Collaborative Project  
Holistic Benchmarking of Big Linked Data  

Project Number: 688227  
Start Date of Project: 2015/12/01  
Duration: 36 months  

Deliverable 5.2.1: First version of the Versioning Benchmark

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due Date of Deliverable</td>
<td>Month 18, 31/05/2017</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 18, 31/05/2017</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP5 - Benchmarks III: Storage and Curation</td>
</tr>
<tr>
<td>Task</td>
<td>T5.2</td>
</tr>
<tr>
<td>Type</td>
<td>Report</td>
</tr>
<tr>
<td>Approval Status</td>
<td>Draft</td>
</tr>
<tr>
<td>Version</td>
<td>0.3</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>35</td>
</tr>
<tr>
<td>Filename</td>
<td>D5.2.1_First_Version_Versioning_Benchmark.pdf</td>
</tr>
</tbody>
</table>

Abstract: This deliverable discusses the first version of HOBBIT’s versioning benchmark.

The information in this document reflects only the author’s views and the European Commission is not liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/ her sole risk and liability.

![European Union Flag]

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 688227.
History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>27/04/2017</td>
<td>First draft created</td>
<td>Vassilis Papakonstantinou</td>
</tr>
<tr>
<td>0.1</td>
<td>01/05/2017</td>
<td>Second draft created</td>
<td>Irini Fundulaki</td>
</tr>
<tr>
<td>0.2</td>
<td>15/05/2017</td>
<td>Peer review comments</td>
<td>Martin Strohbach, Roman Katerinenko, Khurram Mehnoode Cheema</td>
</tr>
<tr>
<td>0.3</td>
<td>17/05/2017</td>
<td>Second revision</td>
<td>Vassilis Papakonstantinou</td>
</tr>
<tr>
<td>1.0</td>
<td>20/05/2017</td>
<td>Final Version</td>
<td>Irini Fundulaki, Vassilis Papakonstantinou</td>
</tr>
</tbody>
</table>

Author List

<table>
<thead>
<tr>
<th>Organization</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTH</td>
<td>Vassilis Papakonstantinou</td>
<td><a href="mailto:papv@ics.forth.gr">papv@ics.forth.gr</a></td>
</tr>
<tr>
<td>FORTH</td>
<td>Irini Fundulaki</td>
<td><a href="mailto:fundul@ics.forth.gr">fundul@ics.forth.gr</a></td>
</tr>
<tr>
<td>FORTH</td>
<td>Giannis Roussakis</td>
<td><a href="mailto:rousakis@ics.forth.gr">rousakis@ics.forth.gr</a></td>
</tr>
<tr>
<td>FORTH</td>
<td>Giorgos Flouris</td>
<td><a href="mailto:fgeo@ics.forth.gr">fgeo@ics.forth.gr</a></td>
</tr>
<tr>
<td>University of Tampere</td>
<td>Kostas Stefanidis</td>
<td><a href="mailto:kostas.stefanidis@uta.fi">kostas.stefanidis@uta.fi</a></td>
</tr>
</tbody>
</table>

1 work performed while the author was affiliated with FORTH
Executive Summary

This report discusses the first version of HOBBIT’s versioning benchmark. It covers the workload conducted from M7 until M18. The Versioning Benchmark aims to test the ability of versioning systems to efficiently manage evolving datasets and queries evaluated across multiple versions of Linked Data datasets.
## Contents

1 Introduction .................................................. 6

2 State of the Art ........................................... 6
   2.1 Versioning Strategies .................................... 7
   2.2 Versioning Queries ....................................... 7
   2.3 Versioning Systems ...................................... 9
   2.4 Versioning Benchmarks .................................. 13

3 Versioning Benchmark .................................... 15
   3.1 Choke Point-based benchmark design .................. 15
   3.2 Data Generation ......................................... 16
   3.3 Task Generation ......................................... 20
   3.4 Evaluation Module ...................................... 23

4 Experiments .................................................. 24

5 Conclusions and Next Steps ............................. 28

Appendices .................................................... 28

A Query Templates .......................................... 28

References ................................................. 33
List of Figures

1. Different types of queries according to their focus and type. .................................................. 8
2. BBC Creative Works Ontology .................................................................................................. 17
3. BBC Company Ontology ........................................................................................................... 17
4. Data generator, types of produced models in generated data. ................................................... 19
5. Versioning benchmark configuration parameters ....................................................................... 20
6. Benchmark Architecture ........................................................................................................... 21
7. Configuration parameters ........................................................................................................... 25
8. Details of experiments with a dataset size of 1M triples ............................................................... 26
9. Ingestion speeds .......................................................................................................................... 26
10. Storage space overhead (in MB) ................................................................................................ 26
11. Execution times for materialization queries (in ms) .................................................................. 27
12. Execution times for single version structured queries (in ms) .................................................... 27
13. Execution times for single delta structured queries (in ms) ........................................................ 27
14. Execution times for cross-version structured queries (in ms) .................................................... 27
15. Execution times for cross-delta structured queries (in ms) ........................................................ 27

List of Tables

1. An overview of RDF versioning systems and frameworks ......................................................... 9
2. The transaction inventory [19] .................................................................................................... 10
3. BBC Core & Domain Ontologies ................................................................................................ 16
4. Versioning Queries ..................................................................................................................... 23

Listings

1. Modern version materialization query template ........................................................................ 28
2. Modern single-version structured query template 1 .................................................................. 28
3. Modern single-version structured query template 2 .................................................................. 28
4. Modern single-version structured query template 3 .................................................................. 29
5. Modern single-version structured query template 4 .................................................................. 29
6. Historical version materialization query template ...................................................................... 29
7. Historical single-version structured query template 1 ................................................................. 30
8. Historical single-version structured query template 2 ................................................................. 30
9. Historical single-version structured query template 3 ................................................................. 30
10. Historical single-version structured query template 4 ............................................................... 31
11. Delta materialization query template ......................................................................................... 31
12. Single-delta structured query template ...................................................................................... 31
13 Cross-delta structured query template ........................................... 31
14 Cross-version structured query template 1 ........................................ 32
15 Cross-version structured query template 2 ........................................ 32
16 Cross-version structured query template 3 ........................................ 33
17 Cross-version structured query template 4 ........................................ 33
1 Introduction

Big Data is one of the key assets of the future. However, the cost and effort required for introducing Big Data technology in a value chain is significant. Mastering the creation of value from Big Data will enhance European competitiveness, will result in economic growth and jobs and will deliver societal benefit. It is thus of utmost importance to reduce the costs and hurdles required to introduce Big Data processing into the European industry. A key step towards abolishing the barriers to the adoption and deployment of Big Data is to provide European companies with open benchmarking reports that allow them to assess the fitness of existing solutions for their purposes. To achieve this goal demands amongst others, the creation of benchmarks that allow one to test the performance of systems that manage Big Data.

In HOBBIT we are working towards the creation of benchmarks for the different steps of the Big Data Value Chain that include, but are not limited to

- Generation & Acquisition
- Analytics & Processing
- Storage & Curation
- Visualization & Services

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 versioning systems (mostly academic) and benchmarks exist for storing versions and testing the proposed solutions respectively. However, the existence of such systems and benchmarks is of the utmost importance, as dynamicity is an indispensable part of the Linked Open Data (LOD) initiative [12, 21]. 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 [27]. 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 report we discuss the benchmark developed in the context of the HOBBIT project for testing the performance of versioning systems for Linked Data. The benchmark is based on Linked Data Benchmark Council’s (LDBC) Semantic Publishing Benchmark. It is based on 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 versioning strategy and system. We followed a choke point-based design for the benchmark, where we extend the SPB queries with features that stress the systems under test.

The outline of the report is the following: in Section 2 we discuss the state of the art of versioning strategies, systems, benchmarks and query types. We present HOBBIT’s versioning benchmark SPBv in Section 3, and experiments are provided in Section 4.

2 State of the Art

This section presents the state of the art of versioning (a) strategies (see Section 2.1) (b) systems (see Section 2.3) and (c) benchmarks (see Section 2.4).

---

2 lbdc.council.org
3 bbc.co.uk
2.1 Versioning Strategies

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

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 [26]. Although there is no processing cost for storing the archives, the main drawback of the full materialization approach concerns scalability issues with respect to storage space: since each version is stored in its entirety, unchanged information between versions is duplicated (possibly multiple times). In scenarios where we have large versions that change often (and no 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 approach is an alternative proposal where one full version of the dataset needs to be stored, and, for each new version, only the set of changes with respect to the previous version (also known as the delta) has to be kept. This strategy has much more modest space requirements when compared to the full materialization approach, as deltas are (typically) much smaller than the dataset itself. However, the delta-based strategy imposes additional computational costs for computing and storing deltas. Also, an extra overhead at query time is introduced, as many queries would require the on-the-fly reconstruction of one or more full versions of the data. Various approaches try to improve the situation, by storing the first version and computing the deltas according to it [6, 9, 25] or storing the latest (current) version and computing reverse deltas with respect to it [11, 13].

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 [19]. This annotation allows us to reconstruct the dataset version at any given time point $t$, by just returning all triples that have been created before $t$ and were deleted after time point $t$ (if at all). An alternative annotation model uses a single annotation value that is used to determine the version(s) in which each triple existed in the dataset [25].

Hybrid Approaches aim at combining the above strategies in order to enjoy most of the advantages of each approach, while avoiding many of their respective drawbacks. This is usually implemented as a combination of the full materialization and delta-based strategies, where several (but not all, or just one) of the versions are materialized explicitly, whereas the rest are only stored implicitly through the corresponding deltas [17]. To determine how many, and which, versions must be materialized, a cost model (such as the one proposed in [20]) could be used to quantify the corresponding overheads (including space overhead for storage, time overhead at storage time, and time overhead at query time), so as to determine the optimal storage strategy. Another combination is the use of delta-based and annotated triples strategies as there are systems that store consecutive deltas, in which each triple is augmented with a value that determines its version [25].

2.2 Versioning Queries

An important novel challenge imposed by the management of multiple versions is the generation of different types of queries (QT) (e.g., queries that access multiple versions and/or deltas). There have been some attempts in the literature [7, 20] to identify and categorize these types of queries. Our suggestion, which is a combination of them, is shown in Figure 1.
Firstly, queries are distinguished when considering their focus, in version and delta queries. Version queries consider complete versions, whereas delta queries consider deltas. Version queries can be further classified to modern and historical, depending on whether they require access to the latest version (the most common case) or a previous one. Obviously, such a categorization cannot be applied to delta queries, as they refer to time changes between versions (i.e., intervals).

In addition, queries can be further classified according to their type, to materialization, single-version and cross-version queries. Materialization queries essentially request the entire respective data (a full version, or the full delta); single-version queries can be answered by imposing appropriate restrictions and filters over a single dataset version or a single delta; whereas cross-version queries request data related to multiple dataset versions (or deltas). Of course, the above categories are not exhaustive; one could easily imagine queries that belong to multiple categories, e.g., a query requesting access to a delta, as well as multiple versions. These types of queries are called hybrid queries. More specifically the types of queries that we consider are:

- **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 queries which are performed on the delta of two consecutive versions. One, for instance, could ask for the new friends that a person obtained between some version and its previous one.

- **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 Versioning Systems

A variety of RDF versioning systems and frameworks have been proposed in recent years; details on these systems are discussed in the subsections below, whereas an overview of their characteristics appears in Table 1. Such characteristics are:

- the versioning strategy that each system/framework implements
- their ability to answer SPARQL queries and to identify equivalent blank nodes across versions
- their ability to support versioning features such as committing, merging, branching etc.

Below we present the different versioning systems that have been developed for Linked Data.

<table>
<thead>
<tr>
<th>System / Framework</th>
<th>versioning Strategy</th>
<th>SPARQL support</th>
<th>Blank nodes support</th>
<th>Versioning features</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-RDF-3X [19]</td>
<td>Annotated Tuples</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SEMVersion [26]</td>
<td>Full Materialization</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cassidy et al. [6]</td>
<td>Delta Based</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Memento [24]</td>
<td>Full Materialization</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>R&amp;Wbase [25]</td>
<td>Annotated Tuples</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>R43Ples [9]</td>
<td>Delta Based</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>TALIR [17]</td>
<td>Hybrid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Im et al. [11]</td>
<td>Delta Based</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DYDRA [1]</td>
<td>Full Materialization</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: An overview of RDF versioning systems and frameworks

X-RDF-3X [19] is an extension of the RDF-3X RDF engine [18]. Even though the original RDF-3X system did not support versioning and versioning features, X-RDF-3X is essentially a timestamp-based temporal RDF engine that supports versioning, time-travel access (i.e., temporal SPARQL queries) and transactions on RDF databases. To achieve such functionality the *annotated triples* strategy was employed. Triples were also augmented with two timestamp fields referring to the *creation* and *deletion* time of each triple. Using these timestamps, the database state of a given point in time can be easily
reconstructed. Ideally, timestamps reflect the commit order of transactions, but unfortunately the commit order is not known when inserting new data. To overcome this problem, a write timestamp is assigned to each transaction once it starts updating the differential indexes (temporal small indexes that are periodically merged to the main ones), and this timestamp is then used for all subsequent operations.

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

<table>
<thead>
<tr>
<th>transId</th>
<th>version #</th>
<th>BOT</th>
<th>EOT</th>
<th>highCV #</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_{101}</td>
<td>100</td>
<td>2009-03-20 16:56:12</td>
<td>2009-03-20 17:00:01</td>
<td>300</td>
</tr>
<tr>
<td>T_{102}</td>
<td>200</td>
<td>2009-03-20 16:58:25</td>
<td>2009-03-20 16:59:15</td>
<td>200</td>
</tr>
<tr>
<td>T_{103}</td>
<td>300</td>
<td>2009-03-20 16:59:01</td>
<td>2009-03-20 16:59:42</td>
<td>300</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 2: THE TRANSACTION INVENTORY [19]

**SemVersion** [26] is inspired by the Concurrent Versioning System (CVS) [4] which was basically used in earlier years to allow collaborative development of source code during software development. **SemVersion** is a java library for providing versioning capabilities to Linked Data stores. More specifically it supports branch and merge operations for versions, as well as the reporting of conflicts. In **SemVersion**, every version is annotated with metadata like its parent version, its branches, a label and a provenance URI used for version identification. **SemVersion** follows the full materialization approach for storing versions, as it focuses more on the management of distributed engineering processes rather than storage space. 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 version of the graph. In both cases, every version of the Linked Data dataset 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.

**Version Control for RDF Triple Stores** [6] is a versioning system for RDF triple stores that is based on Darcs\(^4\) (a version control system built to manage software source code) and its theory of patches. The system uses the delta-based strategy: each version is described as a sequence of patches (deltas) that are all applied sequentially to one version in order to construct the current one. Each of these patches is represented as a named graph consisting of a set of added and deleted triples and is stored in a different Linked Data store than the original data. Optionally, a dependency sub-graph

\(^4\)http://darcs.net/
may be included in the patch, which is a set of triples that must 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
- the *merge* operation can be applied to parallel patches in order to combine them into one

An implementation on MySQL\(^5\) backend for the RedLand store [3] was evaluated which showed that the proposed approach of managing versions adds a significant overhead when compared to the raw RDF datastore. More specifically, query answering becomes four to eight times slower and space consumption increased from two to four times when compared to the raw RDF store of RedLand.

MEMENTO [23] 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, \(i = 1, \ldots, n\)) which are the archived representations of the resource that summarize its state in the past. The time \(t_i\) that a memento was captured is called *Memento-datetime*. MEMENTO can also be adopted in the context of Linked Data [24] as it had been used for providing access to prior versions of DBpedia. To do so, versions of DBpedia are stored in a MySQL database as complete snapshots, so the full materialization approach is followed, and served through a MEMENTO endpoint.

R&Wbase [25] tracks changes and versions by following a hybrid strategy, as it uses the delta-based in conjunction with the annotated triples versioning strategies. In particular, triples are stored in a quad-store as consecutive deltas. Each altered triple is assigned a context value, which is a number from a continuous sequence. More specifically, every new delta obtains an even number \(2 \times y\) that is larger than all preceding delta numbers. Then, each triple of said delta is assigned the value \(2 \times y\) (even for added triples) and \(2 \times y + 1\) (odd for deleted triples). Furthermore, the delta identifier \(2 \times y\) is used in order to store the delta’s provenance metadata in triple format, using the PROV-O vocabulary [15]. These metadata include a UID, the delta’s parent, the responsible person of the changes, the delta’s date etc. By following the above approach, it is possible to significantly reduce the required storage space, as the number of stored triples is relative to the delta size instead of the graph size, which is much smaller in most cases. R&Wbase allows querying the data stored, using SPARQL queries that are translated in such a way that the quad-store is treated as a triple-store. In particular, when a query is applied in a specific version, all version’s ancestors have to be identified, by traversing the metadata of such version, and then the query is applied to the set of returned versions. Finally, being a Git-like tool, R&Wbase supports versioning features like branching and merging of previously committed graphs.

R43ples [9] offers a central repository based on a Copy-Modify-Merge mechanism, where clients get the requested information via SPARQL (copy), work with it locally (modify) and commit their updates also via SPARQL (merge). Much like R&Wbase, R43ples supports the basic versioning features like tagging, branching and merging of versions. To do so, it introduces an enhanced, non-standard version of the SPARQL language that includes a set of new keywords (REVISION, USER, MESSAGE, BRANCH and TAG) to the reserved SPARQL keywords to support the aforementioned operations. R43ples follows the delta-based approach for storing versions. In particular, each version is represented using a temporary graph which is associated with two additional named graphs corresponding to the delta’s sets that contain the added and deleted triples. The association is represented using an extended version of the PROV-O ontology [15], called Revision Management Ontology (RMO). Applying these delta sets to the prior revision will lead to the current one. The aforementioned approach of using

---

\(^5\)https://www.mysql.com/
temporary copies of graphs for storing versions and deltas tends to be rather costly when querying the data, as only medium sized data sets can be handled by R43ples. In fact queries on datasets with more than a few thousand triples take longer than most users are willing to wait [9].

TAILR [17] is a platform for preserving the history of arbitrary linked datasets over time, implemented as a Python web application. It follows the hybrid approach for storing the data: the history of each individual tracked resource is encoded as a series of deltas or deletes based on interspersed snapshots. More specifically, their storage model consists of repositories, changesets and blobs. A repository can be created by users and is actually the linked dataset along with its history. A changeset encodes the information about modifications that happen to the data at a particular time point. According to the versioning 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 try to minimize the storage and retrieval cost of versions. 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 6.

In their experimental evaluation, the authors measured the response times of their Push API and MEMENTO API [23] as well as the growth of the required storage space for an increasing number of versions. For their experiments they used a random sample of 100K resources selected from each version of DBpedia [2] 3.2 to 3.9. Regarding the Push API response times, push requests for the first release took the longest time on average. MEMENTO API response times tend to slightly increase for later revisions due to the longer base/delta chains that result to higher reconstruction costs. Finally, the storage overhead is directly related to the nature of the data and especially to the delta encoding.

A VERSION MANAGEMENT FRAMEWORK FOR RDF TRIPLE STORES [11] is a framework for managing RDF versions on top of relational databases (where all triples are stored in one large triple table). The framework follows the delta-based approach as it stores the last version and the deltas that led to it. To improve the performance of cross-delta queries, the authors introduce aggregated deltas, which associate the latest version with each of the previous ones (not only the last one); obviously, this comes at the cost of increasing space (storage) requirements. The delta of each version is separately stored in an INSERT and a DELETE relational table, so a version can be constructed on the fly using appropriate SQL statements.

For the evaluation, Uniprot dataset [22] versions v1-v9 were used. The implementation employed the Oracle 11g Enterprise edition 7. In the evaluation, the authors compared the proposed approach with the full materialization and sequential deltas approaches. The authors conducted experiments related to storage overhead, version construction and delta computation times, compression ratio and query performance. As expected, the approach is less efficient than the sequential deltas, but outperforms the full materialization approach regarding storage space and deltas computation time. Moreover, their approach highly outperforms the sequential delta approach regarding version reconstruction. In particular, while construction time in the sequential delta approach is proportional to the number of past versions that must be considered, the aggregated delta can compute any version almost at constant time (with respect to the number of past versions). Regarding the query answering performance, the full materialization approach has the best performance for the types of queries that refer to specific versions, but the aggregated delta approach outperforms the sequential delta approach in most cases.

---

6 https://mariadb.org/
DYDRA [1] is an RDF graph service in the cloud that stores and retrieves RDF data through SPARQL, LDF\textsuperscript{8} and LDP\textsuperscript{9} interfaces. In order to have access to the previous store states, in addition to the current one, the service includes a REVISION clause analogous to GRAPH SPARQL clause, as each versioned dataset is stored in its own named graph in a quad store. Queries supported by DYDRA are characterized according to two dimensions: the dataset constitution, and algebra combination. Dataset constitution determines the revisions that are included in the target dataset, and may be: none, single, multiple, a range or a difference of versions and algebra combination concerns how the query combines the later versions.

2.4 Versioning Benchmarks

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 (1) open and easily accessible from all third parties that are interested in testing their systems. Second, it has to be (2) unbiased, which means that there should not exist a conflict of interest between the creators of the benchmark and the creators of the system(s) under test. These features guarantee a fair and reproducible evaluation of the systems under test. To guarantee (additionally) that the benchmark will produce useful results, it should be highly (3) configurable and scalable, in order to cope with the different characteristics and needs of each system. Pertaining to our focus on benchmarks for versioning systems, configurability and scalability may refer to the number of versions that a data generator can produce, the size of each version, the number of changes from version to version etc.

In addition, the benchmark should be (4) agnostic to the different strategies followed by the versioning system. For example, for benchmarks related to RDF versioning systems, one should also take into account the different strategies that are being employed, and should be agnostic with regards to the strategy that an RDF versioning 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 (5) extensible, to be able to test additional features or requirements for a versioning system that may appear in the future. To the best of our knowledge, there have been only two proposed benchmarks for RDF versioning systems in the literature, which are described in detail below.

BEAR [8, 7] 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. Based on the analysis of these RDF versioning strategies, a set of directions is provided that must be followed when evaluating the efficiency of RDF versioning systems, many of which are similar to the requirements we outlined above. The first BEAR directive for benchmarks is that the benchmarks should be agnostic with respect to the used versioning strategy in order for the comparison to be fair. Secondly, queries have to be simple and become more complex as the strategies and systems are better understood. And finally, the benchmark should be extensible as lessons learnt from previous work and new retrieval features arise. As a basis for comparing the different versioning strategies, the BEAR benchmark introduces four features that describe the dataset configuration:

- **Data dynamicity** measures the number of changes between versions, and is described via the

\textsuperscript{8}linkeddatafragments.org/

\textsuperscript{9}https://www.w3.org/TR/ldp/
change ratio and the data growth. The change ratio quantifies how much (what proportion) of the dataset changes from one version to another and the data growth determines how its size changes from one version to another.

- **Data static core** contains the triples that exist in all dataset versions.
- **Total version-oblivious triples** computes the total number of different triples in an archive, independently of their timestamp (i.e., the version in which they appear).
- **RDF vocabulary** represents the different subjects, predicates and objects in an RDF archive.

Regarding the generation of the queries of the benchmark, the result cardinality and selectivity of the query should be considered, keeping in mind that the results of a query can highly vary among different versions. For example, by selecting queries with similar result cardinality and selectivity, one could guarantee that potential retrieval differences in response times could be attributed to the versioning strategy. In order to be able to judge the different systems, authors of BEAR introduced various categories of queries, which are similar to the ones we discuss in Section 2.2 and have been used as a source of inspiration for our categorization. In particular, the authors propose queries on versions (i.e., modern and historical version materialization queries), deltas (delta materialization and structured queries), as well as the so-called change materialization queries, which essentially check the version in which the answer to a query changes with respect to previous versions. Even though BEAR provides a detailed theoretical analysis of the features that are useful in the process of designing a benchmark, it fails to satisfy one of the five requirements that we have previously set. In particular, its data workload is composed of a static dataset, so BEAR is not a benchmark generator; thus, the requirement (3) of configurability and scalability is not met.

**EVOGEN** [16] is a generator for evolving RDF data that is used for benchmarking versioning and change detection systems. EVOGEN is based on the LUBM generator [10], by extending its schema with 10 new RDF classes and 19 new RDF properties in order to support schema evolution. Their benchmarking methodology is based on a set of requirements and parameters that affect the data generation process, the context of the tested application and the query workload, as required by the nature of the evolving data. EVOGEN is a Benchmark Generator, and is extensible and highly configurable in terms of the number of generated versions and the number of changes occurring from version to version. Similarly, the query workload is generated adaptively to the data generation process. EVOGEN takes into account the versioning strategy of the system under test, by providing adequate input data formats (full versions, deltas, etc.) as appropriate.

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

In EVOGEN, the user is able to choose the output format of the generated data by allowing him to request fully materialized versions or deltas; this allows supporting (and testing) systems employing different versioning strategies. The query workload produced by EvoGen leverages the 14 LUBM queries, appropriately adapted to apply for evolving versions. In particular, EvoGen generates the following six types of queries, which are based on the previous generated data and their characteristics:
• Retrieval of a diachronic dataset: a query asking for all the triples in all versions of a dataset.
• Retrieval of a specific version: a query requesting all triples in a specific version (i.e., modern or historical version materialization queries).
• Snapshot queries on the data, i.e., queries accessing a single version (single-version historical queries).
• Longitudinal (temporal) queries that retrieve the timeline of particular subgraphs, through a subset of past versions (cross-version structured queries).
• Queries on changes, which access the deltas (delta materialization or single-delta structured queries).
• Mixed queries which use a mix of sub-queries from the above types (hybrid queries).

EVOGEN is, in practice, a more complete benchmark, as it is a 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 benchmarked systems store information about the low level deltas (addition/deletion of classes, addition/deletion of class instances etc.) in order to be answered. Moreover, to successfully answer the 14 original LUBM queries, the benchmarked systems are required to support reasoning (forward or backward). As a result, versioning systems that do not support reasoning functionalities fail to answer the majority (11 of 14) of generated queries.

3 Versioning Benchmark

In this Section we are going to present the versioning benchmark SPBv, we developed in the context of the HOBBIT project. The full source code of the benchmark can be found on the HOBBIT github page. In Figure 6 we present a short overview of the architecture of SPBv, the components of which will be described in detailed in the following Sections.

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 - taxonomies and databases that include relevant concepts, entities and factual information. This metadata allows publishers to efficiently retrieve relevant content, according to their various 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, etc.

The outline of this Section is the following: in Section 3.1 we present the choke points, a.k.a. technical difficulties that we believe a benchmark should consider in order to challenge the systems under test, data and task generation of SPBv is presented in Sections 3.2 and 3.3 while the KPIs we use for SPBv are discussed in Section 3.4.

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 [5]. So, a benchmark can be characterized as valuable
if its workload stresses those choke points that systems should manage. In Versioning Benchmark the following choke points are considered:

- **VCP1: Storage Space** tests the ability of the systems to efficiently handle storage space growth as new versions are stored.
- **VCP2: 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.
- **VCP3: Parallel Version Reconstruction** tests the ability of the systems, that follow the delta-based or hybrid archiving strategies, to reconstruct in parallel versions when a query asks for information from more than one versions.
- **VCP4: Partial Delta Computation** tests the ability to compute in parallel deltas when a query asks for information from more than one deltas.
- **VCP5: 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 Versioning Benchmark extends SPB’s data generator. SPB’s data generator uses seven **core** and three **domain** RDF ontologies for the data production. The former ontologies define the main entities and their properties, required to describe essential concepts of the benchmark namely, creative works, persons, documents, BBC products (news, music, sport, education, blogs), annotations (tags), provenance of resources and content management system information. Domain ontologies are used to express concepts or properties from a domain of interest such as football, politics, entertainment among others. Also, a set of reference datasets are employed by the data generator to produce the data of interest. These datasets are snapshots of the real datasets provided by BBC; in addition, a GeoNames and DBpedia reference dataset has been included for further enriching the annotations with geo-locations to enable the formulation of geo-spatial queries, and person data. Core and domain ontologies are shown in Table 3. The Creative Works core ontology which is the most important one is shown in details in Figure 2.

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 a meta-data about a real entity (or entities) that exist in reference datasets. A creative work which is the main class (cworkCreativeWork) of the Creative Work ontology, as shown in Figure 2, collects all RDF descriptions of creative works created by the publisher’s editorial team. This class is defined as a subclass of **core:Thing** (subclass of owl:Thing), allowing in this way the creation of complex information graphs. A creative work has a number of properties such as cworktitle, cworkshortTitle,
Figure 2: BBC Creative Works Ontology

Figure 3: BBC Company Ontology
cwork:description, cwork:dateCreated, cwork:audience, cwork:format among others; it has a category (property cwork:category) and can be tagged (property cwork:tag) with anything (i.e., instances of class owl:Thing). The latter property is further specialized (through the rdfs:subPropertyOf relation) to properties cwork:about and cwork:mentions which hold the references to entities. That way a creative work provides meta-data about one or several entities - facts about them and defines relations between them. Creative works can be instances of classes cwork:NewsItem, cwork:Programme and cwork:BlogPost, all defined as subclasses of class cwork:CreativeWork.

SPB’s data generator models three types of relations in the data, as described latter and shown in Figure 4.

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. The number of creative works 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 clusterings with sizes (i.e., number of creative works) of different magnitude. Example of clusterings of data could be news items that are about events starting with a high number of journalistic assets related to them and following a decay in time as they reach the end of time period, a tendency that mirrors a real world scenario in which a 'fresh' event is popular and its popularity decreases as time goes by.

Correlations of entities This correlation effect is produced by generating creative works about two or three entities from reference data in a fixed period of time. Each of the entities is tagged by creative works solely at the beginning and end of the correlation period and in the middle of it, both are used as tags for the same creative work. Such an example of data correlation could be that several 'popular' persons (e.g., Macron and Le Pen) are mentioned together by creative works for a certain period of time.

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 a 5% of all the resources from reference data and mark them as popular when the remaining ones are marked as regular. When creating creative works, 30% percent of them are tagged with randomly selected popular resources and the remaining 70% are linked to the regular ones. Example for random taggings could be every-day events which become less important several days after their start date. This random generation of data concerns only one third of all generated data; the remaining data is generated with correlations and clustering effects modeled as previously described.

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. In particular it extends 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). In its second version, the data generator of SPBv will also support the change of already existing creative works.

The following parameters can be set to configure Versioning Benchmark’s data generator (as shown in Figure 5).

1. Generator seed: used to set the random seed for the data generator
3. Seed year: defines a seed year that will be used as starting point for generating the Creative Works date properties
4. *Generation period*: the period of generated data in years
5. *Size*: defines the size of generated synthetic data in triples produced by the data-generator
6. *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 to extract such entities from the previously described reference datasets. To do so, all instances from the different domain ontologies that exist in the reference datasets are identified. The identification process was a one-time offline process, consisted of the execution of queries that collected data about the stored entities. Once those instances are identified, they are stored in several files in order to be used by the data generator for tagging a generated creative work. After the extraction of entities, the versioning data generator proceeds as follows:

- Retrieve entities from the appropriate file.
- Retrieve DBpedia locations from the appropriate file.
- Retrieved Geonames locations from the appropriate file.
- Select the *popular* and *regular* entities from the previous set of retrieved instances.
- Adjust the number of major/minor events and number of correlations, according to the total size of the dataset that will be generated. This is done in order to let the ratio of the three types of modelled data (clusterings, correlations, random) to be 33%, 33% and 33% respectively.
- Distribute to all available data generators the major/minor events and correlations, as each generator has to produce the whole event/correlation in order for the event to be valid. This is a necessary step, since all benchmarks offered by the HOBBIT platform can retrieve their data through multiple data generators that run in parallel.
- Each data generator 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 6.
- Generate SPARQL queries based on the already generated data, and send them to the Task Generator component, as shown in Figure 6.
- Compute the Gold Standard in order to be able to test the validity of benchmarked system’s results and send it to the Evaluation Storage component.

Figure 4: *Data generator, types of produced models in generated data.*
3.3 Task Generation

As shown in Figure 6, the Task Generators of SPBv are responsible to provide all the tasks, that should be solved, to the benchmarked system which in its turn, send the results to the Evaluation Storage component. In more details there are three types of tasks:

- **Ingestion tasks** are those tasks that trigger the system to report the time required for loading a new coming version.
- **Storage space task** prompts the system to report the total storage space overhead for storing the different versioned datasets.
- **Query performance tasks**: For each one of the eight versioning query types, that were previously described (Section 2.2), a set of SPARQL queries is generated. In particular, for each
query type there is a SPARQL query that uses templates for its parameterization. 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 and is given in the configuration of the benchmark, as shown in Figure 5. Given that there is neither a standard language, nor an official SPARQL extension for querying RDF versioned data, for defining the aforementioned query templates, we 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.

In Table 4 we provide the query tasks that were produced by the task generators. For each of them we provide a text description and we enumerate the versioning choke points (VCPs - Section 3.1) it addresses. Note, that for query types related to structured queries we chose to use 4 of the 12 queries of SPB basic workload as a baseline. All the 12 queries are fully described in [14]. The reason we ended up to those 4 queries was that they fulfill all the choke points [14] SPB queries implement. The SPARQL representation of all query templates can be found in Appendix A.

<table>
<thead>
<tr>
<th>Query</th>
<th>Query Type</th>
<th>Description</th>
<th>Choke Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Modern version materialization</td>
<td>The full current version of the dataset is retrieved</td>
<td>-</td>
</tr>
<tr>
<td>Q2.1</td>
<td>Modern single-version query</td>
<td>Retrieve from the current version, the 10 most recent creative works that are about or mention different topics.</td>
<td>VCP2</td>
</tr>
<tr>
<td>Q2.2</td>
<td>Modern single-version structured query</td>
<td>Retrieve from the current version, those creative works that are instances of classes BlogPost or NewsItem, are about a specific topic and their value of primaryFormat is one of TextualFormat, InteractiveFormat or PictureGalleryFormat. If the creative work has an audience then the creative work should be obtained using this value. Given the retrieved list of creative works order the result in descending order of the value of their creationDate property. The size of the resultset is limited by a random number between 5 and 20.</td>
<td>VCP2</td>
</tr>
<tr>
<td>Q2.3</td>
<td>Modern single-version structured query</td>
<td>Retrieve all creative works, of a specific type from the current version, that are tagged with a topic and order them by creation date. The size of the resultset is limited by a random number between 5 and 20.</td>
<td>VCP2</td>
</tr>
<tr>
<td>Q2.4</td>
<td>Modern single-version structured query</td>
<td>Retrieve from the current version, all creative works that contain a certain word in their title or description. The size of the resultset is limited by a random number between 1 and 1000.</td>
<td>VCP2</td>
</tr>
<tr>
<td>Q3</td>
<td>Historical version materialization</td>
<td>A full past version of the dataset is retrieved.</td>
<td>-</td>
</tr>
<tr>
<td>Q4.1</td>
<td>Historical single-version structured query</td>
<td>Retrieve the 10 most recent creative works, that are about or mention different topics of a specific past version.</td>
<td>VCP2</td>
</tr>
<tr>
<td>Q4.2</td>
<td>Historical single-version structured query</td>
<td>Retrieve from a past version, those creative works that are instances of classes BlogPost or NewsItem, are about a specific topic and their value of primaryFormat is one of TextualFormat, InteractiveFormat or PictureGalleryFormat. If the creative work has an audience then the creative work should be obtained using this value. Given the retrieved list of creative works order the result in descending order of the value of their creation date property. The size of the resultset is limited by a random number between 5 and 20 (full text query).</td>
<td>VCP2</td>
</tr>
<tr>
<td>Q4.3</td>
<td>Historical single-version structured query</td>
<td>Retrieve all creative works, of a specific type from a past version, that are tagged with a topic and order them by creation date. The size of the resultset is limited by a random number between 5 and 20.</td>
<td>VCP2</td>
</tr>
<tr>
<td>Q4.4</td>
<td>Historical single-version structured query</td>
<td>Retrieve all creative works from a specific past version, which contain a certain word in their title or description. The size of the resultset is limited by a random number between 1 and 1000.</td>
<td>VCP2</td>
</tr>
<tr>
<td>Q5</td>
<td>Delta materialization</td>
<td>Retrieve the differences between two given versions.</td>
<td>VCP5</td>
</tr>
<tr>
<td>Q6</td>
<td>Single-delta structured query</td>
<td>Retrieve the descriptions of the latest creative works tagged with a specific location. Consider that the description of each specific CreativeWork is stored in a dedicated named graph. The result should include only the explicit statements about the creative work, without owl:sameAs equivalence and without statements inferred otherwise.</td>
<td>VCP4, VCP5</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Q7</td>
<td>Cross-delta structured query</td>
<td>Retrieve the average number of added News Items from one version to another between versions Vi and Vj.</td>
<td>VCP4, VCP5</td>
</tr>
<tr>
<td>Q8.1</td>
<td>Cross-version structured query</td>
<td>Retrieve the 10 most recent creative works, that are about or mention different topics from a set of specific past versions.</td>
<td>VCP2, VCP3</td>
</tr>
<tr>
<td>Q8.2</td>
<td>Cross-version structured query</td>
<td>Retrieve from a set of different versions, those creative works that are instances of classes BlogPost or NewsItem, are about a specific topic and their value of primaryFormat is one of TextualFormat, InteractiveFormat or PictureGalleryFormat. If the creative work has an audience then the creative work should be obtained using this value. Given the retrieved list of creative works order the result in descending order of the value of their creationDate property. The size of the resultset is limited by a random number between 5 and 20.</td>
<td>VCP2, VCP3</td>
</tr>
<tr>
<td>Q8.3</td>
<td>Cross-version structured query</td>
<td>Retrieve all creative works, of a specific type from set of specific versions, that are tagged with a topic and order them by creation date. The size of the resultset is limited by a random number between 5 and 20.</td>
<td>VCP2, VCP3</td>
</tr>
<tr>
<td>Q8.4</td>
<td>Cross-version structured query</td>
<td>Retrieve all creative works from a set of specific versions, which contain a certain word in their title or description. The size of the resultset is limited by a random number between 1 and 1000.</td>
<td>VCP2, VCP3</td>
</tr>
</tbody>
</table>

Table 4: Versioning Queries

### 3.4 Evaluation Module

Finally, as we can see in Figure 6 by having in Evaluation Storage both the gold standard and the results from the benchmarked system, we are ready to evaluate the system’s performance. Analogous to the types of tasks, there are three performance metrics that can be used to evaluate a versioning system:

1. The space required to store the different versioned datasets is measured. Such metric is essential to understand whether a system can choose the best strategy (e.g. full materialization, delta-based, annotated triples or hybrid, as explained in Section 2.1) for storing the versions.
2. The time that a system needs for storing a new coming 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 defined the following *KPIs*:

- **Initial version ingestion speed** (in triples per second): The total triples that can be loaded per second for the dataset’s initial version. The reason why we distinguish it from the ingestion speed of other versions is that the initial version is in most cases much larger than the ones that follow, so potential optimization may be applied by the versioning systems.

- **Applied changes speed** (in changes per second): Every new coming version, other than the initial one, contains a set of changes (additions or deletions of triples). This KPI measures the average number of such changes that could be stored by the benchmarked systems per second after the loading of all new versions.

- **Storage cost** (in KB): This KPI measures the total storage space required in order to store all versions.

- **Average Query Execution Time** (in ms): The average execution time, in milliseconds for the query.

4 Experiments

For the first version of Versioning benchmark we used OpenLink Virtuoso Opensource\(^{11}\) for testing the benchmark’s implementation on top of the HOBBIT platform. In particular for our System Adapter we used the docker image tenforce/virtuoso\(^{12}\) which dockerize the Virtuoso 7.2.4 release. Virtuoso does not support versioning, but we easily implemented the full materialization versioning strategy (see Section 2.1), by assuming that each version is represented in its own named graph. For tasks related to the ingestion of new triples we used the bulk loading process\(^{13}\) offered by Virtuoso, with 12 RDF loaders so that we can parallelize the data load and hence maximize loading speed.

For our experiments four datasets of different sizes that correspond to around 100K, 500K, 1M and 5M triples produced. 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 divided in 5 different versions of equal time intervals of around of 2 and half months. Regarding the query workload, for each query type, 5 different queries where produced and the average execution time of such queries was computed. All the configuration parameters that were selected for our experiments, except of the varying dataset size, are shown in Figure 7.

In order to get fair enough results we run three experiments per dataset size and we computed the average values for all reported results. E.g. in Figure 8 we can see the output results from the HOBBIT platform interface after executing the three experiments of datasets consisting of 1M triples.

In the left histogram of Figure 9 we can see for all datasets the speed with which the triples of their initial version was inserted into Virtuoso triple store. 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, increase. This is an expected result, as Virtuoso bulk loads files containing much more triples, as dataset size increase. 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. As we observe here, the changes

\(^{11}\)https://virtuoso.openlinksw.com/

\(^{12}\)https://hub.docker.com/r/tenforce/virtuoso/

\(^{13}\)https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtBulkRDFLoader
Figure 7: Configuration parameters

Figure 7: Configuration parameters

are applied much slower than the triples of the initial version. This is happening due to the archiving strategy of *Full Materialization* (Section 2.1) that is used in this case. Remember 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.

In Figure 10 we can see the storage space required for storing the data for all different datasets. Such space, expectedly increases as the total number of triples increase, from 30 MB to 1450 MB. This significant overhead on storage space is due to the archiving strategy used (i.e., Full Materialization).

In Figures 11, 12, 13, 14 and 15 we present for each of the eight versioning query types, as these are described in Section 2.2, their average execution time (in ms) after executing the five queries per query type and the three experiments per dataset size.

In Figure 11 we can see the time required for materializing I) the modern (current) version II) a 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 2nd 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 hardcoded 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 12, 13, 14 and 15 we can see the execution times for all types of structured queries. In most 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 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 A, are all of the same form, we observe that the earlier 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.
Figure 11: Execution times for materialization queries (in ms)

Figure 12: Execution times for single-version structured queries (in ms)

Figure 13: Execution times for single-delta structured queries (in ms)

Figure 14: Execution times for cross-version structured queries (in ms)

Figure 15: Execution times for cross-delta structured queries (in ms)
5 Conclusions and Next Steps

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

In the second version of SPBv 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 modification of existing data as well. Furthermore, we will let the benchmarked systems to decide the generated data format, according to the versioning strategy they implement. So, if a system implements the full materialization versioning strategy, will receive the generated data as separate versions. On the other hand, if a system implements the delta-based strategy 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, as the delta-based ones, 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 versioning strategy they implement. Finally for disseminating our benchmark we plan to implement more System Adapters, just like we did for Virtuoso, by emphasizing on the already existing versioning systems that were described in the report.

Appendices

A Query Templates

```
SELECT ?s ?p ?o
WHERE {
  GRAPH {{(graphVmax)}} { ?s ?p ?o }
}
```

Listing 1: Modern version materialization query template

```
SELECT ?creativeWork
FROM {{(graphVmax)}}
WHERE {
  ?creativeWork {{cwAboutOrMentions}}} {{cwAboutOrMentionsUri}}
    .
  ?creativeWork a cwork:CreativeWork ;
    cwork:dateCreated ?dateCreated ;
    cwork:description ?description ;
    cwork:primaryFormat ?primaryFormat ;
  OPTIONAL { ?creativeWork cwork:shortTitle ?shortTitle . }
  OPTIONAL { ?creativeWork cwork:mentions ?mentions . }
  OPTIONAL { ?creativeWork cwork:altText ?altText . }
}
ORDER BY DESC(?dateCreated)
LIMIT 10
```

Listing 2: Modern single-version structured query template 1
```sql
SELECT DISTINCT ?creativeWork
FROM {{graphVmax}}
WHERE {
  ?creativeWork cwork:dateCreated ?created .
  ?creativeWork cwork:about {{cwAboutUri}} .
  { ?creativeWork cwork:primaryFormat cwork:TextualFormat . }
  UNION
  { ?creativeWork cwork:primaryFormat cwork:InteractiveFormat . }
  UNION
  { ?creativeWork cwork:primaryFormat cwork:PictureGalleryFormat . }
  { ?creativeWork a cwork:NewsItem . }
  UNION
  { ?creativeWork a cwork:BlogPost . }
  OPTIONAL { ?creativeWork cwork:audience ?audience }.
  FILTER (!BOUND(?audience) || ?audience = {{cwAudience}}) .
}
ORDER BY DESC(?created)
LIMIT {{randomLimit}}
```

Listing 3: Modern single-version structured query template 2

```sql
SELECT DISTINCT ?creativeWork
FROM {{graphVmax}}
WHERE {
  ?creativeWork cwork:about {{cwAboutUri}} ;
  cwork:dateCreated ?created ;
  cwork:primaryFormat {{cwFormat}} ;
  a {{cwType}} .
}
ORDER BY DESC(?created)
LIMIT {{randomLimit}}
```

Listing 4: Modern single-version structured query template 3

```sql
SELECT DISTINCT ?creativeWork
FROM {{graphVmax}}
WHERE {
  ?type rdfs:subClassOf cwork:CreativeWork .
  ?creativeWork a ?type ;
  cwork:title ?title ;
  cwork:description ?description .
  OPTIONAL { ?creativeWork cwork:dateCreated ?dateCreated . }
  OPTIONAL { ?creativeWork cwork:category ?category . }
            ?pco bbc:webDocumentType ?webDocType . }
  FILTER (CONTAINS(?title, {{{word}}}) || CONTAINS(?description, {{{word}}})).}
ORDER BY DESC(?dateCreated)
LIMIT {{randomLimit}}
```

Listing 5: Modern single-version structured query template 4
SELECT  ?s  ?p  ?o
WHERE  { GRAPH  { {{graphVxMinusGraphVmax}}}  { ?s  ?p  ?o } }

Listing 6: Historical version materialization query template

SELECT  ?creativeWork
FROM  { {{graphVxMinusGraphVmax}}}  WHERE  {
?creativeWork  {{cwAboutOrMentions}}}  {{cwAboutOrMentionsUri}}} .
?creativeWork  a  cwork:CreativeWork ;
  cwork:dateCreated  ?dateCreated ;
  cwork:description  ?description ;
  cwork:primaryFormat  ?primaryFormat ;
OPTIONAL  { ?creativeWork  cwork:shortTitle  ?shortTitle . }
OPTIONAL  { ?creativeWork  cwork:mentions  ?mentions . }
OPTIONAL  { ?creativeWork  cwork:altText  ?altText . }
}
ORDER BY DESC(?dateCreated)
LIMIT  10

Listing 7: Historical single-version structured query template 1

SELECT DISTINCT  ?creativeWork
FROM  { {{graphVxMinusGraphVmax}}}  WHERE  {
?creativeWork  cwork:dateCreated  ?created .
?creativeWork  cwork:about  {{cwAboutUri}} .
{
  { ?creativeWork  cwork:primaryFormat  cwork:TextualFormat . }
  UNION
  { ?creativeWork  cwork:primaryFormat  cwork:InteractiveFormat . }
  UNION
  { ?creativeWork  cwork:primaryFormat  cwork:PictureGalleryFormat . }
}
{
  { ?creativeWork  a  cwork:NewsItem . }
  UNION
  { ?creativeWork  a  cwork:BlogPost . }
}
FILTER  (!BOUND(?audience)  ||  ?audience  =  {{cwAudience}}) .
}
ORDER BY DESC(?created)
LIMIT  { {{randomLimit}}} 

Listing 8: Historical single-version structured query template 2

SELECT DISTINCT  ?creativeWork
FROM  { {{graphVxMinusGraphVmax}}}  WHERE  {
?creativeWork  cwork:about  {{cwAboutUri}} ;
  cwork:dateCreated  ?created ;
  cwork:primaryFormat  {{cwFormat}} ;
a  {{cwType}} .
}
ORDER BY DESC(?created)
Listing 9: Historical single-version structured query template 3

```sql
LIMIT {{randomLimit}}
```

Listing 10: Historical single-version structured query template 4

```sql
SELECT * WHERE {
    SELECT * WHERE {
        GRAPH {{{graphVx}}} { ?s ?p ?o }
        MINUS { GRAPH {{{graphVyExcludeVx}}} { ?s ?p ?o } }
    }
    UNION {
        SELECT * WHERE {
            GRAPH {{{graphVyExcludeVx}}} { ?s ?p ?o }
            MINUS { GRAPH {{{graphVx}}} { ?s ?p ?o } }
        }
    }
}
```

Listing 11: Delta materialization query template

```sql
SELECT ((?cnt_2 - ?cnt_1) AS ?blog_posts_diff) WHERE {
    SELECT (COUNT(*) AS ?cnt_1) WHERE {
        GRAPH {{{graphVxMinusGraphVmax}}} { ?creativeWork a cwork:BlogPost }
    }

    SELECT (COUNT(*) AS ?cnt_2) WHERE {
        GRAPH {{{graphVxPlus1}}} { ?creativeWork a cwork:BlogPost }
    }
}
```

Listing 12: Single-delta structured query template

```sql
WHERE {
    SELECT (COUNT(*) AS ?ViPlus1_news_items_added ) WHERE {
        GRAPH {{{graphViPlus1}}} { ?creativeWork a cwork:NewsItem }
        MINUS { GRAPH {{{graphVi}}} { ?creativeWork a cwork:NewsItem } }
    }
```
Listing 13: Cross-delta structured query template

```
(SELECT (COUNT(*) AS ?Vj_news_items_added)
WHERE {
  GRAPH {{{graphVj}}} { ?creativeWork a cwork:NewsItem }
  MINUS { GRAPH {{{graphVjMinus1}}} { ?creativeWork a cwork:NewsItem } }
} .
)
```

Listing 14: Cross-version structured query template 1

```
SELECT ?creativeWork
FROM {{{graphVi}}} {{{fromIntermediateVersionsClauses}}} FROM {{{graphVj}}}
WHERE {
  ?creativeWork {{{cwAboutOrMentions}}} {{{cwAboutOrMentionsUri}}} .
  ?creativeWork a cwork:CreativeWork ;
      cwork:dateCreated ?dateCreated ;
      cwork:description ?description ;
      cwork:primaryFormat ?primaryFormat ;
  OPTIONAL { ?creativeWork cwork:shortTitle ?shortTitle . }
  OPTIONAL { ?creativeWork cwork:mentions ?mentions . }
  OPTIONAL { ?creativeWork cwork:altText ?altText . }
}
ORDER BY DESC(?dateCreated)
LIMIT 10
```

Listing 15: Cross-version structured query template 2

```
SELECT DISTINCT ?creativeWork
FROM {{{graphVi}}} {{{fromIntermediateVersionsClauses}}} FROM {{{graphVj}}}
WHERE {
  ?creativeWork cwork:dateCreated ?created .
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  { ?creativeWork cwork:primaryFormat cwork:TextualFormat . } UNION
  { ?creativeWork cwork:primaryFormat cwork:InteractiveFormat . }
  UNION
  { ?creativeWork cwork:primaryFormat cwork:PictureGalleryFormat . }
}
{ ?creativeWork a cwork:NewsItem . }
UNION { ?creativeWork a cwork:BlogPost . }
OPTIONAL { ?creativeWork cwork:audience ?audience . }
FILTER (!BOUND(?audience) || ?audience = {{{cwAudience}}} .)
}
ORDER BY DESC(?created)
LIMIT {{{randomLimit}}}
```
SELECT DISTINCT ?creativeWork
FROM {{graphVi}}
                     {{fromIntermediateVersionsClauses}}
FROM {{graphVj}}
WHERE {
   ?creativeWork cwork:about {{{cwAboutUri}}} ;
   cwork:dateCreated ?created ;
   cwork:primaryFormat {{{cwFormat}}} ;
   a {{{cwType}}}. 
}
ORDER BY DESC(?created)
LIMIT {{randomLimit}}

Listing 16: Cross-version structured query template 3

SELECT DISTINCT ?creativeWork
FROM {{(graphVi)}}
                     {{fromIntermediateVersionsClauses}}
FROM {{(graphVj)}}
WHERE {
   ?type rdfs:subClassOf cwork:CreativeWork .
   ?creativeWork a ?type ;
   cwork:title ?title ;
   cwork:description ?description .
OPTIONAL { ?creativeWork cwork:dateCreated ?dateCreated . }
OPTIONAL { ?creativeWork cwork:category ?category . }
OPTIONAL {
}
FILTER (CONTAINS(?title, {{{word}}}) || CONTAINS(?description, {{{word}}})) .
}
ORDER BY DESC(?dateModified)
LIMIT {{randomLimit}}

Listing 17: Cross-version structured query template 4

References


