Collaborative Project

Holistic Benchmarking of Big Linked Data

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

Deliverable 5.1.1
First 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 18, 31/05/2017</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 18, 31/05/2017</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP5 - Benchmarks III: Storage and Curation</td>
</tr>
<tr>
<td>Task</td>
<td>T5.1</td>
</tr>
<tr>
<td>Type</td>
<td>Report</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>47</td>
</tr>
</tbody>
</table>

Abstract: This deliverable describes the development of the first version of the Data Storage benchmark for HOBBIT. By introducing important modifications in the synthetic data generator and the dataset of the Social Network Benchmark, developed in the context of the FP7 Linked Data Benchmarking Council (LDBC), and transforming its SQL queries to SPARQL, all while preserving its most important features, we were able to introduce a 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.

Project funded by the European Commission’s Horizon 2020 Program.
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>28/04/2017</td>
<td>First draft</td>
<td>Milos Jovanovik</td>
</tr>
<tr>
<td>0.2</td>
<td>02/05/2017</td>
<td>SPARQL queries in Appendix</td>
<td>Milos Jovanovik</td>
</tr>
<tr>
<td>0.3</td>
<td>12/05/2017</td>
<td>Integration into the HOBBIT Platform</td>
<td>Mirko Spasić</td>
</tr>
<tr>
<td>0.4</td>
<td>12/05/2017</td>
<td>Abstract and Executive Summary</td>
<td>Milos Jovanovik</td>
</tr>
<tr>
<td>0.5</td>
<td>19/05/2017</td>
<td>Internal Review</td>
<td>Irini Fundulaki</td>
</tr>
<tr>
<td>0.6</td>
<td>22/05/2017</td>
<td>Modifications after the Internal Review</td>
<td>Milos Jovanovik, Mirko Spasić</td>
</tr>
<tr>
<td>0.7</td>
<td>28/05/2017</td>
<td>MOCHA 2017 Results</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>Milos Jovanovik</td>
<td><a href="mailto:mjovanovik@openlinksw.com">mjovanovik@openlinksw.com</a></td>
</tr>
<tr>
<td>OpenLink</td>
<td>Mirko Spasić</td>
<td><a href="mailto:mspasic@openlinksw.com">mspasic@openlinksw.com</a></td>
</tr>
</tbody>
</table>
Executive Summary

The demand for efficient RDF storage technologies has grown steadily in recent years, due to the increasing number of Linked Data datasets and applications exploiting Linked Data resources. More specifically, a growing number of applications require RDF storage solutions capable of answering SPARQL queries in interactive times. Given the rising number of RDF storage solutions, there is an increasing need for objective means to compare technologies from different vendors. Consequently, there is a growing 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. A number of benchmarks and benchmark generation frameworks for querying Linked Data have been developed over the past decade. After carefully considering all options, by assessing their compliance to general data storage benchmark requirements – 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 loading support, interactive complex read queries, as well as simple lookups, concurrency and high throughput – we chose the Social Network Benchmark, developed in the context of the FP7 Linked Data Benchmarking Council (LDBC), as a starting point. By introducing important modifications in 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.

This deliverable outlines the details of how the first version of the Data Storage benchmark for HOBBIT was developed and tested. It focuses on the synthetic data generator for the benchmark, the dataset, the SPARQL queries and the key performance indicators which constitute the benchmark. The deliverables also describes the integration of the benchmark into the HOBBIT platform, and showcases the benchmark results of the baseline implementation.

1http://www.ldbcouncil.org/
Contents

1 Introduction ................................................................. 6
   1.1 Requirements for the Data Storage Benchmark ................. 7

2 Background ................................................................. 7
   2.1 Related Work ....................................................... 7
   2.2 The LDBC SNB Dataset .......................................... 8
   2.3 Dataset Coherence ................................................ 9
      2.3.1 Measuring RDF Dataset Coherence in Virtuoso ............ 9
      2.3.2 Coherence of the LDBC SNB Dataset ...................... 11

3 Data Storage Benchmark Dataset ..................................... 11
   3.1 Coherence of the Data Storage Benchmark Dataset .......... 13

4 Data Storage Benchmark ................................................ 13
   4.1 Choke Points ..................................................... 14
   4.2 SPARQL Queries (Complex) ..................................... 14
      4.2.1 Query 1 - Friends with a Certain Name .................. 14
      4.2.2 Query 2 - Recent Posts and Comments by Your Friends . 15
      4.2.3 Query 3 - Friends and Friends of Friends That Have Been to Countries X and Y 15
      4.2.4 Query 4 - New Topics ...................................... 15
      4.2.5 Query 5 - New Groups ..................................... 16
      4.2.6 Query 6 - Tag Co-occurrence ............................. 16
      4.2.7 Query 7 - Recent Likes .................................... 16
      4.2.8 Query 8 - Recent Replies .................................. 17
      4.2.9 Query 9 - Recent Posts and Comments by Friends or Friends of Friends .... 17
      4.2.10 Query 10 - Friend Recommendation ........................ 17
      4.2.11 Query 11 - Job Referral ................................... 18
      4.2.12 Query 12 - Expert Search ................................. 18
      4.2.13 Query 13 - Recent Posts and Comments Where Your Friends Are Mentioned .. 18
      4.2.14 Query 14 - New Shared Links .............................. 19
   4.3 SPARQL Queries (Short) .......................................... 19
      4.3.1 Short 1 - Person Profile .................................... 19
      4.3.2 Short 2 - Person Recent Messages .......................... 19
4.3.3 Short 3 - Person Friends ............................................................... 19
4.3.4 Short 4 - Message Content ...................................................... 20
4.3.5 Short 5 - Message Creator ...................................................... 20
4.3.6 Short 6 - Message Forum .......................................................... 20
4.3.7 Short 7 - Message Replies ........................................................ 20
4.4 SPARQL Queries (Update) ............................................................ 20
  4.4.1 Update 1 - Add Person ......................................................... 20
  4.4.2 Update 2 - Add Post Like ..................................................... 20
  4.4.3 Update 3 - Add Comment Like ............................................... 20
  4.4.4 Update 4 - Add Forum .......................................................... 21
  4.4.5 Update 5 - Add Forum Membership ......................................... 21
  4.4.6 Update 6 - Add Post ............................................................. 21
  4.4.7 Update 7 - Add Comment ..................................................... 21
  4.4.8 Update 8 - Add Friendship ................................................... 21
4.5 Key Performance Indicators .......................................................... 21

5 Evaluation ....................................................................................... 22
  5.1 Comparison of the Data Storage Benchmark and Social Network Benchmark ......................................................... 22
  5.2 Validating the Data Storage Benchmark on Different Triple Stores ................................................................. 26

6 Integration into the HOBBIT Platform ............................................. 28
  6.1 DSB Parameters ........................................................................... 28
  6.2 Components .................................................................................. 29
    6.2.1 Benchmark Controller ............................................................ 29
    6.2.2 Data Generator ....................................................................... 29
    6.2.3 Task Generator ...................................................................... 30
    6.2.4 System Adapter ................................................................. 30
    6.2.5 Evaluation Module ............................................................... 31
    6.2.6 Key Performance Indicators (KPIs) ....................................... 31
    6.2.7 Experiments ........................................................................ 32
  6.3 Preliminary Results ....................................................................... 32

References ......................................................................................... 33

A SPARQL queries (Complex) ............................................................. 35
  A.1 Query 1 - Friends with a Certain Name ....................................... 35
A.2 Query 2 - Recent Posts and Comments by Your Friends ......................... 36
A.3 Query 3 - Friends and Friends of Friends That Have Been to Countries X and Y .... 37
A.4 Query 4 - New Topics ..................................................... 37
A.5 Query 5 - New Groups ..................................................... 38
A.6 Query 6 - Tag Co-occurrence .......................................... 38
A.7 Query 7 - Recent Likes .................................................. 39
A.8 Query 8 - Recent Replies ............................................. 39
A.9 Query 9 - Recent Posts and Comments by Friends or Friends of Friends .......... 39
A.10 Query 10 - Friend Recommendation .................................. 40
A.11 Query 11 - Job Referral ............................................... 41
A.12 Query 12 - Expert Search ............................................ 41
A.13 Query 13 - Recent Posts and Comments Where Your Friends Are Mentioned .... 42
A.14 Query 14 - New Shared Links ....................................... 42

B SPARQL queries (Short) .................................................. 42
  B.1 Short 1 - Person Profile ................................................. 42
  B.2 Short 2 - Person Recent Messages ................................... 43
  B.3 Short 3 - Person Friends .............................................. 43
  B.4 Short 4 - Message Content .......................................... 43
  B.5 Short 5 - Message Creator ........................................... 44
  B.6 Short 6 - Message Forum ............................................. 44
  B.7 Short 7 - Message Replies ........................................... 44

C SPARQL queries (Update) .............................................. 45
  C.1 Update 1 - Add Person ................................................ 45
  C.2 Update 2 - Add Post Like .......................................... 45
  C.3 Update 3 - Add Comment Like ..................................... 45
  C.4 Update 4 - Add Forum ............................................... 46
  C.5 Update 5 - Add Forum Membership ................................. 46
  C.6 Update 6 - Add Post .................................................. 46
  C.7 Update 7 - Add Comment ........................................... 46
  C.8 Update 8 - Add Friendship ........................................ 47
1 Introduction

The demand for efficient RDF storage technologies has grown steadily in recent years. This increase is due to the increasing number of Linked Data datasets and applications exploiting Linked Data resources. In particular, a growing number of applications require RDF storage solutions capable of answering SPARQL interactive queries. Given the increasing number of solutions for RDF storage, there is an increasing need for objective means to compare technologies from different vendors. Consequently, there is a growing 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. A number of benchmarks and benchmark generation frameworks for querying Linked Data have been developed over the past decade [2, 12, 19, 21]. Therefore, instead of developing a new benchmark for data storage from scratch, we analyzed the existing ones, taking into account their relevance, their popularity and design, their pros and cons regarding scalability, realness, the key performance indicators they measure, etc. Requirements that potential data storage benchmarks had to meet were: 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 loading support, interactive complex read queries, as well as simple lookups, concurrency and high throughput [3, 13].

Taking into account a number of existing benchmarks, detailed below in Section 2.1, we decided to use the Social Network Benchmark (SNB), developed under the auspices of the Linked Data Benchmark Council (LDBC)\(^2\), as a starting point in constructing the Data Storage benchmark for HOBBIT. LDBC introduced a new choke-point driven methodology for developing benchmark workloads, which combines user input with input from expert systems architects [10]. Unlike other benchmarks which are specific, tied to a single technology, 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.

The Social Network Benchmark was developed as an SQL-based benchmark intended for RDBMS systems and consists of three separate sub-benchmarks (workloads) on a single common dataset, generated by a synthetic data generator (DATAGEN) which models an online social network (OSN), like Facebook. In the first workload, SNB-Interactive, there are 3 kinds of queries: (a) complex read-only queries (touching a significant amount of data, and representing 50% of the benchmark), (b) simple read-only queries (lookups of a single entity, 40% of total), and (c) update queries. The other two workloads, SNB-BI and SNB-Algorithms, are currently under construction, and are intended to contain queries for business analytics and often-used graph analysis algorithms, respectively.

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. This deliverable provides a detailed description of the RDF dataset, the dataset generator, the benchmark queries and the key performance indicators which constitute the Data Storage benchmark for HOBBIT. The deliverables also describes the integration of the benchmark into the HOBBIT platform, and showcases the benchmark results of the baseline implementation.

\(^2\)http://www.ldbcouncil.org/
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
- High throughput

These requirements enable an efficient evaluation of data storage solutions which support interactive workloads in real-world scenarios, over various dataset sizes [3, 13].

2 Background

2.1 Related Work

In order to pick the most appropriate existing benchmark as a starting point for the HOBBIT data storage benchmark, we looked at several options. Here, a brief overview of our findings is presented.

The Lehigh University Benchmark (LUBM) [12], the Berlin SPARQL Benchmark (BSBM) [4] and the SPARQL Performance Benchmark (SP2Bench) [22] use synthetic data and synthetic queries for different scenarios. LUBM provides data over the organizational structure of Universities. SP2Bench uses the DBLP bibliographic database and BSBM was developed for the e-commerce use-case and supports synthetic updates. However, the queries in all three benchmarks either lack complexity, or do not simulate interactive workloads and thus cannot, in our opinion, produce relevant results on the performance of data storage solutions [19].

Even though the DBpedia SPARQL Benchmark (DBPSB) uses real data and real queries, the results in [7], especially in comparison to the results in [19], show that the queries are not complex enough to show new insights.

FEASIBLE challenges this issue by providing real and complex queries, but requires a query log to generate them [21]. Hence, it cannot provide queries for synthetic data, as synthetic datasets do not come with real query logs.

The Waterloo SPARQL Diversity Test Suite (WatDiv) [2] provides a QueryGenerator with 125 query templates. However, the generator is restricted to conjunctive SELECT queries only, which do not cover our requirements.

The Social Network Benchmark (SNB) [10] from LDBC represents a synthetic, but realistic dataset with complex queries. Therefore, it can show new insights of a triple store performance using synthetic data. Another reason which makes SNB queries difficult for RDF storage systems is the real-world
distribution in the dataset, which results in much more challenging tasks for a query optimizer, larger number of potential optimal query plans, and introduces problems in cardinality estimations. For example, in SNB, it is hard to estimate the number of Posts by friends of a Person due to two reasons: the number of friends of a Person can significantly vary, along with the number of Posts per Person. This is a desired scenario in benchmarks which represent real-world use-cases. Some of the well-known and widely accepted benchmarks, such as TPC-H, do not have this feature – TPC-H uses a uniform distribution of entity relations [9].

2.2 The LDBC SNB Dataset

SNB provides a synthetic data generator (DATAGEN)\(^3\), which models an online social network (OSN), like Facebook. The data contains different types of entities and relations, such as persons with friendship relations among them, posts, comments and likes. Additionally, it reproduces many of the structural characteristics observed in real OSNs, summarized below.

**Attribute correlations.** Real data is correlated by nature. For instance, given names to persons are correlated with their country. Similarly, the textual content of a message is correlated with the interests of its creator. For the purpose of benchmarking and performance evaluation, reproducing these correlations is essential since their understanding can be used to generate optimal execution plans or to properly lay out the data in memory. DATAGEN uses dictionaries extracted from DBpedia to create entities with correlated attribute values.

**Degree distributions.** The underlying graph structure of OSNs exhibits skewed degree distributions: most of people have between 20 to 200 friends, while a few of them have over 5000 friends [23]. In a real database system, this skewness complicates the estimation of cardinalities required to produce optimal execution plans, or the load balancing when executing parallel operators over adjacency lists. Also, nodes with a large degree, commonly known as hubs, must be carefully handled, especially in distributed systems, where traversing these nodes can incur in a large inter-node communication. DATAGEN takes the degree distribution of Facebook and empirically reproduces it.

**Structure-Attribute correlations.** The homophily principle states that in a real social network, similar people have a larger probability to be connected, which leads to the formation of many transitive relations between connected people with similar characteristics. As a consequence, even though it is commonly accepted that graph data access patterns are usually random, in practice there is some degree of locality that can be exploited. For instance, people from a given country are more likely to be connected among them than to people from other countries. Thus, this information can be used to lay out data in memory wisely to improve the performance of graph traversal, for instance, by putting all people in a country closer in memory. DATAGEN generates person relationships based on different correlation dimensions (i.e. the interests of a person, the place that person studied, etc.). These correlation dimensions are used to sort persons in such a way that those that are more similar, are placed close in the storage solution. Then, edges are created between close persons, with a probability that decreases geometrically with their distance. As shown in [20], this approach successfully produces attribute correlated networks with other desirable characteristics such as large clustering coefficient, a small diameter or a large largest connected component.

**Spiky activity volume.** In a real social network, the amount of activity is not uniform but reactive to real-world events. If a natural disaster occurs, we will observe people talking about it mostly after the time of the disaster, and the associated activity volume will decay as the hours pass. This translates to a spiky volume activity along the like of the social network, mixing moments with a

\(^3\)https://github.com/ldbc/ldbc_snb_datagen
high load with situations where the load level is small. Also, this means that the amount of messages produced by people and their topics are correlated with given points in time, which complicates the work of a query optimizer when estimating cardinalities for their query plans. This can also make a system unable to cope with the load if it has not been properly overprovisioned to handle these spiky situations. Instead of generating posts and comments uniformly distributed along time, DATAGEN creates virtual events of different degrees of relevance, which are used to drive the generation of the user activity, producing a spiky distribution.

2.3 Dataset Coherence

The comparison of data generated with existing RDF benchmarks (TPC-H, BSBM, LUBM, SP2Bench, etc.) and data found in widely used real RDF datasets (DBpedia, UniProt, etc.) shows that these two have significant structural differences [8]. Real-world RDF datasets are less coherent, i.e. have a lower degree of structuredness than synthetic RDF datasets. This structural difference is important as it has direct consequences on the storage of the data, as well as on the ways the data are indexed and queried. In the same paper, the authors introduce a composite metric, called coherence of a dataset, in order to quantify the structuredness of the dataset \( D \) with respect to the type system \( T \) as follows:

\[
CH(T, D) = \sum_{T \in T} WT(CV(T, D)) \cdot CV(T, D)
\]

This is the weighted sum of the coverage \( CV(T, D) \) of individual types \( T \in T \), where the weight coefficient \( WT(CV(T, D)) \) depends on the number of properties for a type \( T \), the number of entities in dataset \( D \) of type \( T \), and their share in the totality of the dataset \( D \) among the other types. Its rationale is to give higher impact to types with more instances and properties. \( CV(T, D) \) represents the coverage of type \( T \) on the dataset \( D \). It depends on whether the instances of the type \( T \) set a value for all its properties. If that is the case for all the instances, the coverage will be 1 (perfect structuredness), otherwise it will take a value from \([0, 1)\). The conclusion of [8] is that there is a clear distinction in the structuredness, i.e. the coherence between the datasets derived from the existing RDF benchmarks and the real-world RDF datasets. For the former, the values range between 0.79 (for SP2Bench) and 1 (for TPC-H) showing us the characteristics of relational databases, while the coherence values for almost all real-world datasets are below or around 0.6.

2.3.1 Measuring RDF Dataset Coherence in Virtuoso

In order to make a data storage benchmark for RDF and Linked Data which is more realistic, the benchmark dataset should follow the nature of real data. Since the SNB dataset is developed to test not only RDF stores, but also graph database systems, graph programming frameworks, relational and NoSQL database systems, we wanted to measure how this dataset is suitable for RDF benchmarks.

The authors of [8] propose a workflow to compute the coherence of a dataset in a few steps: assembling all triples into a single file, data cleaning and normalization, generating several new files and sorting them in different orders to provide the ability that the corresponding metrics can be collected by making a single pass of the sorted file. The disadvantages of this approach are the memory requirements for storing all files in non-compressed format, and the time required to sort them. Also, the sorting process could use additional temporary space which is not negligible.

Therefore, we developed a new approach to compute coherence of any dataset, using Virtuoso [11], leaving the system to take care of the efficacy and data compression. Virtuoso is a column store with
good compression capabilities, thus we will have a simpler and much more space- and time-efficient procedure for calculating the proposed metric. First, we load the dataset in question in a graph within Virtuoso, with a single command (`ld_dir`). Next, we define a stored procedure in Virtuoso/PL for calculating the coherence, by selecting all types from a dataset, calculating their individual coverage and the weighted sum coefficient. The procedure, along with its supporting procedures, is available on GitHub\(^4\), and is presented below:

```sql
create procedure coherence (in graph VARCHAR) {
    declare a real;
    select sum(coverage * (pred_cnt + inst_cnt)) into a
    from (
        select id_to_iri(O) as type,
            coverage(graph, id_to_iri(O)) as coverage,
            predicates_count(graph, id_to_iri(O)) as pred_cnt,
            instances_count(graph, id_to_iri(O)) as inst_cnt
        from (
            select distinct O from RDF_QUAD
            where G = iri_to_id(graph) and P = iri_to_id('rdf:type')
        ) as tmp
    ) tmp1;
    return a / total_preds_and_insts(graph);
}
```

This procedure selects all the types from a dataset, and calculates coverage \(CV(T, D)\) for each one, along with the number of predicates and instances. The last two numbers are needed for the weighted sum, and are calculated by the `predicates_count()` and `instances_count()` functions, which are simple and not listed here – they contain only a single count query. The `total_preds_and_insts()` function is of a similar nature. However, the `coverage()` function is a bit more complex:

```sql
create procedure coverage (in graph VARCHAR, in t LONG VARCHAR) {
    declare a, b, c bigint;
    select sum(cnt) into a from (
        select t2.P as pred, count(distinct t1.S) as cnt
        from RDF_QUAD t1, RDF_QUAD t2
        where t1.S = t2.S and t2.P <> iri_to_id('rdf:type')
        and t1.G = iri_to_id(graph) and t2.G = iri_to_id(graph)
        and t1.P = iri_to_id('rdf:type') and t1.O = iri_to_id(t)
        group by pred
    ) tmp;
    select count(distinct t2.P) into b from RDF_QUAD t1, RDF_QUAD t2
    where t1.S = t2.S and t2.P <> iri_to_id('rdf:type')
    and t1.G = iri_to_id(graph) and t2.G = iri_to_id(graph)
    and t1.P = iri_to_id('rdf:type') and t1.O = iri_to_id(t);
    select count(distinct S) into c from RDF_QUAD
    where G = iri_to_id(graph)
    and P = iri_to_id('rdf:type') and O = iri_to_id(t);
    return cast (a as real) / (b * c);
}
```

\(^4\)`https://github.com/hobbit-project/DataStorageBenchmark-Dev`
2.3.2 Coherence of the LDBC SNB Dataset

Using the RDF serialization of the original SNB DATAGEN, we prepared the SNB RDF dataset in several sizes, i.e. scale factors, and measured their coherence values (Section 3.1). Since their coherence varies from 0.86 to 0.89, we concluded that the SNB dataset is much more structured than the real-world RDF datasets, thus it is not fully suitable for our Data Storage benchmark – at least not if used in its original form. Our intention is to have and use an RDF dataset which mimics real-world Linked Data datasets, with a structuredness level of around 0.6.

Therefore, we had to work on modifying the SNB dataset and DATAGEN.

3 Data Storage Benchmark Dataset

The original SNB DATAGEN (Section 2.2) reproduces the important structural characteristics observed in real online social networks. However, the authors of [8] show that structuredness of RDF datasets used for benchmarking is important, as well. Therefore, to construct the dataset for the Data Storage Benchmark (DSB), we modified the SNB DATAGEN so that we lower the structuredness, i.e. coherence measure, from around 0.88 to around 0.6, to comply with the structuredness of real-world RDF datasets. Before we started working on this, we first had to work on refining the RDF serialization of SNB DATAGEN, to provide an RDF / Linked Data output, as opposed to the relational database output used as part of the LDBC project.

The authors of [8] propose a generic way of decreasing the coherence metric for any dataset, without using domain specific knowledge. The consequence of this modification is the reduction of the dataset size. We decided to take a different approach and modify the SNB DATAGEN by introducing new predicates and by removing triples from the initial dataset, all while taking into account and keeping the main characteristics of the dataset (described in Section 2.2) intact. This enrichment phase provides a dataset with a coherence value matching the real-world RDF datasets and makes the dataset a bit more complex – complying with the current state and features of real-world social networks. Table 1 depicts the most dominant entity types from the SNB dataset, along with their weights. We omit the other types, as their weights are not significant in this case.

Table 1: Entity types and their weights from the SNB dataset.

<table>
<thead>
<tr>
<th>Type</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment</td>
<td>0.6477</td>
</tr>
<tr>
<td>Post</td>
<td>0.3126</td>
</tr>
<tr>
<td>Forum</td>
<td>0.0282</td>
</tr>
<tr>
<td>Person</td>
<td>0.0031</td>
</tr>
<tr>
<td>University</td>
<td>0.0016</td>
</tr>
</tbody>
</table>

The weight of a type mostly depends on the number of its instances in the dataset. In the original SNB dataset the comments are the most numerous, followed by posts, which is visible in the results shown in Table 1 where we can see their dominance in this regard: together they hold 96% of
the dataset weight. In order to decrease the coherence metric of the dataset, $CH(\mathcal{T}, \mathcal{D})$, we should decrease the coverage $CV(\mathcal{T}, \mathcal{D})$ of each type $\mathcal{T}$ from the dataset. But, if we, for example, decrease the coverage of type $\text{Person}$ from 0.95 to 0 (which is not realistic), this will result in a drop of the coherence measure for less than 0.3%. Bearing in mind that we have to decrease it much more than that, the only reasonable choice for modifications are the $\text{Comment}$ and $\text{Post}$ types.

The common predicates of these two types are $\text{browserUsed}$, $\text{content}$, $\text{creationDate}$, $\text{hasCreator}$, $\text{hasTag}$, $\text{id}$, $\text{isLocatedIn}$, $\text{length}$ and $\text{locationIP}$, while $\text{Comment}$ instances additionally have the $\text{replyOf}$ predicate, and $\text{Post}$ instances can have $\text{language}$ and $\text{imageFile}$ properties if the $\text{Post}$ instance is a photo. One way of decreasing the coverage of specific types is the removal of a high number of triples related to a specific property. But, taking into account the specific domain, we conclude that the only property that can be removed in part of the posts and comments is $\text{isLocatedIn}$. The initial purpose of this property was to specify a country from which the message was issued, and it was determined by the IP address of the location where the message had been created. However, since a lot of users access social networks using their smartphones equipped with GPS receivers, social networks offer the possibility of adding locations to the messages. If we consider this property in that manner, we can remove the location from some messages, as not all messages contain location information. Various research in the domain show that users rarely share their location in the posts: the authors of [15] show that only about 1.2% of Twitter posts contain an explicit location information, while [17] shows that only around 6% of Instagram posts (photos) have a location tagged. Therefore, we remove the location information from 98% of comments and textual posts, and from 94% of photo posts, and with it the coverage of posts and comments gets significantly reduced.

Since it does not make sense to remove any other property, in order to achieve our goal, we decided to introduce new ones. In the initial dataset, all of the comments are textual, while recently social networks added a predefined set of GIFs which can be used as comments [16, 14]. In the initial dataset, one third of all comments are long textual comments, while two thirds are short comments, e.g. “ok”, “great”, “cool”, “thx”, “lol”, “no way!”, “I see”, “maybe”, etc. In order to include GIFs as comments, we introduce the $\text{gifFile}$ property, which we apply in 80% of the short comments as a replacement of their $\text{content}$ property.

In the next step, we add one more property to posts and comments: $\text{mentions}$. Its purpose is to mention a person in a post or a comment. This modification is also in line with what we have on social networks such as Facebook, Twitter, Instagram, etc., where a user can mention another user in a post or a comment, usually to make the other person aware of it. An analysis we performed over the Twitter7 dataset [23] showed that 40% of the tweets contain at least one mention. Therefore, we add this property to 40% of posts and comments, which provides an additional drop in the coherence measure.

A significant issue in operating a social network is privacy. Facebook introduced the possibility for each author of a post/comment to determine its level of privacy: if you want to share it publicly, to your friends only, or to a specific group of people [1]. Therefore, we introduce the $\text{visibility}$ predicate, which is set to a post/comment when it is posted with a privacy setting different from the default one for the user. Therefore, we generate this property for 5% of all messages, using the assumption that users generally use their default privacy setting.

The final change we added to the data generator is the addition of the $\text{link}$ property, which both textual posts and comment can have. This corresponds to the real-world activity of sharing a link in a post, in addition to the text. Based on the analysis of user behavior on social media [18], which found that 43% of Facebook posts contain links, we add the $\text{link}$ property to that share of textual posts and comments. As a value for the $\text{link}$ property, we use a random value from a predefined pool, similar as with other properties filled by the DATAGEN. It will always be fetched at the end of query execution,
without any filtering to introduce estimation of cardinality, so the actual value is irrelevant.

With this, we created a new DATAGEN for the Data Storage Benchmark dataset. The new DATAGEN generates synthetic RDF datasets which mimic an online social network and has real-world coherence. It is publicly available on GitHub\(^5\).

### Table 2: Coherence of the SNB datasets.

<table>
<thead>
<tr>
<th>SF</th>
<th>#Triples</th>
<th>Coherence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46.9M</td>
<td>0.8599</td>
</tr>
<tr>
<td>3</td>
<td>142.6M</td>
<td>0.8702</td>
</tr>
<tr>
<td>10</td>
<td>480.8M</td>
<td>0.8808</td>
</tr>
<tr>
<td>30</td>
<td>1478.2M</td>
<td>0.8883</td>
</tr>
<tr>
<td>100</td>
<td>4804.3M</td>
<td>0.8943</td>
</tr>
</tbody>
</table>

### Table 3: Coherence of the DSB datasets.

<table>
<thead>
<tr>
<th>SF</th>
<th>#Triples</th>
<th>Coherence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45.4M</td>
<td>0.6025</td>
</tr>
<tr>
<td>3</td>
<td>136.5M</td>
<td>0.6049</td>
</tr>
<tr>
<td>10</td>
<td>464.1M</td>
<td>0.6086</td>
</tr>
<tr>
<td>30</td>
<td>1428.4M</td>
<td>0.6115</td>
</tr>
<tr>
<td>100</td>
<td>4645.7M</td>
<td>0.6139</td>
</tr>
</tbody>
</table>

### 3.1 Coherence of the Data Storage Benchmark Dataset

To assess the structuredness of the SNB RDF datasets, generated by the original data generator, and the DSB RDF datasets, generated by our modified version of DATAGEN, we made measurements of the datasets in different sizes: 1, 3, 10, 30, and 100GB. Table 2 and Table 3 depict the results of the measurements: they show the number of triples (in millions), and the coherence metric for all versions of the datasets.

The tables show that the modifications we introduced to design the DSB DATAGEN, yield an RDF dataset with real-world coherence values – a goal we set to achieve for the Data Storage benchmark.

### 4 Data Storage Benchmark

Since we base our Data Storage Benchmark (DSB) on LDBC Social Network Benchmark (SNB), after modifying the synthetic RDF data generator, we took upon defining and modifying the benchmark queries. The natural language formulations and the choke points of the benchmark queries were specified in the scope of LDBC, and only SQL versions of the queries were developed and used [10, 5]. However, due to the RDF / Linked Data nature of the Data Storage Benchmark in the context of HOBBIT and the changes we introduced to the dataset, development of SPARQL queries and modifications to the formulations of the queries in general were necessary. Since two of the original complex SNB queries (Q13 and Q14) cannot be expressed using SPARQL 1.1 syntax, we had to redefine them.

A brief overview of all queries from the three types (complex queries, short queries and update queries) is presented in this section. The relevant choke points for each query are also specified. The details of the choke points which are shortened for brevity, can be implied from the more detailed ones.

\(^5\) [https://github.com/hobbit-project/DataStorageBenchmark-Dev](https://github.com/hobbit-project/DataStorageBenchmark-Dev)
Also, each subsection details the differences between the new version of the SPARQL query for DSB, related to the the original LDBC SNB query.

The SPARQL 1.1 implementation of the queries is provided in Appendix A, B and C.

4.1 Choke Points

The development of SNB was driven by so-called ‘choke-points’, i.e. on a choke-point analysis that was used to identify important technical challenges to be evaluated in a query workload [10]. A choke point is an aspect of query execution or optimization which is known to be problematical for the current generation of various relational, graph and RDF DBMS. In general, the choke points cover usual challenges of query processing – subquery unnesting, complex aggregate performance, detecting dependent group-by keys, etc. – as well as some hard problems that are usually not part of synthetic benchmarks.

Some examples of choke-points include: estimating cardinality in graph traversals with data skew and correlations, choosing the right join order and type, handling scattered index access patterns, parallelism and result reuse, etc. For more details, refer to [10] and [5].

4.2 SPARQL Queries (Complex)

4.2.1 Query 1 - Friends with a Certain Name

**Description:** Given a start Person, find a given number of closest Person entities with a given first name that the start Person is connected to (excluding start Person) by at most 3 steps via Knows relationships. Return Persons, including summaries of the Persons workplaces and places of study.

**Choke points:** This query looks for paths of length one, two or three through the Knows relation, starting from a given Person and ending at a Person with a given name. It requires for a complex aggregation, that is, returning the concatenation of universities, companies, languages and email information of the person. Also, it tests the ability of the optimizer to delay the evaluation of sub-queries functionally dependent on the Person, after the evaluation of the top-k. Performance is highly sensitive to properly estimating the cardinalities in each transitive path, and paying attention not to explore already visited Persons. In all of the SPARQL queries, a very relevant choke point is the translation of internal IDs into external ones. This should be done as late as possible, at a point of minimum cardinality, e.g. after top-k order. RDF and possibly graph models often use a synthetic integer identifier for entities, e.g. a URI. For presentation purposes to the client applications, these identifiers must be translated to their original form, e.g. the URI string that was used when the data loaded. This should be done as late as possible, or at the point of minimal cardinality in the plan.

The usage of optional triple patterns limits the number of possible query plans, because it is evaluated in the end. This may cause a good query plan to not be available, compared to if a triple pattern was not marked as optional. This happens if the triple pattern in question is highly selective.

**Differences:** The new version of this query for DSB differs from the original SNB query, due to the modifications of the dataset, in the following:

- Last name of a Person is optional;
- Birthday of a Person is optional;
• The selectivity of a query is less than the original one because there are less Person entities in
the dataset with the specified first name, due to the missing first names for some Persons.

4.2.2 Query 2 - Recent Posts and Comments by Your Friends

**Description**: Given a start Person, find a given number of most recent Messages from all of that
Person’s friends, that were created before (and including) a given date.

**Choke points**: This is a navigational query looking for paths of length two, starting from a
given Person, going to their friends and from them, moving to their published Posts and Comments.
This query exercises both the optimizer and the ways the data is stored. It tests the ability to create
execution plans taking advantage of the orderings induced by some operators to avoid performing
expensive sorts. This query requires selecting Posts and Comments based on their creation date, which
might be correlated with their identifier, potentially leading to intermediate results with interesting
orders. Also, messages could be stored in an order correlated with their creation date to improve data
access locality. Finally, as many of the attributes required in the projection are not needed for the
execution of the query, it is expected that the query optimizer will move the projection to the end.

**Differences**: The DSB version of the query is also more complicated than the SNB one:

• First and last names of a Person are optional;

• Comments that should be returned could have `gifFile` property instead of `content`, thus there
  are two UNIONs in the SPARQL query instead of just one.

4.2.3 Query 3 - Friends and Friends of Friends That Have Been to Countries X and Y

**Description**: Given a start Person, find a given number of Persons that are their friends and friends
of friends (excluding start Person) who have made Posts/Comments in both of the given Countries,
X and Y, within a given period. Only Persons that are foreign to Countries X and Y are considered,
that is Persons whose Location is not Country X or Country Y. The result Person entities should be
sorted in a descending order by the number of Posts/Comments made in the specified countries.

**Choke points**: In this query, it is very important for the query optimizer to select the most
efficient join ordering, which will depend on the cardinalities of the intermediate results. Since many
friends of friends can be duplicates, then it is expected to eliminate duplicates prior to accessing the
Post and Comments, as well as eliminate those friends from countries X and Y, as the size of the
intermediate results can be severely affected.

**Differences**: As in all queries returning Persons, the DSB version allows optional first and last
names. It has less selectivity because not all of the posts have an associated location.

4.2.4 Query 4 - New Topics

**Description**: Given a start Person, find a given number of most commonly used Tags that are attached
to Posts that were created by that Person’s friends. Only include Tags that were attached to friends’
Posts created within a given time interval, and that were never attached to friends’ Posts created before
this interval.

**Choke points**: Here, the proper join type selection is crucial, which implies accurate estimates
of cardinalities. Depending on the cardinalities of both sides of a join, a hash or an index based join
operator is more appropriate. These cardinalities are clearly affected by the input Person, the number of friends, the variety of Tags, the time interval and the number of Posts, so for different substitution parameters the different join types should be exploited.

**Differences:** None.

### 4.2.5 Query 5 - New Groups

**Description:** Given a start Person, find a given number of Forums which that Person’s friends and friends of friends (excluding start Person) became Members of after a given date. For each Forum find the number of Posts that were created by any of these Persons, and sort the results by the number of Posts. For each Forum consider only those Persons which joined that particular Forum after the given date.

**Choke points:** As in the previous one, this query requests the proper join type selection. Also, it rewards the usage of indexes, but their accesses will be presumably scattered due to the two/three-hop search space of the query, leading to unpredictable and scattered index accesses. The efficiency of index lookup is very different depending on the locality of keys coming to the indexed access. Techniques like vectoring non-local index accesses by simply missing the cache in parallel on multiple lookups vectored on the same thread may have high impact. Also, detecting absence of locality should turn off any locality dependent optimizations if these are costly when there is no locality.

**Differences:** None.

### 4.2.6 Query 6 - Tag Co-occurrence

**Description:** Given a start Person and some Tag, find a given number of most commonly used other Tags that occur together with this Tag on Posts that were created by start Person’s friends and friends of friends (excluding start Person). For each Tag, find the count of Posts that were created by these Persons, which contain both this Tag and the given Tag.

**Choke points:** Rich join order optimization that is relevant here is a choke point that is much more important in the RDF world than in relational databases, since numerous RDF databases use the relational model to store triples [6]. In one such setting, all triple patterns from the SPARQL query should be translated to SQL as a self-join of the corresponding table, with conditions if there are common subjects, predicates and/or objects, e.g. a SPARQL query with \( n \) triple patterns will result with \( n - 1 \) self-joins, giving to the optimizer a lot of possibilities for join order.

**Differences:** None.

### 4.2.7 Query 7 - Recent Likes

**Description:** Given a start Person, find a given number of most recent Likes on any of start Person’s Messages. Find Persons that liked any of start Person’s Messages, the Messages they liked most recently, creation date of that Like, and the latency (in minutes) between creation of Messages and Like. Additionally, return a flag for each Person found, indicating whether the liker is a friend of the start Person. In the case that a Person liked multiple Messages at the same time, return the Message with the lowest identifier.

**Choke points:** This query has a couple of relevant choke points, including late projection, join type selection, and flattening sub-queries. The last one tests the ability of the query optimizer to flatten
execution plans when there are correlated sub-queries. Many queries have correlated sub-queries and their query plans can be flattened, such that the correlated sub-query is handled using an equi-join, outer-join or anti-join.

Differences: There are 2 differences in this DSB version:

- First and last names of a liker are now optional;
- The content of a message could be a textual content or a photo, but also GIF file, as well.

4.2.8 Query 8 - Recent Replies

Description: Given a start Person, find a given number of most recent Comments that are replies to Messages of the start Person. Only consider immediate (1-hop) replies, not the transitive (multi-hop) case. Return the reply Comments, and the Person that created each reply Comment.

Choke points: This query looks for paths of length two, starting from a given Person, going through its created Messages and finishing at their replies. In this query there is temporal locality between the replies being accessed. Thus the top-k order by this can interact with the selection, i.e. do not consider older posts than the 20th oldest seen so far.

Differences: The same as in Q7.

4.2.9 Query 9 - Recent Posts and Comments by Friends or Friends of Friends

Description: Given a start Person, find a given number of most recent Messages created by that Person’s friends or friends of friends (excluding start Person). Only consider the Messages created before a given date (excluding that date).

Choke points: This is one of the most complex queries, and the list of choke points is longer: late projection, join type selection, scattered indexed access pattern, dimensional clustering, etc. This query is expected to touch variable amounts of data with entities of different characteristics, and therefore, properly estimating cardinalities and selecting the proper operators will be crucial.

Differences: The same as in Q7.

4.2.10 Query 10 - Friend Recommendation

Description: Given a start Person, find a given number of most similar Persons of that Person’s friends of friends (excluding start Person, and immediate friends) to the given Person, who were born on or after the 21st of a given month (in any year) and before the 22nd of the following month. Calculate the similarity between each of these Persons and start Person, where similarity for any Person is defined as follows:

- \( \text{common} = \) number of Posts created by that Person, such that the Post has a Tag that start Person is interested in
- \( \text{uncommon} = \) number of Posts created by that Person, such that the Post has no Tag that start Person is interested in
- \( \text{similarity} = \text{common} - \text{uncommon} \)
**Choke points**: This query looks for paths of length two, starting from a Person and ending at the friends of their friends. It does widely scattered graph traversal, and one expects no locality in friends of friends, as these have been acquired over a long time and have widely scattered identifiers. The join order is simple, but one must see that the anti-join for "not in my friends" is better with hash. Also the last pattern in the scalar sub-queries joining or anti-joining the tags of the candidate’s posts to interests of self should be by hash.

**Differences**: First and last names of a friend of a friend are optional.

### 4.2.11 Query 11 - Job Referral

**Description**: Given a start Person, find a given number of that Person’s friends and friends of friends (excluding start Person) who started Working in some Company in a given Country, before a given date (year). Sort the Person entities in the results by the date they started working for the Company.

**Choke points**: Here, join type selection, scattered indexed access pattern and top-k push down are relevant. The last one tests the ability of the query optimizer to perform optimizations based on top-k selections. Many times queries demand retrieval of the top-k elements. Once k results are obtained, extra restrictions in a selection can be added based on the properties of the \( k \)th element currently in the top-k, being more restrictive as the query advances, instead of sorting all elements and picking the highest k.

**Differences**: The same as in Q10.

### 4.2.12 Query 12 - Expert Search

**Description**: Given a start Person, find the Comments that this Person’s friends made in reply to Posts, considering only those Comments that are immediate (1-hop) replies to Posts, not the transitive (multi-hop) case. Only consider Posts with a Tag in a given TagClass or in a descendant of that TagClass. Count the number of these reply Comments, and collect the Tags (with valid tag class) that were attached to the Posts they replied to. Return Persons with at least one reply, the reply count, and the collection of Tags. Sort the results by the reply count, returning the Person entities with most replies first.

**Choke points**: In this query, the optimizer should try to exclude the dependent group-by keys. Sometimes queries require group-by expressions on a set of columns and a key, where the value of the key determines the columns. In this situation, the aggregation operator should be able to exclude certain group-by attributes from the key matching. The scattered indexed access pattern is also present in this query, as well as the execution of a transitive step.

**Differences**: The same as in Q10.

### 4.2.13 Query 13 - Recent Posts and Comments Where Your Friends Are Mentioned

**Description**: Given a start Person, find a given number of most recent Messages where any of the Person’s friend was mentioned, that were created after (and including) a given date.

**Choke points**: This query does not introduce new choke points because they are similar to those from Q2, but with an important difference: instead of going from a Person’s friends to their Posts, where each Post has its own author, the query looks for Posts where friends were mentioned. The property mentions is not so common as hasCreator, and not all posts have mentioned Persons.
This query, along with Query 14, was introduced in order to touch new properties and parts of the DSB dataset, which given their previously discussed characteristics have a different distribution than other, already existing properties.

**Differences**: This is a new query.

### 4.2.14 Query 14 - New Shared Links

**Description**: Given a start Person, find a given number of most shared links in the Messages created by that Person’s friends or friends of friends (excluding start Person). Only consider the Messages created after a given date (excluding that date) and not shared privately.

**Choke points**: This query adds grouping to the choke points of Q9.

**Differences**: This is a new query.

### 4.3 SPARQL Queries (Short)

Some of these 14 queries return Persons, some of them return Posts. For some of the former, the short queries S1, S2 and S3 will be issued, while for the latter, the rest of the short queries will be executed. Also, short queries can trigger themselves off, as some of them return Persons and/or Posts, as well. All of the short queries are very simple as they represent look-up queries, so the choke points for them will be omitted. These short queries are necessary due to two main reasons: (a) a data storage benchmark has to evaluate the system performance for short lookups as well, as specified in the requirements; (b) a real-world use-case involving any type of application requires more extensive use of short lookups than complex queries and business analytics.

#### 4.3.1 Short 1 - Person Profile

**Description**: Given a start Person, retrieve his first name, last name, birthday, IP address, browser, and city of residence.

**Differences**: First and last names of the person are optional.

#### 4.3.2 Short 2 - Person Recent Messages

**Description**: Given a start Person, retrieve the last 10 Messages created by that user. For each message, return that message, the original post in its conversation, and the author of that post. If any of the Messages is a Post, then the original Post will be the same Message, i.e., that Message will appear twice in that result.

**Differences**: First and last names of the author are optional, and the message could have a **gifFile** property instead of **content** or **imageFile**.

#### 4.3.3 Short 3 - Person Friends

**Description**: Given a start Person, retrieve all of their friends, and the date at which they became friends. Order the results by the date in a descending order.

**Differences**: Friends’ first and last names are optional.
4.3.4 Short 4 - Message Content

Description: Given a Message, retrieve its content and creation date.

Differences: The message could have a gifFile property instead of content or imageFile.

4.3.5 Short 5 - Message Creator

Description: Given a Message, retrieve its author.

Differences: The author’s first and last names are optional.

4.3.6 Short 6 - Message Forum

Description: Given a Message, retrieve the Forum that contains it and the Person that moderates that Forum. Since comments are not directly contained in Forums, for Comments, return the Forum containing the original Post in the thread which the Comment is replying to.

Differences: The moderator’s first and last names are optional.

4.3.7 Short 7 - Message Replies

Description: Given a Message, retrieve the (1-hop) Comments that reply to it. In addition, return a boolean flag indicating if the author of the reply knows the author of the original Message. If author is same as original author, return false for "knows" flag. Order the returned Comments by their date, in a descending order.

Differences: The first and last name of the author of a Comment are optional, and the Comment can have a textual content or a GIF.

4.4 SPARQL Queries (Update)

4.4.1 Update 1 - Add Person

Description: Add a Person to the social network.

Differences: As the data schema modifications are already explained, the entity Person can have missing first and last names, and birthday as well, thus a couple of triples less could be inserted.

4.4.2 Update 2 - Add Post Like

Description: Add a Like to a Post of the social network.

Differences: None.

4.4.3 Update 3 - Add Comment Like

Description: Add a Like to a Comment of the social network.

Differences: None.
4.4.4 Update 4 - Add Forum

**Description:** Add a Forum to the social network.

**Differences:** None.

4.4.5 Update 5 - Add Forum Membership

**Description:** Add a Forum membership to the social network.

**Differences:** None.

4.4.6 Update 6 - Add Post

**Description:** Add a Post to the social network.

**Differences:** A Post can have a missing location, and there can be additional properties that need to be inserted: *links*, *mentions* and *visible*.

4.4.7 Update 7 - Add Comment

**Description:** Add a Comment replying to a Post/Comment to the social network.

**Differences:** The same as in U6, and one more: a Comment can have a *gifFile* property instead of *content*.

4.4.8 Update 8 - Add Friendship

**Description:** Add a friendship relation to the social network.

**Differences:** None.

4.5 Key Performance Indicators

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

- **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.

- **Number of Wrong Answers:** The number of SPARQL SELECT queries whose result set is different from the result set obtained from the triple store used as a gold standard.

- **Throughput:** The average number of tasks executed per second.
5 Evaluation

5.1 Comparison of the Data Storage Benchmark and Social Network Benchmark

In order to evaluate the Data Storage benchmark, i.e. to make sure that the modifications introduced to the synthetic dataset and the DSB SPARQL queries still test the same choke points as SNB, we compared the query execution times of the original SNB queries over the original SNB dataset, and the new DSB queries over the new DSB dataset. We developed the original SNB queries as SPARQL queries for this purpose. The two complex queries (Q13 and Q14) which cannot be expressed in SPARQL 1.1 syntax, were written with a Virtuoso-specific SPARQL syntax which allowed us to execute them over the RDF version of the SNB dataset.

We used Virtuoso as a triple store for all measurements. However, the experiment should not be considered as an evaluation of Virtuoso, but as an evaluation of the proposed modifications to create the new Data Storage benchmark. The experiments presented in this section were executed on a dual Xeon(R) E5-2630 @ 2.33GHz machine with 192GB RAM and 2 SSDs with 470GB.

In order to execute the full benchmark run as a standalone benchmark, a benchmark driver is provided. It is a software tool representing a load generator, designed for benchmarking database management systems. It takes a workload definition, a database connector, and a set of configuration parameters as input. It then generates a workload (i.e., stream of operations, long and short queries together with updates) in conformance with the workload definition, and executes those operations against a given system, using the provided database connector. During execution, the driver continuously measures performance metrics, then generates a report upon completion based on these metrics. It is capable of generating parallel workloads (e.g. concurrent reads and writes), while respecting the configured operation mix and ensuring that ordering between dependent operations is maintained. Therefore, to generate a more or less demanding workload from the same workload definition (same operation mix, same operation parameters, same ordering, etc.), the driver provides a mechanism for compressing / stretching an operation stream such that the intervals between operations is increased or decreased, proportionately for the entire stream. For example, a value of 2.0 means the benchmark will run 2x slower / longer, 0.1 will run 10x faster / shorter, and 1.0 (default) will leave the benchmark unchanged.

Figure 1: Average Complex Query Execution Times on the SNB Dataset (Q2-Q7)

Figure 2: Average Complex Query Execution Times on the DSB Dataset (Q2-Q7)

The experiment consists of running all query types with different substitution parameters, sepa-
rately, single threaded, sequentially, allowing the system to execute a query as fast as possible, without the possible influence of concurrent queries. When a real evaluation of a triple store is being executed, this aspect of multi-threaded runs should also be taken into account. The first part of the experiment deals with the original SNB dataset with the SNB queries, proposed in LDBC, run on different sizes of the dataset. The tested scale factors are 1, 3, 10, 30 and 100. Tables 4 and 5 contain the number of Persons and Posts in the SNB and DSB datasets, per different scale factors, and the total numbers of triples representing the entire datasets, as well. All short and long queries are executed 1,000 times with different parameters, forcing the optimizer to use different query plans, while each type of updates was executed 10,000 times. The second part of the experiment was performed in the same manner, but with the new versions of the queries for DSB, against the new DSB dataset. This approach ensured a good base for comparing the original SNB and the new Data Storage benchmark.

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Number of Persons</th>
<th>Number of Posts</th>
<th>Number of Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9,892</td>
<td>3.1M</td>
<td>48.0M</td>
</tr>
<tr>
<td>3</td>
<td>24,328</td>
<td>8.9M</td>
<td>143.1M</td>
</tr>
<tr>
<td>10</td>
<td>65,645</td>
<td>29.2M</td>
<td>486.9M</td>
</tr>
<tr>
<td>30</td>
<td>165,430</td>
<td>87.5M</td>
<td>1,508.2M</td>
</tr>
<tr>
<td>100</td>
<td>448,626</td>
<td>277.2M</td>
<td>4,905.5M</td>
</tr>
</tbody>
</table>

Figures 1, 2, 3 and 4 depict the average query execution times for the queries on different sizes of the datasets. The x-axis depicts the different scale factors, while the y-axis shows the times in milliseconds. Since the y-axis is in a logarithmic scale, the nearly linear plot lines for the queries represent a logarithmic exponential increase of their execution times as the scale factor grows. This is expected, since all queries start with a specific Person (a substitution parameter) and then go to their Posts, Comments or Friends, etc. Actually, 14 read queries have an average complexity of $O(D \log(n))$, $O(D^2 \log(n))$ or $O(D^3 \log(n))$ (where $D$ is the average out-degree of a node in a social graph and does not depend on $n$, which is the total number of triples), depending on whether they touch one-, two- or three-hop friendship circle. The logarithmic component is a result of a corresponding index lookup.

The execution times for Q1 are omitted from the charts, since the current version of Virtuoso
Table 5: DSB Dataset Sizes

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Number of Persons</th>
<th>Number of Posts</th>
<th>Number of Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9,927</td>
<td>3.1M</td>
<td>47.4M</td>
</tr>
<tr>
<td>3</td>
<td>24,333</td>
<td>8.8M</td>
<td>138.7M</td>
</tr>
<tr>
<td>10</td>
<td>65,678</td>
<td>29.2M</td>
<td>476.2M</td>
</tr>
<tr>
<td>30</td>
<td>165,601</td>
<td>86.8M</td>
<td>1,461.7M</td>
</tr>
<tr>
<td>100</td>
<td>449,283</td>
<td>278.6M</td>
<td>4,836.3M</td>
</tr>
</tbody>
</table>

exhibits a problem when compiling the query. The plot lines for Q2, Q3, Q4, Q6, Q8 are almost linear, times are almost equal on both benchmarks, and the visible execution time increase is reasonable. The same conclusion would have been applicable for Q7, as well, if there had not been the problem of a non-optimal query execution plan on the largest scale factor. The results for Q5 and Q9 are too high to be considered interactive, but they are still similar on both benchmarks. This means that the Virtuoso query optimizer has a problem with finding an optimal query execution plan for this query on all dataset sizes, due to the wrong estimate of cardinalities, but it also shows that our modifications of the SNB dataset to construct the DSB dataset do not have any impact on the query performance. Q10 and Q11 are slightly faster in the new benchmark due to the modifications of the dataset, i.e. the decreased structuredness. The line for Q12 clearly shows the problem with this query on scale factors 3 and 10 in the SNB benchmark, while in other cases it does not show any performance issues. A comparison between Q13 and Q14 on the two benchmark versions does not make sense, as they represent completely different queries.

The Figures 5, 6, 7 and 8 show the same measurements, but for short queries and updates. From the figures, we can conclude that the differences between SNB and DSB are even smaller. Since the measurements in these figures are in tens of milliseconds, the plot lines are not as linear as in the previous figures. In the updates, the execution times only depend on the number of triples that should be inserted, thus the lines for U2, U3 and U5 almost overlap, and they are located at the bottom of figures, indicating the smallest number of triples for insertion – 3 triples. Above them, there is a line for U8, with 8 triples for insertion. Posts and Comments have almost the same number of properties, thus the lines representing their updates are almost the same. U1 is responsible for inserting around 20 triples, due to which it is located on the top of the figures.

Figure 5: Average Short Query Execution Times on the SNB Dataset

Figure 6: Average Short Query Execution Times on the DSB Dataset
Figure 9 contains ratios of the average query execution times on the SNB datasets and on the DSB ones. The values around 1 mean similar execution times and their cells are colored in yellow; the green color depicts faster executions on the new DSB dataset, as opposed to the red color. These diagrams and this table could represent excellent guidelines as to where the optimizer of a particular storage solution (in this case Virtuoso) should be improved.

They also prove that the new Data Storage benchmark preserves the testing of the intended choke-points from the existing Social Network benchmark, while at the same time provides a real-world compliant RDF dataset for more realistic benchmarking. This characteristic of the DSB RDF dataset may not be visible in the initial tests on Virtuoso – since it copes well with less coherent data as well – but since the dataset structuredness has a direct impact on data storage, indexing and querying, we are confident that the DSB dataset and queries will provide a better testing environment for RDF storage solutions.
5.2 Validating the Data Storage Benchmark on Different Triple Stores

In order to prove that all SPARQL queries from the Data Storage benchmark (DSB) are compatible with and can be executed on different systems, we provide an initial evaluation of Blazegraph, Fuseki and Virtuoso. The experiment can also serve as a comparison of these three systems, but this comparison should be considered an initial one, due to the following:

- In order to have the final comparison, the full benchmark should be executed, with concurrent reads and writes, with real workload comprised of proposed query mixes, using the provided benchmark driver, etc.
- All systems are used with their default settings, while an improved configuration in any of them can lead to a significant increase in their score in the benchmark.

The experiments from this section were performed with an Intel(R) Xeon(R) CPU E5-2698 v3 CPU with 2.3GHz, 128GB RAM, 2.56TB HDD running Ubuntu 16.04.2 and Java 1.8.

Before the measurement run (90 version with different parameters of each query, single threaded, sequential) and the warm-up run (9 queries for each type), all queries were passed through a SPARQL 1.1 syntax validation.

The advantages of a comparison such as this one, are:

- Validation of query results across systems,
- Improved comparison between systems, query by query, without a possible influence of the concurrency of a benchmark, and thus
- Guidelines for a query optimizer of a system where the available improvements could be made.

Figure 10: Average Complex Query Execution Times on SF1

Figure 11: Average Complex Query Execution Times on SF10

The experiment consists of two different dataset sizes: the smallest one, scale factor 1 (SF1), and the ten-times larger one, scale factor 10 (SF10), showing the behavior of the triple stores against the small- and mid-size datasets. Figures 10, 12, 11 and 13 show the average query execution times (in milliseconds) per query type on all three systems for SF1 and SF10, respectively.

Missing dots represent timeouts (3 minutes) or some other problem which occurred while running the query. For instance, Blazegraph does not support `xsd:duration`, resulting in error messages for Q3s.
and Q4s. On a smaller dataset, Fuseki has timeouts for Q2s and Q9s. On a larger dataset, Blazegraph was unable to finish Q7s, Q8s, Q9s and Q10s on time, while Fuseki added timeouts for Q7s. The maximum query execution time has not limited any of the query executions on Virtuoso.

It should be mentioned that Blazegraph has problems with short queries, containing a pattern with a specified number without a specified type, although the SPARQL standard permits the syntax:

\[ ?\text{subject} \text{snvoc:id} \%ID\% . \]

The queries for Blazegraph were rewritten, adding the type in question, so the graphs in the figures above represent the average execution times for these rewritten versions:

\[ ?\text{subject} \text{snvoc:id} "\%ID\%"^^\text{xsd:long} . \]

The slowest query for all systems on both dataset sizes is Q9. In SF1, Blazegraph solves it on average in 54s, Virtuoso in less than 7s, while Fuseki is unable to execute it in less than 3 minutes. On a larger dataset, Virtuoso needs 30s, while Fuseki and Blazegraph result in timeout. The second slowest for Virtuoso is Q5, but at the same time, Q7 is more complicated for Fuseki and Blazegraph. Q7s are executed by Virtuoso in 0.2s (SF1) and 0.7s (SF10), meaning that Fuseki and Blazegraph have problems with the choke-points that were used when designing Q7. A very similar scenario is present in Q8 – it is one of the simplest queries for Virtuoso, while the other systems execute it in about 10s. The fastest long query for all systems was Q11, and the differences between the systems are minor.

It is obvious that Virtuoso shows the best results in this experiment, which is something to be expected bearing in mind previous benchmark comparisons between them, as seen in [7].

While the results of the benchmark convey that Virtuoso performs best on average, they also allow us to spot several areas where Virtuoso can be further optimized. For example, Fuseki is twice as fast as Virtuoso when Q3 is executed on SF1. Also, we should check the optimality of Q4 on the same dataset, where Fuseki is slightly faster than Virtuoso.

In comparing the numbers for SF1 and SF10, one can expect higher values, i.e. longer execution times, but not by much, as the theoretical complexity of the queries is logarithmic. Therefore, the ratio between the execution times on SF10 and SF1 should not be around 10, but much less. This ratio can be used as an indicator of the system scalability. Figure 14 contains these ratios, per query and system, and can be used as a guideline for improving the performance of the triple store related
to the particular query. For instance, it is obvious that possible improvements for Q1 exist under all systems, since the average execution time of this query increases in a linear fashion. For Virtuoso, the most problematic queries from this perspective are Q4 and Q6, while for Fuseki it is Q4. The worst situation is for Blazegraph, where all measured ratios are around or greater than 10, stressing a poor system scalability. The ratios corresponding to short queries are about 1, showing the optimal scalability related to simple look-up queries for all the systems.

6 Integration into the HOBBIT Platform

The Data Storage benchmark (DSB) has been fully integrated into the HOBBIT platform, and all interested parties can evaluate their system against it by running the benchmark through the platform website\(^7\). Interested parties should follow the instructions from the platform Wiki page\(^8\) – 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\(^9\).

6.1 DSB Parameters

The Data Storage benchmark has parameters which need to be set in order to execute the benchmark. These parameters are independent of the triple store which is evaluated. Figure 15 illustrates an example configuration of the DSB. The required parameters are:

- **Number of operations:** The user must provide the total number of SPARQL queries that should be executed against the tested system. This number includes both SELECT and INSERT queries, given at Appendix A, B and C. The ratio between them, e.g. the number of queries per query type, has been specified in such a way that each query type has the same impact on the overall score of the benchmark. This means that the simpler and faster queries are present much more frequently than the complex and slower ones. This kind of a query mix will be tuned after the Mighty Storage Challenge (MOCHA) at ESWC 2017\(^{10}\), based on the results achieved by all participating systems in the corresponding task of the challenge.

---

\(^7\) [http://master.project-hobbit.eu/](http://master.project-hobbit.eu/)
\(^8\) [https://github.com/hobbit-project/platform/wiki](https://github.com/hobbit-project/platform/wiki)
\(^9\) [https://project-hobbit.eu/challenges/mighty-storage-challenge/msc-tasks/](https://project-hobbit.eu/challenges/mighty-storage-challenge/msc-tasks/)
\(^10\) [https://project-hobbit.eu/events/mighty-storage-challenge-on-eswc-2017/](https://project-hobbit.eu/events/mighty-storage-challenge-on-eswc-2017/)
6.2 Components

Here, a brief description of the main components of the Data Storage benchmark is given. Their implementation can be found as part of the public GitHub project\footnote{https://github.com/hobbit-project/DataStorageBenchmark}.

6.2.1 Benchmark Controller

The Benchmark Controller is used to create and orchestrate all other components of the benchmark. During initialization, it gathers parameters from the Platform Controller, necessary to initialize the other components. After the initialization, the Benchmark Controller executes the Data Storage benchmark by sending start signals to the Data Generator and the Task Generator. Once the Data Generator, the Task Generator and the System Adapter finish their tasks, the Benchmark Controller creates the Evaluation Module of the benchmark, waits for it and sends the received evaluation results of the benchmark to the Platform Storage, from where the platform can read it, and present it to the user.

6.2.2 Data Generator

The Data Generator is the component that creates the dataset for the benchmark. The new DATAGEN for the Data Storage benchmark was already described in Section 3, and the Data Generator component represents a wrapper around it.

The workflow of the Data Storage benchmark, in the context of the HOBBIT platform and its design, makes it unnecessary to generate the synthetic RDF dataset on each individual benchmark run. Due to the different available scale factors for the dataset, the dataset can reach a size of several billions of triples – such a dataset can take several hours to be generated, even on very powerful hardware, using all available resources. Therefore, we changed the benchmark approach and pre-generated the dataset.

- **Scale factor**: The 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 given at Section 3.

![Figure 15: DSB parameters](image-url)
datasets for every size (scale factor). In order to provide fast access, we placed them as part of the benchmark and the platform. An additional reason which supports our decision to use and download a pre-generated dataset instead of generating it, will be explained in the next subsection.

When started, the Data Generator initializes, receives the requested scale factor from the Benchmark Controller, downloads the dataset in question and sends it to the System Adapter for a bulk load. After that, it waits for a signal from the System Adapter indicating the bulk load is done, and then sends the data to the Task Generator, which requires the data to prepare the SPARQL SELECT and INSERT queries.

### 6.2.3 Task Generator

The Task Generator is the component which creates the tasks, i.e. the SPARQL SELECT and INSERT queries, based on the incoming data. It cannot prepare queries without dataset knowledge, due to two reasons:

- The SPARQL INSERT queries should represent insertions of activity in a social network, which happen after the last timestamp used in the bulk loaded dataset – with this the benchmark dataset is divided into two parts: one used for bulk loading, the the other as part of the update streams.

- The SPARQL SELECT queries should be related to existing entities in the dataset.

The answers of the SPARQL SELECT queries from the system being benchmarked should be checked. Due to the nature of the SELECT queries, the answers cannot be calculated easily from the dataset, without using an alternative triple store as a gold standard. There are two ways for achieving this: by querying and updating a gold standard triple store in parallel with the system under test, and by pre-calculating the answers only once and using them every time a system is being benchmarked. The former unnecessarily increases the time necessary for executing a single experiment, along with hardware resources. Therefore, we decided to take the latter approach.

The Task Generator receives the data from the Data Generator, prepares the queries and retrieves the pre-calculated answers to them. It then sends the queries to the System Adapter, and the answers to the Evaluate Storage.

In the first version of the Data Storage benchmark 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. To achieve such behavior, the Task Generator and the Evaluation Storage are synchronized. With this, we examine and appraise the best performance of the tested system for a given query, by allowing the system to have all resources available for the query in question.

### 6.2.4 System Adapter

The System Adapter is a component which implements the API defined by the Data Storage benchmark (DSB) and enables the communication between the benchmark and the benchmarked system. We developed an instance of the System Adapter used for our baseline implementation of the DSB, for which we used the Virtuoso open-source version (VOS). The System Adapter is the part of the same Docker container as the system.
After this component initializes itself, it starts receiving data from the Data Generator – the files with the benchmark dataset. When all files are accepted, indicated by a signal from the Data Generator, the System Adapter starts loading the dataset into the system being tested. Upon completion, it sends a signal to the other components indicating it is ready to start answering the SPARQL queries, which are then sent by the Task Generator. All accepted queries are then executed against the benchmarked system, and their answers are sent to the Evaluation Storage, for validation against the expected answers.

### 6.2.5 Evaluation Module

The Evaluation Module is the component which evaluates the results received from the benchmarked system. For each performed query, the Evaluation Storage saves the execution start-time and end-time, along with the expected result set. Based on that, this component calculates all specified key performance indicators (KPIs), and sends them to the platform as an RDF model.

### 6.2.6 Key Performance Indicators (KPIs)

The KPIs of the experiment running against DSB are the ones described in Section 4.5.

---

Figure 16: DSB results as a proof of concept.
6.2.7 Experiments

In order to test the Data Storage benchmark integration into the HOBBIT platform, we did a test run over the baseline implementation (Figure 16). The experiment parameters were: scale factor = 1, number of operations = 10,000. The results are presented here as a proof of concept and should not be considered as official results, since the underlying system has not been properly tuned, yet.

6.3 Preliminary Results

At the end of the reporting period for the deliverable, the Data Storage Benchmark was part of the Mighty Storage Challenge (MOCHA) 2017\textsuperscript{12}, at ESWC 2017\textsuperscript{13}. The challenge consisted of four tasks, and DSB was the Task 2. The configuration for DSB at MOCHA 2017 was: SF = 1, number of operations = 15,000. The benchmark had a defined maximum time for the experiment of 30 minutes.

For the participation in Task 2 of MOCHA 2017, i.e. for the Data Storage Benchmark, three systems applied and were submitted: Virtuoso 7.2 Open-Source Edition by OpenLink Software, Virtuoso 8.0 Commercial Edition (beta release) by OpenLink Software, and QUAD by Ontos.

Unfortunately, QUAD was not able to finish the experiment in the requested time, i.e. it exhibited a timeout.

Based on the results from the KPIs, shown in Figures 17, 18, 19 and 20, the winning system for the task was Virtuoso 7.2 Open-Source Edition by OpenLink Software.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{loading_time.png}
\caption{Loading Time}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{throughput.png}
\caption{Throughput}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{long_queries.png}
\caption{Long Queries}
\end{figure}

\textsuperscript{12}https://project-hobbit.eu/challenges/mighty-storage-challenge/
\textsuperscript{13}http://2017.eswc-conferences.org/
Figure 20: Short Queries and Updates

References


A SPARQL queries (Complex)

A.1 Query 1 - Friends with a Certain Name

```
SELECT ?fr ?last (min(?dist) as ?mindist)
  ((?bday - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000 as ?birthday)
  ((?since - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000 as ?creationDate)

{ ?fr a snvoc:Person . ?fr snvoc:firstName "%Name%" .
  OPTIONAL { ?fr snvoc:lastName ?last } .
  
  { SELECT DISTINCT ?fr (1 as ?dist)
  WHERE {
    sn:pers%Person% snvoc:knows ?fr.
  }
  }
  UNION
  { SELECT DISTINCT ?fr (2 as ?dist)
  WHERE {
    FILTER (?fr != sn:pers%Person%).
  }
  }
  UNION
  { SELECT DISTINCT ?fr (3 as ?dist)
  WHERE {
    FILTER (?fr != sn:pers%Person%).
  }
  }
{ SELECT ?fr (group_concat (?email; separator = ", " ) as ?emails)
  WHERE {
  }
  GROUP BY ?fr
}
.
{ SELECT ?fr (group_concat (?lng; separator = ", " ) as ?lngs)
  WHERE {
  }
  GROUP BY ?fr
}
.
OPTIONAL {
  SELECT ?fr (group_concat ( concat (?o_name, " ", ?year, " ", ?o_country); separator = ", ") as ?studyAt)
```
A.2 Query 2 - Recent Posts and Comments by Your Friends

SELECT ?fr ?first ?last ?post ?content ((?date - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000 as ?creationDate)
WHERE {
  OPTIONAL { ?fr snvoc:firstName ?first }
  OPTIONAL { ?fr snvoc:lastName ?last }
  {
    { ?post snvoc:content ?content } UNION
    { ?post snvoc:imageFile ?content } UNION
    { ?post snvoc:gifFile ?content }
  }.
  ?post snvoc:creationDate ?date.
  FILTER (?date <= "%Date%"^^xsd:dateTime).
}
ORDER BY DESC(?creationDate) ?post
LIMIT %limit%
A.3 Query 3 - Friends and Friends of Friends That Have Been to Countries X and Y

```
WHERE {
    WHERE {
      { SELECT ?fr (count(*) as ?ct1)
        WHERE {
          ?post snvoc:creationDate ?date .
          FILTER (?date >= "%Date0%"^^xsd:dateTime &&
                     ?date < "%Date0%"^^xsd:dateTime + xsd:duration("P%Duration%D")) .
          ?post snvoc:isLocatedIn dbpedia:%Country1%
        }
        GROUP BY ?fr
      }
      { SELECT ?fr (count(*) as ?ct2)
        WHERE {
          ?post2 snvoc:creationDate ?date2 .
          FILTER (?date2 >= "%Date0%"^^xsd:dateTime &&
                     ?date2 < "%Date0%"^^xsd:dateTime + xsd:duration("P%Duration%D")) .
          ?post2 snvoc:isLocatedIn dbpedia:%Country2%
        }
        GROUP BY ?fr
      }
      { sn:pers%Person% snvoc:knows ?fr. } UNION
        FILTER (?fr != sn:pers%Person%) }
    }
    OPTIONAL { ?fr snvoc:firstName ?first }.
    OPTIONAL { ?fr snvoc:lastName ?last }.
    ?fr snvoc:isLocatedIn ?city.
    FILTER(!exists {?city snvoc:isPartOf dbpedia:%Country1%}).
    FILTER(!exists {?city snvoc:isPartOf dbpedia:%Country2%}).
  }
  FILTER (?ct1 > 0 && ?ct2 > 0) .
}
ORDER BY DESC(?sum) ?fr
LIMIT %limit%
```

A.4 Query 4 - New Topics

```
SELECT ?tagname (COUNT(*) as ?cnt)
WHERE {
  ?post snvoc:creationDate ?date. %Person% snvoc:knows ?fr.
  FILTER (?date >= "%Date%"^^xsd:dateTime && ?date <= "%Date%"^^xsd:dateTime + xsd:duration("%Duration%"))
  FILTER (!exists {
  })
```
A.5 Query 5 - New Groups

SELECT ?title ?group (COUNT(?post) as ?cnt)
WHERE {
{SELECT distinct ?fr
  WHERE {
    { %Person% snvoc:knows ?fr. } UNION
      FILTER (?fr != %Person%)
    }
  }
}
FILTER (?date >= "%Date%"^^xsd:dateTime)
}
GROUP BY ?title ?group
ORDER BY desc(?cnt) ?group
LIMIT %limit%

A.6 Query 6 - Tag Co-occurrence

SELECT ?tagname (COUNT(*) as ?cnt)
WHERE {
{ SELECT DISTINCT ?fr
  WHERE {
    { %Person% snvoc:knows ?fr. } UNION
      FILTER (?fr != %Person%)
    }
  }
}
?post snvoc:hasTag ?tag1. ?tag1 foaf:name '%Tag%'.
FILTER (?tagname != '%Tag%')
}
GROUP BY ?tagname
ORDER BY desc(?cnt) ?tagname
LIMIT %limit%
A.7 Query 7 - Recent Likes

SELECT ?liker ?first ?last  
((?max_ldt - xsd:dateTime("1970-01-01T00:00:00+00:00")) * 1000 as ?max_ldate) 
(not exists {%Person% snvoc:knows ?liker} as ?is_new)  
?post ?content ((?max_ldt - ?dt)/60 as ?lag)
WHERE {
  {{ SELECT ?liker (max(?ldt) as ?max_ldt)  
    WHERE {
      ?post snvoc:hasCreator %Person% .  
    }  
    GROUP BY ?liker  
    ORDER BY DESC(?max_ldt)  
    LIMIT %limit% 
  }  
  {{ ?lk1 snvoc:hasPost ?post } UNION { ?lk1 snvoc:hasComment ?post }} .  
  OPTIONAL { ?liker snvoc:firstName ?first }  
  OPTIONAL { ?liker snvoc:lastName ?last }  
  ?post snvoc:creationDate ?dt. 
}
ORDER BY DESC(?max_ldate) ?liker

A.8 Query 8 - Recent Replies

SELECT ?from ?first ?last  
((?dt - xsd:dateTime("1970-01-01T00:00:00+00:00")) * 1000 as ?creationDate)  
?rep ?content
WHERE {
  { SELECT ?rep ?dt  
    WHERE {
      ?post snvoc:hasCreator %Person%.  
    }  
    ORDER BY DESC (?dt)  
    LIMIT %limit% 
  }
  ?rep snvoc:hasCreator ?from.  
  OPTIONAL { ?from snvoc:firstName ?first }  
  OPTIONAL { ?from snvoc:lastName ?last }  
  {{?rep snvoc:content ?content} UNION {?rep snvoc:gifFile ?content}} 
}
ORDER BY DESC(?creationDate) ?rep

A.9 Query 9 - Recent Posts and Comments by Friends or Friends of Friends

((?date - xsd:dateTime("1970-01-01T00:00:00+00:00")) * 1000 as ?creationDate)
WHERE {
  { SELECT distinct ?fr
WHERE {
  { %Person% snvoc:knows ?fr. } UNION
    FILTER (?fr != %Person%)
  } }
}
OPTIONAL { ?fr snvoc:firstName ?first }
OPTIONAL { ?fr snvoc:lastName ?last }
FILTER (?date < "%Date%"^^xsd:dateTime)
{ { ?post snvoc:content ?content } UNION
  { ?post snvoc:imageFile ?content } UNION
  { ?post snvoc:gifFile ?content }
} }
ORDER BY DESC(?creationDate) ?post
LIMIT %limit%

A.10 Query 10 - Friend Recommendation

SELECT ?first ?last ((coalesce(?s1, 0) - coalesce(?s2,0)) as ?score) ?fof ?gender ?locationname
WHERE {
  { SELECT distinct ?fof
    WHERE {
      sn:pers%Person% snvoc:knows ?fr .
      FILTER (?fof != sn:pers%Person%).
      MINUS { sn:pers%Person% snvoc:knows ?fof } .
    }
  } .
  OPTIONAL { ?fof snvoc:firstName ?first } .
  OPTIONAL { ?fof snvoc:lastName ?last } .
  ?fof snvoc:isLocatedIn ?based .
  ?based foaf:name ?locationname .
  FILTER (1 = if (month (?bday) = %HS0%, if (day(?bday) >= 21, 1, 0),
    if (month (?bday) = %HS1%, if (day(?bday) < 22, 1, 0), 0)).
  }
  OPTIONAL {
    SELECT ?fof (count (distinct ?post) as ?s1)
    WHERE {
      ?post a snvoc:Post .
      ?post snvoc:hasCreator ?fof .
      sn:pers%Person% snvoc:hasInterest ?tag
    }
    GROUP BY ?fof
  } .
  OPTIONAL {
    SELECT ?fof (count (distinct ?post) as ?s2)
    WHERE {
      ?post a snvoc:Post .
      ?post snvoc:hasCreator ?fof .
    }
  } .
FILTER (!exists { sn:pers%Person% snvoc:hasInterest ?tag.
=post snvoc:hasTag ?tag .})

GROUP BY ?fof
}
}
ORDER BY DESC(?score) ?fof
LIMIT %limit%

A.11 Query 11 - Job Referral

WHERE {
?org snvoc:isLocatedIn ?country. ?country foaf:name '%Country%'.
FILTER (?startdate < %Date%)
{ SELECT DISTINCT ?fr
WHERE {
  { %Person% snvoc:knows ?fr. } UNION
    FILTER (?fr != %Person%)}
}
}
OPTIONAL { ?fr snvoc:firstName ?first }
OPTIONAL { ?fr snvoc:lastName ?last }
}
ORDER BY ?startdate ?fr desc(?orgname)
LIMIT %limit%

A.12 Query 12 - Expert Search

SELECT ?exp ?first ?last (group_concat(distinct ?tagname; separator=', ') as ?tags)
(COUNT (distinct ?reply) as ?cnt)
WHERE {
%Person% snvoc:knows ?exp.
OPTIONAL { ?exp snvoc:firstName ?first }
OPTIONAL { ?exp snvoc:lastName ?last }
?reply snvoc:hasCreator ?exp.
?tag a ?type.
?type rdfs:subClassOf* ?type1. ?type1 rdfs:label "%TagType%".
}
GROUP BY ?exp ?first ?last
ORDER BY DESC(?cnt) ?exp
LIMIT %limit%
A.13  Query 13 - Recent Posts and Comments Where Your Friends Are Mentioned

   (((?datetime - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000) as ?creationDate)
WHERE {
   %Person% snvoc:knows ?fr.
   ?post snvoc:hasCreator ?author.
   ?post snvoc:creationDate ?datetime.
   FILTER (?datetime >= "%Date%"^^xsd:dateTime).
   { { ?post snvoc:content ?content } UNION
     { ?post snvoc:imageFile ?content } UNION
     { ?post snvoc:gifFile ?content }
   }.
   OPTIONAL { ?author snvoc:firstName ?fname }.
   OPTIONAL { ?author snvoc:lastName ?lname }.
}
ORDER BY DESC (?datetime) ?post
LIMIT %limit%

A.14  Query 14 - New Shared Links

SELECT ?link (COUNT (*) as ?shares)
WHERE {
   { SELECT DISTINCT ?fr
     WHERE {
       { %Person% snvoc:knows ?fr. } UNION
         FILTER (?fr != %Person%)
       }
     }
   }
   ?post snvoc:creationDate ?datetime.
   FILTER (?datetime >= "%Date%"^^xsd:dateTime).
   FILTER (!exists {?post snvoc:visible "false"^^xsd:boolean }).
}
GROUP BY ?link
ORDER BY DESC (?shares) ?link
LIMIT %limit%

B  SPARQL queries (Short)

B.1  Short 1 - Person Profile

   (((?p_birthday - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000) as ?p_bd)
   (((?p_creationdate - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000) as ?p_cd)
WHERE {
   ?person snvoc:id %Id%.
OPTIONAL { ?person snvoc:firstName ?p_firstname }.
OPTIONAL { ?person snvoc:lastName ?p_lastname }.
OPTIONAL { ?person snvoc:birthday ?p_birthday }.
?person snvoc:creationDate ?p_creationdate.
?person snvoc:isLocatedIn ?p_place.
}

B.2 Short 2 - Person Recent Messages

SELECT ?post ?con
  { ((?cd - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000 as ?cdate) }.
WHERE {
  ?post snvoc:hasCreator ?pers.
  ?post snvoc:content ?con.
}

ORDER BY DESC(?cd)
LIMIT 10

B.3 Short 3 - Person Friends

SELECT ?fr ?p_friendfirstname ?p_friendlastname
  { ((?k_since - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000 as ?k_s) }.
WHERE {
  ?tmp snvoc:creationDate ?k_since.
  ?tmp snvoc:hasPerson ?fr.
}

ORDER BY DESC(?k_s)
LIMIT 10

B.4 Short 4 - Message Content

SELECT ?con
  { ((?dt - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000 as ?date) }.
WHERE {
  ?post snvoc:id %Id%.
B.5 Short 5 - Message Creator

```sparql
SELECT ?creator ?p_firstname ?p_lastname
WHERE {
?post snvoc:id %Id%.
?post snvoc:hasCreator ?creator.
OPTIONAL { ?creator snvoc:firstName ?p_firstname }.
OPTIONAL { ?creator snvoc:lastName ?p_lastname }.
}
```

B.6 Short 6 - Message Forum

```sparql
WHERE {
?post snvoc:id %Id%.
?orig a snvoc:Post.
?forum snvoc:hasModerator ?moderator.
OPTIONAL { ?moderator snvoc:firstName ?first }.
OPTIONAL { ?moderator snvoc:lastName ?last }.
}
```

B.7 Short 7 - Message Replies

```sparql
SELECT ?comment ?content 
((?dt - xsd:dateTime("1970-01-01T00:00:00.000+00:00")) * 1000 as ?date)
?creator ?creatorfirstname ?creatorlastname
(exists { ?creator snvoc:knows ?author } as ?knows)
WHERE {
?post snvoc:id %Id%.
?post snvoc:hasCreator ?author.
{ { ?comment snvoc:content ?content } UNION
{ ?comment snvoc:gifFile ?content }
}.
?comment snvoc:creationDate ?dt.
?comment snvoc:hasCreator ?creator.
OPTIONAL { ?creator snvoc:firstName ?creatorfirstname }.
OPTIONAL { ?creator snvoc:lastName ?creatorlastname }.
}
ORDER BY DESC(?date) ?creator
```
C  SPARQL queries (Update)

C.1 Update 1 - Add Person

INSERT DATA {
  %Person% a snvoc:Person .
  %Person% snvoc:firstName "%FirstName%" .
  %Person% snvoc:lastName "%LastName%" .
  %Person% snvoc:gender "%Gender%" .
  %Person% snvoc:birthDay "%Birthday%"^^xsd:date .
  %Person% snvoc:creationDate "%CreationDate%"^^xsd:dateTime .
  %Person% snvoc:locationIP "%LocationIP%" .
  %Person% snvoc:browserUsed "%BrowserUsed" .
  %Person% snvoc:isLocatedIn <%CityURI%> .
  %Person% snvoc:id "%PersonId%"^^xsd:long .
  %Person% snvoc:speaks "%Language1%" .
  %Person% snvoc:speaks "%Language2%" .
  %Person% snvoc:speaks "%Language3%" .
  %Person% snvoc:email "%Email1%" .
  %Person% snvoc:email "%Email2%" .
  %Person% snvoc:hasInterest <%Tag1URI%> .
  %Person% snvoc:hasInterest <%Tag2URI%> .
  %Person% snvoc:hasInterest <%Tag3URI%> .
  %Person% snvoc:studyAt [
    snvoc:hasOrganisation <%Organization1URI%> ;
    snvoc:classYear "%StudyYear%"
  ] .
  %Person% snvoc:workAt [
    snvoc:hasOrganisation <%Organization2URI%> ;
    snvoc:workFrom "%WorkStartYear%"
  ] .
}

C.2 Update 2 - Add Post Like

INSERT DATA {
  %Person% snvoc:likes [
    snvoc:hasPost <%PostURI%> ;
    snvoc:creationDate "%CreationDate%"^^xsd:dateTime
  ] .
}

C.3 Update 3 - Add Comment Like

INSERT DATA {
  %Person% snvoc:likes [
    snvoc:hasComment <%CommentURI%> ;
    snvoc:creationDate "%CreationDate%"^^xsd:dateTime
  ] .
}
C.4 Update 4 - Add Forum

INSERT DATA {
  %Forum% a snvoc:Forum .
  %Forum% snvoc:title "%ForumTitle%" .
  %Forum% snvoc:creationDate "%CreationDate%"^^xsd:dateTime .
  %Forum% snvoc:hasModerator <%ModeratorURI%> .
  %Forum% snvoc:id "%ForumId%"^^xsd:long .
  %Forum% snvoc:hasTag <%Tag1URI%> .
  %Forum% snvoc:hasTag <%Tag2URI%> .
}

C.5 Update 5 - Add Forum Membership

INSERT DATA {
  %Forum% snvoc:hasMember [snvoc:hasPerson <%MemberURI%> ;
    snvoc:joinDate "%JoinDate%"^^xsd:dateTime .
  ] .
}

C.6 Update 6 - Add Post

INSERT DATA {
  %Post% a snvoc:Post .
  %Post% snvoc:locationIP "%LocationIP%" .
  %Post% snvoc:creationDate "%CreationDate%"^^xsd:dateTime .
  %Post% snvoc:browserUsed "%BrowserUsed%" .
  %Post% snvoc:language "%Language%" .
  %Post% snvoc:content "%Content%" .
  %Post% snvoc:length "%Length%" .
  %Post% snvoc:hasCreator <%AuthorURI%> .
  %Post% snvoc:id "%PostId%"^^xsd:long .
  %Post% snvoc:containerOf <%PostURI%> .
  %Post% snvoc:isLocatedIn <%CountryURI%> .
  %Post% snvoc:hasTag <%Tag1URI%> .
  %Post% snvoc:hasTag <%Tag2URI%> .
  %Post% snvoc:hasTag <%Tag3URI%> .
  %Post% snvoc:mentions <%Person1URI%> .
  %Post% snvoc:mentions <%Person2URI%> .
  %Post% snvoc:visible "%Visibility%"^^xsd:boolean .
  %Post% snvoc:links "%Link%" .
}

C.7 Update 7 - Add Comment

INSERT DATA {
  %Comment% a snvoc:Comment .
  %Comment% snvoc:locationIP "%LocationIP%" .
  %Comment% snvoc:creationDate "%CreationDate%"^^xsd:dateTime .
  %Comment% snvoc:browserUsed "%BrowserUsed%" .
  %Comment% snvoc:content "%Content%" .
}
C.8 Update 8 - Add Friendship

INSERT DATA {
  %Person1% snvoc:knows [
    snvoc:hasPerson %Person2% ;
    snvoc:creationDate "%CreationDate%^^xsd:dateTime
  ] .
  %Person2% snvoc:knows [
    snvoc:hasPerson %Person1% ;
    snvoc:creationDate "%CreationDate%^^xsd:dateTime
  ]
  %Person1% snvoc:knows %Person2% .
  %Person2% snvoc:knows %Person1% .
}