Congenial Benchmarking of RDF Storage Solutions

Axel-Cyrille Ngonga Ngomo
axel.ngonga@upb.de
Data Science Group, Paderborn University
Paderborn, Germany

Felix Conrads
felix.conrads@upb.de
Data Science Group, Paderborn University
Paderborn, Germany

Maximilian Pensel
maximilian.pensel@tu-dresden.de
Chair for Automata Theory, TU Dresden
Dresden, Germany

Anni-Yasmin Turhan
anni-yasmin.turhan@tu-dresden.de
Chair for Automata Theory, TU Dresden
Dresden, Germany

ABSTRACT

Many SPARQL benchmark generation techniques rely on SPARQL query templates or on selecting representative queries from a set of input queries by inspecting their syntactic features. Hence, prototype queries from such benchmarks mainly capture combinations of SPARQL features, but not the semantics nor the conceptual association between queries. We present *congenial benchmarks*—a novel type of benchmark that can detect conceptual associations and thus reflect prototypical user intentions when selecting prototype queries. We study Sparrow, an instantiation of congenial benchmarks, where the conceptual associations of SPARQL queries are measured by concept similarity measures. To this end, we transform unary acyclic conjunctive SPARQL queries into $\mathcal{ELH}$-description logic concepts. Our evaluation of three popular triple stores on two datasets shows that the benchmarks generated by Sparrow differ considerably from benchmarks generated using a feature-based approach. Moreover, our evaluation suggests that Sparrow can characterize the performance of common triple stores with respect to user needs by exploiting conceptual associations to detect prototypical user needs.

CCS CONCEPTS

• **Information systems → World Wide Web**.

KEYWORDS

Information systems Web data description languages

ACM Reference Format:
Axel-Cyrille Ngonga Ngomo, Felix Conrads, Maximilian Pensel, and Anni-Yasmin Turhan. 2018. Congenial Benchmarking of RDF Storage Solutions. In *10th International Conference on Knowledge Capture (K-CAP'19), November 19–21, 2019, Marina Del Rey, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

K-CAP '19, November 19–21, 2019, Marina Del Rey, CA, USA

ACM ISBN 978-1-4503-7008-0/19/11…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Performance evaluations of triple stores based on realistic benchmarks are vital to assess and constantly improve such systems [8, 16]. In recent years, a large collection of benchmarks and benchmark generation techniques has been developed (see [17] for an overview). Two families of benchmarks for triple stores dominate this collection. Most of the early benchmarks for triple stores are *synthetic benchmarks* (e.g., [1, 8]), which commonly generate a synthetic dataset upon which a set of predefined query templates is instantiated. However, works on the structure of RDF data point out a high discrepancy between synthetic and real RDF data [6, 13]. Hence, an increasing number of benchmarks belong to the second family of benchmarks, which rely on *real datasets and real queries* [12, 18, 19]. Most of these benchmark generation techniques analyze a set of queries in regard of which (syntactic) SPARQL features they use [16]. The queries are then clustered according to the results of the feature analysis. A prototype query is selected from each cluster. The prototype queries then constitute the resulting feature-based benchmark.

A performance evaluation based on such a feature-based benchmark can only supply insights on the relationship between syntactical features and performance. While this can be sufficient to tailor the performance of triple stores for many applications, this is certainly not meaningful for all. In many SPARQL-based applications (e.g., search and question answering [21], automatic exploration [9], machine learning [2]), the SPARQL queries are relaxed to avoid empty result sets, or strengthened to prune large result sets. This alteration of the query may be done either manually by a user, by a query editing system [20] or even automatically in structured machine learning applications [2]. The corresponding query logs reveal that queries often address similar "targets" reflecting the user's intent for posing the query [15]. Essentially, there is a strong conceptual association between the queries for this kind of application which in turn gives rise to clusters of such congenial queries.

Traditional SPARQL feature-based benchmarks are oblivious to such kind of conceptual associations between queries and do not allow the performance of the query engine to be tuned to this kind of query collection. More precisely, the overlap of features used in queries does not indicate whether these queries retrieve things alike or are serving similar or even the same user intents. Consequently, the analysis of features and the resulting clustering yields benchmarks that simply do not reflect how well existing triple stores can serve clusters of queries obtained from prototypical user

intents. Instead, they evaluate how well triple stores implement particular SPARQL features or combinations thereof. Our aim is to *provide benchmarks that challenge triple stores in the same way as do applications producing a proliferation of similar queries due to the user intent*. We call such benchmarks *congenial benchmarks*. To produce such benchmarks, we analyze real input query logs with regard to the semantic similarity between queries and then perform the clustering accordingly.

The central task for generating congenial benchmarks is to measure the similarity between the queries. This can be done by applying similarity measures, which are functions that compute a value from the unit interval, where 1 indicates high similarity (even equivalence) and 0 indicates total dissimilarity. The similarity measure that we want to employ for our purpose needs to **a)** regard the semantics of the query and **b)** abstract from its syntactic form as we want to cluster those queries that formalize a similar intent. Furthermore, our similarity measure needs to **c)** be robust in the sense that it has (formal) properties that ensure "wanted" behavior, such as symmetry or equivalence invariance. While there are many similarity measures available for concepts written in ontology languages, there are hardly any for conjunctive or SPARQL queries [5]. An approach to obtain a measure with the desired properties is the simi framework [11], which generates concept similarity measures (CSM) for concepts written in the description logic $\mathcal{ELH}$. The application of the CMS to the queries then yields clusters from which we select prototype queries that constitute the cases in the congenial benchmark.

In this paper, we examine the approach of congenial benchmarks and study an instantiation of it with SPARQL queries from query logs generated by applications for RDF data series. The simi framework yields CSMs that are only applicable to a subclass of SPARQL queries, namely those that correspond to $\mathcal{ELH}$ concepts, i.e., tree-shaped queries with only one answer variable. Despite this, we adopt simi since the resulting measures abstract from the syntactic form and consider the semantics of the queries. Furthermore, there is an implementation of the simi framework available. To use as many queries from the logs as possible, we devise a translation function from SPARQL to $\mathcal{ELH}$ concepts and carry out a generation of congenial benchmarks in two variants: first for queries from the fragment of $\mathcal{ELH}$ concepts that can be translated without loss and, second, for those queries that can be approximated by an $\mathcal{ELH}$ concept with some loss of information. We use a clustering approach to detect prototype queries from a truncated query similarity graph generated using the simi framework. The resulting set of queries yields a congenial benchmark which we then use to benchmark triple stores.

Interestingly, despite the limitation to $\mathcal{ELH}$ concepts during the generation of the benchmark, our evaluation on the original queries shows that triple stores exhibit different behavior on congenial benchmarks and on feature-based benchmarks. In particular, while the mix of SPARQL query features is not significantly different across the benchmarks, the clusters and prototype queries generated by the two benchmarks are clearly different (see Figure 1). Moreover, the distributions of the query runtimes suggest that the ability of triple stores to perform well on feature-based benchmarks measured in previous works is not a good predictor for their performance on congenial benchmarks (see Figure 2). This insight is supported by the difference in ranking of triple stores obtained with the two categories of benchmarks (see Figure 3).

The paper is structured as follows: Next, we introduce $\mathcal{ELH}$, the simi framework, and our CSM *BM-simi*. Section 3 describes our translations from SPARQL queries to $\mathcal{ELH}$ concepts. The process of benchmark generation is detailed in Section 4. Section 5 reports on experiments and their results. The paper ends with a brief discussion.

2 PRELIMINARIES

We briefly sketch the DL $\mathcal{ELH}$ and introduce the simi framework for generating CSMs from which we derived the particular CSM used in our experiments.[1]

2.1 The description logic $\mathcal{ELH}$

In DLs, ontologies are built from *concept names*, *role names* and *individual names*. Let $\mathsf{N_C}$ be a set of concept names, $\mathsf{N_R}$ a set of role names, and $\mathsf{N_I}$ a set of individual names. In $\mathcal{ELH}$, complex concepts are built according to the concept constructors shown in the upper part of Table 1. The semantics of $\mathcal{ELH}$ concepts are given by interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is the *interpretation domain* and $\cdot^{\mathcal{I}}$ is the *interpretation mapping* that assigns to each $A \in \mathsf{N_C}$ a subset of $\Delta^{\mathcal{I}}$, to each $r \in \mathsf{N_R}$ a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and to each $a \in \mathsf{N_I}$ an element from $\Delta^{\mathcal{I}}$. The interpretation of complex concepts is given according to the semantics listed in Table 1.

Table 1: Syntax and semantics of $\mathcal{ELH}$.

Name	Syntax	Semantics
atomic concept	$A \ (A \in \mathsf{N_C})$	$A^{\mathcal{I}}$
top concept	$\top$	$\Delta^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
existential restriction	$\exists r.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y \in C^{\mathcal{I}} \wedge \langle x, y\rangle \in r^{\mathcal{I}}\}$
concept definition	$A = C \ (A \in \mathsf{N_C})$	$A^{\mathcal{I}} = C^{\mathcal{I}}$
role inclusion	$r \sqsubseteq s \ (r, s \in \mathsf{N_R})$	$r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$
concept assertion	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
role assertion	$r(a, b)$	$\langle a^{\mathcal{I}}, b^{\mathcal{I}}\rangle \in r^{\mathcal{I}}$

A complex $\mathcal{ELH}$ concept can be assigned a concept name via a *concept definition*, as displayed in the lower part of Table 1. If the ontology contains only concept definitions that are acyclic, i.e., no concept name on the left-hand side of a definition refers to itself (indirectly), then the ontology is called *acyclic*. In such a case, the concept definitions can be used like macros and defined concepts can simply be expanded to exemplify the TBox information. In $\mathcal{ELH}$, roles can be assigned super-roles by *role inclusion* axioms. A *TBox* $\mathcal{T}$ is a set of concept definitions and role inclusion axioms. An interpretation $\mathcal{I}$ is a *model* of a TBox $\mathcal{T}$, if it fulfills the semantics of all concept definitions and role inclusion axioms from $\mathcal{T}$ (displayed in Table 1). Please note that $\mathcal{ELH}$-concepts (together with an acyclic TBox) can only specify graph structures that are trees.

[1] We refer the reader to the specifications of SPARQL at https://www.w3.org/TR/sparql11-query/ and RDF at https://www.w3.org/RDF/.

Let $a, b \in \mathsf{N_I}$. Then, assertions can either be *concept assertions* or *role assertions* as shown in Table 1. A set of assertions is called an *ABox*. An interpretation $\mathcal{I}$ is a *model* of an ABox $\mathcal{A}$, if it fulfills the semantics of all assertions from $\mathcal{A}$ (displayed in Table 1).

Two prominent inferences for DLs are testing subsumption and answering concept queries. Given two concepts C and D, *subsumption* (denoted $C \sqsubseteq_{\mathcal{T}} D$) decides whether C is a sub-concept of D w.r.t. $\mathcal{T}$. Formally, $C \sqsubseteq_{\mathcal{T}} D$ holds iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds in all models of $\mathcal{T}$. *Equivalence* of concepts w.r.t. $\mathcal{T}$ (denoted $C \equiv_{\mathcal{T}} D$), holds if $C \sqsubseteq_{\mathcal{T}} D$ and $D \sqsubseteq_{\mathcal{T}} C$ hold. Given a concept C, *answering the concept query* C, is to compute the set of all individuals from $\mathcal{A}$ that are instances of C w.r.t. $\mathcal{T}$. Formally, $C(a)$ holds w.r.t. $\langle \mathcal{T}, \mathcal{A} \rangle$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$ holds for all models $\mathcal{I}$ of $\mathcal{T}$ and $\mathcal{A}$. $\mathcal{ELH}$ concept queries correspond to SPARQL queries that are tree-shaped conjunctive queries with one projection variable. We concentrate on this kind of queries, since we want to employ "well-behaved" CSMs for congenial benchmarks—as supplied by the simi framework.

2.2 The simi framework and our CSM *BM-simi*

For our proof of concept of congenial benchmarks, we need an assessment of similarity that regards the semantics rather than the syntax of the queries. The simi framework [11] generates CSMs that realize this property. By various parameters, this framework allows tailored CSMs to be built and guarantees formal properties for the resulting CSMs. If used w.r.t. an acyclic $\mathcal{ELH}$-TBox $\mathcal{T}$, a CSM *simi*() generated by the framework has (among others) the following properties:

- *symmetric,* iff $simi(C, D) = simi(D, C)$;
- *equivalence invariant,* iff for all $C \equiv_{\mathcal{T}} D$ and all concepts E it holds that
  $simi(C, E) = simi(D, E)$;
- *equivalence closed,* iff $C \equiv_{\mathcal{T}} D \iff simi(C, D) = 1$;

For the conceptual association, the important property is equivalence invariance, which is achieved by preprocessing of the concepts (see [11] for details). The resulting concepts are conjunctions and can be represented as a set of conjuncts (denoted $\widehat{C}$ for a concept C). Symmetry is achieved by a *directed CSM* $simi_d$, a fuzzy connector $\otimes$ and defining:

$$simi(C, D) := simi_d(C, D) \otimes simi_d(D, C). \tag{1}$$

In our measure *BM-simi*, the values obtained from *BM-*$simi_d$ are combined by the average as the fuzzy connector $\otimes$.

The directed CSM $simi_d$ works analogously to the Jaccard index, which is a similarity measure for sets [10], but with the important difference that $simi_d$ needs to treat complex syntactic sub-concepts appearing in existential restrictions recursively. Thus, $simi_d$ follows the inductive definition of $\mathcal{ELH}$-concepts, where the base case are pairs of named concepts from $\mathsf{N_C}$ (and of roles from $\mathsf{N_R}$). These pairs are assessed by a *primitive measure*, which is a function $pm : N_C^2 \cup N_r^2 \longrightarrow [0, 1]$ where for all $A, B \in N_C$ and $r, s, t \in N_r$ the following holds:

- $pm(A, B) = 1 \iff A = B$
- $pm(r, s) = 1 \iff s \sqsubseteq_{\mathcal{T}} r$
- $s \sqsubseteq_{\mathcal{T}} r \implies pm(s, r) > 0$, and
- $t \sqsubseteq_{\mathcal{T}} s \implies pm(r, s) \leq pm(r, t)$.

Our instantiation *BM-*$simi_d$ uses the "default primitive measure" from [11] that yields 1 for syntactically equal (concept or role) names, 0.01 for roles r, s with $r \sqsubseteq_{\mathcal{T}} s$ and $s \not\sqsubseteq_{\mathcal{T}} r$, and 0 otherwise.

Another parameter of $simi_d$ is the *weighting function* (denoted $g()$) for elements from $\widehat{C}$. It allows some concept names or existential restrictions to contribute more to the similarity value and can highlight thematic sub-domains of the ontology. As we don't want such imbalance, we take a uniform weighting function $g()$ for *BM-*$simi_d$.

To combine similarity values obtained from the recursive calls, $simi_d$ uses a weight $w \in (0, 1)$ that sets for existential restrictions, say $\exists r.C$, the ratio of how much the role r versus the concept C contributes to the combined similarity value. *BM-*$simi_d$ employs $w = 0.8$ which lets the existence of a related object contribute more to the similarity value than the class of that object. This weight essentially also dampens the contribution of concepts nested existential restrictions according to their nesting depth.

Let E, F be $\mathcal{ELH}$ concepts, C, D be $\mathcal{ELH}$ concepts distinct from $\top$, $A, B \in \mathsf{N_C}$ and $r, s \in \mathsf{N_R}$. Then the directed measure $simi_d$ is a function that maps pairs of $\mathcal{ELH}$ concepts to $[0, 1]$ and can be parameterized by a bounded t-conorm $\oplus$, a primitive measure pm, a weighting function g and $w \in (0, 1)$. The definition of simi_d is

$$\begin{aligned} simi_d(\top, \top) &:= simi_d(\top, D) := 1, \\ simi_d(C, \top) &:= 0, \\ simi_d(C, D) &:= \frac{\sum_{C' \in \widehat{C}} \left[g(C') \cdot \bigoplus_{D' \in \widehat{D}} simi_a(C', D') \right]}{\sum_{C' \in \widehat{C}} g(C')}, \end{aligned}$$

where $simi_a$ measures the similarity of $C' \in \widehat{C}$ to $D' \in \widehat{D}$ and is defined as

$$\begin{aligned} simi_a(A, B) &:= pm(A, B), \\ simi_a(\exists r.E, A) &:= simi_a(A, \exists r.E) := 0, \\ simi_a(\exists r.E, \exists s.F) &:= pm(r, s) \cdot [w + (1 - w) simi_d(E, F)]. \end{aligned}$$

Our CSM *BM-simi* is an instantiation of the simi framework that uses average for $\otimes$, probabilistic sum for $\oplus$, the default primitive measure pm, a uniform weighting function g, and $w = 0.8$. *BM-simi* fulfills the formal properties listed above. We have obtained a prototype implementation of the simi framework from the authors of [11] that we use to generate and compute *BM-simi*. For the equivalence tests of concepts and roles, we use the HermiT reasoner [7]. Before applying *BM-simi* to SPARQL queries, a conversion of these queries to $\mathcal{ELH}$ concept queries is necessary.

3 CONVERTING SPARQL TO $\mathcal{ELH}$ CONCEPT QUERIES

Only a subset of all SPARQL queries are also $\mathcal{ELH}$ concept queries. We present two approaches to converting SPARQL queries to $\mathcal{ELH}$ queries. The *lossless approach* only converts a SPARQL query q into an $\mathcal{ELH}$ concept C if the result set of q without solution modifiers is equal to the extension $C^{\mathcal{I}}$ of C. The *lossy approach* is a best-effort conversion of acyclic SPARQL queries q to $\mathcal{ELH}$ concepts C that needs not preserve equality between the result sets of q and C. This approach relies on heuristics to determine the parts of the

query that are to be converted.[2] In the following, we characterize the queries which can be converted by the lossless approach and describe the steps of the conversion. Thereafter, we present the heuristics at the core of our lossy approach.

3.1 Lossless Approach

This approach converts SPARQL queries q into $\mathcal{ELH}$ concepts C with $[[q]] = C^{\mathcal{I}}$, where $[[q]]$ stands for the interpretation of q without solution modifiers, i.e., for its result set.[3] For example, $C^{\mathcal{I}}$ for C = `Person` is equal to the interpretation of the query `SELECT ?x WHERE {?x a Person}`. Mapping SPARQL queries to $\mathcal{ELH}$ concepts comes with the following limitations: **a)** Only queries with one projection variable (i.e., unary queries) can be captured. **b)** The projection variable cannot be a role. **c)** The query must be acyclic and conjunctive. **d)** The query must be free of filters. Concept queries that fulfill these conditions are called ACQ (acyclic conjunctive queries). Importantly, while we consider queries with optional patterns and solution modifiers when running the benchmarks, we ignore both during the conversion process.

Preprocessing. Let $\boldsymbol{Q}$ be a set of ACQs (e.g., from a query log). The basic intuition behind our conversion strategy is to transform each ACQ $q \in \boldsymbol{Q}$ into an edge-labeled graph $G(q) = (V, E, l)$ with the following properties: **a)** V is the set of all subjects and objects (including variables) occurring in q. **b)** $e = (s, o) \in E$ iff $\exists p \in V : (s, p, o)$ is a triple pattern in q. **c)** If (s, p, o) is a triple pattern in q, then $l((s, o)) = p$.

Let ?x be the unique projection variable of the input query. To transform this graph into a tree with ?x as root, we first need to introduce extensions for properties, nominals and a universal property. Hence, we preprocess our input data as follows: **a)** For each individual o, we add a class O with $O^{\mathcal{I}} = \{o\}$, which is safe as we consider only tree structures in the end. **b)** We create a role r_0 with $r_0^{\mathcal{I}} = \bigcup_{r \in \mathsf{N_R}} r^{\mathcal{I}}$. **c)** For each role r we create a new role r^- with $(r^-)^{\mathcal{I}} = \{(b, a) : (a, b) \in r^{\mathcal{I}}\}$ representing the inverse role. Note that we do not include inverse roles in the TBox of the underlying ontology as it would exceed the expressivity of $\mathcal{ELH}$.

We transform the labeled graph $G(q)$ into a tree as follows: Let $\delta : V \times V \rightarrow \mathbb{N}$ be the graph distance between two nodes, i.e., the length of the shortest path between these nodes. For any edge (s, o) with label p, if $\delta(?x, s) > \delta(?x, o)$, then replace (s, o) with (o, s) and set $l(o, s) = p^-$. Consider for example the query shown in the following, which retrieves entities that live in capital cities:

```
SELECT ?x WHERE
            { ?x :livesIn ?y. ?z :hasCapital ?y.}
```

The distance from the subject of the second triple (?z) to the root (?x) is larger than the distance of the object (?y). Hence, we would transform it into the following query:

```
SELECT ?x WHERE
            {?x :livesIn ?y. ?y :hasCapital⁻¹ ?z.}
```

[2]Note that we assume that all triples entailed by $\mathcal{ELH}$ have been materialized in the knowledge base.

[3]Technically, $[[q]]$ is a sequence of partial variable bindings, which is a table and $C^{\mathcal{I}}$ is a set. Still, we can define the equality of the two by mapping the rows of $[[q]]$ to a set.

$G(q)$ is now a tree that we transform q into an $\mathcal{ELH}$ concept by using the transformation τ, which we define in the following.

Converting triple patterns to $\mathcal{ELH}$. If the query q contains exactly one triple pattern, then $G(q) = (s, p, o)$. In this case, $\tau(s, p, o)$ is computed as shown in Table 2 below.

Table 2: Conversion of triple patterns into $\mathcal{ELH}$ concepts.

Triple pattern	$\mathcal{ELH}$ concept
`?u p Class`	$\exists p$.Class
`?u p o`	$\exists p$.O
`?u rdf:type Class`	Class
`Class p ?u`	$\exists p^-$.Class
`s p ?u`	$\exists p$.s
`?u ?p Class`	$\exists r_0$.Class
`?u p ?o`	$\exists p.\top$
`Class ?p ?u`	$\exists r_0^-$.Class
`?u p ?x`	$\exists p^-.\top$
`?x ?p ?o`	$\exists r_0.\top$
`?s ?p ?x`	$\exists r_0^-.\top$

- `Class` stands for a class.
- `s` and `o` stand for individuals.
- All letters preceding by a question mark are variables.
- `?u` stands for the join variable or the root of the tree.

Converting queries to $\mathcal{ELH}$. For the sake of readability, we use the triple (s, p, o) to denote an edge $(s, o) \in E$ with the label p. The translation function τ is defined as follows: first compute all paths α_i from the root ?x to all leaves of $G(q)$ and set $\tau(G(q)) := \bigsqcap_i \tau(\alpha_i)$. The translation of paths is carried out as follows: if α_i is of length 1, then $\tau(\alpha_i)$ is a triple pattern and can be translated using Table 2. Otherwise, α_i is a sequence of triple patterns: $\alpha_i = [(s_i^1, p_i^1, o_i^1); \ldots; (s_i^z, p_i^z, o_i^z)]$, where $o_i^k = s_i^{k+1}$ for $k \in [1, z-1]$. We define $\tau(\alpha_i) := \tau([(s_i^1, p_i^1, o_i^1); \ldots; (s_i^z, p_i^z, o_i^z)])$ recursively as:

Rule 1: $\tau(\alpha_i) = \tau([(s_i^2, p_i^2, o_i^2); \ldots; (s_i^z, p_i^z, o_i^z)])$ if p_1 is `rdf:type`,
Rule 2: $\tau(\alpha_i) = \exists p_i^1.\tau([(s_i^2, p_i^2, o_i^2); \ldots; (s_i^z, p_i^z, o_i^z)])$ if p_i^1 is not a variable and is not `rdf:type`,
Rule 3: $\tau(\alpha_i) = \exists r_0.\tau([(s_i^2, p_i^2, o_i^2); \ldots; (s_i^z, p_i^z, o_i^z)])$, otherwise.

As an example to illustrate such a conversion consider the query q:

```
SELECT ?x WHERE
          { ?x a :Person. ?x :livesIn ?y.
          ?y :locatedIn Europe. }
```

q has two paths from the answer variable ?x to the leaves, namely α_1 = `?x a Person` and α_2 = `?x livesIn ?y. ?y locatedIn Europe`. Hence the conversion is $\tau(q) = \tau(\alpha_1) \sqcap \tau(\alpha_2)$. Since α_1 is a single triple pattern, it is translated into the concept C = `Person`. Since α_2 is a path, it must be transformed recursively. Given that the predicate in the first triple pattern in α_2 is not a variable and is not a `rdf:type`, Rule 2 applies and yields $\tau(\alpha_2) = \exists$ `livesIn`. τ(`?y locatedIn Europe`). Translating the inner part results in $\tau(q)$ = Person$\sqcap\exists$ livesIn.($\exists$locatedIn.{Europe}). The individual Europe is finally replaced with a singleton class. Note that the conversion step described in Subsection 3.3 needs to be applied for large query logs.

3.2 Lossy Approach

The lossy approach converts SPARQL queries by simply removing the parts that cannot be expressed in $\mathcal{ELH}$. It addresses the limitations presented above as follows: **a)** *Only queries with one projection variable.* If a query has more than one answer variable, the lossy approach selects a random projection variable and ignores the remaining ones. **b)** *The projection variable cannot be a role.* The lossy approach simply omits such queries. **c)** *The query must be conjunctive and acyclic.* Queries with UNIONs are transformed into disjunctive normal form and each of the conjunctions is considered independently. Join variables which lead to a cycle are replaced by as many distinct variables as the number of their occurrence. **d)** *The query must be free of filters.* Filters are simply ignored. The possibly pruned query is then converted using the procedure from the lossless approach. Generally, the lossy approach provides a means for the conversion of a large portion of SPARQL queries and admits in our case the generation of congenial benchmarks based on a broader set of queries.

3.3 Conversion of ABox Individuals

The two conversion approaches can cause a large number of singleton classes generated by mapping single individuals to concepts. This leads to impractical compute times and memory requirements if the set Q of ACQs to convert is large. Hence, we generalize over the results of the lossless and lossy approach in the following way. For all individuals s occurring in a translated query $\tau(q)$, we compute the most specific class S with $s^I \in S^I$. To this end, we compute the class hierarchy of all classes whose extension contains s. We then select (randomly) one of the leaves of this hierarchy and replace s with that leaf. As this may lead to identical $\mathcal{ELH}$ concepts for different queries, we only consider one of these $\mathcal{ELH}$ concepts and subsume the queries under this concept for the matter of performance.

4 GENERATION OF CONGENIAL BENCHMARKS

Let C be the set of all distinct $\mathcal{ELH}$ concepts extracted from query log Q and converted to $\mathcal{ELH}$ concepts. The aim of our benchmark generation approach is to compute a set of representative concepts to stand for C according to their semantic similarity. To this end, we use the simple but time-efficient graph clustering approach described in the subsequent subsection. The input graph SG is a truncated similarity graph whose set of nodes is C. $(C_1, C_2) \in C^2$ is an edge in SG iff $BM\text{-}simi(C_1, C_2) \geq \theta$, where $\theta \in [0, 1]$. The weight of the edge (C_1, C_2) is then set to $BM\text{-}simi(C_1, C_2)$. In our experiments we set θ to 0.45 to assure a middle ground between very specific and very broad clusters.

4.1 Clustering $\mathcal{ELH}$ query concepts by *BM-simi*

An overview of the method is given in Algorithm 1. We define the similarity between an $\mathcal{ELH}$ concept C and a cluster $\mathcal{D}$ as

$$sim(C, \mathcal{D}) = \frac{|\mathcal{D}|}{N} \cdot \sum_{D \in \mathcal{D}} BM\text{-}simi(C, D),$$

where N stands for the average size of all clusters. Our approach begins with an empty set of clusters. For each $\mathcal{ELH}$ concept, say C, our algorithm checks if there is a cluster which contains a concept D with $BM\text{-}simi(C, D) = 1$. In this case, these two concepts are equivalent, since *BM-simi* is equivalence closed, and $\mathcal{ELH}$ concept C is added to this cluster. Otherwise, the algorithm checks if there exists a cluster where the similarity to the concept is higher than the given threshold θ (0.45 in our experiments). If there are several clusters with a similarity higher than the threshold, the $\mathcal{ELH}$ concept is added to the one with the highest similarity. Else, a new cluster is created for C.

4.2 Query Selection from Clusters

The final step of the process for generating congenial benchmarks is the selection of the prototypical concept from a cluster. For each cluster, we select (one of) the concepts with the highest average similarity to all other concepts in the cluster, i.e., the medoid of the cluster. The overall procedure of conversion, clustering and query

Algorithm 1 SPARROW's cluster algorithm.

```
procedure ComputeClusters(SG, θ)
Input: Similarity Graph SG, threshold θ
Output: Set of clusters
    Set ClusterSet = new Set()
    for Concept C in SG.getConcepts() do
        int maxSimi = 0
        Best = BestCluster(C, ClusterSet)
        Best.add(C)
        if Best ∉ ClusterSet then
            ClusterSet.add(Best)
        end if
    end for
    return ClusterSet
end procedure
function BestCluster(C, ClusterSet)
    Cluster bestCluster = new Cluster()
    for Cluster in ClusterSet do
        int simi = 0
        for Concept D in Cluster do
            if SG.getSim(C, D)==1 then
                return Cluster
            else
                simi = simi + SG.getSim(C, D)
            end if
        end for
        simi = simi∗Cluster.size() / avgSize(ClusterSet)
        if simi ≥ θ ∧ simi > maxSimi then
            bestCluster = Cluster
            maxSimi = simi
        end if
    end for
    return bestCluster
end function
```

selection was implemented in our system for generating congenial benchmarks called Sparrow.[4]

5 EXPERIMENTS AND RESULTS

The goal of our evaluation was twofold. First, we wanted to *characterize the difference between congenial benchmarks and benchmarks based on SPARQL features*. In [18], the benchmark generation approach Feasible was compared with other approaches such as [12] and it was concluded that they achieve the smallest compounded error during the benchmark creation process. Hence, we used Feasible to generate benchmarks based on SPARQL features for such a comparison. We compared the clusters generated by Sparrow and by Feasible using the positive predictive value as measure (short PPV, see Section 5.3 for a formal definition). Second, we wanted to *test whether benchmarking results computed using a congenial and a SPARQL feature-based benchmark differ*. We hence executed the benchmarks generated by Sparrow and Feasible on the same original datasets using the same hardware and compared the behavior of state-of-the-art triple stores on these two kinds of benchmarks.

5.1 Experimental Data and Setup

Hardware: All experiments were executed on a desktop machine with an Intel(R) Core(TM) i7-3770 CPU @ 3.4 GHZ CPU, 32 GB RAM and a 4 TB HDD running on Ubuntu 16.04.4 LTS and Java 1.8. Each experiment was executed using the benchmark execution framework Iguana v2.0.0 [4][5], which we chose because it is open-source and thus ensures that our experiments can easily be repeated.

Datasets: We used two datasets and two corresponding query logs. *Semantic Web Dog Food* (*SWDF*)[6] is a small dataset with 304,592 triples and 185 distinct properties. For this dataset, we used a log with 1,411,932 queries provided by LSQ.[7] We used a subset of *DBpedia*[8] with 361,404,364 triples and 60,677 distinct properties as our second dataset. For this dataset, we relied on a log with 2,311,262 queries.[9] During our experiments, we bulk loaded each of the datasets into the triple stores before starting the evaluation procedure for the said dataset. During each run of the experiment, the triple stores contained only the dataset upon which the benchmarking was being carried out.

Triple stores: We selected triple stores that were openly available, were able to store the DBpedia dataset and that offered a SPARQL 1.1 HTTP endpoint at the time of writing. The following three triple stores were hence used in our evaluation: **a)** Virtuoso, 7.2.5.3229-pthreads, **b)** Fuseki (Jena TDB), 3.5.0, and **c)** GraphDB, GRAPHDB_LITE v8.3.1+sha.4a20f47. Each triple store was allocated 16 GB RAM. For Virtuoso we used the setting *NumberOfBuffers*=1,360,000 and *MaxDirtyBuffers*=1,000,000.

Benchmark Generation: We applied the lossless and the lossy approaches to convert the queries from the DBpedia and SWDF query logs into $\mathcal{ELH}$ concepts. We used the set of queries which could be converted into $\mathcal{ELH}$ as input for the benchmark generation in Feasible. The set of corresponding $\mathcal{ELH}$ concepts was used as input for the benchmark generation of Sparrow.

Benchmark Execution: To ensure *realistic query loads*, each $\mathcal{ELH}$ concept selected by Sparrow was mapped back to the original SPARQL query from which it was computed. We executed these real queries (not their $\mathcal{ELH}$ conversions) throughout our benchmarking experiments based on Sparrow. The benchmarking experiments based on Feasible were carried out with the real queries Feasible had selected as prototypes. Each experiment was executed in a single-threaded stress test with 60 minutes runtime and a timeout of 3 minutes per query. After each run, the tested triple store was restarted to avoid the possible effects of caching and buffering.

5.2 Benchmark Characteristics from Benchmark Generation

An overview of the number of queries converted and the size of the benchmarks generated by our conversion approaches is shown in Table 3. For SDWF, the lossless approach converted 525 queries and the lossy approach converted 33,413 queries. The queries were used to generate 185 resp. 271 clusters. DBpedia led to more concepts, with the lossless approach generating more than 180,000 concepts and 6,734 clusters. Like in Feasible, we used the 250 most densely populated clusters to generate our congenial benchmark. As the lossy approach generated too many queries for clustering, we only considered a random subset of $2 \cdot 10^5$ queries from the DBpedia query log. More than 51% of these queries were transformed into $\mathcal{ELH}$ concepts. Again, only the 250 most dense clusters from the 4,116 generated clusters were considered for the benchmark.

Table 3: Overview of the conversion of SPARQL queries into $\mathcal{ELH}$ concepts.

	SWDF		DBpedia	
Approach	lossless	lossy	lossless	lossy
Size of query log	1,411,932	1,411,932	2,311,262	200,000
$\mathcal{ELH}$ concepts	525	33,413	180,128	102,028
Number of clusters	185	271	6,734	4,116
Benchmark size	185	271	250	250

5.3 Comparing Congenial and Feature-based Benchmarks

The first question to address regarded the similarity between the benchmarks generated by Feasible and Sparrow. As in previous works (e.g. see [3]), we measured the positive predictive value (PPV) of Sparrow w.r.t. Feasible and vice-versa. Let $\mathcal{S}$ be the set of clusters generated by Sparrow and $\mathcal{F}$ be the set of clusters generated by Feasible. The PPV of Sparrow w.r.t. Feasible is defined as

$$PPV(\mathcal{S},\mathcal{F}) = \frac{1}{|\mathcal{S}|} \cdot \sum_{S \in \mathcal{S}} \frac{\max_{F \in \mathcal{F}} |S \cap F|}{|S|}.$$

The results of our comparison are shown in Figure 1. The average

[4] https://github.com/dice-group/sparrow
[5] http://iguana-benchmark.eu/
[6] https://drive.google.com/file/d/1R1bmqOTfdOV3paVRGfZbERbQwcUjEBRO
[7] The LSQ datasets can be found at http://lsq.aksw.org. A copy of the query log used for SWDF is at http://goo.gl/3q52Ka.
[8] That is, all the files listed at http://downloads.dbpedia.org/2016-10/core/.
[9] All queries can be found at https://bit.ly/2FAKbfk

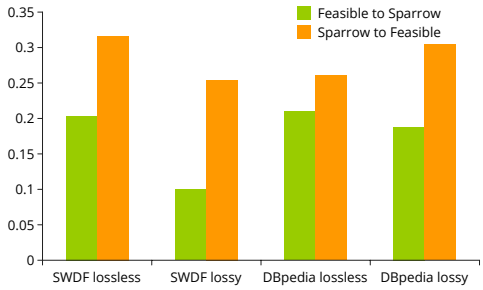

Figure 1: Positive predictive value of Sparrow w.r.t. Feasible and vice-versa.

PPV from Feasible to Sparrow is below 0.21, i.e., less than 21% of the queries contained in the clusters from Feasible are contained in the best matching cluster in Sparrow. A similar observation holds in the other direction, with the threshold being around 32%. We can conclude **a)** that the benchmarks generated by Sparrow and Feasible for our experiments are different in their composition and **b)** that the semantic (Sparrow) and structural (Feasible) similarity of SPARQL queries differ considerably. These findings support our initial hypothesis: *the prototype queries needed to measure the performance of triple stores on congenial queries are clearly different from the prototype queries computed by feature-based benchmarks.*

5.4 Performance Evaluation of Triple stores based on Congenial vs. Feature-based Benchmarks

We measured the number of Queries per Second (QpS) as well as the Query Mixes per Hour (QMpH) achieved by the triple stores Virtuoso, Fuseki, and GraphDB in four different settings (2 datasets and lossless vs. lossy conversion). Figure 2 shows the distribution of QpS in all four settings for both, Sparrow and Feasible. As expected, there is a considerable difference between the behavior of triple stores faced with (Sparrow implementing) a congenial benchmark and their behavior when benchmarked with (Feasible implementing) a feature-based benchmark. On SWDF (see Figures 2a and 2c), the mean of the QpS is commonly higher for Sparrow. For example, GraphDB reaches a median of 422.28 on Sparrow (lossless) while it only achieves a median of 363.64 on Feasible (lossless). Similar observations can be made on the other datasets. However, the number of outliers is considerably higher than with Feasible. A similar but less pronounced effect can be seen on the DBpedia dataset (see Figures 2b and 2d).

A comparison of the QMpH achieved using Sparrow and Feasible also suggests a considerable difference between the average runtimes of the queries selected from the same subset of queries by Sparrow and Feasible (see Figure 3). While one could assume that this difference is due to the combination of SPARQL features used in Feasible queries, our study of the distribution of SPARQL features across the queries in the two benchmarks shows that there is no significant difference between the use of SPARQL features in Sparrow and Feasible.[10] This result substantiates the claim made in the introduction of the paper, i.e., *that the ability of triple stores to cater to prototypical user needs expressed as queries—as measured by congenial benchmarks—differs from their ability to execute prototypical combinations of SPARQL features—as measured by current state-of-the-art benchmarks.*

An analysis of Figure 3 reveals another important insight unveiled by our results: *the fastest triple stores w.r.t. stress tests based on SPARQL features are not necessarily the best to cater to congenial queries.* In particular, Virtuoso is the fastest triple store according to Feasible but it is outperformed by GraphDB on Sparrow in all experimental setups. This means in particular that *the ability to cater for particular combinations of SPARQL features differ from that of catering for prototypical user needs.*

This result is of central importance as it suggests that while Virtuoso implements prototypical combinations of SPARQL features efficiently, applications based on generating a flurry of similar queries (e.g., structured machine learning applications) might be better served using GraphDB as a backend. These results further emphasize the need for congenial benchmarks by suggesting that congenial benchmarks provide more adequate results pertaining to the performance of triple stores as backends of applications, such as structured machine learning, browsing and relationship finding, which all rely on sequences of semantically similar queries.

6 DISCUSSION

We presented and evaluated a novel approach towards generating benchmarks for triple stores: congenial benchmarks. While current benchmarks based on real data and real queries (such as Feasible) often measure the ability of triple stores to deal with prototypical syntactic features of SPARQL queries, congenial benchmarks (such as generated by Sparrow) measure how well triple stores can cater for prototypical users' needs by considering the semantic similarity of SPARQL queries. Our evaluation revealed that these two tasks differ considerably. This means that *congenial benchmarks allow the exploration of a portion of the behavior of triple stores, which has not been considered in the literature.*

Our implementation of a congenial benchmark generation system revealed a large number of research avenues. First, semantic similarity frameworks often do not scale to large numbers of queries. The computation of bounded similarity has been well studied in the deduplication and link discovery literature. However, semantic similarity measures such as simi have never been studied in this respect. As the development of an efficient means for the detection of semantically near-duplicates went beyond the scope of this paper, we had to implemented a rather naïve indexing technique for accelerating the runtime of our benchmark generation procedure. The efficient generation of congenial benchmarks from large query logs will however demand the development of efficient techniques for computing the semantic similarity of SPARQL query concepts.

We had to use $\mathcal{ELH}$ concepts as coarse approximation to measure the similarity of user needs expressed in SPARQL queries, which is a strong restriction. While our results suggest that this approach still suffices to show that congenial benchmarks differ from benchmarks based on SPARQL features, our implementation also shows that more than 50% of the SPARQL queries cannot be compared using this conversion. Hence, there is clearly a need

[10] Detailed numbers can be found in the supplementary materials.

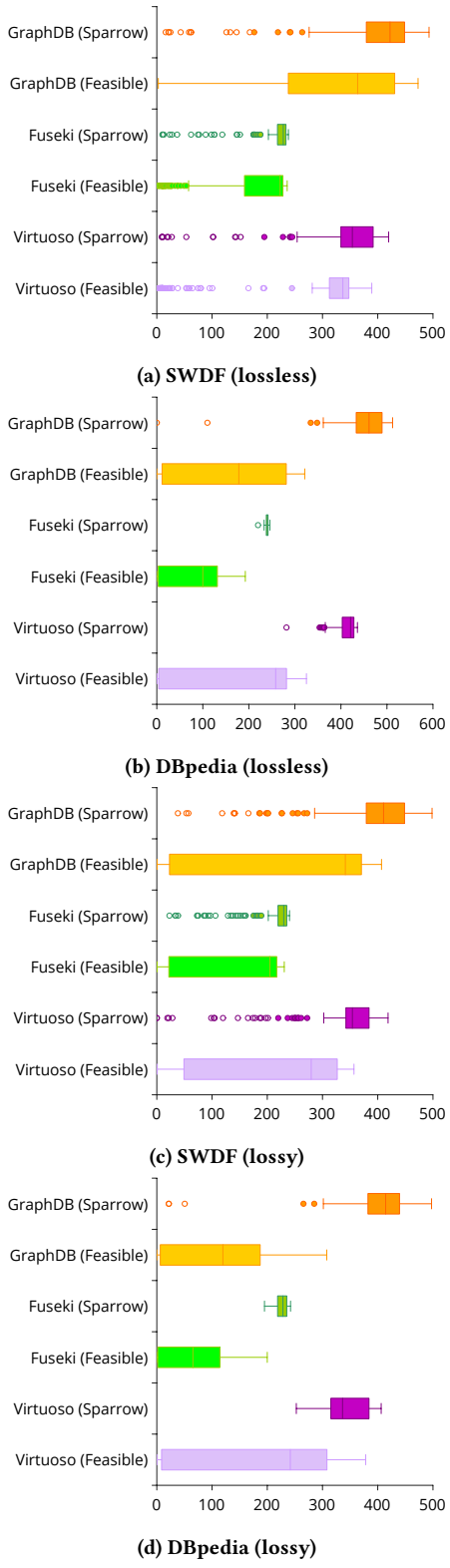

Figure 2: Distribution of queries per second (QPS) on SWDF and DBpedia.

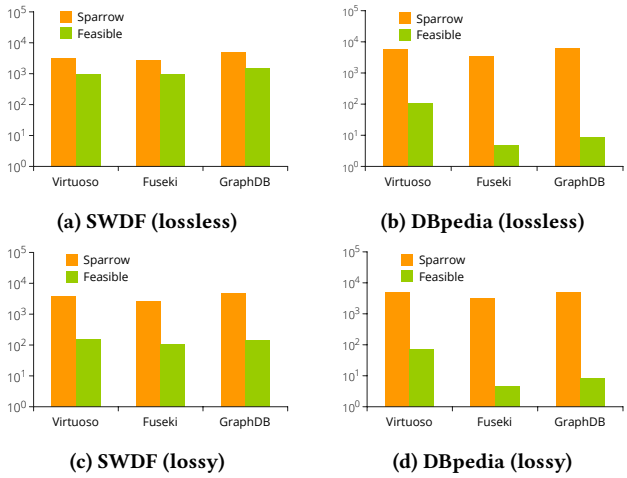

Figure 3: QMpH on SWDF and DBpedia. The y-axis is in logarithmic scale.

to develop frameworks for measuring the semantic similarity of SPARQL queries. First steps in this direction could be to build semantic similarities for monotone SPARQL queries based on their newly presented canonical form [14].

REFERENCES

[1] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *ISWC*. 197–212.

[2] Simon Bin, Lorenz Bühmann, Jens Lehmann, and Axel-Cyrille Ngonga Ngomo. 2016. Towards SPARQL-based Induction for Large-scale RDF Data Sets. In *ECAI*. IOS Press, 1551–1552.

[3] Sylvain Brohee and Jacques Van Helden. 2006. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC bioinformatics* 7, 1 (2006), 488.

[4] Felix Conrads, Jens Lehmann, Muhammad Saleem, Mohamed Morsey, and Axel-Cyrille Ngonga Ngomo. 2017. IGUANA: A Generic Framework for Benchmarking the Read-Write Performance of Triple Stores. In *International Semantic Web Conference (ISWC)*.

[5] Renata Queiroz Dividino and Gerd Gröner. 2013. Which of the following SPARQL Queries are Similar? Why?. In *Proceedings of the First International Workshop on Linked Data for Information Extraction (LD4IE'13) (CEUR Workshop Proceedings)*, Vol. 1057. CEUR-WS.org.

[6] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. 2011. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 145–156.

[7] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. 2014. HermiT: An OWL 2 Reasoner. *J Autom Reasoning* 53, 3 (2014), 245–269.

[8] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. In *Web Semantics*, Vol. 3. Elsevier, 158–182.

[9] Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. 2009. RelFinder: Revealing relationships in RDF knowledge bases. In *International Conference on Semantic and Digital Media Technologies*. Springer, 182–187.

[10] Paul Jaccard. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin de la Societe Vaudoise des Sciences Naturelles* 37 (1901), 547–579.

[11] Karsten Lehmann and Anni-Yasmin Turhan. 2012. A Framework for Semantic-based Similarity Measures for $\mathcal{ELH}$-Concepts. In *Proc. of the Europ. Conf. on Logics in AI*. Springer, 307–319.

[12] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data. In *International Semantic Web Conference*, Vol. 7031. Springer Heidelberg, 454–469.

[13] Shi Qiao and Z. Meral Özsoyoglu. 2015. RBench: Application-Specific RDF Benchmarking. In *SIGMOD*. ACM, 1825–1838. https://doi.org/10.1145/2723372.2746479

[14] Jaime Salas and Aidan Hogan. 2018. Canonicalisation of monotone SPARQL queries. In *International Semantic Web Conference*. Springer, 600–616.
[15] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: The linked sparql queries dataset. In *ISWC*. Springer, 261–269.
[16] Muhammad Saleem, Ali Hasnainb, and Axel-Cyrille Ngonga Ngomo. 2017. LargeRDFBench: A Billion Triples Benchmark for SPARQL Endpoint Federation. In *Journal of Web Semantics (JWS)*.
[17] Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. 2015. A fine-grained evaluation of SPARQL endpoint federation systems. *Semantic Web* (2015), 1–26.
[18] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. Feasible: A Feature-Based SPARQL Benchmark Generation Framework. In *International Semantic Web Conference*. Springer, 52–69.
[19] Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. 2011. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In *International Semantic Web Conference*. 585–600.
[20] Ahmet Soylu, Martin Giese, Ernesto Jimenez-Ruiz, Evgeny Kharlamov, Dmitriy Zheleznyakov, and Ian Horrocks. 2014. Towards exploiting query history for adaptive ontology-based visual query formulation. In *Research Conference on Metadata and Semantics Research*. Springer, 107–119.
[21] Christina Unger, Corina Forascu, Vanessa Lopez, Axel-Cyrille Ngonga Ngomo, Elena Cabrio, Philipp Cimiano, and Sebastian Walter. 2014. Question answering over linked data (QALD-4). In *Working Notes for CLEF Conf.*