without tears. *Journal of Web Semantics*, 19, 2013.
12. T. Neumann and G. Weikum. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *VLDB Endowment*, 3(1-2):256–263, 2010.
13. K. Stefanidis, I. Chrysakis, et al. On designing archiving policies for evolving RDF datasets on the Web. In *ER*, pages 43–56. Springer, 2014.
14. G. Fernandez, D. Javier, et al. BEAR: Benchmarking the Efficiency of RDF Archiving. Technical report, 2015.
15. M. Meimaris, G. Papastefanatos, et al. A query language for multi-version data web archives. *Expert Systems*, 33(4):383–404, 2016.
16. M. Meimaris and G. Papastefanatos. The EvoGen Benchmark Suite for Evolving RDF Data. *MeDAW*, 2016.
17. V. Papakonstantinou, G. Flouris, et al. Versioning for linked data: Archiving systems and benchmarks. In *BLINK@ ISWC*, 2016.
18. J. D. Fernández, J. Umbrich, et al. Evaluating query and storage strategies for rdf archives. In *Proceedings of the 12th International Conference on Semantic Systems*, pages 41–48. ACM, 2016.
19. Y. Guo, Z. Pan, et al. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the WWW*, 3(2):158–182, 2005.
20. V. Papakonstantinou, G. Flouris, et al. Spbv: Benchmarking linked data archiving systems. 2017.
21. V. Kotsev, N. Minadakis, et al. Benchmarking RDF Query Engines: The LDBC Semantic Publishing Benchmark. In *BLINK*, 2016.