Deliverable 5.2.2:
Second 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 30, 29/05/2018</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 30, 29/05/2018</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>Other</td>
</tr>
<tr>
<td>Approval Status</td>
<td>Final</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Abstract: This deliverable presents the final version of the Semantic Publishing Versioning Benchmark (SPVB) for the HOBBIT platform.

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.

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>11/05/2018</td>
<td>First draft created</td>
<td>Vassilis Papakonstantinou (FORTH)</td>
</tr>
<tr>
<td>0.1</td>
<td>11/05/2018</td>
<td>First version</td>
<td>Vassilis Papakonstantinou (FORTH)</td>
</tr>
<tr>
<td>0.2</td>
<td>23/05/2018</td>
<td>Peer review comments</td>
<td>Gayane Sedrakyan (imec) and Erik Mannens (imec)</td>
</tr>
<tr>
<td>0.3</td>
<td>25/05/2018</td>
<td>Reviewed Version</td>
<td>Vassilis Papakonstantinou (FORTH)</td>
</tr>
<tr>
<td>1.0</td>
<td>29/05/2018</td>
<td>Final version</td>
<td>Vassilis Papakonstantinou (FORTH), Giorgos Flouris (FORTH) and Irini Fundulaki (FORTH)</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>Giorgos Flouris</td>
<td><a href="mailto:fgeo@ics.forth.gr">fgeo@ics.forth.gr</a></td>
</tr>
</tbody>
</table>
Executive Summary

This report discusses the second version of HOBBIT’s Semantic Publishing Versioning Benchmark (SPVB). It covers the workload conducted until M36. The SPVB 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 .......................................................... 7

2 Background .......................................................... 8
   2.1 Versioning Strategies .......................................... 8
   2.2 Query Types .................................................. 8

3 Semantic Publishing Versioning Benchmark ................. 9
   3.1 Data Generation ............................................... 10
      3.1.1 Configuration Parameters ................................. 10
      3.1.2 Producing synthetic added triples ...................... 10
      3.1.3 Producing synthetic deleted triples .................... 13
      3.1.4 Evolution of real DBpedia data ......................... 13
   3.2 Tasks Generation ............................................. 14
      3.2.1 QT1 - Modern version materialization (Appendix A.1) .......... 15
      3.2.2 QT2 - Modern single-version structured queries (Appendix A.2) .......... 15
      3.2.3 QT3 - Historical version materialization (Appendix A.3) .......... 16
      3.2.4 QT4 - Historical single-version structured queries (Appendix A.4) .......... 16
      3.2.5 QT5 - Delta materialization (Appendix A.5) & QT6 - Single-delta structured query (Appendix A.6) .......... 17
      3.2.6 QT7 - Cross-delta structured queries (Appendix A.7) .......... 17
      3.2.7 QT8 - Cross-version structured queries (Appendix A.8) .......... 18
   3.3 Evaluation Module ........................................... 21

4 Experiments .......................................................... 21
   4.1 Benchmarked systems .......................................... 21
      4.1.1 OSTRICH .................................................. 22
      4.1.2 R43PLES 0.8.8 ............................................ 22
      4.1.3 VIRTUOSO ................................................ 22
      4.1.4 APACHE JENA FUSEKI 3.6.0 ............................... 23
      4.1.5 GRAPHDB Free 8.5 ...................................... 23
      4.1.6 BLAZEGRAPH ............................................. 23
   4.2 Experimental setup ............................................ 23
   4.3 Results ......................................................... 24
4.3.1 MOCHA-ESWC2018 results ........................................... 29

5 Conclusions and Future work ........................................... 33

Appendices ........................................................................ 34

A Query Templates ............................................................ 34
   A.1 QT1 - Modern version materialization query template .......... 34
   A.2 QT2 - Modern single-version structured query templates ....... 34
   A.3 QT3 - Historical version materialization query template ....... 36
   A.4 QT4 - Historical single-version structured query templates .... 37
   A.5 QT5 - Delta materialization query template ....................... 39
   A.6 QT6 - Single-delta structured query template ..................... 39
   A.7 QT7 - Cross-delta structured query template ..................... 40
   A.8 QT8 - Cross-version structured query templates ................ 40

References ........................................................................... 43
List of Figures

1. Different queries organized by focus and type. [14] ................................. 8
2. Configuration parameters of SPVB ............................................................... 11
3. Architecture of SPVB ................................................................................. 12
4. Replacement value for $\text{graphVmax}$ of QT1 and QT2, assuming that we have 10 versions ................................................................. 15
5. Replacement values for $\text{graphVhistorical}$ of QT3 and QT4, assuming that we have 10 versions ................................................................. 17
6. Replacement values for $\text{graphVmax}$ (green) and $\text{graphVhistorical}$ (red) of QT5 and QT6, assuming that we have 10 versions ......................................................... 18
7. Replacement values for $\text{graphVi}$ (green), $\text{graphViPlus1}$ (red) and placeholder $\text{graphViPlus2}$ (blue) of QT7, assuming that we have 10 versions ................................................................. 19
8. Replacement values for $\text{graphVstart}$ (green), versions participate in $\text{fromClauses}$ (blue) and $\text{graphVmax}$ (red) of QT8, assuming that we have 10 versions ................................................................. 20
9. Initial version ingestion speed in triples/second ........................................ 24
10. Applied changes speed in changes/second ................................................. 24
11. Throughput in queries/second ................................................................. 25
12. Percentage of succeeded queries ............................................................. 25
13. Storage cost in MB .................................................................................... 25
14. Average execution time for Modern version materialization queries .......... 26
15. Average execution time for Historical version materialization queries ...... 26
16. Average execution time for Delta materialization ...................................... 26
17. Average execution time for Modern single-version structured queries ....... 27
18. Average execution time for Historical single-version structured queries .. 27
19. Average execution time for Single-delta structured queries ...................... 27
20. Average execution time for Cross-version structured queries ................... 28
21. Average execution time for Cross-delta structured queries ....................... 28
22. MOCHA-ESWC2018 results for the systems implement the Delta-based versioning strategy ................................................................. 31
23. MOCHA-ESWC2018 results for the systems implement the Full Materialization versioning strategy ................................................................. 32
24. Final scores for all systems participated in MOCHA-ESWC2018 .............. 33
List of Tables

1 BBC Core & Domain Ontologies .................................................. 12
2 Evolution of DBpedia data ........................................................ 13
3 Replacing mechanism for QT1 placeholders .............................. 15
4 Replacing mechanism for QT2 placeholders .............................. 15
5 Replacing mechanism for QT3 placeholders .............................. 16
6 Replacing mechanism for QT4 placeholders .............................. 16
7 Replacing mechanism for QT5 & QT6 placeholders ..................... 17
8 Replacing mechanism for QT7 placeholders .............................. 18
9 Replacing mechanism for QT8 placeholders .............................. 20
10 Results for R43PLES system ..................................................... 29
11 Results for OSTRICH system .................................................... 29
12 Average execution times for OSTRICH system ......................... 30
13 Weights for the four most important KPIs ............................... 31

List of Query Templates

1 Historical single-version structured query template .................... 14
2 Modern version materialization query template ....................... 34
3 Modern single-version structured query template 1 ................... 34
4 Modern single-version structured query template 2 ................... 34
5 Modern single-version structured query template 3 ................... 35
6 Modern single-version structured query template 4 ................... 35
7 Modern single-version structured query template 5 ................... 36
8 Modern single-version structured query template 6 ................... 36
9 Historical version materialization query template ................... 36
10 Historical single-version structured query template 1 ............... 37
11 Historical single-version structured query template 2 ............... 37
12 Historical single-version structured query template 3 ............... 37
13 Historical single-version structured query template 4 ............... 38
14 Historical single-version structured query template 5 ............... 38
15 Historical single-version structured query template 6 ............... 39
16 Delta materialization query template ................................. 39
17 Single-delta structured query template ................................. 39
18 Cross-delta structured query template ................................. 40
19 Cross-version structured query template 1 ............................ 40
20 Cross-version structured query template 2 ............................ 41
21 Cross-version structured query template 3 ............................ 41
22 Cross-version structured query template 4 ............................ 41
23 Cross-version structured query template 5 ............................ 42
24 Cross-version structured query template 6 ............................ 42
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 storage benchmarks that test the ability of Linked Data systems to store and query data in an efficient way that do not address the management of data versions. 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 (as described in D5.2.1 [15]).

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 [7, 17]. 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 [20]. 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. 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 report, we discuss the Semantic Publishing Versioning Benchmark (SPVB) developed in the context of the H2020 European HOBBIT project 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 (the way that versions are stored, as discussed in [15]) 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) Semantic Publishing Benchmark (SPB) as well as DBpedia [1] data. SPB leverages the scenario of the BBC media organization, which makes heavy 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 (as will be described in Section 2.2) and are partially based on a subset of the 25 query templates defined in the context of DBpedia SPARQL Benchmark (DBPSB) [12]. SPVB evaluates the correctness and performance of the system under test through 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 report is the following. Section 2 summarizes all the background information the reader needs to comprehend the following sections (have been described in detail in D5.2.1[3]). In Section 3 we provide a detailed description of all different components that compose the SPVB. Section 4 describes the experiments we have conducted using the SPVB for a set of baseline systems. Finally, Section 5 concludes and outlines future work.

---

1 https://project-hobbit.eu/
2 ldbc.council.org
Background

This Section summarizes all the background information the reader needs to comprehend the following sections. In particular, we briefly introduce the available versioning strategies used for storing versioned data, as long as the versioning query types that we have defined. Both of them have been described in detail in D5.2.1.

Versioning Strategies

Three alternative RDF versioning strategies have been proposed in the literature. The first one is the full materialization, where all different versions are explicitly stored [19]. 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 [2, 5, 18, 6, 8]. Finally, the annotated triples strategy, is based on the idea of augmenting each triple with its temporal validity, which is usually composed of two timestamps that determine when the triple was created and deleted [13]. Hybrid strategies [16] 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.

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 [4, 16, 10, 11] to identify and categorize these types of queries. Our suggestion, which is a combination of such efforts and was presented in [14], is shown in Figure 1.

Figure 1: Different queries organized by focus and type. [14]
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 must be evaluated on several versions of the dataset, thereby retrieving information common in many versions.

Semantic Publishing Versioning Benchmark

In this section, we present the second version of the SPVB, developed in the context of the HOBBIT H2020 project. The benchmark is build on top of the HOBBIT platform which is available online\(^5\), but it can be locally deployed as well\(^6\). The source code of the benchmark can be found in the project’s github repository\(^7\). 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. Compared to its first version SPVB have been dramatically changed. In particular, in the first version only additions were supported: 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 wanted to be tested for versioned datasets with many versions containing small/large number of changes or with few versions containing small/large number of changes. In the second version, SPVB allows configurable amount of additions and deletions to be produced. This is a critical feature, since now produces 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. Also, in the first version of the benchmark data was sent to the system only as independent copies whereas in the second version, the data generator can send data in different forms (independent copies, change-sets). Finally,

\(^5\)https://master.project-hobbit.eu/
\(^6\)https://github.com/hobbit-project/platform
\(^7\)https://github.com/hobbit-project/versioning-benchmark
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 3.

### Data Generation

The Data Generator, as shown in Figure 3, is the main component of the 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.

#### Configuration Parameters

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. Such size include the triples of the BBC Core & Domain Ontologies as well as the triples of the initial version of DBpedia, that will be described in detail later.

3. **Version insertion ratio** proposed in [3] defines the proportion of added triples between two consecutive versions. Given two versions $V_i$ and $V_{i+1}$, the version insertion ratio $\delta^+_{i,i+1}$ is computed by the formula $\delta^+_{i,i+1} = |\Delta^+_{i,i+1}|/|V_i|$, where $|\Delta^+_{i,i+1}|$ is the number of added triples from version $i$ to version $i+1$, and $|V_i|$ is the total number of triples of version $i$ (where triples are added).

4. **Version deletion ratio** proposed in [3] defines the proportion of deleted triples between two consecutive versions. Given two versions $V_i$ and $V_{i+1}$, the version deletion ratio $\delta^-_{i,i+1}$ is computed by the formula $\delta^-_{i,i+1} = |\Delta^-_{i,i+1}|/|V_i|$, where $|\Delta^-_{i,i+1}|$ is the number of deleted triples from version $i$ to version $i+1$, 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 (Section 2.1), so it requires the generated data in a specific form. SPVB’s data generator can output the data as i) an **Independent Copy** (suitable for systems that implement the full materialization strategy), ii) as a **Changeset** – set of added and deleted triples (suitable for systems implementing the delta-based or annotated triples versioning strategies) or iii) both as an independent copy and changeset (suitable for systems implementing a hybrid strategy).

6. **Generator seed** is used to set the random seed for the data generator. This seed is used to control all random data generation happening in SPVB.

### Producing synthetic added triples

Based on the aforementioned parameters the generator can produce a version of the original dataset that contains both realistic synthetic data and real DBpedia data.
Regarding the part of the synthetic data, the data generator of SPVB uses the data generator of SPB [9], for producing the initial version of the dataset as well as the triples that will be added from one version to another. SPB’s data generator uses seven core and three domain RDF ontologies (see Table 1) and a reference dataset of DBpedia for producing the data.

The SPB data generator produces RDF descriptions of creative works that are valid instances of the BBC Creative Work core ontology. A creative work can be defined as metadata about a real entity (or entities) that exist in 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. That way a creative work provides metadata about one or several entities and defines relations between them. SPB’s data generator models the following three types of relations in the data, and for each one produces 1/3 of the number of creative works.
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.

So, for producing the initial version of the dataset the SPB data generator runs with the Size configuration parameter as input. For producing the triples that have to be added, we first compute the number of triples to be added using the version insertion ratio, and then we run the SPB data generator with the number of added triples as input. As a result, the set of creative works that have to be added are produced. At this point we let the SPB data generator to only produce random data distributions, as we do not want to “break” the existing relations between creative works and entities (clustering or the correlation of entities) of the initial version.
Table 2: Evolution of DBpedia data

<table>
<thead>
<tr>
<th>DBpedia version</th>
<th>1000 entities triples</th>
<th>Deleted triples</th>
<th>Added triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>V₀: 2012-09-14</td>
<td>40.362</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>V₂: 2014-09-30</td>
<td>64.520</td>
<td>16.888</td>
<td>45.991</td>
</tr>
<tr>
<td>V₃: 2015-06-02</td>
<td>60.094</td>
<td>20.480</td>
<td>16.054</td>
</tr>
<tr>
<td>V₄: 2016-09-26</td>
<td>71.021</td>
<td>21.958</td>
<td>32.885</td>
</tr>
</tbody>
</table>

Producing synthetic deleted triples

For producing the triples that have to 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, and we choose creative works in a random manner until we reach the targeted number of triples that have to be deleted. 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 here, 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 that have to be deleted exceeds the total randomly produced, we use a system-imposed threshold of 33% for $\delta_{i,i+1}$.

Evolution of real DBpedia data

As we mentioned earlier, except from the versioning of the creative works dataset, SPVB’s data generator supports the versioning of the reference dataset of DBpedia employed by 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 SPB data generator 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, 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 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 (as shown in Table 2). 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. In cases that the data generator is configured to produce less than 5 versions (lets 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. So, we can later evaluate the produced SPARQL queries and compute the expected results, that are required by the Evaluation Module (Section 3.3) to assess the correctness of
the results reported by the benchmarked system.

Tasks Generation

As shown in Figure 3, 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 that retrieves creative works that are about different topics, along with the topics type from a past version. The full list of them can be seen in Appendix A. 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.

```
# Query Type: 4, Historical single-version structured query
# Query Description :
# Retrieve from a historical version, the creative works that are about or mention different topics, along with the topics type.

PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?creativeWork ?v1
FROM {{{graphVhistorical}}}
WHERE {

  ?creativeWork cwork:about {{{cwAboutUri}}} .
  {{{cwAboutUri}}} rdf:type ?v1 .
}
```

Query Template 1: Historical single-version structured query template

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) [12] 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 11 of Listing 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 and the generator is configured to produce \( n \) versions in total. The \{{{\text{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’ replacing mechanism for all query types described in detail later. 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 ran 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.

In particular, the placeholders of each query template, shown in Appendix A, is replaced using the following mechanism:

**QT1 - Modern version materialization (Appendix A.1)**

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Replacement values</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{{\text{graphVmax}}}}</td>
<td>The graph name corresponds to the current version</td>
</tr>
</tbody>
</table>

Table 3: Replacing mechanism for QT1 placeholders

**QT2 - Modern single-version structured queries (Appendix A.2)**

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Replacement values</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{{\text{graphVmax}}}}</td>
<td>The graph name corresponds to the current version</td>
</tr>
<tr>
<td>{{{\text{cwAboutUri}}}}</td>
<td>A URI of a DBpedia entity. 5 random values out of the list of possible ones, that have previously been extracted are selected.</td>
</tr>
<tr>
<td>{{{\text{cwMentionsUri}}}}</td>
<td></td>
</tr>
<tr>
<td>{{{\text{dbpsb}}}}</td>
<td>A DBpedia URI. Select 5 random values out of the list of possible ones, that have previously been extracted.</td>
</tr>
</tbody>
</table>

Table 4: Replacing mechanism for QT2 placeholders

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
V_0 & V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 & V_8 & V_9 \\
\hline
\end{array}
\]

Figure 4: Replacement value for \{{{\text{graphVmax}}}\} of QT1 and QT2, assuming that we have 10 versions
QT3 - Historical version materialization (Appendix A.3)

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Replacement values</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{{graphVhistorical}}}</td>
<td>The graph name corresponds to a historical version. Assuming that we have a dataset of n versions ($V_0$ to $V_{n-1}$), placeholder is replaced with the following three values:</td>
</tr>
<tr>
<td></td>
<td>1. The $V_0$</td>
</tr>
<tr>
<td></td>
<td>2. The $V_x$ version, where $x = \text{round}(\frac{n-2}{2})$</td>
</tr>
<tr>
<td></td>
<td>3. The $V_{n-2}$ version</td>
</tr>
</tbody>
</table>

Table 5: Replacing mechanism for QT3 placeholders

QT4 - Historical single-version structured queries (Appendix A.4)

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Replacement values</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{{graphVhistorical}}}</td>
<td>The graph name corresponds to a historical version. Assuming that we have a dataset of n versions ($V_0$ to $V_{n-1}$), placeholder is replaced with the following three values:</td>
</tr>
<tr>
<td></td>
<td>1. The $V_0$</td>
</tr>
<tr>
<td></td>
<td>2. The $V_x$ version, where $x = \text{round}(\frac{n-2}{2})$</td>
</tr>
<tr>
<td></td>
<td>3. The $V_{n-2}$ version</td>
</tr>
<tr>
<td>{{{cwAboutUri}}}</td>
<td>A URI of a DBpedia entity. For each replacement value of {{{graphVhistorical}}} placeholder (1, 2 and 3 cases shown above), select a random value out of the list of possible ones, that have previously been extracted</td>
</tr>
<tr>
<td>{{{cwMentionsUri}}}</td>
<td></td>
</tr>
<tr>
<td>{{{dbpsb}}}</td>
<td>A DBpedia URI. For each replacement value of {{{graphVhistorical}}} placeholder (1, 2 and 3 cases shown above), select a random value out of the list of possible ones, that have previously been extracted</td>
</tr>
</tbody>
</table>

Table 6: Replacing mechanism for QT4 placeholders
Figure 5: Replacement values for `{{{graphVhistorical}}}` of QT3 and QT4, assuming that we have 10 versions

**Table 7: Replacing mechanism for QT5 & QT6 placeholders**

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Replacement values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{{{graphVmax}}}</code></td>
<td>The graph name corresponds to the current version</td>
</tr>
<tr>
<td><code>{{{graphVhistorical}}}</code></td>
<td>The graph name corresponds to a historical version. Assuming that we have a dataset of ( n ) versions (( V_0 ) to ( V_{n-1} )), placeholder is replaced with the following four values:</td>
</tr>
<tr>
<td></td>
<td>1. The ( V_0 )</td>
</tr>
<tr>
<td></td>
<td>2. The ( V_x ) version, where ( x = \text{round}(\frac{n-2}{3}) )</td>
</tr>
<tr>
<td></td>
<td>3. The ( V_y ) version, where ( y = \text{round}(2 \times \frac{n-2}{3}) )</td>
</tr>
<tr>
<td></td>
<td>4. The ( V_{n-2} ) version</td>
</tr>
</tbody>
</table>

**QT5 - Delta materialization (Appendix A.5) & QT6 - Single-delta structured query (Appendix A.6)**

**QT7 - Cross-delta structured queries (Appendix A.7)**

Cross-delta structured queries refer to deltas between consecutive versions. We want to cover a high spectrum of queried deltas so we let them to be at the beginning, in the middle and in the end of the dataset timeline.
Figure 6: Replacement values for `{{graphVmax}}` (green) and `{{graphVhistorical}}` (red) of QT5 and QT6, assuming that we have 10 versions

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Replacement values</th>
</tr>
</thead>
</table>
| `{{graphVi}}` | Assuming that we have a dataset of $n$ versions $(V_0$ to $V_{n-1})$, the placeholder is replaced with the following three values:  
1. The $V_0$  
2. The $V_x$ version, where $x = \text{round}(\frac{n-3}{2})$  
3. The $V_{n-3}$ version |
| `{{graphViPlus1}}` | The graph name determines the $V_{i+1}$ version |
| `{{graphViPlus2}}` | The graph name determines the $V_{i+2}$ version |

Table 8: Replacing mechanism for QT7 placeholders

QT8 - Cross-version structured queries (Appendix A.8)

We distinguish the cross-version structured queries in different cases, according to the number of queried versions and how far such versions are from each other. We want to cover a high spectrum of such cases, so we let the queried versions to be 2, 3 or 4 (where the total number of versions permits) and their "distance" to be the nearest or the longest.
Figure 7: Replacement values for {{{graphVi}}} (green), {{{graphViPlus1}}} (red) and placeholder-graphViPlus2 (blue) of QT7, assuming that we have 10 versions.

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Replacement values</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{{graphVstart}}}</td>
<td>The graph name determining the starting queried version. Assuming that we have a dataset of ( n ) versions (( V_0 ) to ( V_{n-1} )), the placeholder is replaced with the following values:</td>
</tr>
<tr>
<td></td>
<td>1. Nearest distance</td>
</tr>
<tr>
<td></td>
<td>(a) 2 queried versions: The ( V_{n-2} ) version</td>
</tr>
<tr>
<td></td>
<td>(b) 3 queried versions: The ( V_{n-3} ) version</td>
</tr>
<tr>
<td></td>
<td>(c) 4 queried versions: The ( V_{n-4} ) version</td>
</tr>
<tr>
<td></td>
<td>2. Longest distance</td>
</tr>
<tr>
<td></td>
<td>(a) 2, 3 or 4 queried versions: The ( V_0 ) version</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{{{fromClauses}}}</td>
<td>SPARQL “FROM” clauses for the intermediate queried versions. Assuming that we have a dataset of ( n ) versions (( V_0 ) to ( V_{n-1} )), the placeholder is replaced with the following values:</td>
</tr>
<tr>
<td></td>
<td>1. Nearest distance</td>
</tr>
<tr>
<td></td>
<td>(a) 2 queried versions: -</td>
</tr>
<tr>
<td></td>
<td>(b) 3 queried versions: FROM clause for the ( V_{n-2} ) version</td>
</tr>
<tr>
<td></td>
<td>(c) 4 queried versions: FROM clauses for the ( V_{n-3} ) and ( V_{n-2} ) versions</td>
</tr>
<tr>
<td></td>
<td>2. Longest distance</td>
</tr>
<tr>
<td></td>
<td>(a) 2 queried versions: -</td>
</tr>
<tr>
<td></td>
<td>(b) 3 queried versions: FROM clauses for the ( V_x ) where ( x = round(\frac{n-1}{3}) )</td>
</tr>
<tr>
<td></td>
<td>(c) 4 queried versions: FROM clauses for the ( V_x ) and ( V_y ) versions,</td>
</tr>
<tr>
<td></td>
<td>where ( x = round(\frac{n-1}{3}) ) and ( y = round(2 \times \frac{n-1}{3}) )</td>
</tr>
</tbody>
</table>
The $V_{n-1}$ version

A URI of a DBpedia entity. For each set of replacement values of

\{
\{\{graphVstart\}\}, \{\{fromClauses\}\} and \{\{graphVmax\}\}\}

placeholders (out of the 6 possible ones), select a random value from the
list that have previously been extracted

A DBpedia URI. For each set of replacement values of

\{
\{\{graphVstart\}\}, \{\{fromClauses\}\} and \{\{graphVmax\}\}\}

placeholders (out of the 6 possible ones), select a random value from the
list that have previously been extracted

Table 9: Replacing mechanism for QT8 placeholders

<table>
<thead>
<tr>
<th>2 queried versions</th>
<th>3 queried versions</th>
<th>4 queried versions</th>
<th>2 queried versions</th>
<th>3 queried versions</th>
<th>4 queried versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nearest distance</td>
<td>V0 V1 V2 V3 V4 V5 V6 V8 V9</td>
<td>Nearest distance</td>
<td>V0 V1 V2 V3 V4 V5 V6 V7 V8 V9</td>
<td>Nearest distance</td>
<td>V0 V1 V2 V3 V4 V5 V6 V7 V8 V9</td>
</tr>
<tr>
<td>Longest distance</td>
<td>V0 V1 V2 V3 V4 V5 V6 V7 V8 V9</td>
<td>Longest distance</td>
<td>V0 V1 V2 V3 V4 V5 V6 V7 V9</td>
<td>Longest distance</td>
<td>V0 V1 V2 V3 V4 V5 V6 V8 V9</td>
</tr>
</tbody>
</table>

Figure 8: Replacement values for \{\{graphVstart\}\} (green), versions participate in \{\{fromClauses\}\} (blue) and \{\{graphVmax\}\} (red) of QT8, assuming that we have 10 versions

After replacing all the placeholders of each query template following the previous described mechanism, a set of similar queries is generated. As shown in Figure 3, 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.

\(^9\)http://vos.openlinksw.com
under test and the expected results to the *Evaluation Storage* component. By term “sequentially” we mean that the Task Provider waits for each task to be executed by the system before sending the next one. 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*.

**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)*:

- **Query failures**: The number of queries that failed to execute. Failure refers to the fact that the system under test return 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, for time saving, only condition i) is checked.

- **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 such 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.

**Experiments**

In this section we are going to present the experiments we conducted with SPVB on top of the HOBBIT platform\(^\text{10}\), as long as the results of MOCHA-ESWC2018 challenge (for the SPVB - Task 3). We will describe i) the systems that SPVB used for benchmarking, ii) the hardware setup of the HOBBIT platform where the experiments conducted, as well as the configuration parameters that chosen for such experiments, and iii) the results themselves.

**Benchmarked systems**

Our decision for selecting the systems to experiment with, based on two factors: i) their ability to have versioning capabilities and ii) their ability to provide an open source solution. So, firstly, the two versioning systems of OSTRICH and R43PLES, are used for benchmarking. Since, there is only a limited number of versioning systems we decided to also assess the performance of different triplestores

\(^{10}\text{https://master.project-hobbit.eu}\)
assuming that they store each version in a different named graph (full materialization versioning strategy). So, we also used the RDF triplestores of Virtuoso, Apache Jena Fuseki, GraphDB Free and Blazegraph. For all of them, a System Adapter implemented in the Java programming language, following a predefined Common API. From our side we implemented the System Adapters for the systems of R43ples, Virtuoso, Apache Jena Fuseki and GraphDB Free. A participant of MOCHA-ESWC2018 challenge implemented the OSTRICH System Adapter and finally, a HOBBIT partner implemented the Blazegraph one.

**OSTRICH**

OSTRICH is a triple store that allows multiple dataset versions to be stored and queried. The store is a hybrid between full materialization, delta-based and annotated triples storage, which provides a good trade-off between storage size and query time. Version materialized, version and delta materialized queries can be performed efficiently over the different versions. These queries support limits and offsets for any triple pattern, as results are returned as triple streams. Insertion is done by first inserting a dataset snapshot, which is encoded in HDT\(^{12}\). After that, deltas can be inserted, which contain additions and deletions based on the last delta or snapshot.\(^{13}\)

**R43ples 0.8.8**

R43ples (Revision for Triples) is an open source Revision Management Tool for the Semantic Web. It provides different revisions of named graphs via a SPARQL interface. All information about revisions, changes, commits, branches and tags are stored in additional named graphs beside the original graph in an attached external triple store. R43ples uses an internal Jena TDB which is attached to an existing SPARQL endpoint of a triplestore and acts as another endpoint both for normal SPARQL queries as well as for revision-enhanced SPARQL queries, named R43ples queries. The R43ples endpoint allows to specify revisions which should be queried for each named graph used inside a SPARQL query. The whole revision information is stored in additional graphs in the attached Jena TDB\(^{14}\).

**Virtuoso**

Virtuoso is a middleware and database engine hybrid that combines the functionality of a traditional Relational database management system (RDBMS), Object-relational database (ORDBMS), virtual database, RDF, XML, free-text, web application server and file server functionality in a single system\(^{15}\). For the System Adapter implementation we used the 7.2.4 release (25/04/2016), configured to return more than 1048576 results via HTTP (which was the default limit) and also let it to use the 2/3 of system’s RAM (which is 256GB). Regarding the loading phase, Virtuoso is able to parallelize the process, so we let it use the recommended number_of_cores/2.5 bulk loaders.

\(^{11}\)https://project-hobbit.eu/challenges/mighty-storage-challenge2018/msc-tasks/#common-api

\(^{12}\)http://www.rdfhdt.org/

\(^{13}\)https://github.com/rdfostrich/ostrich

\(^{14}\)http://plt-tud.github.io/r43ples/

\(^{15}\)https://en.wikipedia.org/wiki/Virtuoso_Universal_Server
Apache Jena Fuseki 3.6.0

Apache Jena Fuseki is a SPARQL server. It can run as a operating system service, as a Java web application (WAR file), and as a standalone server. It provides security (using Apache Shiro) and has a user interface for server monitoring and administration. It provides the SPARQL 1.1 protocols for query and update as well as the SPARQL Graph Store protocol. Fuseki is tightly integrated with TDB to provide a robust, transactional persistent storage layer, and incorporates Jena text query and Jena spatial query. It can be used to provide the protocol engine for other RDF query and storage systems\footnote{https://jena.apache.org/documentation/fuseki2/}. Jena provides several TDB command line utilities. One of the most important is the bulk loader and index builder “tdbloader2”\footnote{https://jena.apache.org/documentation/tdb/commands.html#tdbloader2} which is used by the System Adapter for the bulk loading phase. tdbloader2 is the most efficient option to load data into a TDB-back model but can only be used to create a database as it overwrites existing data. As a result, we only used it for loading the first version. For the rest versions we used the “tdbloader”\footnote{https://jena.apache.org/documentation/tdb/commands.html#tdbloader} in which the graph name where the data are going to be loaded can be determined. Unfortunately, tdbloader and tdbloader2 are not transactional\footnote{https://jena.apache.org/documentation/tdb/tdb_transactions.html} so it could not be possible to parallelize the bulk loading phase.

GraphDB Free 8.5

GraphDB Free is a family of highly-efficient, robust and scalable RDF databases that streamlines the load and use of linked data cloud datasets. It implements the RDF4J\footnote{http://rdf4j.org/about/} framework interfaces, the W3C SPARQL Protocol specification\footnote{https://www.w3.org/TR/sparql11-overview/} and supports all RDF serialisation formats.

Blazegraph

Blazegraph is a fully open-source high-performance graph database supporting the RDF data model. It operates as an embedded database or over a client/server REST API. Blazegraph supports high-availability and dynamic sharding and supports both the Blueprints and Sesame APIs. It supports optional transactions, very high concurrency, and very high aggregate IO rates\footnote{https://www.blazegraph.com/product/product-description/}.

Experimental setup

All the experiments where conducted in the online instance of the HOBBIT platform\footnote{https://master.project-hobbit.eu} which is deployed on a server cluster. Each of the Data Generator, Task Provider, Evaluation Module and System Adapter 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. Unfortunately, the monitoring services provided by the HOBBIT platform cannot give reliable results regarding the storage space yet, so we manually compute the “Storage space cost” KPI wherever possible.

For our experiments we produced 3 datasets of different initial sizes that correspond to 100K, 500K and 1M triples. 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,
were produced a final version of 141K, 627K and 1.235M triples respectively. The number of triples in such final versions include 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 all systems, since the HOBBIT platform sequentially runs the submitted experiments for more than twenty benchmarks. So, in order to have a good use of the platform this time limit was necessary.

Results

Regarding the full materialization strategy that implemented on top of Virtuoso, Apache Jena Fuseki, GraphDB Free and Blazegraph, all the systems except of Blazegraph managed to run the experiments for all dataset sizes. In particular, for the dataset sizes of 500K and 1M triples, Blazegraph failed to complete the experiments in the defined timeout of 1 hour. Unfortunately, since the System Adapter implemented by a HOBBIT partner we did not have the ability to further investigate the reasons of such failure.

First, in Figure 9 we can see for all datasets and for all systems the initial version ingestion speed. The speed ranges from 24K to 140K triples per second and increases as the dataset size, and consequently the size of its initial version, increases. As we can see Virtuoso outperforms all the other systems for all dataset sizes. This is happening due to the fact that it is able to parallelly execute the loading phase and hence maximize the loading speed. Also, we can see that Apache Jena Fuseki outperforms GraphDB Free since for the first version it uses the bulk loader of “tdbloader2”.

The same holds for the applied changes speed, shown in Figure 10, which ranges from 7K to 53K changes per second. We can observe here that for all dataset sizes and for all systems, 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. Also, we can see that the applied changes speed of Apache Jena Fuseki is now similar to that of GraphDB Free. This is happening due to the fact that the slower “tdbloader” bulk loader used for loading the data of other versions than the initial one, in which we are able to determine the graph name where the triples will be loaded into.
Regarding the query execution rate, we can see in Figure 11 that as expected, the smaller is the dataset the higher is the throughput. In addition, GRAPHDB FREE is the system that evaluates the queries in the highest rate, followed by BLAZEGRAHP, APACHE JENA FUSEKI and last VIRTUOSO, although (as shown in Figure 12) it fails to return the correct results for all executed queries. Recall here that VIRTUOSO which succeed to return the correct results for all queries, is the triplestore that used for the computation of the expected answers.

Next, in Figure 13 we can see the storage space required for storing the data for all different datasets and for all systems. The space requirements as expected increase as the total number of triples increase from 80MB to 2GB. This significant overhead on storage space is due to the archiving strategy used (i.e., Full Materialization), since duplicated data from version to version have to be stored. Also, we can observe that for the larger datasets APACHE JENA FUSEKI uses almost two times the storage space used by the other systems for storing the same data. This is happening because it uses an on-disk database format where all the data and the triple and quad indexes (e.g. SPO, PSO, GSPO, etc.) stored in the disk.

In Figures 14, 15, 17, 18, 16, 19, 21 and 20 we present the average execution time (in msec) for...
the queries of each versioning query type, for each dataset size and for all systems.

In particular, in Figures 14 and 15 we can see the time required for materializing the modern (current) version and a historical (past) one 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 the execution times are small enough, considering the number of returned triples, as all the versions are already materialized in the triple store. Also, we can see that GraphDB Free responds much faster than its competitors when a full version is requested.

In Figure 16 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. Except of Virtuoso and Blazegraph for the smaller dataset we can see that all other systems failed to correctly materialize a delta since they returned a little more or fewer triples.

Finally, in Figures 17, 18, 19, 20 and 21 we can see the execution times for all structured
queries. In all of the cases, similarly to materialization queries, the execution time increases as the number of triples increases and the queries refer to historical versions require slightly less time to be executed, since such versions composed of less triples. Also, although one would expect the delta-based queries 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.

![Figure 17: Average execution time for Modern single-version structured queries](image1)

![Figure 18: Average execution time for Historical single-version structured queries](image2)

![Figure 19: Average execution time for Single-delta structured queries](image3)
Regarding the pure versioning systems of R43ples and OSTRICH, for the first we managed to run experiments only for the dataset of 100K triples as for the remaining datasets the experiment time exceed the timeout of one hour. For the later, although all the experiments succeeded to run within the defined timeout of one hour, many of the returned results were not the expected ones, so for both R43ples and OSTRICH we do not report the results graphically.

For the R43ples system, in the left table of Table 10, we can see the initial version ingestion speed and the 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 the triplestores of Virtuoso, Apache Jena Fuseki, GraphDB Free and Blazegraph, 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 Table, is similar to the other triplestores. Someone would expect R43ples to outperform them, as the Full Materialization strategy implemented on top of them, but that seems not to be the case.

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.

The OSTRICH versioning system, as we can see in the Table 11, for all dataset sizes, succeeded to load the initial version at high speed, especially compared to the other versioning system of 43ples, since it stores the first version as a snapshot. On the other hand the changes are applied in a much slower rate than the triples of the initial version are loaded. This is happening due to the fact that OSTRICH uses a hybrid strategy between full materialization, delta-based and annotated triples for storing the data. As a result, for storing every new version it needs to quantify the corresponding overheads in order to determine the optimal storage strategy at loading time. The storage space overhead could not be manually quantified since the OSTRICH System Adapter implemented by a MOCHA-ESWC2018 participant.

Regarding the queries execution we can observe that for the smallest dataset OSTRICH managed to execute successfully most of the queries, but as the dataset size increase the queries that failed to be executed increase as well. As shown in the Table 12, one of the reasons of such failures is the query...
timeout of one minute that is defined by OSTRICH. Another one, is the fact that it failed to correctly load all the versioned data.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial version ingestion speed (triples/sec)</td>
<td>3502.39</td>
</tr>
<tr>
<td>Applied changes speed (changes/sec)</td>
<td>2767.56</td>
</tr>
<tr>
<td>Storage cost (MB)</td>
<td>192</td>
</tr>
<tr>
<td>Throughput (queries/second)</td>
<td>0.09</td>
</tr>
<tr>
<td>Queries failed</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 10: Results for R43ples system

<table>
<thead>
<tr>
<th>Metric</th>
<th>Result</th>
<th>Succeeded queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT1 (ms)</td>
<td>13887.33</td>
<td>0/1</td>
</tr>
<tr>
<td>QT2 (ms)</td>
<td>146.28</td>
<td>25/30</td>
</tr>
<tr>
<td>QT3 (ms)</td>
<td>18265.78</td>
<td>0/3</td>
</tr>
<tr>
<td>QT4 (ms)</td>
<td>11681.49</td>
<td>13/18</td>
</tr>
<tr>
<td>QT5 (ms)</td>
<td>31294.00</td>
<td>0/4</td>
</tr>
<tr>
<td>QT6 (ms)</td>
<td>12299.58</td>
<td>4/4</td>
</tr>
<tr>
<td>QT7 (ms)</td>
<td>35294.33</td>
<td>2/3</td>
</tr>
<tr>
<td>QT8 (ms)</td>
<td>19177.33</td>
<td>30/36</td>
</tr>
</tbody>
</table>

Table 11: Results for OSTRICH system

### MOCHA-ESWC2018 results

The aim of the Mighty Storage Challenge (MOCHA-ESWC2018) was to test the performance of solutions for SPARQL processing in aspects that are relevant for modern applications. These include ingesting data, answering queries and serving as backend for applications driven by Linked Data. In this report we only focus on the performance of each system after being benchmarked by SPVB (Task 3 of MOCHA-ESWC2018).

Since SPVB gives the ability to the systems under test to retrieve the data of each version as Independent Copies (IC), Change-Sets (CS) or both as IC and CS (Section 3.1.1), three sub-tasks defined in the context of the challenge which only differed in terms of the “generated data form”

24 [https://project-hobbit.eu/challenges/mighty-storage-challenge2018/](https://project-hobbit.eu/challenges/mighty-storage-challenge2018/)
Table 12: Average execution times for OSTRICH system

<table>
<thead>
<tr>
<th>Metric</th>
<th>100K</th>
<th>500K</th>
<th>1M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Result</td>
<td>Succeeded queries</td>
<td>Result</td>
</tr>
<tr>
<td>QT1 (ms)</td>
<td>39699</td>
<td>1/1</td>
<td>60095</td>
</tr>
<tr>
<td>QT2 (ms)</td>
<td>2396</td>
<td>25/30</td>
<td>3165.75</td>
</tr>
<tr>
<td>QT3 (ms)</td>
<td>33732</td>
<td>3/3</td>
<td>60017</td>
</tr>
<tr>
<td>QT4 (ms)</td>
<td>456</td>
<td>15/18</td>
<td>9846</td>
</tr>
<tr>
<td>QT5 (ms)</td>
<td>29352</td>
<td>2/4</td>
<td>60023</td>
</tr>
<tr>
<td>QT6 (ms)</td>
<td>106.25</td>
<td>0/4</td>
<td>425</td>
</tr>
<tr>
<td>QT7 (ms)</td>
<td>22.33</td>
<td>0/3</td>
<td>12</td>
</tr>
<tr>
<td>QT8 (ms)</td>
<td>1261.38</td>
<td>26/36</td>
<td>74</td>
</tr>
</tbody>
</table>

configuration parameter. So, each participant was able to submit his/her system in the correct sub-task according to the implemented versioning strategy. In particular all the systems benchmarked using the following common parameters:

- A seed for data generation: 100
- Initial version size (in triples): 200000
- Number of versions: 5
- Version Deletion Ratio (%): 10
- Version Insertion Ratio (%): 15

As shown in Figures 22 and 23 all the systems except of the R43pies one, were managed to be tested. The latest, did not manage to load and answer all the queries in the defined timeout of 1 hour.

In order to be able to decide who is the winner for the Task 3 of the challenge, we combined the results of the four most important KPIs and calculated a final score that ranges from 0 to 1. Note here that due to reliability issues mentioned earlier, the “Storage space cost” KPI excluded from the final score formula. So, to compute the final score, we assigned weights to those KPIs, whose sum equals to 1. The four KPIs (in order of importance) along with the assigned weights are shown in Table 13.
Table 13: Weights for the four most important KPIs

<table>
<thead>
<tr>
<th>Order</th>
<th>KPI</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Throughput</td>
<td>0.4</td>
</tr>
<tr>
<td>2</td>
<td>Queries failed</td>
<td>0.3</td>
</tr>
<tr>
<td>3</td>
<td>Initial version ingestion speed</td>
<td>0.15</td>
</tr>
<tr>
<td>4</td>
<td>Applied changes speed</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Next, we applied feature scaling to normalize the results of the Throughput, Initial version ingestion speed and Applied changes speed by using the following formula:

\[ x' = \frac{x - \min(x)}{\max(x) - \min(x)} \]

where \( x \) is the original value and \( x' \) is the normalized value.

Regarding the Queries failed KPI, since the lower is the result, the better the system performs, the aforementioned formula applied on the percentage of succeeded queries and not to the number of

\footnote{Bring all results into the range of [0,1]}
queries that failed to be executed.

Having all the results normalized in the range of \([0 - 1]\) and the weights of each KPI, we computed the final scores as the sum of the weighted normalized results. As shown in Figure 24, Virtuoso v8.0 was the system that performed better.

Figure 23: MOCHA-ESWC2018 results for the systems implement the *Full Materialization* versioning strategy.

<table>
<thead>
<tr>
<th>System</th>
<th>VOS for all</th>
<th>Virtuoso v8.0</th>
<th>Apache Jena Fuseki 3.6.0</th>
<th>Blazegraph</th>
<th>GraphDB Free 9.5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KPIs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applied changes speed (changes/sec)</td>
<td>21689.8467093125</td>
<td>19831.20793125</td>
<td>7895.408203125</td>
<td>6961.072753909625</td>
<td>11451.9423828125</td>
</tr>
<tr>
<td>Initial version ingestion speed (trips/sec)</td>
<td>43802.671875</td>
<td>43793.0803125</td>
<td>28332.5351625</td>
<td>22477.298828125</td>
<td>15589.913859375</td>
</tr>
<tr>
<td>QT1, average execution time (ms)</td>
<td>21412.0</td>
<td>1661.0</td>
<td>17836.0</td>
<td>8651.0</td>
<td>5918.0</td>
</tr>
<tr>
<td>QT2, average execution time (ms)</td>
<td>139.066666404140625</td>
<td>147.6664717529296875</td>
<td>229.0</td>
<td>121.393358764648475</td>
<td>129.7333740234975</td>
</tr>
<tr>
<td>QT3, average execution time (ms)</td>
<td>16781.0</td>
<td>11944.0</td>
<td>7647.66650390625</td>
<td>5523.0</td>
<td>3063.0</td>
</tr>
<tr>
<td>QT4, average execution time (ms)</td>
<td>120.9444427490234375</td>
<td>142.388885498046875</td>
<td>202.388885498046875</td>
<td>107.41176605224609375</td>
<td>153.27777699809375</td>
</tr>
<tr>
<td>QT5, average execution time (ms)</td>
<td>11475.0</td>
<td>7186.0</td>
<td>0.0</td>
<td>7501.0</td>
<td>0.0</td>
</tr>
<tr>
<td>QT6, average execution time (ms)</td>
<td>172.25</td>
<td>165.75</td>
<td>195.25</td>
<td>184.0</td>
<td>260.25</td>
</tr>
<tr>
<td>QT7, average execution time (ms)</td>
<td>28.6666660308270905</td>
<td>17.0</td>
<td>47.0</td>
<td>122.0</td>
<td>26.0</td>
</tr>
<tr>
<td>QT8, average execution time (ms)</td>
<td>177.0</td>
<td>155.638855498046875</td>
<td>448.566680908203125</td>
<td>83.18161610107421875</td>
<td>210.117645263671875</td>
</tr>
<tr>
<td>Queries failed</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Storage cost (MB)</td>
<td>0.03090625</td>
<td>0.0</td>
<td>384.80893975</td>
<td>304.2966875</td>
<td>232.3671875</td>
</tr>
<tr>
<td>Throughput (queries/second)</td>
<td>0.754659096340179444</td>
<td>1.2058429932250976</td>
<td>1.32686459415435791</td>
<td>1.84617877000536717</td>
<td>3.036572217941284175</td>
</tr>
</tbody>
</table>
In more details, as shown in Figures 22 and 23 we can see that, similarly to the experiments we manually have conducted, VIRTUOSO v8.0 along with VOS FOR ALL outperforms the other systems in terms of loading times. This is happening, since they have the ability to use multiple bulk loaders that can be run in parallel. Also, we notice that both VOS FOR ALL and VIRTUOSO v8.0 return correct enough results. This was an expected behavior since, as we mentioned in Section 3.2, Virtuoso Open Source is the triplestore used by SPVB for computing the expected results. Regarding the query execution rate, we can observe that GraphDB Free, which also achieved a high score, managed to answer most of the queries (and especially the materialization ones) faster than its competitors.

Conclusions and Future work

In this report we described the final version of the Versioning Benchmark SPVB. In particular, we provided a detailed description of all the components that compose the SPVB along with a set of experimental results after asses the performance of VIRTUOSO, APACHE JENA FUSEKI, GRAPHDB FREE, BLAZEGRAH, OSTRICH and R43PLES systems with SPVB. Finally, we presented the results of MOCHA-ESWC2018 challenge for SPVB-related task.

Compared to the first version of SPVB, we:

- reached a final number of six benchmarked systems (from the previous two);
- enhanced the data generation process by:
  - using real DBpedia data;
  - adding support for producing additions and deletions;
  - sending the generated data to the systems on multiple forms (independent copies, change-sets, etc.);
changed the task generation process by using real DBpedia queries that arise from real world situations.

Since we plan to keep SPVB an alive benchmark, so that we can attract more systems to use it, we have plans for further improvements. In more details, we want to move the expected results computation out of the Data Generator component. By doing so, SPVB will be able to generate in parallel data through multiple Data Generators. Also, 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 want to add functionalities that the second version of the HOBBIT platform offers. In particular, we want to make KPIs to be able to be visualized graphically on the GUI of the platform (wherever possible).

Appendices

Query Templates

**QT1 - Modern version materialization query template**

```
# Query Type: 1, Modern version materialization
# Query Description:
# The full current version of the dataset is retrieved
SELECT ?s ?p ?o
WHERE {
  GRAPH {{{graphVmax}}} { ?s ?p ?o }
}
```

Query Template 2: Modern version materialization query template

**QT2 - Modern single-version structured query templates**

```
# Query Type: 2, Modern single-version structured query
# Query Description:
# Retrieve from the current version, the creative works that are about or mention different topics, along with the topics type.
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?creativeWork ?v1
FROM {{{graphVmax}}}
WHERE {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  {{{cwAboutUri}}} rdf:type ?v1 .
}
```

Query Template 3: Modern single-version structured query template 1

```
# Query Type: 2, Modern single-version structured query
# Query Description:
```
# Retrieve from the current version, the creative works that are about or mention different topics, along with the topics type.

```sql
PREFIX dbp-prop: <http://dbpedia.org/property/>
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>

FROM {{(graphVmax)}}
WHERE {
  { ?creativeWork cwork:about {{{cwAboutUri}}} .
    {{{cwAboutUri}}} dbp-prop:subsid ?v3
    OPTIONAL { ?v2 {{{cwAboutUri}}} dbp-prop:parent }
    OPTIONAL { {{{cwAboutUri}}} dbp-prop:divisions ?v4 }
  }
  UNION {
    ?creativeWork cwork:about {{{cwAboutUri}}} .
    ?v2 {{{cwAboutUri}}} dbp-prop:parent
    OPTIONAL { {{{cwAboutUri}}} dbp-prop:subsid ?v3 }
    OPTIONAL { {{{cwAboutUri}}} dbp-prop:divisions ?v4 }
  }
  UNION {
    ?creativeWork cwork:about {{{cwAboutUri}}} .
    {{{cwAboutUri}}} dbp-prop:divisions ?v4
    OPTIONAL { {{{cwAboutUri}}} dbp-prop:subsid ?v3 }
    OPTIONAL { ?v2 {{{cwAboutUri}}} dbp-prop:parent }
  }
}
```

Query Template 4: Modern single-version structured query template 2

---

# Query Type: 2, Modern single-version structured query

```sql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT *
FROM {{(graphVmax)}}
WHERE {
  { ?creativeWork cwork:about {{{cwAboutUri}}} .
    {{{cwAboutUri}}} rdfs:comment ?v0 .
    FILTER (lang (?v0) = 'en')
  }
  UNION { {{{cwAboutUri}}} foaf:depiction ?v1 }
  UNION { {{{cwAboutUri}}} foaf:homepage ?v2 }
}
```

Query Template 5: Modern single-version structured query template 3

---

# Query Type: 2, Modern single-version structured query

```sql
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>

FROM {{(graphVmax)}}
WHERE {
  {{{cwMentionsUri}}} ?v4 ?v5 .
```

---
Query Template 6: Modern single-version structured query template 4

```sparql
# Query Type: 2, Modern single-version structured query
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
FROM {{{graphVmax}}}
WHERE {
  { ?creativeWork cwork:about {{{cwAboutUri}}} .
    FILTER ((STR(?v3) = 'http://www.w3.org/2000/01/rdf-schema#label' &&
              lang(?v4) = 'en') ||
             (STR(?v3) = 'http://dbpedia.org/ontology/abstract' &&
              lang(?v4) = 'en') ||
             (STR(?v3) = 'http://www.w3.org/2000/01/rdf-schema#comment' &&
              lang(?v4) = 'en') ||
             (STR(?v3) != 'http://dbpedia.org/ontology/abstract' &&
              STR(?v3) != 'http://www.w3.org/2000/01/rdf-schema#comment' &&
              STR(?v3) != 'http://www.w3.org/2000/01/rdf-schema#label'))
  }
  UNION
  { ?creativeWork cwork:about {{{cwAboutUri}}} .
    ?v5 ?v3 {{{cwAboutUri}}} .
    FILTER (STR(?v3) = 'http://dbpedia.org/ontology/owner' ||
             STR(?v3) = 'http://dbpedia.org/property/redirect')
  }
}
```

Query Template 7: Modern single-version structured query template 5

```sparql
# Query Type: 2, Modern single-version structured query
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?creativeWork ?v2
FROM {{{graphVmax}}}
WHERE {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  ?v2 ?v3 {{{cwAboutUri}}} .
  FILTER (STR(?v3) = 'http://dbpedia.org/ontology/owner' ||
           STR(?v3) = 'http://dbpedia.org/property/redirect')
}
```

Query Template 8: Modern single-version structured query template 6

```
```

QT3 - Historical version materialization query template

```sparql
# Query Type: 3, Historical version materialization
```
**Query Template 9: Historical version materialization query template**

```
SELECT ?s ?p ?o
WHERE {
  GRAPH {{{graphVhistorical}}} { ?s ?p ?o }
}
```

**Query Template 10: Historical single-version structured query template 1**

```
SELECT DISTINCT ?creativeWork ?v1
FROM {{{graphVhistorical}}}
WHERE {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  {{{cwAboutUri}}} rdf:type ?v1 .
}
```

**Query Template 11: Historical single-version structured query template 2**

```
FROM {{{graphVhistorical}}}
WHERE {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  {{{cwAboutUri}}} dbp-prop:subsid ?v3
  OPTIONAL { ?v2 {{{cwAboutUri}}} dbp-prop:parent }
  OPTIONAL { {{{cwAboutUri}}} dbp-prop:divisions ?v4 }
}
UNION {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  ?v2 {{{cwAboutUri}}} dbp-prop:parent
  OPTIONAL { {{{cwAboutUri}}} dbp-prop:subsid ?v3 }
  OPTIONAL { {{{cwAboutUri}}} dbp-prop:divisions ?v4 }
}
UNION {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  {{{cwAboutUri}}} dbp-prop:divisions ?v4
  OPTIONAL { {{{cwAboutUri}}} dbp-prop:subsid ?v3 }
  OPTIONAL { ?v2 {{{cwAboutUri}}} dbp-prop:parent }
}
```

**QT4 - Historical single-version structured query templates**

# Query Type: 4, Historical single-version structured query
# Query Description :
# Retrieve from a historical version, the creative works that are about or mention different topics, along with the topics type.

**PREFIX** cwork: <http://www.bbc.co.uk/ontologies/creativework/>  
**PREFIX** rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

**SELECT** DISTINCT ?creativeWork ?v1
**FROM** {{{graphVhistorical}}}
**WHERE** {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  {{{cwAboutUri}}} rdf:type ?v1 .
}
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT *
FROM {{(graphVhistorical)}}
WHERE {
    ?creativeWork cwork:about {{{cwAboutUri}}} .
    {{{cwAboutUri}}} rdfs:comment ?v0 .
    FILTER (lang (?v0) = 'en')
}
UNION { {{{cwAboutUri}}} foaf:depiction ?v1 }
UNION { {{{cwAboutUri}}} foaf:homepage ?v2 }

Query Template 12: Historical single-version structured query template 3

# Query Type: 4, Historical single-version structured query

PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>

FROM {{(graphVhistorical)}}
WHERE {
    {{{cwMentionsUri}}} ?v4 ?v5 .
        FILTER (langMatches(lang(?v6), 'en') || (!langMatches(lang(?v6), '*'))) } .
    FILTER (langMatches(lang(?v5), 'en') || (!langMatches(lang(?v5), '*'))) .
    OPTIONAL {?v4 rdf:label ?v7 }
}

Query Template 13: Historical single-version structured query template 4

# Query Type: 4, Historical single-version structured query

PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>

FROM {{(graphVhistorical)}}
WHERE {
    ?creativeWork cwork:about {{{cwAboutUri}}} .
    FILTER ((STR(?v3) = 'http://www.w3.org/2000/01/rdf-schema#label' && lang(?v4) = 'en') ||
    (STR(?v3) = 'http://dbpedia.org/ontology/abstract' && lang (?v4) = 'en') ||
    (STR(?v3) = 'http://www.w3.org/2000/01/rdf-schema#comment' &&
    lang (?v4) = 'en') ||
    (STR(?v3) != 'http://dbpedia.org/ontology/abstract' &&
    STR(?v3) != 'http://www.w3.org/2000/01/rdf-schema#comment' &&
    STR(?v3) != 'http://www.w3.org/2000/01/rdf-schema#label')) }
}
UNION {
    ?creativeWork cwork:about {{{cwAboutUri}}} .
}
# Query Type: 4, Historical single-version structured query

```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>  
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?creativeWork ?v2  
FROM {{{graphVhistorical}}}
WHERE {  
    ?v3 rdf:type {{{dbpsb}}}  
}
```

Query Template 14: Historical single-version structured query template 5

QT5 - Delta materialization query template

```sparql
# Query Type: 5, Delta materialization

SELECT *  
WHERE {  
    {  
        GRAPH {{{graphVmax}}} { ?s ?p ?o } .  
        FILTER (NOT EXISTS { GRAPH {{{graphVhistorical}}} { ?s ?p ?o } })  
    }  
    UNION  
    {  
        GRAPH {{{graphVhistorical}}} { ?s1 ?p1 ?o1 } .  
        FILTER (NOT EXISTS { GRAPH {{{graphVmax}}} { ?s1 ?p1 ?o1 } })  
    }  
}
```

Query Template 15: Historical single-version structured query template 6

QT6 - Single-delta structured query template

```sparql
# Query Type: 6, Single-delta structured query
# Query Description:  
# Retrieve the difference in the number of Blog Posts between the current and a historical version

PREFIX cwork:<http://www.bbc.co.uk/ontologies/creativework/>

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

Query Template 16: Delta materialization query template
Query Template 17: Single-delta structured query template

Query Template 18: Cross-delta structured query template

Query Template 19: Cross-version structured query templates
Query Template 19: Cross-version structured query template 1

```sql
# Query Type: 8, Cross-version structured query
# Query Description:
# Retrieve from different versions, the creative works that are about or mention different topics, along with the topics type.

PREFIX dbp-prop: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>

FROM {{{graphVstart}}}
FROM {{{fromClauses}}}
FROM {{{graphVmax}}}
WHERE {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  {{{cwAboutUri}}} dbp-prop:subsid ?v3
  OPTIONAL { ?v2 {{{cwAboutUri}}} dbp-prop:parent } 
  OPTIONAL { {{{cwAboutUri}}} dbp-prop:divisions ?v4 }
}
UNION {
  ?v2 [[[cwAboutUri]]] dbp-prop:parent
  OPTIONAL { [[[cwAboutUri]]] dbp-prop:subsid ?v3 } 
  OPTIONAL { [[[cwAboutUri]]] dbp-prop:divisions ?v4 }
}
UNION {
  [[[cwAboutUri]]] dbp-prop:divisions ?v4
  OPTIONAL { [[[cwAboutUri]]] dbp-prop:subsid ?v3 } 
  OPTIONAL { ?v2 [[[cwAboutUri]]] dbp-prop:parent }
}
}
```

Query Template 20: Cross-version structured query template 2

```sql
# Query Type: 8, Cross-version structured query

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?creativeWork ?v0 ?v1 ?v2
FROM {{{graphVstart}}}
FROM {{{fromClauses}}}
FROM {{{graphVmax}}}
WHERE {
  ?creativeWork cwork:about {{{cwAboutUri}}} .
  {{{cwAboutUri}}} rdfs:comment ?v0 .
  FILTER (lang (?v0) = 'en')
}
UNION { [[[cwAboutUri]]] foaf:depiction ?v1 }
UNION { [[[cwAboutUri]]] foaf:homepage ?v2 }
```

Query Template 21: Cross-version structured query template 3
# Query Type: 8, Cross-version structured query

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

FROM {{graphVstart}}
{{fromClauses}}
FROM {{graphVmax}}
WHERE {
  {{{cwMentionsUri}}} ?v4 ?v5 .
    FILTER (langMatches(lang(?v6),'en') || (!langMatches(lang(?v6),'*'))) } .
  FILTER (langMatches(lang(?v5),'en') || (!langMatches(lang(?v5),'*'))) .
  OPTIONAL {?v4 rdfs:label ?v7 }
}
```

Query Template 22: Cross-version structured query template 4

```
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>

FROM {{graphVstart}}
{{fromClauses}}
FROM {{graphVmax}}
WHERE {
  {?creativeWork cwork:about {{{cwAboutUri}}} .
   FILTER ((STR(?v3) = 'http://www.w3.org/2000/01/rdf-schema#label' && lang(?v4) = 'en') ||
    (STR(?v3) = 'http://dbpedia.org/ontology/abstract' && lang(?v4) = 'en') ||
    (STR(?v3) = 'http://www.w3.org/2000/01/rdf-schema#comment' && lang(?v4) = 'en') ||
  }
  UNION {
    {?creativeWork cwork:about {{{cwAboutUri}}} .
     ?v5 ?v3 {{{cwAboutUri}}} .
     FILTER (STR(?v3) = 'http://dbpedia.org/ontology/owner' ||
       STR(?v3) = 'http://dbpedia.org/property/redirect')
    }
  }
}
```

Query Template 23: Cross-version structured query template 5

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX cwork: <http://www.bbc.co.uk/ontologies/creativework/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?creativeWork ?v2
```
FROM {{graphVstart}}
{{fromClauses}}
FROM {{graphVmax}}
WHERE {
  ?v3 rdf:type {{dbpsb}}}

Query Template 24: Cross-version structured query template 6

References


