FEASIBLE: A Featured-Based SPARQL Benchmark Generation Framework

Muhammad Saleem[1], Qaiser Mehmood[2], and Axel-Cyrille Ngonga Ngomo[1]

[1] Universität Leipzig, IFI/AKSW, PO 100920, D-04009 Leipzig
{lastname}@informatik.uni-leipzig.de
[2] Insight Center for Data Analytics, National University of Ireland, Galway
qaiser.mehmood@insight-centre.org

Abstract. Benchmarking is indispensable when aiming to assess technologies with respect to their suitability for given tasks. While several benchmarks and benchmark generation frameworks have been developed to evaluate triple stores, they mostly provide a one-fits-all solution to the benchmarking problem. This approach to benchmarking is however unsuitable to evaluate the performance of a triple store for a given application with particular requirements. We address this drawback by presenting FEASIBLE, an automatic approach for the generation of benchmarks out of the query history of applications, i.e., query logs. The generation is achieved by selecting prototypical queries of a user-defined size from the input set of queries. We evaluate our approach on two query logs and show that the benchmarks it generates are accurate approximations of the input query logs. Moreover, we compare four different triple stores with benchmarks generated using our approach and show that they behave differently based on the data they contain and the types of queries posed. Our results suggest that FEASIBLE generates better sample queries than the state of the art. In addition, the better query selection and the larger set of query types used lead to triple store rankings which partly differ from the rankings generated by previous works.

1 Introduction

Triple stores are the data backbone of many Linked Data applications [9]. The performance of triple stores is hence of central importance for Linked-Data-based software ranging from real-time applications [8,13] to on-the-fly data integration frameworks [1,15,18]. Several benchmarks (e.g., [2,4,7,9,16,17]) for assessing the performance of the triple stores have been proposed. However, many of them (e.g., [2,4,7,17]) rely on synthetic data or on synthetic queries. The main advantage of such synthetic benchmarks is that they commonly rely on data generators that can produce benchmarks of different data sizes and thus allow to test the scalability of triple stores. However, they often fail to reflect reality. In particular, previous works [5] point out that artificial benchmarks are typically highly structured while real Linked Data sources are most commonly weakly structured. Moreover, synthetic queries most commonly fail to reflect the characteristics of the real queries sent to applications [3,11]. Thus, synthetic benchmark results are rarely sufficient to detect the most suitable triple store for a particular real application. The DBpedia SPARQL Benchmark (DBPSB) [9] addresses a portion of these drawbacks by evaluating the performance of triple stores based on real DBpedia query logs.

The main drawback of this benchmark is however that it does not consider important data-driven and structural query features (e.g., number of join vertices, triple patterns selectivities or query execution times etc.) which greatly affect the performance of triple stores [2,6] during the query selection process. Furthermore, it only considers `SELECT` queries. The other three basic SPARQL query forms, i.e., `ASK`, `CONSTRUCT`, and `DESCRIBE` are not included.

In this paper we present FEASIBLE, a benchmark generation framework able to generate benchmarks from a set of queries (in particular from query logs). Our approach aims to generate customized benchmarks for given use cases or needs of an application. To this end, FEASIBLE assumes that it is given a set of queries well as the number of queries (e.g., 25) to be included into the benchmark as input. Then, our approach computes a sample of the selected subset that reflects the distribution of the queries in the input set of queries. The resulting queries can then be fed to a benchmark execution framework to benchmark triple stores. The contributions of this work are as follows:

1. We present the first structure and data-driven feature-based benchmark generation approach from real queries. By comparing FEASIBLE with DBPSB, we show that considering data-driven and structural query features leads to benchmarks that are better approximations of the input set of queries.
2. We present a novel sampling approach for queries based based on exemplars [10] and medoids.
3. Beside SPARQL `SELECT`, we conduct the first evaluation of 4 triple stores w.r.t. to their performance on `ASK`, `DESCRIBE` and `CONSTRUCT` queries separately.
4. We show that the performance of triple stores varies greatly across the four basic forms of SPARQL query. Moreover, we show that the features used by FEASIBLE allow for a more fine-grained analysis of our benchmarking results.

The rest of this paper is structured as follows: We begin by providing an overview of the key SPARQL query features that need to be considered while designing SPARQL benchmarks. Then, we compare existing benchmarks against these key query features systematically (Section 3) and point out the weaknesses of current benchmarks that are addressed by FEASIBLE. Our benchmark generation process is presented in Section 4. A detailed comparison with DBPSB and an evaluation of the state-of-the-art triple stores follows thereafter. The results are then discussed and we finally conclude. FEASIBLE is open-source and available online at `https://code.google.com/p/feasible/`. A demo can be found at `http://feasible.aksw.org/`.

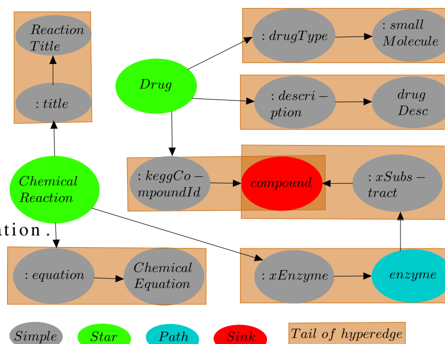
2 Preliminaries

In this section, we define key concepts necessary to understand the subsequent sections of this work. We represent each basic graph pattern (BGP) of a SPARQL query as a directed hypergraph (DH) according to [14]. We chose this representation because it allows representing property-property joins, which previous works [2,6] do not allow to model. The DH representation of a BGP is formally defined as follows:

Definition 1. *Each basic graph patterns BGP_i of a SPARQL query can be represented as a DH $HG_i = (V, E, \lambda_{vt})$, where*

```
SELECT DISTINCT * WHERE
{
?drug :description ?drugDesc.
?drug :drugType :smallMolecule.
?drug :keggCompoundId ?compound.
?enzyme :xSubstrate ?compound.
?Chemicalreaction :xEnzyme ?enzyme.
?Chemicalreaction :equation ?ChemicalEquation.
?Chemicalreaction :title ?ReactionTitle .
}
```

(a) Examplary SPARQL query



(b) Corresponding hypergraph

Fig. 1: DH representation of the SPARQL query. Prefixes are ignored for simplicity

- $V = V_s \cup V_p \cup V_o$ *is the set of vertices of* HG_i*,* V_s *is the set of all subjects in* HG_i*,* V_p *the set of all predicates in* HG_i *and* V_o *the set of all objects in* HG_i*;*
- $E = \{e_1, \ldots, e_t\} \subseteq V^3$ *is a set of directed hyperedges (short: edge). Each edge* $e = (v_s, v_p, v_o)$ *emanates from the triple pattern* $<v_s, v_p, v_o>$ *in* BGP_i*. We represent these edges by connecting the head vertex* v_s *with the tail hypervertex* (v_p, v_o)*. We use* $E_{in}(v) \subseteq E$ *and* $E_{out}(v) \subseteq E$ *to denote the set of incoming and outgoing edges of a vertex* v*;*
- λ_{vt} *is a vertex-type-assignment function. Given a vertex* $v \in V$*, its vertex type can be 'star', 'path', 'hybrid', or 'sink' if this vertex participates in at least one join. A 'star' vertex has more than one outgoing edge and no incoming edge. A 'path' vertex has exactly one incoming and one outgoing edge. A 'hybrid' vertex has either more than one incoming and at least one outgoing edge or more than one outgoing and at least one incoming edge. A 'sink' vertex has more than one incoming edge and no outgoing edge. A vertex that does not participate in any join is of type 'simple'.*

The representation of a complete SPARQL query as a DH is the union of the representations of query's BGPs. As an example, the DH representation of the query in Figure 1a is shown in Figure 1b. Based on the DH representation of SPARQL queries we can define the following features of SPARQL queries:

Definition 2 (Number of Triple Patterns). *From Definition 1, the total number of triple patterns in a* BGP_i *is equal to the number of hyperedges* $|E|$ *in the DH representation of the* BGP_i*.*

Definition 3 (Number of Join Vertices). *Let* $ST = \{st_1, \ldots, st_j\}$ *be the set of vertices of type 'star',* $PT = \{pt_1, \ldots, pt_k\}$ *be the set of vertices of type 'path',* $HB = \{hb_1, \ldots, hb_l\}$ *be the set of vertices of type 'hybrid', and* $SN = \{sn_1, \ldots, sn_m\}$ *be the set of vertices of type 'sink' in a DH representation of a SPARQL query, then the total number of join vertices in the query* $\#JV = |ST| + |PT| + |HB| + |SN|$*.*

Definition 4 (Join Vertex Degree). *Based on the DH representation of SPARQL queries, the join vertex degree of a vertex v is $JVD(v) = |E_{in}(v)| + |E_{out}(v)|$, where $E_{in}(v)$ resp $E_{out}(v)$ is the set of incoming resp. outgoing edges of v.*

Definition 5 (Triple Pattern Selectivity). *Let tp_i be a triple pattern and d be a* relevant *source for tp_i. Furthermore, let N be the total number of triples in d and N_m be the total number of triples in d that matches tp_i, then the selectivity of tp_i w.r.t. d is $Sel(tp_i, d) = N_m/N$.*

According to previous works [2,6], a SPARQL query benchmark should vary the queries it contains w.r.t. the following *query characteristics*: number of triple patterns, number of join vertices, mean join vertex degree, query result set sizes, mean triple pattern selectivities, join vertex types ('star', 'path', 'hybrid', 'sink'), and SPARQL clauses used (e.g., `LIMIT`, `OPTIONAL`, `ORDER BY`, `DISTINCT`, `UNION`, `FILTER`, `REGEX`). In addition, a SPARQL benchmark should contain (or provide options to select) all four SPARQL query forms (i.e., `SELECT`, `DESCRIBE`, `ASK`, and `CONSTRUCT`). Furthermore, the benchmark should contain queries of varying runtimes, ranging from small to reasonably large query execution times. In the next section, we compare state-of-the-art SPARQL benchmarks based on these query features.

3 A Comparison of Existing Benchmarks and Query Logs

Different benchmarks have been proposed to compare triple stores for their query execution capabilities and performance. Table 1 provides a detailed summary of the characteristics of the most commonly used benchmarks as well as of two real query logs. All benchmark executions and result set computations were carried out on a machine with 16 GB RAM and a 6-Core i7 3.40 GHz CPU running Ubuntu 14.04.2. All synthetic benchmarks were configured to generate 10 million triples. We ran LUBM [7] on OWLIM-Lite as it requires reasoning. All other benchmarks were ran on virtuoso 7.2 with NumberOfBuffers = 1360000, and MaxDirtyBuffers = 1000000. As query logs, we used (1) the portion of the DBpedia 3.5.1 query log (a total of 3,159,812 queries) collected between April 30th, 2010 and July 20th, 2010[3] as well as (2) the portion of the Semantic Web Dog Food (SWDF) query log (a total of 1,414,391 queries) gathered between May 16th, 2014 and November 12th, 2014. Note that we only considered queries (called *cleaned queries*) which produce at least 1 result after the query execution (130,466 queries from DBpedia and 64,029 queries from SWDF).[4] In the following, we compare these benchmarks and query logs w.r.t. the features shown in Table 1.

LUBM was designed to test the triple stores and reasoners for their reasoning capabilities. It is based on a customizable and deterministic generator for synthetic data. The queries included in this benchmark commonly lead to query results sizes ranges from 2 to 3200, query triple patterns ranges from 1 to 6, and all the queries consist of a single BGP. LUBM includes a fixed number of `SELECT` queries (i.e., 15) where none of the clauses shown in Table 1 is used.

[3] We chose this query log because it was used by DBPSB.

[4] The datadumps, query logs and cleaned queries for both datasets can be downloaded from project home page

Table 1: Comparison of SPARQL benchmarks and query logs (**F-DBP** = FEASIBLE Benchmarks from DBpedia query log, **DBP** = DBpedia query log, **F-SWDF** = FEASIBLE Benchmark from Semantic Web Dog Food query log, **SWDF** = Semantic Web Dog Food query log, **TPs** = Triple Patterns, **JV** = Join Vertices, **MJVD** = Mean Join Vertices Degree, **MTPS** = Mean Triple Pattern Selectivity, **S.D.** = Standard Deviation). Runtime(ms)

		LUBM	BSBM	SP2Bench	WatDiv	DBPSB	F-DBP	DBP	F-SWDF	SWDF
	#Queries	15	125	12	125	125	125	130466	125	64030
Forms (%)	SELECT	100	80	91.67	100	100	95.2	97.9	92.8	58.7
	ASK	0	0	8.33	0	0	0	1.93	2.4	0.09
	CONSTRUCT	0	4	0	0	0	4	0.09	3.2	0.04
	DESCRIBE	0	16	0	0	0	0.8	0.02	1.6	41.1
Clauses (%)	UNION	0	8	16.67	0	36	40.8	7.97	32.8	29.3
	DISTINCT	0	24	41.6	0	100	52.8	4.1	50.4	34.18
	ORDER BY	0	36	16.6	0	0	28.8	0.3	25.6	10.67
	REGEX	0	0	0	0	4	14.4	0.2	16	0.03
	LIMIT	0	36	8.33	0	0	38.4	0.4	45.6	1.79
	OFFSET	0	4	8.33	0	0	18.4	0.03	20.8	0.14
	OPTIONAL	0	52	25	0	32	30.4	20.1	32	29.5
	FILTER	0	52	58.3	0	48	58.4	93.3	29.6	0.72
	GROUP BY	0	0	0	0	0	0.8	7.6E-6	19.2	1.34
Results	Min	3	0	1	0	197	1	1	1	1
	Max	1.3E+4	31	4.3E+7	4.1E+9	4.6E+6	1.4E+6	1.4E+6	3.0E+5	3.0E+5
	Mean	4.9E+3	8.3	4.5E+6	3.4E+7	3.2E+5	5.2E+4	404	9091	39.5
	S.D.	1.1E+4	9.03	1.3E+7	3.7E+8	9.5E+5	1.9E+5	1.2E+4	4.7E+4	2208
BGPs	Min	1	1	1	1	1	1	0	0	0
	Max	1	5	3	1	9	14	14	14	14
	Mean	1	2.8	1.5	1	2.69	3.17	1.67	2.68	2.28
	S.D.	0	1.70	0.67	0	2.43	3.55	1.66	2.81	2.9
TPs	Min	1	1	1	1	1	1	0	0	0
	Max	6	15	13	12	12	18	18	14	14
	Mean	3	9.32	5.9	5.3	4.5	4.8	1.7	3.2	2.5
	S.D.	1.81	5.17	3.82	2.60	2.79	4.39	1.68	2.76	3.21
JV	Min	0	0	0	0	0	0	0	0	0
	Max	4	6	10	5	3	11	11	3	3
	Mean	1.6	2.88	4.25	1.77	1.21	1.29	0.02	0.52	0.18
	S.D.	1.40	1.80	3.79	0.99	1.12	2.39	0.23	0.65	0.45
MJVD	Min	0	0	0	0	0	0	0	0	0
	Max	5	4.5	9	7	5	11	11	4	5
	Mean	2.02	3.05	2.41	3.62	1.82	1.44	0.04	0.96	0.37
	S.D.	1.29	1.63	2.26	1.40	1.43	2.13	0.33	1.09	0.87
MTPS	Min	3.2E-4	9.4E-8	6.5E-5	0	1.1E-5	2.8E-9	1.2E-5	1.0E-5	1.0E-5
	Max	0.432	0.045	0.53	0.011	1	1	1	1	1
	Mean	0.01	0.01	0.22	0.004	0.119	0.140	0.005	0.291	0.0238
	S.D.	0.074	0.01	0.20	0.002	0.22	0.31	0.03	0.32	0.07
Runtime	Min	2	5	7	3	11	2	1	4	3
	Max	3200	99	7.1E+5	8.8E+8	5.4E+4	3.2E+4	5.6E+4	4.1E+4	4.1E+4
	Mean	437	9.1	2.8E+5	4.4E+8	1.0E+4	2242	30.4	1308	16.1
	S.D.	320	14.5	5.2E+5	2.7E+7	1.7E+4	6961	702.5	5335	249.6

The Berlin SPARQL Benchmark (BSBM) [4] uses a total of 125 query templates to generate any number of SPARQL queries for benchmarking. Multiple use cases such as explore, update, and business intelligence are included in this benchmark. Furthermore, it also includes many of the important SPARQL clauses of Table 1. However, the queries included in this benchmark are rather simple with an average query runtime of 9.1 ms and a largest query result set size of 31.

SP^2Bench mirrors vital characteristics (such as power law distributions or Gaussian curves) of the data in the DBLP bibliographic database. The queries given in benchmark are mostly complex. For example, the mean (across all queries) query result size is above one million and the query runtimes are in the order of 10^5 ms (see Table 1).

The Waterloo SPARQL Diversity Test Suite (WatDiv) [2] addresses the limitations of previous benchmarks by providing a synthetic data and query generator to generate large number of queries from a total of 125 queries templates. The queries cover both simple and complex categories with varying number of features such as result set sizes, total number of query triple patterns, join vertices and mean join vertices degree. However, this benchmark is restricted to conjunctive `SELECT` queries (single BGPs). This means that non-conjunctive SPARQL queries (e.g., queries which make use of the `UNION` and `OPTIONAL` features) are not considered. Furthermore, WatDiv does not consider other important SPARQL clauses, e.g., `FILTER` and `REGEX`. However, our analysis of the query logs of DBpedia3.5.1 and SWDF given in table 1 shows that 20.1% resp. 7.97% of the DBpedia queries make use of `OPTIONAL` resp. `UNION` clauses. Similarly, 29.5% resp. 29.3% of the SWDF queries contain `OPTIONAL` resp. `UNION` clauses.

While the distribution of query features in the benchmarks presented so far is mostly static, the use of different SPARQL clauses and triple pattern join types varies greatly from data set to data set, thus making it very difficult for any synthetic query generator to reflect real queries. For example, the DBpedia and SWDF query log differ significantly in their use of `DESCRIBE` (41.1% for SWDF vs 0.02% for DBpedia), `FILTER` (0.72% for SWDF vs 93.3% for DBpedia) and `UNION` (29.3% for SWDF vs 7.97% for DBpedia) clauses. Similar variations have been reported in [3] as well. To address this issue, the DBpedia SPARQL Benchmark (DBPSB) [9] (which generates benchmark queries from query logs) was proposed. However, this benchmark does not consider key query features (i.e., number of join vertices, mean join vertices degree, mean triple pattern selectivities, the query result size and overall query runtimes) while selecting query templates. Note that previous works [2,6] pointed that these query features greatly affect the triple stores performance and thus should be considered while designing SPARQL benchmarks.

In this work we present FEASIBLE, a benchmark generation framework which is able to generate a customizable benchmark from any set of queries, esp. from query logs. FEASIBLE addresses the drawbacks on previous benchmark generation approaches by taking all of the important SPARQL query features of Table 1 into consideration when generating benchmarks. In the following, we present our approach in detail.

4 FEASIBLE Benchmark Generation

The benchmark generation behind our approach consists of 3 main steps. The first step is the cleaning step. Thereafter, the features of the queries are normalized. In a final

step, we then select a sample of the input queries that reflects the cleaned input queries and return this sample. The sample can be used as seed in template-based benchmark generation approaches such as DBSBM and BSBM.

4.1 Data Set Cleaning

The aim of the data cleaning step is to remove erroneous and zero-result queries from the set of queries used to generate benchmarks. This step is not of theoretical necessity but leads to practically reliable benchmarks. To clean the input data set (here query logs), we begin by excluding all syntactically incorrect queries. The syntactically correct queries which lead to runtime errors[5] as well as queries which return zero results are removed from the set of relevant queries for benchmarking. We attach all 9 SPARQL clauses (e.g., `UNION`, `DISTINCT`) and 7 query features (i.e., runtime, join vertices, etc.) given in Table 1 to each of the queries. For the sake of simplicity we call these 16 (i.e., 9+7) properties *query features* in the following. All unique queries are then stored in a file[6] and given as input to the next step.

4.2 Normalization of Feature Vectors

The query selection process of FEASIBLE requires distances between queries to be computed. To ensure that dimensions with high values (e.g., the result set size) do not bias the selection, we normalize the query representations to ensure that all queries are located in a unit hypercube. To this end, each of the queries gathered from the previous step is mapped to a vector of length 16 which stores the corresponding *query features* as follows: For the SPARQL clauses, which are binary (e.g., `UNION` is either used or not used), we store a value 1 if that clause in used in the query. Otherwise we store a 0. All non-binary feature vectors are normalized by dividing their value with the overall maximal value in the data set. Therewith, we ensure that all entries of the query representations are values between 0 to 1.

4.3 Query Selection

The query selection process is based on the idea of exemplars used in [10] and is shown in Algorithm 1. We assume that we are given (1) a number $e \in \mathbb{N}$ of queries to select as benchmark queries as well as (2) a set of queries L with $|L| = n >> e$, where L is the set of all cleaned and normalized queries. The intuition behind our selection approach is to compute an e-sized partition $\mathcal{L} = \{L_1, \ldots, L_e\}$ of L such that (1) the average distance between the points in two different elements of the partition is high and (2) the average distance of points within a partition is small. We can then select the point closest to the average of each L_i (i.e., the medoid of L_i) to be a prototypical example of a query from L and include it into the benchmark generated by FEASIBLE. We implement this intuition formally by (1) selecting e exemplars (i.e., points that represent a portion of the space) that are as far as possible from each

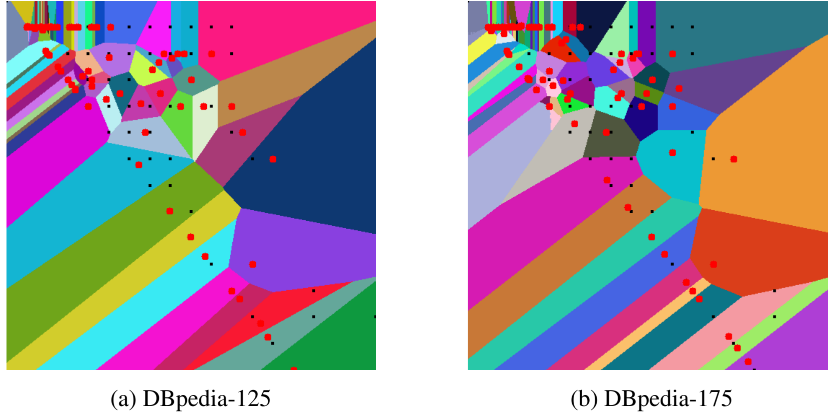
[5] The runtime errors were measured using Virtuoso 7.2.
[6] A sample file can be found at `http://goo.gl/YUSU9A`

Algorithm 1: Query Selection Approach

```
   Data: Set of queries L; Size of the benchmark e
   Result: Benchmark (set of queries) B
1  L̃ = 1/|L| Σ_{q∈L} q ;
2  X_1 = {argmin_{x∈L} d(L̃, x)} ;
3  𝒳 = {X_1} ;
4  for i = 2; i ≤ e; i++ do
5  |   X_i = {argmax_{y∈L\𝒳} d(y, 𝒳)};
6  |   𝒳 = 𝒳 ∪ {X_i};
7  end
8  ℒ = ∅;
9  for i = 1; i ≤ e; i++ do
10 |   L_i = {X_i};
11 |   ℒ = ℒ ∪ {L_i};
12 end
13 for i = 1; i ≤ e; i++ do
14 |   L_i = {q ∈ L\𝒳 : X_i = argmin_{X∈𝒳} d(X, q)}
15 end
16 B = ∅;
17 for i = 1; i ≤ e; i++ do
18 |   L̃_i = 1/|L_i| Σ_{q∈L_i} q;
19 |   b_i = argmin_{q∈L_i} d(L̃_i, q);
20 |   B = B ∪ {b_i};
21 end
22 return B;
```

other, (2) partitioning L by mapping every point of L to one of these exemplars to compute a partition of the space at hand and (3) selecting the medoid of each of the partitions of space as a query in the benchmark. In the following, we present each of these steps formally. For the sake of clarity, we use the following running example: $L = \{q_1 = [0.2, 0.2], q_2 = [0.5, 0.3], q_3 = [0.8, 0.5], q_4 = [0.9, 0.1], q_5 = [0.5, 0.5]\}$ and assume that we need a benchmark with $e = 2$ queries. Note for the sake of simplicity, we used feature vectors of length 2 instead of 16.

Selection of Exemplars We implement an iterative approach to the selection of exemplars (see lines 1-7 of Algorithm 1). We begin by finding the average $\tilde{L} = \frac{1}{n} \sum_{q \in L} q$ of all representations of queries $q \in L$. In our example, this point has the coordinates $[0.58, 0.32]$. The first exemplar X_1 is the point of L that is closest to the average and is given by $X_1 = \arg\min_{x \in L} d(\tilde{L}, x)$, where d stands for the Euclidean distance. In our example, this is the query q_2 with a distance of 0.08. We follow an iterative procedure to extending the set $\mathcal{X}$ of all exemplars: We first find $\eta = \arg\max_{y \in L \setminus \mathcal{X}} \left(\sum_{x \in \mathcal{X}} d(x, y) \right)$. η is the point that is furthest away from all exemplars. In our example, that is the query q_4 with a distance of 0.45 from q_2. We then add η to $\mathcal{X}$ and repeat the procedure for finding η until $|\mathcal{X}| = e$. Given that $e = 2$ in our example, we get the set $\mathcal{X} = \{q_2, q_4\}$ as set of exemplars.



(a) DBpedia-125

(b) DBpedia-175

Fig. 2: Voronoi diagrams for benchmarks generated by FEASIBLE along the two axes with maximal entropy. Each of the red points is a benchmark query. Several points are superposed as the diagram is a projection of a 16-dimensional space unto 2 dimensions.

Selection of Benchmark Queries Let $\mathcal{X} = \{X_1, \ldots, X_e\}$ the set of all exemplars. The selection of benchmark queries begins with partitioning the space according to $\mathcal{X}$. The partition L_i is defined as $L_i = \{q \in L : \forall j \neq i : d(q, X_i) \leq d(q, X_j)\}$ ((see lines 8-15 of Algorithm 1). It is simply the set of queries that are closer to X_i than to any other exemplar. In case of a tie, i.e., $d(q, X_i) = d(q, X_j)$ with $i \neq j$, we assign q to $\min(i, j)$. In our example, we get the following partition: $\mathcal{X} = \{\{q_1, q_2, q_3, q_5\}, \{q_4\}\}$. Finally, we perform the selection of prototypical queries from each partition (see lines 17-22 of Algorithm 1). For each partition L_i we begin by computing the average $\tilde{L_i}$ of all representations of queries in L_i. We then select the query $b_i = \underset{q \in L_i}{\arg\min}\, d(\tilde{L_i}, q)$. The set B of benchmark queries is the set of all queries b_i over all L_i. Note that $|B| = e$. In our example, q_4 being the only query in the second partition means that q_4 is selected as representative for the second partition. The average of the first partition is located at $[0.5, 0.375]$. The query q_2 is the closest to the average, leading to q_2 being selected as representative for the first partition. Our approach thus returns a benchmark with the queries $\{q_2, q_4\}$ as result.

Figures 2a and 2b show Voronoi diagrams of the results of our approach for benchmarks of size 125 and 175 derived from the DBpedia 3.5.1 query log presented in Table 1 along the two dimensions with the highest entropy. Note that some of the queries are superposed in the diagram.

5 Complexity Analysis

In the following, we study the complexity of our benchmark generation approach. We denote the number of features considered during the generation process with d. e is the number of exemplars and $|L|$ the size of the input data set. *Reading and cleaning* the file can be carried out in $O(|L|d)$ as each query is read once and the features are extracted

one at a time. We now need to *compute the exemplars*. We begin by computing the average A of all queries, which can be carried out using $O(|L|d)$ arithmetic operations. Finding the query that is nearest to A has the same complexity. The same approach is used to detect the other exemplars, leading to an overall complexity of $O(e|L|d)$ for the computation of exemplars. *Mapping each point* to the nearest exemplar has an a-priori complexity of $O(e|L|d)$ arithmetic operations. Given that the distances between the exemplars and all the points in L are available from the previous step, we can simply look up the distances and thus gather this information in $O(1)$ for each pair of exemplar and point, leading to an overall complexity of $O(e|L|)$. Finally, *the selection of the representative in the cluster* demands averaging the elements of the cluster and selecting the query that is closest to this point. For each cluster of size $|Cl|$, we need $(d|Cl|)$ arithmetic operations to find the average point. The holds for finding the query nearest to the average. Given that the sum of the sizes of all the clusters is $|L|$, we can conclude that the overall complexity of the selection step is $O(d|L|)$. Overall, the worst-case complexity of our algorithm is thus $O(d|L||E|)$.

In the best case, no queries passes the cleaning test, leading to no further processing and to the same complexity as reading the data, which is $O(|L|d)$. The same best-case complexity holds when a benchmark is generated. Here, the filtering step returns exactly e queries, leading to the exemplar generation step being skipped and thus to a complexity of $O(|L|d)$.

6 Evaluation and Results

Our evaluation comprises two main parts. First, we compare FEASIBLE with DBPSB w.r.t. how well the benchmarks represent the input data. To this end, we use the composite error function defined below. In the second part of our evaluation, we use FEASIBLE benchmarks to compare triple stores w.r.t. their query execution performance.

6.1 Composite Error Estimation

The benchmarks we generate aim to find typical queries for a given query log. From the point of view of statistics, this is equivalent to computing a subset of a population that has the same characteristics (here mean and standard deviation) as the original population. Thus, we measure the quality of the sampling approach of a benchmark by how much the mean and standard deviation of the features of its queries deviates from that of the query log. We call μ_i resp. σ_i the mean resp. the standard deviation of a given distribution w.r.t. to the ith feature of the said distribution. Let B be a benchmark extracted from a set of queries L. We use two measures to compute the difference between B and L, i.e., the error on the means E_μ and deviations E_σ

$$E_\mu = \frac{1}{k}\sum_{j=1}^{k}(\mu_i(L) - \mu_i(B))^2 \text{ and } E_\sigma = \frac{1}{k}\sum_{j=1}^{k}(\sigma_i(L) - \sigma_i(B))^2. \quad (1)$$

We define a composite error estimation E as the harmonic mean of E_μ and E_σ:

$$E = \frac{2E_\mu E_\sigma}{E_\mu + E_\sigma}. \quad (2)$$

6.2 Experimental Setup

Data sets and Query Logs: We used the DBpedia 3.5.1 (232.5M triples) and SWDF (294.8K triples) data sets for triple store evaluation. As queries (see Section 3), we used 130,466 cleaned queries for DBpedia and 64,029 cleaned queries for SWDF.

Benchmarks for Composite Error Analysis: In order to compare FEASIBLE with DBPSB, we generated benchmarks of sizes 15, 25, 50, 75, 100, 125, 150, and 175 queries from the DBpedia 3.5.1 query log. Recall this is exactly the same query log used in DBPSB. DBPSB contains a total of 25 query templates derived from 25 real queries. A single query was generated per query template in order to generate a benchmark of 25 queries. Similarly, 2 queries were generated per query template for a benchmark of 50 queries and so on. The 15-query benchmark of DBPSB was generated from the 25-query benchmark by randomly choosing 15 of the 25 queries. We chose to show results on a 15-query benchmark because LUBM contains 15 queries while SP^2Bench contains 12. We also generated benchmarks of the same size (15-175) from SWDF to compare FEASIBLE's composite errors as well as the performance of triple stores across different data sets.

Triple Stores: We used four triple stores in our evaluation: (1) *Virtuoso Open-Source Edition version 7.2* with NumberOfBuffers = 680000, MaxDirtyBuffers = 500000; (2) *Sesame Version 2.7.8* with Tomcat 7 as HTTP interface and native storage layout. We set the spoc, posc, opsc indices to those specified in the native storage configuration. The Java heap size was set to 6GB; (3) *Jena-TDB (Fuseki) Version 2.0* with a Java heap size set to 6GB and (4) *OWLIM-SE Version 6.1* with Tomcat 7.0 as HTTP interface. We set the entity index size to 45,000,000 and enabled the predicate list. The rule set was empty and the Java heap size was set to 6GB. Ergo, we configured all triple stores to use 6GB of memory and used default values otherwise.

Benchmarks: Most of the previous evaluations were carried out on `SELECT` queries only (see Table 1). Here, beside evaluating the performance of triples stores on `SELECT` evaluation, we also wanted to compare triple stores on the other three forms of SPARQL queries. To this end, we generated DBpedia-ASK-100 (100-ASK-query benchmark derived from DBpedia) and SWDF-ASK-50 (50-ASK-query benchmark derived from SWDF)[7] and compared the selected triple stores for their `ASK` query processing performances. Similarly, we generated DBpedia-CONSTRUCT-100 and SWDF-CONSTRUCT-23, DBpedia-DESCRIBE-25 and SWDF-DESCRIBE-100, and DBpedia-SELECT-100 and SWDF-SELECT-100 benchmarks to test the selected systems for `CONSTRUCT`, `DESCRIBE`, and `SELECT` queries, respectively. Furthermore, we generated DBpedia-Mix-175 (DBpedia benchmark of 175 mix queries of all the four query forms) and SWDF-Mix-175 to test the selected triple stores for their general query processing performance.

[7] We chose to select only 50 queries because the SWDF log we used does not contain enough `ASK` queries to generate a 100-query benchmark.

Benchmark Execution: The evaluation was carried out one triple store at a time on one machine. First, all data sets were loaded into the selected triple store. Once the triple store had completed the data loading, the 2-phase benchmark execution phase began: (1) *Warm-up Phase*: To measure the performance of the triple store under normal operational conditions, a warm-up phase was used where random queries from the query log were posed to triple stores for 10 minutes; (2) *Hot-run Phase:* During this phase, the benchmark query mixes were sent to the tested store. We kept track of the average execution time of each query as well as of the number of query mixes per hour (QMpH). This phase lasted for two hours for each triple store. Note that the benchmark and the triple store were run on the same machine to avoid network latency. We set the query timeout to 180 seconds. The query was aborted after that and maximum time of 180 seconds was used as the query runtime for all queries which timed out. All the data (data dumps, benchmarks, query logs, FEASIBLE code) to repeat our experiments along with complete evaluation results are available at the project website.

6.3 Experimental Results

Composite Error Table 2 shows a comparison of the composite errors of DBPSB and FEASIBLE for different benchmarks. Note that DBPSB queries templates are only available for the DBpedia query log. Thus, we were not able to calculate DBPSB's composite errors for SWDF. As an overall composite error evaluation, FEASIBLE's composite error is 54.9% smaller than DBPSB. The reason for DBPSB's error being higher that FEASIBLE's lies in the fact that it only considers the number of query triple patterns and the SPARQL clauses `UNION`, `OPTIONAL`, `FILTER`, `LANG`, `REGEX`, `STR`, and `DISTINCT` as features. Important query features (such as query result sizes, execution times, triple patterns and join selectivities, and number of join vertices) were not considered when generating the 25 query templates.[8] Furthermore, DBPSB only includes `SELECT` queries. The other three SPARQL query forms, i.e., `CONSTRUCT`, `ASK`, and `DESCRIBE` are not considered. In contrast, our approach considers all of the query forms, SPARQL clauses, and query features reported in Table 1.[9] It is important to mention that FEASIBLE's overall composite error across both data sets is only 0.038.

Triple Store Performance Figure 3 shows a comparison of the selected triple stores in terms of *queries per second* (QpS) and *query mixes per hour* (QMpH) for different benchmarks generated by FEASIBLE. Table 3 shows the overall rank-wise query distributions of the triple stores. Our ranking is partly different from the DBPSB ranking. Overall, (for mix DBpedia and SWDF benchmarks of 175 queries each, Figure 3e to Figure 3g), Virtuoso ranks first followed by Fuseki, OWLIM-SE, and Sesame. Virtuoso is 59% faster than Fuseki. Fuseki is 1.7% faster than OWLIM-SE, which in turn 16% faster than Sesame.[10]

[8] Queries templates available at: `http://goo.gl/1oZCZY`

[9] See FEASIBLE online demo for the customization of these features

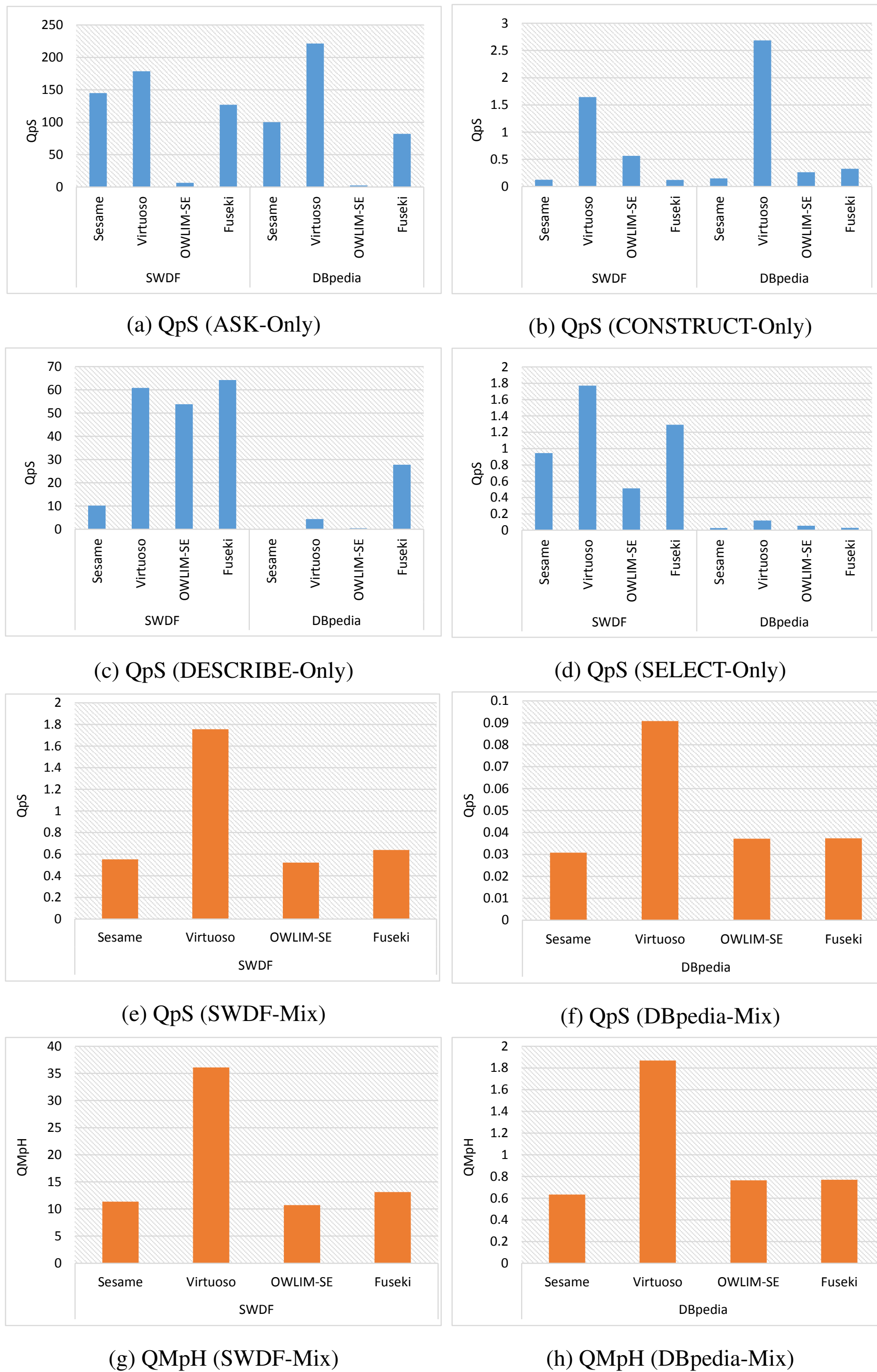
[10] Note the percentage improvements are calculated from the QMpH values as A is (1-QMpH(A)/QMpH(B)*100) percent faster than B.

Table 2: Comparison of the Mean E_μ , Standard Deviation E_σ and Composite E errors for different benchmark sizes of DBpedia and Semantic Web Dog Food query logs. FEASIBLE outperforms DBPSB across all dimensions.

Benchmark	FEASIBLE			DBPSB			Benchmark	FEASIBLE		
	E_μ	E_σ	E	E_μ	E_σ	E		E_μ	E_σ	E
DBpedia-15	0.045	0.054	0.049	0.139	0.192	0.161	SWDF-15	0.019	0.043	0.026
DBpedia-25	0.041	0.054	0.046	0.113	0.139	0.125	SWDF-25	0.034	0.051	0.041
DBpedia-50	0.045	0.056	0.050	0.118	0.132	0.125	SWDF-50	0.036	0.052	0.043
DDBpedia-75	0.053	0.061	0.057	0.096	0.095	0.096	SWDF-75	0.035	0.051	0.042
DDBpedia-100	0.054	0.064	0.059	0.130	0.132	0.131	SWDF-100	0.036	0.050	0.042
DDBpedia-125	0.054	0.064	0.058	0.088	0.082	0.085	SWDF-125	0.034	0.048	0.040
DBpedia-150	0.055	0.064	0.059	0.107	0.124	0.115	SWDF-150	0.033	0.046	0.038
DBpedia-175	0.055	0.065	0.059	0.127	0.144	0.135	SWDF-175	0.033	0.045	0.038
Average	**0.050**	**0.060**	**0.055**	0.115	0.130	0.121	Average	0.032	0.048	0.039

A more fine-grained look at the evaluation reveals surprising findings: On `ASK` queries, Virtuoso is clearly faster than the other frameworks (45% faster than Sesame, which is 16% faster than Fuseki, which is in turn 96% faster than OWLIM-SE, see Figure 3a). The ranking changes for `CONSTRUCT` queries: While Virtuoso is still first (87% faster than OWLIM-SE), OWLIM-SE is now faster that 14% faster than Fuseki, which in turn is 42% faster than Sesame (Figure 3b). The most drastic change occurs on the `DESCRIBE` benchmark, where Fuseki ranks first (66% faster than Virtuoso, which is 86% faster than OWLIM-SE, which in turns 47% faster than Sesame, see Figure 3c). Yet another ranking emerges from the `SELECT` benchmarks, where Virtuoso is overall 55% faster than OWLIM-SE, which is 41% faster than Fuseki, which in turns 11% faster than Sesame (Figure 3d). These results show that the performance of triple stores varies greatly across the four basic SPARQL forms and none of the system is the sole winner across all query forms. Moreover, the ranking also varies across the different datasets (see, e.g., `ASK` benchmark for DBpedia and SWDF). Thus, our results suggest that (1) a benchmark should comprise a mix of SPARQL `ASK`, `CONSTRUCT`, `DESCRIBE`, and `SELECT` queries that reflects the real intended usage of the triple stores to generate accurate results and (2) there is no universal winner amongst triple stores, which points again towards the need to create customized benchmarks for applications when choosing their backend. FEASIBLE addresses both of these requirements by allowing users to generate dedicated benchmarks from their query logs.

Some interesting observations were revealed by the rank-wise queries distributions of triple stores shown in Table 3: First, none of the system is sole winner or loser for a particular rank. Overall, Virtuoso's performance mostly lies in the higher ranks, i.e., rank 1 and 2 (68.29%). This triple store performs especially well on `CONSTRUCT` queries. Fuseki's performance is mostly in the middle ranks, i.e., rank 2 and 3 (65.14%). In general, it is faster for `DESCRIBE` queries and is on a slower side for `CONSTRUCT` and queries containing `FILTER` and `ORDER BY` clauses. While OWLIM-SE's performance is usually on the slower side, i.e., rank 3 and 4 (60.86 %), it performs well on complex queries with large result set sizes and complex SPARQL clauses. Finally, Sesame is



(a) QpS (ASK-Only)

(b) QpS (CONSTRUCT-Only)

(c) QpS (DESCRIBE-Only)

(d) QpS (SELECT-Only)

(e) QpS (SWDF-Mix)

(f) QpS (DBpedia-Mix)

(g) QMpH (SWDF-Mix)

(h) QMpH (DBpedia-Mix)

Fig. 3: Comparison of the triple stores in terms of Queries per Second (QpS) and Query Mix per Hour (QMpH), where a Query Mix comprise of 175 distinct queries.

Table 3: Overall rank-wise ranking of triple stores. All values are in percentages.

	SWDF				DBpedia				Overall			
Triple Store	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th
Virtuoso	38.29	24.57	21.71	15.43	54.86	18.86	15.43	10.86	46.57	21.71	18.57	13.14
Fuseki	17.14	39.43	32.00	11.43	24.00	34.86	24.00	17.14	20.57	37.14	28.00	14.29
OWLIM-SE	10.29	30.29	21.14	38.29	13.14	24.57	25.14	37.14	11.71	27.43	23.14	37.71
Sesame	37.71	12.00	29.14	21.14	25.71	16.57	32.57	25.14	31.71	14.29	30.86	23.14

either fast or slow. For example, for 31.71% of the queries, it achieve the rank 1 (second best after Virtuoso) and but achieves rank 4 on 23.14% of the queries (second worse after OWLIM-SE). In general Sesame is very efficient on simple queries with small result set sizes, a small number of triple triple patterns, and a few SPARQL clauses. However, it performs poorly as soon as the queries grow in complexity. These results shows yet another aspect of the importance of taking structural and data-driven features into consideration while generating benchmarks as they allow deeper insights into the type of queries on which systems perform well or poorly.

Finally, we also looked into the number of query timeouts during the complete evaluation. Most of the systems time out for `SELECT` queries. Overall, Sesame has the highest number of timeouts (43) followed by Fuseki (32), OWLIM-SE (22), and Virtuoso (14). For Virtuoso, the timeout queries have at least one triple pattern with an unbound subject, an unbound predicate and an unbound object (i.e., a triple pattern of the form `?s ?p ?o`). The corresponding result sets were so large that they could not be computed in 3 minutes. The other three systems mostly timeout for the same queries. OWLIM-SE generally performs better for complex queries with large result set sizes. Fuseki has problems with queries containing `FILTER` (12/32) and `ORDER BY` clauses (11/32 queries). Sesame's performance is slightly worse for complex queries containing many triple patterns and joins as well as complex SPARQL clauses. Note that Sesame also times out for 8 `CONSTRUCT` queries. All the timeout queries for each triple store are provided at the project website.

7 Conclusion

In this paper we presented FEASIBLE, a customizable SPARQL benchmark generation framework. We compared FEASIBLE with DBPSB and showed that our approach is able to produce high-quality (in terms of their composite error) benchmarks. In addition, our framework allows users to generate customized benchmarks suited for a particular use case, which is of utmost importance when aiming to gather valid insights into the real performance of different triple stores for a given application. This is demonstrated by our triple store evaluation, which shows that the ranking of triple stores varies greatly across different types of queries as well as across datasets. Our results thus suggest that all of the four query forms should be included in the future SPARQL benchmarks. For the sake of future work, we have started converting linked data query logs into RDF and made available through the LSQ [12] endpoint. Beside the key queries characteristics discussed in Table 1, we have attached many of the SPARQL 1.1 features to each of the

query. We will extend FEASIBLE to query the LSQ SPARQL endpoint directly so as to gather queries for the benchmark creation process.

Acknowledgements

This work was partially supported by projects GeoKnow (GA: 318159) and SAKE (Grant No. 01MD15006E).

References

1. Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. Anapsid: An adaptive query processing engine for sparql endpoints. In *ISWC*, 2011.
2. Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of rdf data management systems. In *ISWC*. 2014.
3. Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world SPARQL queries. *CoRR*, 2011.
4. Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *IJSWIS*, 2009.
5. Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: A comparison of rdf benchmarks and real rdf datasets. In *SIGMOD*, 2011.
6. Olaf Görlitz, Matthias Thimm, and Steffen Staab. Splodge: Systematic generation of sparql benchmark queries for linked open data. In *ISWC*. 2012.
7. Yuanbo Guo and Jeff Heflin. LUBM: A benchmark for owl knowledge base systems. *JWS*, 2005.
8. M Kamdar, Aftab Iqbal, Muhammad Saleem, H Deus, and Stefan Decker. Genomesnip: Fragmenting the genomic wheel to augment discovery in cancer research. In *CSHALS*, 2014.
9. Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Dbpedia sparql benchmark - performance assessment with real queries on real data. In *ISWC*, 2011.
10. Axel-Cyrille Ngonga Ngomo and Sören Auer. LIMES - A time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*, 2011.
11. Francois Picalausa and Stijn Vansummeren. What are real sparql queries like? In *SWIM*, 2011.
12. Muhammad Saleem, Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: The linked sparql queries dataset. In *ISWC*, 2015.
13. Muhammad Saleem, Maulik R Kamdar, Aftab Iqbal, Shanmukha Sampath, Helena F Deus, and Axel-Cyrille Ngonga Ngomo. Big linked cancer data: Integrating linked tcga and pubmed. *JWS*, 2014.
14. Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. HiBISCuS: Hypergraph-based source selection for sparql endpoint federation. In *ESWC*, 2014.
15. Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, Josiane Xavier Parreira, HelenaF. Deus, and Manfred Hauswirth. DAW: Duplicate-aware federated query processing over the web of data. In *ISWC*, 2013.
16. Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *ISWC*, 2011.
17. Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. Sp2bench: a sparql performance benchmark. In *ICDE*, 2009.
18. Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *ISWC*, 2011.