Collaborative Project
Holistic Benchmarking of Big Linked Data
Project Number: 688227  Start Date of Project: 2015/12/01  Duration: 36 months

Deliverable 5.1.2
Second Version of the Data Storage Benchmark

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due Date of Deliverable</td>
<td>Month 30, 31/05/2018</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 30, 31/05/2018</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP5 - Benchmarks III: Storage and Curation</td>
</tr>
<tr>
<td>Task</td>
<td>T5.1</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>
<tr>
<td>Number of Pages</td>
<td>30</td>
</tr>
</tbody>
</table>

Abstract: This deliverable describes the development of the second version of the Data Storage benchmark for HOBBIT. By introducing new features, such as the parallel execution of the benchmark queries in a scenario based on real-world load, and the time compression ratio parameter, we were able to introduce an extended version of our benchmark for assessing Big Linked Data storage solutions for interactive applications.

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>0.1</td>
<td>11/05/2018</td>
<td>First draft</td>
<td>Mirko Spasić, Milos Jovanovik</td>
</tr>
<tr>
<td>0.2</td>
<td>13/05/2018</td>
<td>Review</td>
<td>Irini Fundulaki</td>
</tr>
<tr>
<td>1.0</td>
<td>25/05/2018</td>
<td>Final version</td>
<td>Mirko Spasić, Milos Jovanovik</td>
</tr>
</tbody>
</table>

Author List

<table>
<thead>
<tr>
<th>Organization</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenLink</td>
<td>Mirko Spasić</td>
<td><a href="mailto:mspasic@openlinksw.com">mspasic@openlinksw.com</a></td>
</tr>
<tr>
<td>OpenLink</td>
<td>Milos Jovanovik</td>
<td><a href="mailto:mjovanovik@openlinksw.com">mjovanovik@openlinksw.com</a></td>
</tr>
</tbody>
</table>
Executive Summary

The demand for efficient RDF storage technologies has been steadily growing in recent years, due to the increasing number of available Linked Data datasets and applications exploiting Linked Data resources. With this, a growing number of applications require RDF storage solutions capable of answering SPARQL queries in interactive times, often over very large datasets. Given the number of available RDF storage solutions, there is an need for objective means to compare technologies from different vendors. Consequently, there is a need for representative benchmarks that mimic the actual workloads present in real-world applications. In addition to helping developers, such benchmarks aim to stimulate technological progress among competing systems and thereby accelerate the maturing process of Big Linked Data software tools.

One aspect in such benchmarking efforts is the benchmarking of data storage. As part of the HOBBIT project, we created the Data Storage benchmark, using the LDBC Social Network Benchmark as a starting point. By introducing important modifications to its synthetic data generator and dataset, and by modifying and transforming its SQL queries to SPARQL, all while preserving the benchmark’s most relevant features, we were able to create the first version of the Data Storage benchmark for HOBBIT. Now, building on our previous work, we introduce new features and provide an extended new version of the benchmark.

This deliverable outlines the details of how the second version of the Data Storage benchmark for HOBBIT was developed and tested. It focuses on the newly introduced parallel execution of the benchmark queries, and the time compression ratio parameter, both of which enable storage evaluation based on a real-world workload. The deliverables also describes the integration of the new version of the benchmark into the HOBBIT platform, and showcases the benchmark results of the baseline implementation.
Contents

1 Introduction 4
   1.1 Requirements for the Data Storage Benchmark 4

2 Data Storage Benchmark Dataset 5

3 Data Storage Benchmark (DSB v2.0) 5
   3.1 Key Performance Indicators 7

4 Integration into the HOBBIT Platform 7
   4.1 DSB Parameters 7
   4.2 Components 8
      4.2.1 Benchmark Controller 9
      4.2.2 Data Generator 9
      4.2.3 Task Generator 10
      4.2.4 System Adapter 11
      4.2.5 Evaluation Module 11
      4.2.6 Common API 12

5 Evaluation 13
   5.1 Sequential Tasks 14
   5.2 Parallel Tasks 14

6 MOCHA Preliminary Challenge Results 18

References 20

A Tables 21

B Virtuoso Configuration 26
1 Introduction

The demand for efficient RDF storage technologies has been steadily growing in recent years. This increase is a result of the growing number of available Linked Data datasets and applications exploiting Linked Data resources. With this, a growing number of applications require RDF storage solutions capable of answering SPARQL queries in interactive times, often over very large datasets. Given the number of available RDF storage solutions, there is a need for objective means to compare technologies from different vendors. Consequently, there is a need for representative benchmarks that mimic the actual workloads present in real-world applications. In addition to helping developers, such benchmarks aim to stimulate technological progress among competing systems and thereby accelerate the maturing process of Big Linked Data software tools.

One aspect in such benchmarking efforts is the benchmarking of data storage. After a careful consideration of all available benchmarks and benchmark generation frameworks for querying Linked Data, we developed the first version of the Data Storage benchmark (DSB) for HOBBIT [6]. The benchmark is based on and extends the Social Network Benchmark (SNB), developed under the auspices of the Linked Data Benchmark Council (LDBC)\(^1\). LDBC introduced a new choke-point driven methodology for developing benchmark workloads, which combines user input with input from expert systems architects [4]. Unlike other benchmarks which are specifically tied to a single technology [2, 7], SNB is much more generic, and can be used for the evaluation of pure graph database systems, systems intended to manage Semantic Web data conforming to the RDF data model, distributed graph processing systems and traditional relational database systems that support recursive SQL.

In the first version of the Data Storage benchmark, we have created a benchmark based on the SNB-Interactive SQL version, along with a synthetic RDF dataset generator based on SNB’s DATAGEN, which was used in LDBC to generate a relational database. For the second version, we introduce parallel execution of the benchmark queries, which simulates a real-world workload for the tested system. We also introduce a new parameter, time compression ratio, which allows an increase / decrease to the speed of query issuing, i.e. modifies the speed of the social network activity simulation. This deliverable provides a detailed description of these new features as part of the second version of the Data Storage benchmark for HOBBIT. The deliverables also describes the integration of the new version into the HOBBIT platform, and showcases the benchmark results of the baseline implementation.

1.1 Requirements for the Data Storage Benchmark

The requirements that the Data Storage benchmark for HOBBIT has to meet, are:

- High insert rate with time-dependent and largely repetitive or cyclic data
- Possible exploitation of structure and physical organization adapted to the key dimensions of the data
- Bulk load support
- Interactive complex read queries
- Simple lookups
- Concurrency

\(^1\)http://www.ldbcouncil.org/
• High throughput

These requirements enable an efficient evaluation of data storage solutions which support interactive workloads in real-world scenarios, over various dataset sizes [1, 5].

2 Data Storage Benchmark Dataset

The second version of the Data Storage benchmark uses the same dataset as the first version. The dataset is a synthetic RDF dataset, created by our data generator (DATAGEN). The DATAGEN was based on the Social Network Benchmark data generator, but was significantly modified and improved for the first version of DSB [6]. It generates synthetic RDF datasets which mimic an online social network and has real-world coherence. It can generate synthetic RDF datasets with a scale factor of 1, 3, 10, 30, 100, etc. The number of triples per scale factor is shown in Table 1.

Table 1: Number of triples in the DSB datasets of different size.

<table>
<thead>
<tr>
<th>SF</th>
<th># Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45.4M</td>
</tr>
<tr>
<td>3</td>
<td>136.5M</td>
</tr>
<tr>
<td>10</td>
<td>464.1M</td>
</tr>
<tr>
<td>30</td>
<td>1428.4M</td>
</tr>
<tr>
<td>100</td>
<td>4645.7M</td>
</tr>
</tbody>
</table>

The DSB DATAGEN is publicly available on GitHub².

3 Data Storage Benchmark (DSB v2.0)

Our Data Storage Benchmark (DSB) is based on the LDBC Social Network Benchmark (SNB) and introduces some important modifications related to the synthetic dataset and the benchmark queries. These changes, along with the choke points on which the queries are based, are described in extended detail in deliverable D5.1.1 [6], where the first version of DSB is presented. The improvements introduced in this version of the benchmark relate to the way the queries are executed, i.e. the query workload. In this second version of DSB, the query workload is more realistic, practical and pragmatic.

DSB v1.0 was single-threaded - at any point in the benchmark run there is only one running query, with the next one starting only after the previous one completes. With this approach, we examined and appraised the best performance of the tested system for each query, by allowing the system to have all resources available for the currently running query.

The KPIs collected using the first version of the benchmark are not useful for the deployment of a system which would serve a social network. Primarily, the purpose of the benchmark is the comparison

²https://github.com/hobbit-project/DataStorageBenchmark-Dev
of different triple stores, but also, in a real scenario, the benchmark should provide the relevant parties with answers to the following question: is it possible to use a particular triple store (e.g. Virtuoso Commercial) as a storage component in the back-end of a social network application, if we have in mind the approximate number of users and available hardware resources? So, for that purpose, the average execution time of specific queries is not very relevant. Here, the throughput of concurrent queries is much more important, and the ability of the system to provide the answers to the multiple queries submitted at the same time, or almost at the same time, in reasonable and interactive times. This perspective served as a guideline for the second version of our benchmark.

The second version of DSB is multithreaded, allowing us to test the concurrency of the system with multiple queries running simultaneously. The majority of benchmarks test the parallel execution of multiple queries by simply having multiple query queues and running query-by-query per queue [3]. Our intent was not to just submit a large amount of queries at the same time, but to provide a workload which mimics real-world activity in the social network domain, i.e. a workload which will be driven by updates in the following sense.

Here, the updates represent social network events, such as adding a person to the social network, joining a group, posting a photo or a comment, liking a post, becoming the friend of a person, etc. All of these events, present in the real-world based synthetic dataset, have timestamps of when they occurred, so we can simulate a workload by using them. For instance, if there is a time delay between two specific events of 10ms, the updates (as INSERT queries) corresponding to them will take place in a time span of 10ms. The workload, just like in SNB, is constructed by taking the updates (as INSERT queries) from the dataset and mixing them with SELECT queries which have predefined frequencies (e.g. Query 1 should be performed once in every 132 update operation). The resulting query mix has two goals in mind [4]: to be somewhat realistic, and to be challenging for the query engine. For the first goal, the ratio between INSERT and SELECT queries is realistically determined (e.g. for each 500 reads there is 1 write in Facebook’s database [8]). For the second goal, the idea is to make sure each query type takes approximately equal amount of CPU time (i.e. queries that touch more data run less frequently) to avoid the workload being dominated by a single query [4]. This way, the workload will mimic the real-world usage of the data store serving the social network application.

This kind of workload will provide us with an answer to the following question: is the system under test capable of serving a social network of the specific size? However, this does not seem enough to test the scalability of the system and the system behavior in a more demanding workload. How can we stress it further? Having in mind the graph characteristics of the dataset, the most common approach here is to provide a larger dataset, a dataset with more triples, i.e. a social network with more users where there are more updates per unit of time [4]. This is obvious because more users provide more posts, comments and activity in general, represented as updates, or more precisely SPARQL INSERT queries. They are bundled with SELECT queries by specified ratio, giving us more SPARQL SELECT queries, as well. So, the benchmark scales well with increasing size of the dataset.

Is it possible to have more demanding workload, while keeping the size of the dataset unmodified? Here, we propose a time compression ratio (TCR), as a target throughput to test. It is used for "squeezing" together or "stretching" apart the queries of the workload. If TCR is equal to 1, the queries (INSERT and SELECT) will be executed according to the real times, mimicking the real distribution of the queries. With increasing TCR, the queries will be submitted faster and faster. The higher the values of TCR, the more demanding the workload. For example, if TCR is equal to 2, the queries which represent activities from the dataset spanning 2 minutes, will be executed (more precisely - started) within 1 minute as part of the benchmark. If TCR is less than 1, for instance 0.5, the workload will be easier on the system than the real one, as the same queries (representing 2 minutes of activity in the dataset) will be executed (i.e. started) within 4 minutes in the benchmark.
3.1 Key Performance Indicators

The key performance indicators (KPIs) for the second version of the Data Storage benchmark are the same as for the first version:

- **Bulk Loading Time**: The total time in milliseconds needed for the initial bulk loading of the dataset.
- **Average Task Execution Time**: The average SPARQL query execution time.
- **Average Task Execution Time Per Query Type**: The average SPARQL query execution time per query type.
- **Query failures**: The number of SPARQL SELECT queries whose result set is different (in any aspect) from the result set obtained from the triple store used as a gold standard.
- **Throughput**: The average number of tasks executed per second.

The only difference is that throughput is no longer the main KPI for the benchmark. In DSB v1.0, it could be calculated as 1000 divided by the average task execution time in ms, because of the sequentiality in the execution of queries. Here, this value completely depends on the specified time compression ratio, which is the benchmark parameter specified at the beginning of the experiment. This dependence is already explained in the previous section. So, we could remove it completely from the KPIs, as it can vary very little when fixing the TCR (in the case when the system under test cannot serve the specified workload). The rationale for keeping it among other KPIs will be given in the next section.

4 Integration into the HOBBIT Platform

The second version of the Data Storage benchmark (DSB v2.0) has been integrated into the HOBBIT platform. All interested parties can evaluate their systems against it by running the benchmark through the platform website[^3]. The users who want to use the platform intensively for testing and debugging, can also deploy it locally. We also have our own deployment at OpenLink Software, which is publicly available[^4]. With the latter approach, the users will not have to wait for the experiments of other users, as is the case at the global project platform, which is important during the development phase.

Interested parties should follow the instructions from the platform Wiki page[^5] – in brief, they need to provide their system in the form of a Docker image and provide a system adapter which implements the corresponding API[^6]. The guidelines for implementing the system adapter are provided at the project website[^7].

4.1 DSB Parameters

The DSB v2.0 parameters which need to be set in order to execute the benchmark are the following:

[^3]: http://master.project-hobbit.eu/
[^4]: http://hobbit-demo.openlinksw.com
[^5]: https://github.com/hobbit-project/platform/wiki
• **Number of operations:** The user must provide the total number of SPARQL queries (SELECT and INSERT) that should be executed against the tested system.

• **Scale factor:** DSB can be executed using different sizes of the dataset, i.e. with different scale factors. The total number of triples per scale factor, along with several other properties of the dataset, are available in D5.1.1 [6].

• **Seed:** The user has to specify the seed for generating the pseudo-random numbers used for non-deterministic selection of query substitution parameters.

• **Time compression ratio (TCR):** This value is used for "squeezing" together or "stretching" apart the queries of the workload, thus making it more or less demanding. The parameter is explained in Section 3.

• **Sequential Tasks:** This check-box is disabled by default, and this means that the multithreaded approach of the benchmark will be applied. We did not want to remove all useful features of DSB v1.0, and therefore we provided an option to integrate them into DSB v2.0. If the check-box is enabled, the queries will be submitted to the tested system sequentially, one query after the other. This option is very useful when someone wants to compare systems with similar performance results. For example, we did a lot of internal experiments comparing Virtuoso v7.2 against Virtuoso v8.0, in order to track the performance of the system across versions. With using DSB v2.0, we were unable to notice some minor performance differences between these two systems, because there is a lot of compensation between queries running at the same time. So, for a finer comparison between similar systems, or between different configurations of a single system, this option is very useful. If this option is enabled, the Time Compression Ratio will be ignored, thus the Throughput KPI will be relevant.

• **Enable/Disable Query Type:** This parameter controls if all query types should be included in the workload. There are different scenarios where it would be useful to disable specific query type(s) for the upcoming experiment, e.g. when a system is unable to cope with a specific type of query, or it takes too long to process it. If a user does not want to invalidate all other results by running the problem-inducing queries as well, this option can be applied. By default, all query types are enabled and a user can provide a string for this parameter containing '0' at the position corresponding to the query type that should be disabled. For example, the string '01100' will disable Q1, Q4 and Q5 in the experiment, while all the other will be enabled. It would be easier for the user to have a separate check-box for each query type that should be included in the workload, but it would result in too many benchmark parameters; an unwelcome UI complication.

• **Warm-up Percent:** This parameter presents the percentage of total number of operations that should be used in the warm-up phase. These queries will not be evaluated later, and their purpose is only for system warm-up.

Figure 1 illustrates an example configuration of the Data Storage benchmark.

### 4.2 Components

Here, a brief description of the main components of the Data Storage benchmark is given, highlighting the differences compared to the first version. A detailed presentation of the first version of DSB is provided in D5.1.1 [6]. The implementation of the components can be found as part of the public
GitHub project\(^8\). The current implementation is in the master branch, but the previous versions are tagged with specific tags: v1.0, v1.1 and v1.2.

### 4.2.1 Benchmark Controller

The Benchmark Controller is the main component used to create and synchronize the other components of the benchmark. During the initialization phase, it gathers parameters from the Platform Controller, provided by a platform user, necessary to initialize the other components. Then the Benchmark Controller executes the Data Storage benchmark by sending start signals to the Data Generator and the Task Generator. When they are done, it creates the Evaluation Module of the benchmark, waits for it to finish and sends the received evaluation results of the benchmark to the Platform Storage. From there, the platform can read the results, and present them to the user. This is a common scenario for the controller component - the same scenario is implemented by all other benchmarks in the platform. This is the reason why this component has been undergone only minimal changes from DSB v1.0 to DSB v2.0. These minimal changes are related to using and passing the new DSB parameters.

### 4.2.2 Data Generator

The Data Generator is the component responsible for providing the dataset. Due to the nature of the synthetic RDF dataset(s) generated with our DSB DATAGEN, there is no need to generate them for each benchmark run separately. Additionally, the generation of the larger datasets can take a couple of hours. So, instead, DSB uses pre-generated datasets when used within the HOBBIT platform\(^6\). After the initialization phase, the Data Generator is aware of the designated scale factor, downloads the specified dataset and sends it to the System Adapter to be bulk loaded. When it receives the signal indicating that the bulk loading phase is finished, it starts sending the dataset to the Task Generator,
based on which it prepares the queries. The main differences introduced in the second version of the benchmark for this component are related to the ability of generating more datasets of larger scales, and providing the relevant data to the Task Generator needed for queries, instead of sending the pre-calculated queries to it. This way, the Task Generator has the freedom to generate the queries by itself, opening up opportunities for more realistic workloads.

4.2.3 Task Generator

The Task Generator is the component responsible for creating the workload. It creates the tasks, i.e. the SPARQL SELECT and INSERT queries, based on the incoming data. The deliverable for the first version of the benchmark explained two approaches related to the process of checking the correctness of the query results, and gave the rationale for using a pre-calculated set of results [6]. This component receives the data from the Data Generator, prepares the queries, retrieves the pre-calculated answers to them and then sends the queries to the System Adapter, and the answers to the Evaluate Storage. Compared to the DSB v1.0, this component has gone through a significant amount of changes. In the first version we use sequential tasks, i.e. the Task Generator sends the tasks one by one, waits for the system under test to process a task in full before sending the next one to the System Adapter. This is done by synchronizing the Task Generator and the Evaluation Storage. In the second version of DSB, we have two types of Task Generators: one which is very similar to the Task Generator from DSB v1.0 where sequential tasks are used, and a second one which runs the tasks in parallel and is more realistic and more useful for comparing different systems.

The Task Generator of DSB 2.0 does not synchronize with the other components - the queries are submitted according to the specified workload which encompasses the real-world frequencies, distributions and mutual ratios of the separate queries. The behavior of the Task Generator in this phase is also affected by the designated time compression ratio, which provides an additional way of testing the scalability, as we explained before. Additionally, the Task Generator of DSB 2.0 implements the possibility to disable one or more types of queries, in order to get a workload without them. As we already pointed out, this is very useful for debugging and tweaking the system under test.

The following is a pseudo-code for the most important method of the Task Generator, responsible for generating the tasks:

```java
protected void generateTask(byte[] data) {
    // convert the data received from the Data Generator into suitable form
    String dataString = convert(data);

    // extract and prepare the exact time when the task has to be submitted
    long newSimulatedTime = extract_and_prepare(dataString);
    // current time
    long newRealTime = System.currentTimeMillis();

    // wait for the right time having in mind the time compression ratio
    long waitingTime = calculate_wait_time(newSimulatedTime, newRealTime, tcr)
    wait(waitingTime);

    // prepare SPARQL INSERT query
    String updateString = prepare_update(dataString);

    // if that type is not disabled, submit the query
    submit(updateString);
}
```
if (!disabled(update_type(updateString)))
    sendTaskToSystemAdapter(updateString);

    // for all SPARQL SELECT queries
    for (type : select_types) {
        // if it is the right time to submit the specific query type according to workload
        // and the query type is not disabled
        if (can_submit(type) && !disable(type)) {
            // prepare text of a query
            String selectString = prepareSelectQuery(type);
            // prepare answer of a query
            String answer = prepareAnswer(selectString);

            // sent query to the System Adapter
            sendTaskToSystemAdapter(taskIdString, selectString);
            // sent answer and the timestamp of the current time to the Evaluation Storage
            sendTaskToEvalStorage(current_timestamp(), answer);
        }
    }

4.2.4 System Adapter

The System Adapter is a component which enables the communication between the benchmark and the system under test. It receives data from Data Generator, loads it into the system, and then informs the other components that it is ready to start executing the SPARQL queries. It then gets the queries from the Task Generator and forwards them to the system being tested. The results of the queries are sent to the Evaluation Storage for validation and evaluation, where the correctness, timeliness and other significant characteristics are measured.

We developed a couple of instances of the System Adapter. One of them is used for our baseline implementation of DSB, for which we used the Virtuoso open-source version (VOS). We developed other instances for the commercial versions of Virtuoso v7.2, v7.5, v8.0 and v8.1 (beta). For each of them the System Adapter is mostly the same, and it is part of the same Docker container as the system.

This benchmark is a part of the Mighty Storage Challenge (MOCHA), where we have a common API for all benchmarks which comprise it (Section 4.2.6). That means that all systems which follow the same API could be tested against any benchmark from the MOCHA challenge (ODIN, DSB, Versioning, and Faceted benchmark). Practically, all the systems developed as a baseline for any of these tasks could be used with our benchmark, and vice versa. Apart from our systems, we tested Blazegraph, GraphDB and Apache Jena Fuseki, for which the system adapters were developed within different work packages.

4.2.5 Evaluation Module

The Evaluation Module is the component which validates and evaluates the results received from the benchmarked system. The answers to the queries are pre-calculated using the golden standard, and then are compared to the actual answers that the benchmarked system provided. For each SPARQL
query, the Evaluation Storage saves the execution start-time and end-time, along with the expected result set. Based on these measurements, this component calculates all specified key performance indicators (KPIs) described in Section 3.1, and sends them to the platform (as an RDF model) to be presented to the user.

The KPIs of the experiments running against our second version of DSB are the ones described in Section 3.1, and they can be summarized and compared within the platform results page.

4.2.6 Common API

In the scope of the HOBBIT project, we have 4 different benchmarks that evaluate the performance of triple stores from different aspects:

- ODIN: An RDF data ingestion benchmark that measures how well systems can ingest streams of RDF data.
- DSB: The Data storage benchmark that measures how data stores perform with different types of queries.
- Versioning Benchmark: The Versioning benchmark measures how well versioning and archiving systems for Linked Data perform when they store multiple versions of large datasets.
- Faceted Benchmark: The Browsing Benchmark checks existing solutions for how well they support applications that need browsing through large datasets.

We developed the components of these benchmarks following the common API that we jointly approved. All systems should follow the same API, thus will be ready to be evaluated with any of these benchmarks. In this section, we present the common API.

A SPARQL based system has four phases during the benchmarking:

1. Initialization phase: As every system, it has to initialize itself, e.g., start needed components, load configurations, etc. This phase ends as soon as it sends the SYSTEM_READY_SIGNAL on the command queue (as described in the wiki and implemented in the AbstractSystemAdapter).

2. Loading phase: After the system is running and the benchmark is started, it can receive data from the data queue which it should load into its triple store. This can be done as bulk load. The benchmark controller will send a BULK_LOAD_DATA_GEN_FINISHED signal on the command queue when the data from the data generator has been placed in the data queue. The BULK_LOAD_DATA_GEN_FINISHED message comes with an integer value (4 bytes) as additional data, representing the number of data messages the system adapter should get from the data queue. Note that the benchmark controller might have to collect the number of generated data messages from the data generators. In addition, the BULK_LOAD_DATA_GEN_FINISHED messages contains a flag that determines whether there are more data that have to be sent by the benchmark controller. This flag lets the system enter the querying phase, or makes it wait for additional data to come. More specifically, the system will read the remaining data from the data queue, bulk load it into the store and send a BULK_LOADING_DATA_FINISHED signal on the command queue to the benchmark controller, to indicate that it has finished the loading. If the

---

9 https://github.com/hobbit-project/core/blob/master/src/main/java/org/hobbit/core/components/AbstractSystemAdapter.java
flag of `BULK_LOAD_DATA_GEN_FINISHED` command was `false`, it waits for the next set of data to come, bulk load it into the store and send the `BULK_LOADING_DATA_FINISHED` signal again on the command queue. If the flag is `true` it can proceed to the querying phase. The values of the aforementioned commands are:

```
BULK_LOADING_DATA_FINISHED = (byte) 150;
BULK_LOAD_DATA_GEN_FINISHED = (byte) 151;
```

The data received at that time is structured in the following way:

- Integer value (4 bytes) containing the length of the graph URI
- Graph URI (UTF-8 encoded String)
- NTriples (UTF-8 encoded String; the rest of the package/data stream)

Example workflow:

01. lastBulkLoad = false
02. while lastBulkLoad is false do
03.  numberOfMessages = X
04.  benchmark sends data to system
05.  if there are no more data for sending then
06.     lastBulkLoad = true
07.  end if
08.  benchmark sends `BULK_LOAD_DATA_GEN_FINISHED` { numberOfMessages, lastBulkLoad }
09.  system loads data
10.  system sends `BULK_LOADING_DATA_FINISHED`
11.  done
12.  system enters querying phase

For the benchmarks that measure the time it takes a system to load the data, the times from step 8 to 10 are measured.

3. Querying phase: During this phase the system can get two types of input:

- Data from the data queue that should be inserted into the store in the form of INSERT SPARQL queries.
- Tasks from the task queue, i.e., SPARQL queries (SELECT, INSERT, etc.) that it has to execute. The results for the single tasks (in JSON format) have to be sent together with the id of the task to the result queue.

4. Termination phase: As described in the wiki, the third phase ends when the system receives the `TASK_GENERATION_FINISHED` command and has consumed all the remaining messages from the task queue. (The `AbstractSystemAdapter` already contains this step.)

5 Evaluation

As we mentioned in Section 4.2.4, in the scope of the HOBBIT project we developed four different baseline systems implementing the common API, and that they could be tested against DSB. This
section should not be considered as a comparison of the tested systems, for two reasons: (a) the experiments we ran were intended to prove that our benchmark works as intended and expected, and (b) the baseline systems developed within other workpackages are not capable of answering queries in reasonable time, even for the smallest dataset (also verified with the MOCHA challenge results in Section 6). Because of that, in the experiments that follow we use Virtuoso open-source (VOS) as the system under test. The version of VOS is 07.20.3217 and its configuration file is given in Appendix B. The experiments presented in this section were executed on the HOBBIT platform.

5.1 Sequential Tasks

Here, we present the experiments where we enabled the benchmark parameter sequential tasks, i.e. we ran the sequential version of the Data Storage benchmark (DSB v1.0). This parameter is explained in more details in Section 4.1. Before running any of the large, complex and demanding experiments, we wanted to have an estimation of how complex are the queries for the system when they are executed sequentially, one after the other (Table 3). In D5.1.1, we had similar experiments, and the results were very similar.

For the largest scale factor tested (SF=30), the average query execution time is almost 1s. This is so high, because there are a couple of query types for which the query execution plan is not good, and their execution takes much more time than expected. Figure 2 shows the slowly increasing execution times over scale factors.

![Figure 2: Sequential Tasks: Long Queries (Q1-Q14), Short Queries (S1-S7), Updates (U2-U8).](image)

5.2 Parallel Tasks

From the previous chapter, it is obvious that some of the query types are more complex for the system than others, and some of them cannot be considered as interactive. More precisely, the queries
themselves are developed to be interactive, and the audited results of non-RDF systems tested within the LDBC project against the SQL version of these queries are the proof of that.\textsuperscript{10} But here, the tested triple stores are not capable of executing these queries in a short enough time so that they can be considered as interactive (i.e. execute in less than a couple of seconds). If we include these query types in the real workload, where the queries are submitted very fast, and several at once, they will take a lot of time to execute, they will pile up, and after some time there will be lot of them executing at once, occupying all hardware resources and paralyzing the newly arrived queries. The delay will be increasing constantly, thus invalidating the whole experiment.

With this in mind, we decided to temporarily disable these long lasting query types, and keep only fast ones for the following set of experiments. We achieved this by setting the benchmark parameter Enable/Disable Query Type to 01001111011010111111111111111. We executed VOS against DSB, while varying different parameters:

- Scale Factors (SF): 1, 3, 10, 30
- Time Compression Ratio (TCR): 0.03, 0.1, 0.3, 1, 3, 10, 30, 100, 300

In all experiments, the number of operations (SPARQL queries) is 20000, out of which 20\% are used for system warm-up.

In Table 4, we present the benchmark parameters and the achieved KPIs for six different experiments, all dealing with the smallest dataset (SF=1). The only varying parameter is the Time Compression Ratio (TCR). We start from a less demanding workload where TCR is equal to 3, increasing it up to 1000. From Table 4 given in Appendix 6 and from the graph charts on Figure 3 it is obvious that our system can cope with all tested TC\(\text{Rs},\) except the last one, where the average query execution time is more than 20s. In that experiment, the throughput is more than 350 queries per second and the system is not able to serve the queries with a reasonable delay. More precisely, Figure 3 shows that the average query execution time per query type is almost the same for the first four experiments (TCR=3, TCR=10, TCR=30 and TCR=100), which tells us that during these experiments the system is not under a full load. For the next experiment (TCR=300), we have a noticeable increase in the execution times, but they are still interactive times. In the last one, the increase is very large and the delay accumulates over time. The real timespan covering the social network activity in this workload is about 270 minutes long, but it is simulated in about 16 seconds (TCR=1000).

In the next set of experiments we use the larger dataset (SF=3), with varying values for TCR. There is no need to test the same system with the parameter TCR=1000, because we already saw that the system was not able to deal with that throughput even with the smallest DSB dataset. So, now the dataset is 3 times larger, and the values for TCR are 3 times lower (1, 3, 10, 30, 100 and 300). Table 5 and Figure 4 give us the same conclusions as for the smallest dataset of SF=1.

We continued to test our system with larger datasets. Table 6 and Figure 5 correspond to SF=10, while the larger experiments (SF=30) are presented in Table 7 and Figure 6. Every time we increase the dataset by a factor of 3, we decrease the corresponding values for TCR by 3 as well. Therefore, in the larger experiments (SF=30) with a 4.6 billion triple dataset, we show that our system can cope with the real world throughput (TCR=1), but also with a throughput which is 3 times higher (TCR=3).

Figure 7 explains the workload in general and show how throughput depends on the time compression ratio and the scale factor. It is obvious that throughput linearly increases in accordance with the scale factor and the time compression ratio, as well. The small anomaly that is seen in the left part of Figure 7 related to the last tested time compression ratio for each scale factor, is present because of the

\textsuperscript{10}http://ldbcouncil.org/benchmarks/snb
accumulated delays for the highest throughput. If we consider SF=1, the throughput for TCR=1000 should be 3 times higher than for TCR=300, e.g., about 1100 queries per second, but the achieved throughput is only 354. This can also be seen on the right part of Figure 7.
Figure 5: Scale Factor 10: Long Queries (Q2-Q13), Short Queries (S1-S7), Updates (U1-U8).

Figure 6: Scale Factor 30: Long Queries (Q2-Q13), Short Queries (S1-S7), Updates (U2-U8).

Figure 8 shows the dependence of the average query execution time on the time compression ratio and the scale factor.

Figure 9 depicts the average loading time per scale factor.
6 MOCHA Preliminary Challenge Results

At the end of the reporting period for the deliverable, the Data Storage Benchmark was part of the Mighty Storage Challenge (MOCHA) 2018\textsuperscript{11}, at ESWC 2018\textsuperscript{12}. The challenge consisted of four tasks, and DSB was Task 2. The benchmark had a defined maximum time for the experiment of 3 hours. Table 2 shows the configuration for DSB at MOCHA 2018.

For the participation in Task 2 of MOCHA 2018, i.e. for the Data Storage Benchmark, five systems applied and were submitted: Virtuoso 7.2 Open-Source Edition by OpenLink Software, Virtuoso 8.0

\textsuperscript{11} https://project-hobbit.eu/challenges/mighty-storage-challenge2018/
\textsuperscript{12} https://2018.eswc-conferences.org/
Commercial Edition by OpenLink Software, Blazegraph, GraphDB Free 8.5, and Apache Jena Fuseki 3.6.0.

Table 2: MOCHA 2018 - DSB Configuration.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale Factor</td>
<td>30</td>
</tr>
<tr>
<td>Number of operations</td>
<td>15000</td>
</tr>
<tr>
<td>Time Compression Ratio</td>
<td>0.5</td>
</tr>
<tr>
<td>Enable/Disable Query Type</td>
<td>*</td>
</tr>
<tr>
<td>Seed</td>
<td>100</td>
</tr>
<tr>
<td>Sequential Tasks</td>
<td>false</td>
</tr>
<tr>
<td>Warmup Percent</td>
<td>20</td>
</tr>
</tbody>
</table>

Unfortunately, Blazegraph, GraphDB and Jena Fuseki were not able to finish the experiment in the requested time, i.e. it exhibited a timeout. Based on the results from the main KPIs (Figure 10) and the rest of the KPIs (Figure 11), the winning system for the task was Virtuoso 8.0 Commercial Edition by OpenLink Software.

Figure 10: MOCHA 2018 - Task 2: Short Queries and Updates

* = 01100111010101111111111111111
Figure 11: MOCHA 2018 - Task 2: Average Query Execution Time per Query Type

References


## A Tables

### Table 3: Sequential Tasks.

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>1</th>
<th>3</th>
<th>10</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCR</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Seed</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Enable/Disable Query Type</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Enable Sequential Tasks</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>Number of Operations</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>78.14</td>
<td>187.35</td>
<td>379.25</td>
<td>899.42</td>
</tr>
<tr>
<td>Average Query01 Execution Time</td>
<td>3595.73</td>
<td>3906.22</td>
<td>6819.36</td>
<td>15088.27</td>
</tr>
<tr>
<td>Average Query02 Execution Time</td>
<td>224.82</td>
<td>426.72</td>
<td>550.97</td>
<td>669.51</td>
</tr>
<tr>
<td>Average Query03 Execution Time</td>
<td>111.19</td>
<td>118.74</td>
<td>218.19</td>
<td>524.94</td>
</tr>
<tr>
<td>Average Query04 Execution Time</td>
<td>891.63</td>
<td>4987.81</td>
<td>27999.67</td>
<td>91529.82</td>
</tr>
<tr>
<td>Average Query05 Execution Time</td>
<td>2881.18</td>
<td>21781.49</td>
<td>19201.00</td>
<td>35698.52</td>
</tr>
<tr>
<td>Average Query06 Execution Time</td>
<td>312.00</td>
<td>621.60</td>
<td>1895.40</td>
<td>496.60</td>
</tr>
<tr>
<td>Average Query07 Execution Time</td>
<td>432.12</td>
<td>1587.03</td>
<td>924.57</td>
<td>1753.32</td>
</tr>
<tr>
<td>Average Query08 Execution Time</td>
<td>38.83</td>
<td>47.73</td>
<td>68.15</td>
<td>70.04</td>
</tr>
<tr>
<td>Average Query09 Execution Time</td>
<td>8343.25</td>
<td>14961.38</td>
<td>32335.38</td>
<td>54364.00</td>
</tr>
<tr>
<td>Average Query10 Execution Time</td>
<td>544.94</td>
<td>781.56</td>
<td>1519.05</td>
<td>3307.79</td>
</tr>
<tr>
<td>Average Query11 Execution Time</td>
<td>31.99</td>
<td>39.20</td>
<td>49.83</td>
<td>76.11</td>
</tr>
<tr>
<td>Average Query12 Execution Time</td>
<td>198.22</td>
<td>235.18</td>
<td>336.18</td>
<td>432.32</td>
</tr>
<tr>
<td>Average Query13 Execution Time</td>
<td>126.53</td>
<td>305.73</td>
<td>398.48</td>
<td>638.28</td>
</tr>
<tr>
<td>Average Query14 Execution Time</td>
<td>1441.26</td>
<td>5282.79</td>
<td>13054.95</td>
<td>17006.48</td>
</tr>
<tr>
<td>Average S1 Execution Time</td>
<td>23.01</td>
<td>29.37</td>
<td>28.64</td>
<td>32.68</td>
</tr>
<tr>
<td>Average S2 Execution Time</td>
<td>54.54</td>
<td>60.83</td>
<td>67.64</td>
<td>77.79</td>
</tr>
<tr>
<td>Average S3 Execution Time</td>
<td>10.08</td>
<td>10.41</td>
<td>12.73</td>
<td>14.79</td>
</tr>
<tr>
<td>Average S4 Execution Time</td>
<td>8.40</td>
<td>8.85</td>
<td>9.94</td>
<td>11.50</td>
</tr>
<tr>
<td>Average S5 Execution Time</td>
<td>6.11</td>
<td>6.22</td>
<td>7.29</td>
<td>7.81</td>
</tr>
<tr>
<td>Average S6 Execution Time</td>
<td>14.57</td>
<td>14.04</td>
<td>14.76</td>
<td>19.49</td>
</tr>
<tr>
<td>Average S7 Execution Time</td>
<td>28.74</td>
<td>27.67</td>
<td>29.71</td>
<td>39.88</td>
</tr>
<tr>
<td>Average Update2 Execution Time</td>
<td>10.67</td>
<td>11.68</td>
<td>14.91</td>
<td>30.21</td>
</tr>
<tr>
<td>Average Update3 Execution Time</td>
<td>10.26</td>
<td>15.73</td>
<td>14.71</td>
<td>20.49</td>
</tr>
<tr>
<td>Average Update4 Execution Time</td>
<td>10.39</td>
<td>15.40</td>
<td>19.08</td>
<td>20.10</td>
</tr>
<tr>
<td>Average Update5 Execution Time</td>
<td>9.74</td>
<td>9.85</td>
<td>13.42</td>
<td>13.91</td>
</tr>
<tr>
<td>Average Update6 Execution Time</td>
<td>10.54</td>
<td>12.26</td>
<td>17.40</td>
<td>21.76</td>
</tr>
<tr>
<td>Average Update7 Execution Time</td>
<td>13.29</td>
<td>17.43</td>
<td>23.15</td>
<td>35.73</td>
</tr>
<tr>
<td>Average Update8 Execution Time</td>
<td>15.22</td>
<td>14.77</td>
<td>19.00</td>
<td>28.61</td>
</tr>
<tr>
<td>Loading Time</td>
<td>254592</td>
<td>319601</td>
<td>939616</td>
<td>3392135</td>
</tr>
<tr>
<td>Throughput</td>
<td>12.23</td>
<td>5.23</td>
<td>2.62</td>
<td>1.11</td>
</tr>
</tbody>
</table>
Table 4: Scale Factor 1.

| TCR  | Scale Factor | Seed   | Enable/Disable Query Type | Enable Sequential Tasks | Warmup Percent | Number of Operations | Average Query Execution Time | Average Query02 Execution Time | Average Query03 Execution Time | Average Query06 Execution Time | Average Query07 Execution Time | Average Query08 Execution Time | Average Query10 Execution Time | Average Query11 Execution Time | Average Query13 Execution Time | Average S1 Execution Time | Average S2 Execution Time | Average S3 Execution Time | Average S4 Execution Time | Average S5 Execution Time | Average S6 Execution Time | Average S7 Execution Time | Average Update1 Execution Time | Average Update2 Execution Time | Average Update3 Execution Time | Average Update4 Execution Time | Average Update5 Execution Time | Average Update6 Execution Time | Average Update7 Execution Time | Average Update8 Execution Time | Loading Time | Query Failures | Throughput |
|------|--------------|--------|---------------------------|-------------------------|-----------------------|--------------------|-----------------------------|-------------------------------|-------------------------------|-------------------------------|-------------------------------|-------------------------------|-------------------------------|-------------------------------|-------------------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 3    | 10           | 30     | 100                       | false                   | false                | 20                | 20000                       | 51.45                         | 268.04                        | 150.44                        | 192.00                        | 635.30                        | 58.16                        | 496.37                        | 53.14                        | 157.37                        | 53.32                        | 91.79                        | 32.86                        | 31.59                        | 37.90                        | 54.60                        | 182.00                        | 29.55                        | 29.15                        | 37.70                        | 54.60                        | 25.70                        | 27.66                        | 39.89                        | 43.37                        | 253923                       | 8                         | 369.11           | 3.69            |
| 10   | 30           | 100    | 100                       | false                   | false                | 20                | 20000                       | 46.48                         | 261.32                        | 150.41                        | 249.45                        | 651.13                        | 56.27                        | 474.67                        | 52.99                        | 163.68                        | 48.09                        | 89.97                        | 26.02                        | 24.76                        | 31.62                        | 29.58                        | 51.79                        | 121.00                        | 22.99                        | 23.86                        | 31.26                        | 44.83                        | 54.00                        | 251.00                        | 22.19                        | 22.86                        | 25857                         | 4                         | 354.39           |
| 30   | 100          | 100    | 100                       | false                   | false                | 20                | 20000                       | 43.21                         | 289.66                        | 149.28                        | 382.00                        | 498.84                        | 52.93                        | 601.16                        | 49.16                        | 156.60                        | 42.39                        | 85.97                        | 22.44                        | 21.25                        | 29.58                        | 27.28                        | 70.56                        | 144.00                        | 21.22                        | 21.39                        | 29.86                        | 45.06                        | 51.00                        | 253.00                        | 17.15                        | 17.01                        | 257857                        | 8                         | 369.11           |
| 100  | 1000         | 100    | 100                       | false                   | false                | 20                | 20000                       | 40.69                         | 230.22                        | 138.00                        | 355.09                        | 560.09                        | 49.07                        | 557.58                        | 45.05                        | 153.93                        | 37.93                        | 70.56                        | 19.85                        | 19.21                        | 27.28                        | 45.06                        | 630.63                        | 124.00                        | 21.22                        | 21.39                        | 29.58                        | 45.06                        | 610.10                        | 253.00                        | 17.15                        | 17.01                        | 257857                        | 8                         | 354.39           |
| 300  | 1000         | 100    | 100                       | false                   | false                | 20                | 20000                       | 624.04                        | 910.39                        | 702.38                        | 729.36                        | 1021.10                       | 604.83                       | 1004.16                       | 600.19                       | 681.35                        | 594.08                        | 630.63                       | 582.90                       | 592.24                       | 650.09                       | 584.28                       | 601.10                       | 207.00                       | 687.79                       | 709.11                       | 605.37                       | 575.56                       | 709.11                       | 25575.00                     | 6                         | 369.11           |
| 1000 | 1000         | 100    | 100                       | false                   | false                | 20                | 20000                       | 20448.63                      | 20601.23                      | 20423.81                      | 20679.18                      | 20922.58                      | 20424.53                      | 20822.38                      | 20405.22                      | 20477.95                      | 20404.14                      | 20436.31                      | 20396.16                      | 20398.24                      | 20398.33                      | 20407.21                      | 20423.27                      | 20530.10                      | 20597.83                      | 21703.37                      | 20309.23                      | 20916.91                      | 20728.67                      | 21446.65                     | 209264                       | 6                         | 354.39           |
Table 5: Scale Factor 3.

<table>
<thead>
<tr>
<th>TCR</th>
<th>1</th>
<th>3</th>
<th>10</th>
<th>30</th>
<th>100</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale Factor</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Seed</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Enable/Disable Query Type</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Enable Sequential Tasks</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Warmup Percent</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Number of Operations</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>61.48</td>
<td>54.47</td>
<td>53.04</td>
<td>48.00</td>
<td>10215.03</td>
<td>45024.27</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>374.60</td>
<td>382.68</td>
<td>456.68</td>
<td>360.02</td>
<td>10422.48</td>
<td>45291.61</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>161.94</td>
<td>157.94</td>
<td>158.00</td>
<td>152.53</td>
<td>10220.22</td>
<td>44879.34</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>645.91</td>
<td>645.18</td>
<td>657.09</td>
<td>654.09</td>
<td>10817.09</td>
<td>45098.64</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>2231.96</td>
<td>714.86</td>
<td>2213.90</td>
<td>1949.81</td>
<td>11332.00</td>
<td>47582.35</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>66.85</td>
<td>61.27</td>
<td>57.39</td>
<td>49.83</td>
<td>10162.09</td>
<td>44964.15</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>824.21</td>
<td>819.09</td>
<td>825.07</td>
<td>819.23</td>
<td>10973.86</td>
<td>45705.73</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>64.14</td>
<td>62.90</td>
<td>55.89</td>
<td>48.17</td>
<td>10153.89</td>
<td>44926.73</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>255.48</td>
<td>364.93</td>
<td>265.93</td>
<td>302.41</td>
<td>10433.94</td>
<td>45217.93</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>51.31</td>
<td>46.51</td>
<td>37.68</td>
<td>38.53</td>
<td>10141.17</td>
<td>44930.25</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>83.47</td>
<td>92.60</td>
<td>84.95</td>
<td>66.47</td>
<td>10187.32</td>
<td>44967.41</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>32.01</td>
<td>30.38</td>
<td>21.04</td>
<td>16.90</td>
<td>10139.24</td>
<td>44929.59</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>30.04</td>
<td>29.19</td>
<td>20.06</td>
<td>16.61</td>
<td>10142.30</td>
<td>44932.97</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>24.92</td>
<td>23.61</td>
<td>15.39</td>
<td>13.00</td>
<td>10142.86</td>
<td>44933.60</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>38.05</td>
<td>35.95</td>
<td>28.32</td>
<td>23.65</td>
<td>10152.35</td>
<td>44945.21</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>58.99</td>
<td>51.85</td>
<td>47.15</td>
<td>47.20</td>
<td>10170.24</td>
<td>44964.51</td>
</tr>
<tr>
<td>Average Update Execution Time</td>
<td>28.05</td>
<td>24.77</td>
<td>20.30</td>
<td>17.98</td>
<td>10391.47</td>
<td>45623.29</td>
</tr>
<tr>
<td>Average Update Execution Time</td>
<td>28.96</td>
<td>25.46</td>
<td>19.10</td>
<td>17.16</td>
<td>10415.29</td>
<td>45149.80</td>
</tr>
<tr>
<td>Average Update Execution Time</td>
<td>37.40</td>
<td>31.57</td>
<td>24.37</td>
<td>18.53</td>
<td>10052.63</td>
<td>43690.07</td>
</tr>
<tr>
<td>Average Update Execution Time</td>
<td>27.26</td>
<td>25.14</td>
<td>17.43</td>
<td>17.34</td>
<td>10435.24</td>
<td>45543.89</td>
</tr>
<tr>
<td>Average Update Execution Time</td>
<td>28.63</td>
<td>26.63</td>
<td>17.67</td>
<td>20.05</td>
<td>10012.84</td>
<td>42705.29</td>
</tr>
<tr>
<td>Average Update Execution Time</td>
<td>34.95</td>
<td>28.71</td>
<td>27.49</td>
<td>25.61</td>
<td>10395.02</td>
<td>45644.99</td>
</tr>
<tr>
<td>Average Update Execution Time</td>
<td>46.74</td>
<td>51.27</td>
<td>33.71</td>
<td>37.61</td>
<td>10622.35</td>
<td>47749.81</td>
</tr>
<tr>
<td>Loading Time</td>
<td>316664</td>
<td>316494</td>
<td>328159</td>
<td>317247</td>
<td>322208</td>
<td>324097</td>
</tr>
<tr>
<td>Query Failures</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Throughput</td>
<td>3.42</td>
<td>10.27</td>
<td>34.25</td>
<td>102.80</td>
<td>270.79</td>
<td>181.21</td>
</tr>
</tbody>
</table>

* = 01100111011011111111111111111111
Table 6: Scale Factor 10.

<table>
<thead>
<tr>
<th>TCR</th>
<th>0.3</th>
<th>1</th>
<th>3</th>
<th>10</th>
<th>30</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale Factor</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Seed</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Enable/Disable Query Type</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Enable Sequential Tasks</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Warmup Percent</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Number of Operations</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>71.31</td>
<td>79.07</td>
<td>59.40</td>
<td>57.96</td>
<td>23338.14</td>
<td>57513.94</td>
</tr>
<tr>
<td>Average Query02 Execution Time</td>
<td>615.01</td>
<td>589.00</td>
<td>831.86</td>
<td>570.33</td>
<td>23668.28</td>
<td>57880.89</td>
</tr>
<tr>
<td>Average Query03 Execution Time</td>
<td>267.78</td>
<td>267.13</td>
<td>239.50</td>
<td>206.34</td>
<td>23407.41</td>
<td>57356.19</td>
</tr>
<tr>
<td>Average Query06 Execution Time</td>
<td>2064.00</td>
<td>7489.18</td>
<td>2358.73</td>
<td>2460.00</td>
<td>25537.91</td>
<td>67469.82</td>
</tr>
<tr>
<td>Average Query07 Execution Time</td>
<td>835.83</td>
<td>3256.71</td>
<td>880.54</td>
<td>797.68</td>
<td>23919.01</td>
<td>58094.57</td>
</tr>
<tr>
<td>Average Query08 Execution Time</td>
<td>72.89</td>
<td>67.13</td>
<td>64.36</td>
<td>55.85</td>
<td>23276.22</td>
<td>57441.14</td>
</tr>
<tr>
<td>Average Query09 Execution Time</td>
<td>1549.42</td>
<td>2193.72</td>
<td>1645.80</td>
<td>2457.39</td>
<td>24820.22</td>
<td>59606.41</td>
</tr>
<tr>
<td>Average Query11 Execution Time</td>
<td>83.51</td>
<td>75.18</td>
<td>72.02</td>
<td>59.97</td>
<td>23261.33</td>
<td>57395.05</td>
</tr>
<tr>
<td>Average Query12 Execution Time</td>
<td>410.22</td>
<td>401.34</td>
<td>379.43</td>
<td>413.39</td>
<td>23692.61</td>
<td>57717.26</td>
</tr>
<tr>
<td>Average S1 Execution Time</td>
<td>56.47</td>
<td>47.90</td>
<td>42.35</td>
<td>34.94</td>
<td>23249.06</td>
<td>57398.58</td>
</tr>
<tr>
<td>Average S2 Execution Time</td>
<td>92.34</td>
<td>82.38</td>
<td>87.02</td>
<td>67.44</td>
<td>23290.71</td>
<td>57442.18</td>
</tr>
<tr>
<td>Average S3 Execution Time</td>
<td>40.88</td>
<td>29.49</td>
<td>23.06</td>
<td>18.50</td>
<td>23246.12</td>
<td>57399.56</td>
</tr>
<tr>
<td>Average S4 Execution Time</td>
<td>35.65</td>
<td>27.02</td>
<td>21.79</td>
<td>18.05</td>
<td>23249.51</td>
<td>57403.08</td>
</tr>
<tr>
<td>Average S5 Execution Time</td>
<td>32.03</td>
<td>22.40</td>
<td>17.05</td>
<td>15.58</td>
<td>23250.32</td>
<td>57404.22</td>
</tr>
<tr>
<td>Average S6 Execution Time</td>
<td>46.36</td>
<td>34.00</td>
<td>31.79</td>
<td>28.21</td>
<td>23260.46</td>
<td>57413.95</td>
</tr>
<tr>
<td>Average S7 Execution Time</td>
<td>68.28</td>
<td>58.64</td>
<td>51.07</td>
<td>49.07</td>
<td>23279.05</td>
<td>57434.54</td>
</tr>
<tr>
<td>Average Update2 Execution Time</td>
<td>122.00</td>
<td>149.00</td>
<td>82.00</td>
<td>55.00</td>
<td>29279.00</td>
<td>84379.00</td>
</tr>
<tr>
<td>Average Update2 Execution Time</td>
<td>29.17</td>
<td>24.67</td>
<td>20.79</td>
<td>21.01</td>
<td>23495.05</td>
<td>57514.88</td>
</tr>
<tr>
<td>Average Update3 Execution Time</td>
<td>36.04</td>
<td>24.72</td>
<td>20.62</td>
<td>21.05</td>
<td>23188.52</td>
<td>56102.77</td>
</tr>
<tr>
<td>Average Update4 Execution Time</td>
<td>36.15</td>
<td>21.38</td>
<td>34.08</td>
<td>29.85</td>
<td>24836.38</td>
<td>64149.62</td>
</tr>
<tr>
<td>Average Update5 Execution Time</td>
<td>36.49</td>
<td>23.79</td>
<td>18.39</td>
<td>20.31</td>
<td>23473.63</td>
<td>57513.23</td>
</tr>
<tr>
<td>Average Update6 Execution Time</td>
<td>30.80</td>
<td>28.02</td>
<td>24.64</td>
<td>23.92</td>
<td>24737.08</td>
<td>63496.75</td>
</tr>
<tr>
<td>Average Update7 Execution Time</td>
<td>49.50</td>
<td>34.79</td>
<td>32.18</td>
<td>31.02</td>
<td>23726.20</td>
<td>58689.27</td>
</tr>
<tr>
<td>Average Update8 Execution Time</td>
<td>49.38</td>
<td>50.88</td>
<td>47.18</td>
<td>53.38</td>
<td>23692.05</td>
<td>58586.46</td>
</tr>
<tr>
<td>Loading Time</td>
<td>941103</td>
<td>987098</td>
<td>983863</td>
<td>949311</td>
<td>992662</td>
<td>975562</td>
</tr>
<tr>
<td>Query Failures</td>
<td>29</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Throughput</td>
<td>3.30</td>
<td>10.99</td>
<td>32.99</td>
<td>109.05</td>
<td>200.50</td>
<td>148.32</td>
</tr>
</tbody>
</table>

* = 01100111011010111111111111111
Table 7: Scale Factor 30.

<table>
<thead>
<tr>
<th>TCR</th>
<th>0.1</th>
<th>0.3</th>
<th>1</th>
<th>3</th>
<th>10</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale Factor</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Seed</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Enable/Disable Query Type</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Enable Sequential Tasks</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Number of Operations</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
</tr>
<tr>
<td>Average Query Execution Time</td>
<td>85.42</td>
<td>74.33</td>
<td>78.54</td>
<td>1071.05</td>
<td>51371.66</td>
<td>69811.91</td>
</tr>
<tr>
<td>Average Query02 Execution Time</td>
<td>785.66</td>
<td>840.20</td>
<td>718.20</td>
<td>1644.73</td>
<td>51866.84</td>
<td>70330.30</td>
</tr>
<tr>
<td>Average Query03 Execution Time</td>
<td>568.72</td>
<td>497.28</td>
<td>468.90</td>
<td>1422.16</td>
<td>51456.90</td>
<td>69838.94</td>
</tr>
<tr>
<td>Average Query06 Execution Time</td>
<td>520.45</td>
<td>499.00</td>
<td>544.55</td>
<td>1444.09</td>
<td>52099.82</td>
<td>70959.36</td>
</tr>
<tr>
<td>Average Query07 Execution Time</td>
<td>1222.68</td>
<td>803.68</td>
<td>3572.74</td>
<td>6122.39</td>
<td>52064.20</td>
<td>72338.55</td>
</tr>
<tr>
<td>Average Query08 Execution Time</td>
<td>61.58</td>
<td>44.95</td>
<td>38.86</td>
<td>974.69</td>
<td>51256.17</td>
<td>69692.00</td>
</tr>
<tr>
<td>Average Query10 Execution Time</td>
<td>3121.66</td>
<td>3121.28</td>
<td>3285.66</td>
<td>4259.03</td>
<td>55008.43</td>
<td>73202.39</td>
</tr>
<tr>
<td>Average Query11 Execution Time</td>
<td>106.85</td>
<td>96.30</td>
<td>87.20</td>
<td>1013.01</td>
<td>51258.74</td>
<td>69668.95</td>
</tr>
<tr>
<td>Average Query13 Execution Time</td>
<td>602.64</td>
<td>670.90</td>
<td>595.22</td>
<td>1541.65</td>
<td>51821.60</td>
<td>70312.09</td>
</tr>
<tr>
<td>Average S1 Execution Time</td>
<td>56.71</td>
<td>51.97</td>
<td>38.24</td>
<td>972.80</td>
<td>51239.66</td>
<td>69653.65</td>
</tr>
<tr>
<td>Average S2 Execution Time</td>
<td>107.07</td>
<td>82.85</td>
<td>73.20</td>
<td>1008.97</td>
<td>51276.60</td>
<td>69701.00</td>
</tr>
<tr>
<td>Average S3 Execution Time</td>
<td>38.01</td>
<td>29.98</td>
<td>22.40</td>
<td>972.80</td>
<td>51242.94</td>
<td>69660.60</td>
</tr>
<tr>
<td>Average S4 Execution Time</td>
<td>35.68</td>
<td>27.41</td>
<td>20.72</td>
<td>973.24</td>
<td>51246.65</td>
<td>69664.82</td>
</tr>
<tr>
<td>Average S5 Execution Time</td>
<td>29.90</td>
<td>22.18</td>
<td>16.35</td>
<td>973.52</td>
<td>51248.24</td>
<td>69666.65</td>
</tr>
<tr>
<td>Average S6 Execution Time</td>
<td>43.53</td>
<td>34.88</td>
<td>29.72</td>
<td>985.35</td>
<td>51258.67</td>
<td>69677.59</td>
</tr>
<tr>
<td>Average S7 Execution Time</td>
<td>66.86</td>
<td>62.92</td>
<td>50.06</td>
<td>1004.98</td>
<td>51280.17</td>
<td>69699.69</td>
</tr>
<tr>
<td>Average Update2 Execution Time</td>
<td>30.60</td>
<td>26.41</td>
<td>23.08</td>
<td>1295.73</td>
<td>51575.84</td>
<td>70025.64</td>
</tr>
<tr>
<td>Average Update3 Execution Time</td>
<td>52.02</td>
<td>25.93</td>
<td>20.52</td>
<td>1154.88</td>
<td>51464.15</td>
<td>69816.06</td>
</tr>
<tr>
<td>Average Update4 Execution Time</td>
<td>35.67</td>
<td>29.61</td>
<td>29.94</td>
<td>1213.39</td>
<td>48567.50</td>
<td>63767.28</td>
</tr>
<tr>
<td>Average Update5 Execution Time</td>
<td>27.93</td>
<td>23.28</td>
<td>18.57</td>
<td>1173.89</td>
<td>51403.92</td>
<td>69658.45</td>
</tr>
<tr>
<td>Average Update6 Execution Time</td>
<td>29.24</td>
<td>24.86</td>
<td>25.73</td>
<td>864.57</td>
<td>52352.22</td>
<td>71315.66</td>
</tr>
<tr>
<td>Average Update7 Execution Time</td>
<td>66.61</td>
<td>28.90</td>
<td>36.86</td>
<td>1069.77</td>
<td>52350.20</td>
<td>71551.28</td>
</tr>
<tr>
<td>Average Update8 Execution Time</td>
<td>52.44</td>
<td>41.79</td>
<td>49.90</td>
<td>1375.27</td>
<td>52377.71</td>
<td>71582.88</td>
</tr>
<tr>
<td>Loading Time</td>
<td>3390361</td>
<td>3450266</td>
<td>3384711</td>
<td>3513919</td>
<td>3509870</td>
<td>3432508</td>
</tr>
<tr>
<td>Query Failures</td>
<td>30</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Throughput</td>
<td>3.51</td>
<td>10.52</td>
<td>34.08</td>
<td>95.82</td>
<td>133.66</td>
<td>113.75</td>
</tr>
</tbody>
</table>

* = 01100111011011111111111111
B Virtuoso Configuration

; virtuoso.ini
;
; Configuration file for the OpenLink Virtuoso VDBMS Server
;
; To learn more about this product, or any other product in our portfolio, please check out our web site at:
;
; http://virtuoso.openlinksw.com/
;
; or contact us at:
;
; general.information@openlinksw.com
;
; If you have any technical questions, please contact our support staff at:
;
; technical.support@openlinksw.com
;
;
; Database setup
;
[Database]
DatabaseFile = /myvol/db/virtuoso.db
ErrorLogFile = /myvol/db/virtuoso.log
LockFile = /myvol/db/virtuoso.lck
TransactionFile = /myvol/db/virtuoso.trx
xa_persistent_file = /myvol/db/virtuoso.pxa
ErrorLogLevel = 7
FileExtend = 200
MaxCheckpointRemap = 2000
Striping = 0
TempStorage = TempDatabase

[TempDatabase]
DatabaseFile = /myvol/db/virtuoso-temp.db
TransactionFile = /myvol/db/virtuoso-temp.trx
MaxCheckpointRemap = 2000
Striping = 0

; Server parameters
;
[Parameters]
ServerPort = 1111
LiteMode = 0
DisableUnixSocket = 1
DisableTcpSocket = 0
;SSLServerPort = 2111
;SSLCertificate = cert.pem
;SSLPrivateKey = pk.pem
;X509ClientVerify = 0
When running with large data sets, one should configure the Virtuoso process to use between 2/3 to 3/5 of free system memory and to stripe storage on all available disks.

Uncomment next two lines if there is 2 GB system memory free

```
NumberOfBuffers = 170000
MaxDirtyBuffers = 130000
```

Uncomment next two lines if there is 4 GB system memory free

```
NumberOfBuffers = 340000
MaxDirtyBuffers = 250000
```

Uncomment next two lines if there is 8 GB system memory free

```
NumberOfBuffers = 680000
MaxDirtyBuffers = 500000
```

Uncomment next two lines if there is 16 GB system memory free

```
NumberOfBuffers = 1360000
MaxDirtyBuffers = 1000000
```

Uncomment next two lines if there is 32 GB system memory free

```
NumberOfBuffers = 2720000
MaxDirtyBuffers = 2000000
```

Uncomment next two lines if there is 48 GB system memory free

```
NumberOfBuffers = 4000000
MaxDirtyBuffers = 3000000
```

Uncomment next two lines if there is 64 GB system memory free

```
NumberOfBuffers = 5450000
```
;MaxDirtyBuffers = 4000000
;
;; Note the default settings will take very little memory
;; but will not result in very good performance
;;
NumberOfBuffers = 14900000
MaxDirtyBuffers = 11000000

[HTTPServer]
ServerPort = 8890
ServerRoot = /usr/local/virtuoso-opensource/var/lib/virtuoso/vsp
MaxClientConnections = 10
DavRoot = DAV
EnabledDavVSP = 0
HTTPProxyEnabled = 0
TempASPXDdir = 0
DefaultMailServer = localhost:25
ServerThreads = 10
MaxKeepAlives = 10
KeepAliveTimeout = 10
MaxCachedProxyConnections = 10
ProxyConnectionCacheTimeout = 15
HTTPThreadSize = 280000
HttpPrintWarningsInOutput = 0
Charset = UTF-8

HTTPLogFile = http.log
MaintenancePage = atomic.html
EnabledGzipContent = 1

[AutoRepair]
BadParentLinks = 0

[Client]
SQL_PREFETCH_ROWS = 100
SQL_PREFETCH_BYTES = 16000
SQL_QUERY_TIMEOUT = 0
SQL_TXN_TIMEOUT = 0
;
SQL_NO_CHAR_C_ESCAPE = 1
;
SQL_UTF8_EXECS = 0
;
SQL_NO_SYSTEM_TABLES = 0
;
SQL_BINARY_TIMESTAMP = 1
;
SQL_ENCRYPTION_ON_PASSWORD = -1

[VDB]
ArrayOptimization = 0
NumArrayParameters = 10
VDBDisconnectTimeout = 1000
KeepConnectionOnFixedThread = 0

[Replication]
ServerName = db-D602566B774E
ServerEnable = 1
QueueMax = 500000
;
; Striping setup
;
; These parameters have only effect when Striping is set to 1 in the 
; [Database] section, in which case the DatabaseFile parameter is ignored.
;
; With striping, the database is spawned across multiple segments 
; where each segment can have multiple stripes.
;
; Format of the lines below:
;   Segment<number> = <size>, <stripe file name> [, <stripe file name> .. ]
;
; <number> must be ordered from 1 up.
;
; The <size> is the total size of the segment which is equally divided 
; across all stripes forming the segment. Its specification can be in 
; gigabytes (g), megabytes (m), kilobytes (k) or in database blocks 
; (b, the default)
;
; Note that the segment size must be a multiple of the database page size 
; which is currently 8k. Also, the segment size must be divisible by the 
; number of stripe files forming the segment.
;
; The example below creates a 200 meg database striped on two segments 
; with two stripes of 50 meg and one of 100 meg.
;
; You can always add more segments to the configuration, but once 
; added, do not change the setup.
;
[Striping]
Segment1 = 100M, db-seg1-1.db, db-seg1-2.db
Segment2 = 100M, db-seg2-1.db
;

;[TempStriping]
;Segment1 = 100M, db-seg1-1.db, db-seg1-2.db
;Segment2 = 100M, db-seg2-1.db
;

;[Ucms]
;UcmPath = <path>
;Ucm1 = <file>
;Ucm2 = <file>
;

[Zero Config]
ServerName = virtuoso (D602566B774E)
;ServerDSN = ZDSN
;SSLServerName = 
;SSLServerDSN = 

[Mono]
;MONO_TRACE = Off
;MONO_PATH = <path_here>
;MONO_ROOT = <path_here>
;MONO_CFG_DIR = <path_here>
;virtclr.dll =

[URIQA]
DynamicLocal = 0
DefaultHost = localhost:8890

[SPARQL]
;ExternalQuerySource = 1
;ExternalXsltSource = 1
;DefaultGraph = http://localhost:8890/dataspace
;ImmutableGraphs = http://localhost:8890/dataspace
ResultSetMaxRows = 100000000
MaxQueryCostEstimationTime = 400; in seconds
MaxQueryExecutionTime = 60; in seconds
DefaultQuery = select distinct ?Concept where {{} a ?Concept} LIMIT 100
DeferInferenceRulesInit = 0; controls inference rules loading
;PingService = http://rpc.pingthesemanticweb.com/

[Plugins]
LoadPath = /usr/local/virtuoso-opensource/lib/virtuoso/hosting
Load1 = plain, wikiv
Load2 = plain, mediawiki
Load3 = plain, creolewiki
;Load4 = plain, im
;Load5 = plain, wbxml2
;Load6 = plain, hslookup
;Load7 = attach, libphp5.so
;Load8 = Hosting, hosting_php.so
;Load9 = Hosting, hosting_perl.so
;Load10 = Hosting, hosting_python.so
;Load11 = Hosting, hosting_ruby.so
;Load12 = msdtc, msdtc_sample