SQCFramework: SPARQL Query Containment Benchmark Generation Framework

Muhammad Saleem
AKSW, Uni Leipzig, Germany
saleem@informatik.uni-leipzig.de

Claus Stadler
AKSW, Uni Leipzig, Germany
stadler@informatik.uni-leipzig.de

Qaiser Mehmood
INSIGHT, NUIG, Ireland
qaiser.mehmood@insight-centre.org

Jens Lehmann
Uni Bonn, Germany
Fraunhofer IAIS, Bonn, Germany
jens.lehmann@cs.uni-bonn.de
jens.lehmann@iais.fraunhofer.de

Axel-Cyrille Ngonga Ngomo
AKSW, Uni Leipzig, Germany
Uni-Paderborn, Germany
ngonga@informatik.uni-leipzig.de
axel.ngonga@upb.de

ABSTRACT

Query containment is a fundamental problem in data management with its main application being in global query optimization. A number of SPARQL query containment solvers for SPARQL have been recently developed. To the best of our knowledge, the Query Containment Benchmark (QC-Bench) is the only benchmark for evaluating these containment solvers. However, this benchmark contains a fixed number of synthetic queries, which were handcrafted by its creators. We propose SQCFramework, a SPARQL query containment benchmark generation framework which is able to generate customized SPARQL containment benchmarks from real SPARQL query logs. The framework is flexible enough to generate benchmarks of varying sizes and according to the user-defined criteria on the most important SPARQL features to be considered for query containment benchmarking. This is achieved using different clustering algorithms. We compare state-of-the-art SPARQL query containment solvers by using different query containment benchmarks generated from DBpedia and Semantic Web Dog Food query logs. In addition, we analyze the quality of the different benchmarks generated by SQCFramework.

CCS CONCEPTS

•**General and reference** →*Metrics; Evaluation; Performance;*

ACM Reference format:
Muhammad Saleem, Claus Stadler, Qaiser Mehmood, Jens Lehmann, and Axel-Cyrille Ngonga Ngomo. 2017. SQCFramework: SPARQL Query Containment Benchmark Generation Framework. In *Proceedings of Knowledge Capture, Texas, USA, December 2017 (K-CAP'2017),* 8 pages.
DOI:

1 INTRODUCTION

Query containment is the problem of deciding whether the result set of one query is included in the result set of another. It is a well-known problem and is used in a number of tasks, such as devising efficient query planners and caching mechanisms, data integration, view maintenance, determining independence of queries from updates, and query rewriting etc. [4, 9]. For example, suppose query $q1$ is contained in query $q2$. In case $q2$'s execution is time-expensive, the result of $q1$ may be obtained more efficiently from $q2$'s result set. Similarly, it is possible to check if two queries are equivalent by simply checking their mutual containment. The significant growth of the Web of Data has motivated a considerable amount of work on SPARQL query containment [1, 3, 5–8, 10, 14, 15]. To the best of our knowledge, the SPARQL Query Containment Benchmark (SQC-Bench) [2] is the only benchmark designed to test SPARQL query containment solvers. This benchmark contains a fixed number of 76 query containment tests handcrafted by the authors. Even though the benchmark contains a variety of tests of varying complexities, the number of tests is fixed and all tests are synthetic. While SQC-Bench serves the purposes of its developers well, it provides developers limited insights into the performance of their systems on real datasets [12]. In addition, this benchmark does not allow users to generate benchmarks tailored towards specific use-cases.

We propose SQCFramework, a framework for the automatic generation of SPARQL query containment benchmarks from real SPARQL query logs. The framework is able to generate benchmarks customized by its user in terms of (1) the number of containment tests, (2) the structural features of the SPARQL queries (e.g., number of triple patterns, number of projection and join variables, join vertex degree etc.), and (3) the use of SPARQL constructs (e.g., UNION, OPTIONAL, FILTER, REGEX etc.). For example, the user can opt to create a benchmark with 100 benchmark queries, in which super-queries (the query that contains another query) have a minimal number of triples patterns equal to 4, having less than 3 join vertices and at least a UNION clause. The framework generates the desired benchmark from the query log by using different clustering methods, while considering the customized selection criteria specified by the user. The contributions of this work are as follows: (1) We present the first (to the best of our knowledge) SPARQL query containment benchmarks generation framework from real queries. (2) Our framework allows the automatic generation of flexible benchmarks according to the given use-case criteria specified by the user. (3) We identify the key SPARQL features that should be considered while designing SPARQL query containment benchmarks. We then apply the well-known algorithms – KMeans++, Aglomerative, DBSCAN+KMeans++, FEASIBLE and FEASIBLE-Exemplars

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
K-CAP'2017, Texas, USA
© 2017 Copyright held by the owner/author(s). ...$15.00
DOI:

[12], Random selection – to select the benchmark queries from the given set of containment queries. The detailed analysis of the generated benchmarks enable users to select the best-fit benchmark according to their needs. (4) We compare state-of-the-art SPARQL query containment solvers using the benchmarks generated by our framework and discuss the result.

SQCFramework is open-source and available online along with a live demo[1].

2 RELATED WORK

The query containment problem has a rich literature in relational databases. In this work, we focus on SPARQL query containment. In [1, 15], the SPARQL query containment problem is reduced to a formula satisfiability test using μ-calculus. While these works provide theoretical proofs for the model they propose, they do not provide any prototype implementation. The Alternation Free two-way μ-calculus (AFMU) [14] is a prototype implementation of the formula satisfiability solver for the alternation-free fragment of the μ-calculus. In AFMU, the problem of SPARQL query containment is also reduced to a formula satisfiability test. The framework supports projections, union conjunctive queries (UCQ), blank nodes, and RDFs reasoning on SPARQL queries [2]. Another solver, the XML tree logic solver (TreeSolver), is presented in [6]. It performs static analysis on XPath queries by transforming them into μ-calculus. It is a satisfiability solver which can be used for decision problems such as containment, equivalence, overlap, and coverage.

Letelier et al. [8] studied the fundamental problem of SPARQL query containment in the form of subsumption relation. SPARQL-Algebra is the SPARQL query containment solver based on the theoretical study presented in [8]. This implementation supports conjunctive and OPTIONAL queries with no projection [2]. The complexity of well-designed SPARQL query containment, restricted to the conjunction and OPTIONAL operators and with extensions by UNION and/or projection is studied in [10]. Their complexity results range from NP-completeness to undecidability. The problem of SPARQL query containment for an expressive class of navigational queries called Extended Property Paths (EPPs) is studied in [3]. Finally, [5] reduced the SPARQL query containment problem to the problem of deciding whether a sub graph isomorphism exists between the (normalized) algebra expression tree of a view and a sub-tree of a user query.

To the best of our knowledge, SQC-Bench [2] is the only SPARQL query containment benchmark that compared AFMU, SPARQL-Abebra, and TreeSolver containment solvers. The benchmark contains three test suites: (1) **Conjunctive Queries with No Projection (CQNoProj):** This test suite contains a total of 20 containment test cases and is designed for the containment of Basic Graph Patterns (BGPs). It contains conjunctive queries with no projection. Each test-case checks containment between two queries. (2) **Union of Conjunctive Queries with Projection (UCQProj):** This test suite comprises 28 test cases: 14 tests contain projections, 6 tests contain UNIONs and 2 tests contain both. (3) **Union of Conjunctive Queries under RDFS reasoning (UCQrdfs):** There is a total of 28 test cases in this category. It tests the query containment solver under RDFs reasoning. In comparison to the previous two test suites, the queries in this category have a low complexity, especially in terms of number of triple patterns and number of join variables. This test suite makes use of 4 different small schemas to test the correctness of solvers.

SQC-Bench provides a good variety in terms of number of triple patterns, number of variables, number of joins and size of the ontology. In addition, it also takes the type of graph pattern connectors (`UNION`, `OPTIONAL`, `FILTER` etc.) and the type of ontology (no schema, RDFS, OWL, etc.) into account. However, the benchmark is synthetic and contains a fixed number of 76 test cases. As acknowledged by the authors, the used schemas are not realistic [2]. Users are not able to generate customized containment benchmarks, pertaining to a specific use case. In addition, it is likely that real queries issued by the user differ significantly from the synthetic queries presented in SQC-Bench [11, 12]. SQCFramework addresses these drawbacks and allows users to generate customized benchmarks from the SPARQL queries log, i.e., real queries posted by real agents.

3 PRELIMINARIES

In this section, we define key concepts necessary to understand the subsequent sections of this work. To denote the set of solution mappings that is defined as the result of a SPARQL query Q over an RDF graph G we write $[\![Q]\!]_G$. According to [2], we define the arity of a query as the arity of the query answers. Note that when there is an outer projection (i.e., explicitly mentioned in the query), it is defined by its distinguished variables. Otherwise, it is defined by all free variables of the query. For a given RDF graph, we say a query is contained in another query if all of its answers are included in those of other query.

Definition 3.1 (SPARQL query containment). Given two queries Q1 and Q2 with the same arity, Q1 is Contained in Q2 denoted by Q1 $\sqsubseteq$ Q2, if and only if $[\![Q1]\!]_G \subseteq [\![Q2]\!]_G$ for every RDF graph G. We call Q2 a super-query and Q1 a sub-query.

As mentioned, [2] identifies that structural SPARQL query features (e.g., number of BGPs, number of triple patterns, number of join vertices, number of projection variables etc.) are important considerations while designing query containment benchmarks. We also considered these features while generating automatic SPARQL containment benchmarks. In the following, we define these SPARQL query features formally.

We represent any basic graph pattern (BGP) of a given SPARQL query as a *directed hypergraph* (DH) [13], a generalization of a directed graph in which a hyperedge can join any number of vertices. In our specific case, every hyperedge captures a triple pattern. The subject of the triple becomes the source vertex of a hyperedge and the predicate and object of the triple pattern become the target vertices. For instance, consider the query in Figure 1 whose hypergraph is illustrated in Figure 1. Unlike common SPARQL representation, in which the subject and object of the triple pattern are connected by an edge, our hypergraph-based representation contains nodes for all three components of the triple patterns. As a result, we can capture joins that involve predicates of triple patterns. Formally, our hypergraph representation is defined as follows:

Definition 3.2 (Directed hypergraph of a BGP). The hypergraph representation of a BGP B is a directed hypergraph $HG = (V, E)$

[1]https://github.com/AKSW/SQCFramework

whose vertices are all the components of all triple patterns in B, i.e., $V = \bigcup_{(s,p,o)\in B}\{s,p,o\}$, and that contains a hyperedge $(S,T) \in E$ for every triple pattern $(s,p,o) \in B$ such that $S = \{s\}$ and $T = (p,o)$.

Listing 1: Examplary SPARQL query

```
SELECT DISTINCT ?drug ?title WHERE {
?drug db:cat dbc:mic.  ?drug db:regNo ?id.
?keggDrug rdf:type kegg:Drug.
?keggDrug bio2rdf:xRef ?id.
?keggDrug purl:title ?title. }
```

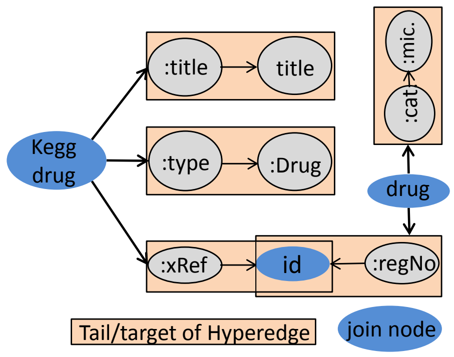


Figure 1: DH representation of the SPARQL query given in Listing 1. Prefixes are ignored for succinctness.

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

Definition 3.3 (Join Vertex). For every vertex $v \in V$ in such a hypergraph we write $E_{\text{in}}(v)$ and $E_{\text{out}}(v)$ to denote the set of incoming and outgoing edges, respectively; i.e., $E_{\text{in}}(v) = \{(S,T) \in E \mid v \in T\}$ and $E_{\text{out}}(v) = \{(S,T) \in E \mid v \in S\}$. If $|E_{\text{in}}(v)| + |E_{\text{out}}(v)| > 1$, we call v a *join vertex.*

Definition 3.4 (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.

4 SQCFRAMEWORK BENCHMARK GENERATION

In this section, we present the benchmark generation process in the SQCFramework. We first discuss the selection of real queries that we use for benchmarking. We then discuss the key query features that we considered while generating the benchmarks using different clustering methods. We finally discuss the generation of customized benchmarks.

4.1 Input Queries

Our framework takes a set of queries as input and selects the required sample of queries according to the user-defined criteria. The input queries commonly originate from the query logs of SPARQL endpoints or can be manually provided by the user. In this work, we aim to generate benchmarks from real user queries. To this end, we use the Linked SPARQL Queries (LSQ) datasets [11], which provide real queries extracted from the logs of public SPARQL endpoints. The LSQ datasets provide various statistics (e.g., number of joins, number of triple patterns, number of join vertices etc.) about queries that we considered important (discussed in next section) for benchmark generation. Currently, the LSQ project contains datasets extracted from 20 SPARQL endpoints.[2]

4.2 Important Query Features

A query containment benchmark should comprise queries/tests of varying complexities. Hence, we consider the following query features while generating containment benchmarks. **(1) Number of entailments/sub-queries:** Unlike QC-Bench where a super-query commonly has one sub-query, it is possible for a given super-query to have multiple sub-queries in a real SPARQL queries log. It is hence possible that a given containment solver can successfully identify one sub-query and fail to identify another sub-query for the same super-query.
(2) Number of projection variables: As pointed out by [2], this dimension is of significant importance and should be considered while designing containment benchmarks. This is because the query containment checks if the answers (solution mappings to projection variables) of one query are contained in the answers of another. The higher the number of projection variables, the harder to check for containment. The number of projection variables for the query shown in Figure 1 is 2, i.e., drug and title.
(3) Number of BGPs: A BGP is a sequence of triple patterns with optional `Filters`. Adding other graph patterns such as `UNION` and `OPTIONAL` terminates the BGP[3]. This feature is considered by [2] in the UCQProj test suites. Previous work [8] pointed out that the SPARQL `OPTIONAL` is one of the most difficult constructors for SPARQL query containment checks. The number of BGPs for the query given in Figure 1 is 1.
(4) Total number of triple patterns: The total number of triple patterns in a query has a direct relation with how expensive the containment check is: the more the triple patterns, the harder the query containment check [2]. The number of triple patterns for the query given in Figure 1 is 5.
(5) Max. and Min. BGP triple patterns: BGPs are the most essential and important building blocks of SPARQL queries, which virtually every SPARQL-related system has to support. This feature enables the selection of queries by the sizes of their BGPs. As an example, the query given in Figure 1 has a single BGP with 5 triple patterns, hence, the Min. and Max. numbers of BGP triple patterns is 5.
(6) Number of join vertices: This feature is also considered by [2]. The answer size of a given query varies significantly depending on the join vertices in the query. It is possible that a given triple pattern has a large number of solution mappings which are excluded after performing *join* with the solution mappings of another triple

[2]The LSQ datasets are available from SQCFramework website
[3]BGP: https://www.w3.org/TR/sparql11-query/#BasicGraphPatterns

pattern. The number of join vertices for query given in Figure 1 is 3, i.e. the variables *keggDrug*, *id*, and *drug*.

(7) Mean join vertex degree: The number of solution mappings pertaining to a given projection variable is directly affected by number of joins and their degree. Therefore, it is important to consider both of these query features for containment benchmarking. For the query given in Figure 1, the degrees of join vertices *keggDrug*, *id*, and *drug* are 3,2,2, respectively. The mean join vertex degree becomes 7/3 = 2.33.

(8) Number of LSQ features: The Linked SPARQL Queries (LSQ) [11] stores additional SPARQL features such as use of `DISTINCT`, `REGEX`, `FILTER`, `LIMIT`, `BIND`, `VALUES`, `ORDERY BY`, `HAVING`, `GROUP BY`, `OFFSET` aggregate functions (e.g., `COUNT`, `SUM`, `Min`, `Max` etc.), `SERVICE`, `OPTIONAL`, `UNION`, property path etc. We make a count of all of these SPARQL operators and functions and use it as a single query dimension. The number of LSQ features for query given in Figure 1 is 2, i.e., `DISTINCT` and `LIMIT` clauses.

The LSQ datasets contains most of the statistics about queries mentioned above. The only feature that is missing is the number of entailments, i.e., the number sub-queries for a every super-query in the dataset (i.e., LSQ does not store the containment relationships among queries). We made use of the existing SPARQL query containment solvers [5, 6, 8, 14] and actually running queries to identify the containment relationships among queries in LSQ. We verified the containment results by actually executing the both super- and sub-queries over the underlying dataset and programmatically checking if the result of the sub-query is indeed included in the result of the super-query. In our benchmarks, we only consider those LSQ queries which participate in at least one containment relation, i.e., the queries are either a super-query or a sub-query for another query in the dataset.

4.3 Benchmark Generation

Our benchmark generation problem is defined as follows:

Definition 4.1 (Benchmark Generation Problem). Let L represent the set of input (LSQ) queries. Our goal is to select the N queries that best represent L as well as more diverse in features, with $N << |L|$.

The user provides the input LSQ dataset and the required number N of super-queries and the selection criteria to be considered in the generated benchmark. Then, the benchmark generation is carried out in the following four main steps: (1) Select all the super-queries along with the required features from the input LSQ dataset. (2) Generate feature vectors and their normalization for the selected super-queries.(3) Generate N number of clusters from the super-queries. (4) Select single most representative super-query from each cluster to be included in the final benchmark.

Note that since the number of sub-queries for a given selected super-query can be greater than or equal to 1, the number of query containment tests in a benchmark will be greater than or equal to the number of super-queries. Now we discuss each of these steps in detail.

4.3.1 Selection of super-queries. Since LSQ contains RDF datasets, we can select super-queries along with the statistics by simply using SPARQL queries. Our framework retrieves the set of super-queries along with the required features (discussed in previous section)

Listing 2: Super-queries selection along with required features from LSQ dataset

```
Prefix lsq: <http://lsq.aksw.org/vocab#>
SELECT  DISTINCT ?sup count(DISTINCT ?sub)
    as ?entailments ?projVars ?bgps ?tps ?
    joinVertices ?maxBGPTriples ?
    minBGPTriples ?meanJVD count(?feature)
    as ?noFeatures {
?sub  lsq:isEntailed ?sup .
?sup  lsq:hasStructuralFeatures ?sf .
?sf   lsq:projectVars ?projVars .
?sf   lsq:bgps ?bgps . ?sf  lsq:tps ?tps .
?sf   lsq:joinVertices ?joinVertices .
?sf   lsq:maxBGPTriples ?maxBGPTriples .
?sf   lsq:minBGPTriples ?minBGPTriples .
?sf   lsq:meanJoinVertexDegree ?meanJVD .
?sf   lsq:usesFeature  ?feature
FILTER (str(?sub) != str(?sup) ) }
```

by using the SPARQL query given in Listing 2. The result of this query execution is stored in a map that is used in the subsequent benchmark generation steps. In Section 4.4, we show how this query can be modified to generate customized query containment benchmarks.

4.3.2 Normalized Feature Vectors. The cluster generation algorithms (explained in the next section) require distances between queries to be computed. To this end, each of the queries from the input LSQ dataset is mapped to a vector of length 9 which stores the corresponding *query features* discussed in Section 4.2 (Min. and Max. BGPs from Section 4.2 are two features). Assuming the number of sub-queries is 2, the feature vector for the query given in Figure 1 is [2, 2, 1, 5, 5, 5, 3, 2.33, 2]. To ensure that dimensions with high values (e.g., the number of LSQ features) do not bias the selection of queries for benchmarking, we normalize the query feature vectors with values between 0 and 1. This is to ensure that all queries are located in a unit hypercube. To this end, each of the individual values in every feature vector is divided by the overall maximal value (across all the vectors) for that query feature. Suppose the values [10, 8, 6, 12, 5, 10, 10, 5, 30] represent the maximal feature vector (i.e., each individual feature value is the maximum across all vectors), the normalized feature vector for the query given in Figure 1 becomes [0.2, 0.25, 0.16, 0.41, 1, 0.5, 0.33, 0.46, 0.06].

4.3.3 Generation of Clusters. As a next step, we generate N clusters from the given input LSQ queries represented as normalized feature vectors. For this step we used 5 existing well-known algorithms – FEASIBLE, FEASIBLE-Exemplars, KMeans++, DBSCAN+KMeans++, Random selection – which allow the generation of the required fixed number of clusters. Note DBSCAN+KMeans++ means that we applied DBSCAN first to remove the outlier queries and then applied KMeans++ to generate the required number of clusters. Note that we need an additional normalization of the remaining vectors after outliers are removed. Moreover, note that our framework is flexible enough to integrate any other clustering algorithm which allow the generation of a fix number of clusters.

Our framework provides a detailed analysis (explained in Section 5.1) of the benchmarks generated using different methods, thus allowing users to pick the best method for the required containment benchmark.

4.3.4 Selection of Most Representative Queries. Finally, we perform the selection of a single prototypical query from each cluster. This step is exactly the same as performed in FEASIBLE: For each cluster S, compute the centroid c which is the average of the feature vectors of all the queries in S. Following this, compute the distance of each query in S with c and select the query of minimum distance to include in the resulting benchmark.

Note that our framework also allows the generation of benchmarks using random selection. In addition, it allows the generation of benchmarks using Agglomerative clustering. However, Agglomerative clustering does not allow the creation of fixed size benchmarks. The aforementioned benchmark website contains the CLI options for the generation of benchmarks.

4.4 Benchmark Personalization

As mentioned before, our framework allows customized benchmark generation according to the criteria specified by the user. This can be done by simply specializing the query given in Listing 2. For example, imagine the user wants to generate customized benchmarks with the following features: The number of projection variables in the super-queries should be at most 2. The number of BGPs should be greater than 1 or the number of triple patterns should be greater than 3. The benchmark should be selected from the most recently executed 1000 queries, rather than considering the whole LSQ queries for benchmarking.

The query for the selection of such a personalized benchmark is given in Listing 3. Note that LSQ datasets also contain agent information, thus creating a query containment benchmark from a single specific agent queries is also possible. This can be particularly helpful when developing caching mechanisms or optimizing SPARQL queries for a particular user/system.

Listing 3: Benchmark personalization

```
Prefix lsq: <http://lsq.aksw.org/vocab#>
Prefix prov: <http://www.w3.org/ns/prov#>
SELECT DISTINCT ?sup count(?sub) as ?
    entailments ?projVars ?bgps ?tps ?
    joinVertices ?maxBGPTriples ?
    minBGPTriples ?meanJvd count(?feature)
    as ?noFeatures {
?sub   lsq:isEntailed ?sup .
?sup   lsq:hasStructuralFeatures ?sf .
?sf    lsq:projectVars ?projVars ;
lsq:bgps ?bgps ; lsq:tps ?tps ;
lsq:joinVertices ?joinVertices ;
lsq:maxBGPTriples ?maxBGPTriples ;
lsq:minBGPTriples ?minBGPTriples ;
lsq:meanJoinVertexDegree ?meanJVD ;
lsq:usesFeature  ?feature .
FILTER (str(?sub) != str(?sup))
# ——————— Personalization of the benchmark ———————
?sup   lsq:hasRemoteExec ?rExe .
?rExe  prov:atTime ?exeTime .
FILTER(?projVars <=2 && (?bgps > 1 || ?tps
    >3)) }
ORDER BY DESC(?exeTime) Limit 1000
```

5 EVALUATION AND RESULTS

Our evaluation comprises two main parts. First, we provide an analysis of the benchmarks generated by the available SQCFramework's methods. In the second part of our evaluation, we use the selected benchmarks generated by our framework to compare existing SPARQL query containment solvers w.r.t. their query containment precision and the execution time required to perform the query containment tasks.

5.1 SQCFramework Benchmarks Analysis

A benchmark should not be solely comprised of query containment tests that are very similar in their features. Sufficient diversity in the benchmark queries is important to ensure the overall quality of the generated benchmark. To this end, we analyze the benchmarks generated by our framework in terms of: 1) how well they represent the input LSQ queries, and 2) the diversity of the benchmark queries generated by our framework. From the point of view of statistics, the first measure is equivalent to the selection of a sample from a population that has the characteristics (here mean and standard deviation) similar to those of the original population. Thus, like FEASIBLE, we measure how much the mean and standard deviation of the features of benchmark queries deviate from those of the input LSQ queries. We call μ_i the mean and σ_i the standard deviation of a given distribution w.r.t. 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

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

Given that the harmonic mean is biased towards smaller values, we define a composite error estimation E (also called similarity error) by using the harmonic mean of $(1 - E_\mu)$ and $(1 - E_\sigma)$:

$$E = 1 - \frac{2(1 - E_\mu)(1 - E_\sigma)}{((1 - E_\mu) + (1 - E_\sigma))}. \quad (2)$$

The diversity score D is the average standard deviation of the query features k included in the benchmark B:

$$D = \frac{1}{k}\sum_{j=1}^{k}(\sigma_i(B). \quad (3)$$

Note that for a given benchmark, the smaller the composite error the better it represents the overall input query log, and the higher the diversity score, the more diverse the queries included in the benchmark. It is important to note that we are using the normalized features values while computing E and D, thus the values lie between 0 and 1.

5.2 Experiment Setup

LSQ datasets: For the experimental evaluation in this work, we used Semantic Web Dog Food (SWDF) and DBpedia LSQ datasets. Other LSQ datasets (a total of 20) are available from the project website and can be used in the benchmark generation after adding the query containment relationships. We chose the SWDF and DBpedia LSQ datasets because they were used in the FEASIBLE [12] evaluation.

Benchmarks for Composite Error and Diversity Analysis: In order to compare the quality of the benchmarks generated by the different SQCFramework methods, we generated benchmarks of sizes (number of super-queries) 15, 25, 50, 75, 100, and 125 from SWDF and benchmarks of sizes 2, 4, 6, 9, 12, 15 from DBpedia. The SWDF pattern of benchmarks was used in FEASIBLE as well. In DBpedia we sought to generate small sized benchmarks in order to allow users to perform a quick comparison of the query containment solvers based on a reasonably small number of containment tests (see Table 1). In addition, we wanted to compare the clustering algorithms for various benchmark sizes.

Benchmark generation methods: We generated the above mentioned benchmarks using (1) FEASIBLE, (2) KMeans++, (3) DBSCAN+KMeans++, (4) Random selection, and (5) FEASIBLE-exemplars and provide a detailed analysis of their composite errors and diversity scores.

SPARQL containment solvers: We evaluated four SPARQL query containment solvers: (1) SPARQL Algebra, (2) AFMU, (3) TreeSolver, and (4) JSAC. To the best of our knowledge, these are the state of the art available SPARQL containment solvers. We run each of the query containment solvers in its default settings.

Benchmarks for Containment Evaluation: We selected a 10 super-queries benchmark generated by KMeans++ from SWDF LSQ dataset. This benchmark contains a total of 1192 SPARQL queries containment tests (ref., see Figure 2).

Hardware: We performed all of the experiments on an Ubuntu 16.04.02 LTS machine with 16GB RAM DDR4 and a 2.4GhZ Intel I7-7700HQ CPU.

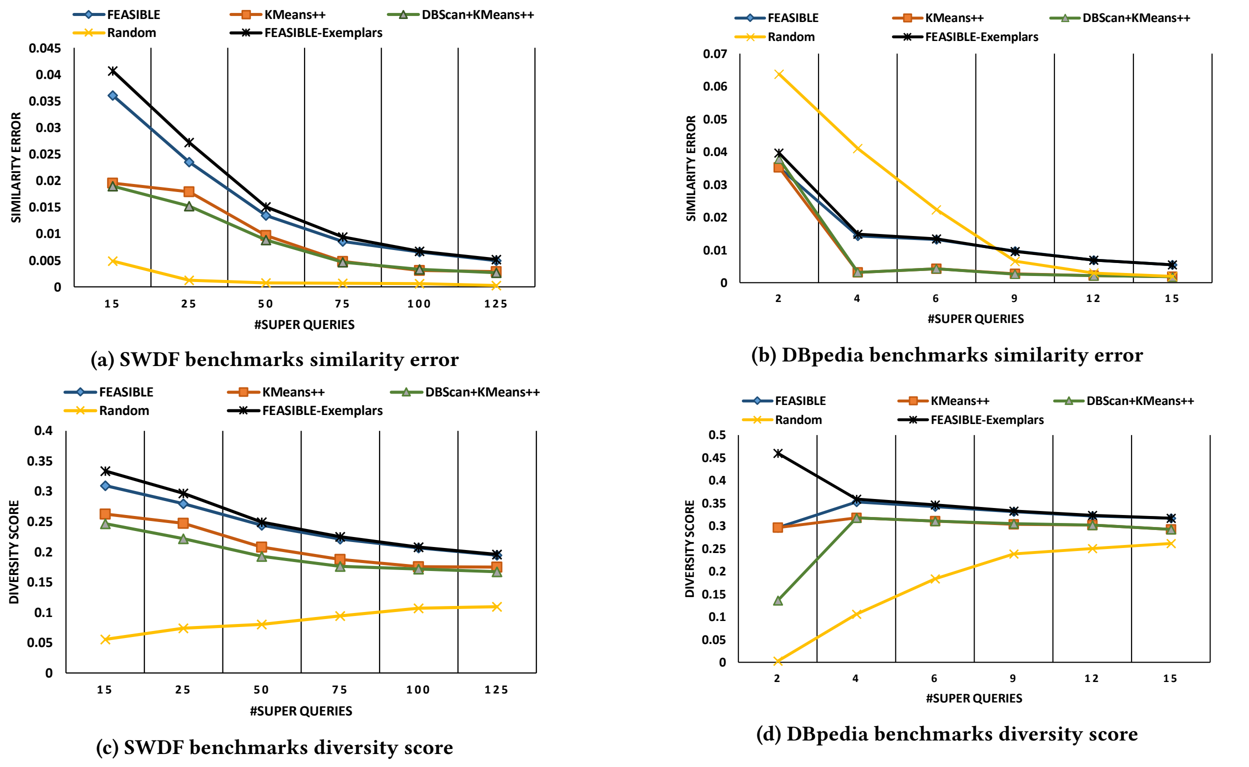
Metrics: As mentioned, to analyze the benchmarks generated by our framework, we define the similarity error and the diversity score. For the comparison of the SPARQL query containment solvers we choose two standard metrics: (1) Query Mixes per Hour (QMpH) along with the standard deviation, and (2) the precision of the containment solvers in terms of the number of correctly identified containment tests. This metric denotes how long a particular containment solver takes to check the query containment between two queries. This metric is generally used in the triple store's benchmark evaluation [12]. The second metric evaluates the accuracy of the containment solvers. Note that each benchmark is called a test suite or a query mix. We set the query timeout to 5 seconds. The query was aborted after that and maximum time of 5 seconds (sufficient for 99% of queries) was used as the query containment runtime for all queries which times out. All the data (codes, benchmarks, results) to repeat our experiments along with complete evaluation results are available at the project website.

5.3 Experiment Results

5.3.1 Composite Error and Diversity. Figure 2 shows the composite errors and the diversity scores for the different benchmarks generated from SWDF and DBpedia using the SQCFramework's currently supported methods. In terms of similarity/composite error shown in Figure 2a and Figure 2b, the DBSCAN+KMeans++'s selection outperforms the other methods. DBSCAN+KMeans++'s overall (across all the generated benchmarks) similarity error is 2% smaller than KMeans++ which is 28% smaller than Random selection which in turn is 18% smaller than FEASIBLE which is 9% smaller than FEASIBLE-Exemplars. In terms of diversity scores shown in Figure 2c and Figure 2d, the FEASIBLE-Exemplars outperforms the other methods. FEASIBLE-Exemplars' overall (across all the generated benchmarks) diversity score is 4% larger than FEASIBLE which is 21% larger than KMeans++ which in turn is 6% larger than DBSCAN+KMeans++ which is 47% larger than Random selection.

In summary, we have the following key observations: (1) If the aim is to test the query containment solvers with a benchmark that best reflects the overall input query log then DBSCAN+KMeans++ is the best method. However, the downside of this approach is that the benchmark queries will not be sufficiently diverse. This means it is highly possible that many of the tests are too easy or may even bias the overall results for a best suited query containment solver. DBSCAN+KMeans++'s smaller error might be due to the fact that DBSCAN first removes the outlier queries from the population, leading KMeans++ to select more representative samples for the distribution. (2) If the aim is to test the query containment solvers with a benchmark that contains diverse tests then FEASIBLE-Exemplars is the best method. The reason for FEASIBLE-Exemplars' highest diversity scores is the method it follows to select the exemplars: first it selects the middle query as first exemplar in the multi dimensional space. The second exemplar is the query that has the longest distance from the first exemplars. Similarly, the third exemplar is the one that has the longest distance from the first two exemplars and so on. This means FEASIBLE-Exemplars is a method that will always give the best possible diverse queries benchmarks. However, on the down side this benchmark may not well-reflect the overall query log. (3) Our final observation is interesting: the two measures are inverse to each other using cluster-based selection. If a method is good in similarity error it performs poor in diversity score and vice versa. The reason for this is that selecting outliers are always more diverse but they do not represent the normal distribution. However, this might not be true for random selection.

Table 1 shows the number of query containment tests and the corresponding benchmark generation time for different benchmark sizes of SWDF and DBpedia LSQ datasets. Note that we refer the number of super-queries to be the size of the benchmark. We can see the number of containment tests (in which two queries are checked for containment) for each of the benchmarks is significantly larger than the number of super-queries. This is simply because each super-query can have multiple sub-queries and we include all sub-queries for a given super-query in the resulting benchmark. Overall, the clustering methods have comparable benchmark generation time.



(a) SWDF benchmarks similarity error

(b) DBpedia benchmarks similarity error

(c) SWDF benchmarks diversity score

(d) DBpedia benchmarks diversity score

Figure 2: Comparison of the benchmarks generated by our framework in terms of similarity error and diversity score

Table 1: Comparison of the number of containment tests (#T) and the benchmark generation time (G in seconds) for different benchmark sizes of SWDF and DBpedia. (KM++ = KMeans++, DB = DBSCAN,FSBL = FEASIBLE,Ex = Exemplars #Q = number of super-queries or the benchmark size)

	SWDF										DBpedia										
	FSBL		KM++		DB+KM++		Random		FSBLE-Ex			FSBL		KM++		DB+KM++		Random		FSBLE-Ex	
#Q	#T	G	#T	G	#T	G	#T	G	#T	G	#Q	#T	G	#T	G	#T	G	#T	G	#T	G
15	1739	6	1694	7	906	5	408	1	1877	2	2	9	1	9	1	34	1	4	1	36	1
25	2433	7	1902	7	1134	6	179	1	2721	2	4	40	1	55	1	55	1	38	1	38	1
50	4563	7	2153	7	1372	7	400	1	4530	2	6	74	2	64	1	64	1	11	1	72	1
75	4716	7	2457	7	1972	7	496	1	4697	2	9	126	2	105	1	102	1	117	1	123	1
100	4863	8	3009	7	2218	7	1304	1	4978	2	12	163	2	140	1	111	1	151	1	160	1
125	5149	9	3541	7	3083	7	1522	1	5153	2	15	197	2	146	1	146	1	180	1	195	1

5.3.2 SQC-Bench Vs. SQCFramework. As shown in Figure 2, FEASIBLE-Exemplar generates the most diverse benchmarks. In Figure 3, we compare the diversities of the FEASIBLE-Exemplar with SQC-Bench across the 9 selected features. This analysis is based on the 27 super queries benchmark. This is because SQC-Bench contains a total of 27 unique super queries. Thus we generated 27 super queries benchmark using FEASIBLE-Exemplar. As an overall diversity score (i.e., the last bar of Figure 3), our approach generated more diverse benchmark (0.29 vs. 0.24 diversity score) in comparison with the SQC-Bench. Across the individual features, the benchmark generated by our framework is more diverse than SQC-Bench in 7/9 individual features. Only in number of projection variables and number of BGPs the SQC-Bench standard deviation is higher than our approach.

5.3.3 Containment Solvers Evaluation Result. Table 2 shows a comparison of the selected query containment solvers for a benchmark of 10 super-queries generated from SWDF by using the KMeans++ clustering algorithm. Overall, JSAC clearly outperforms other solvers as it is the only isomorphism-based one and can handle all 1192 test cases. The QMpH of AFMU is about twice as larger of JSAC, suggesting it can quickly check the query containment test cases. However, since TS, AFMU and TreeSolver can only handle at most 5/1192 test-cases, the QMpH rate rather suggests how fast

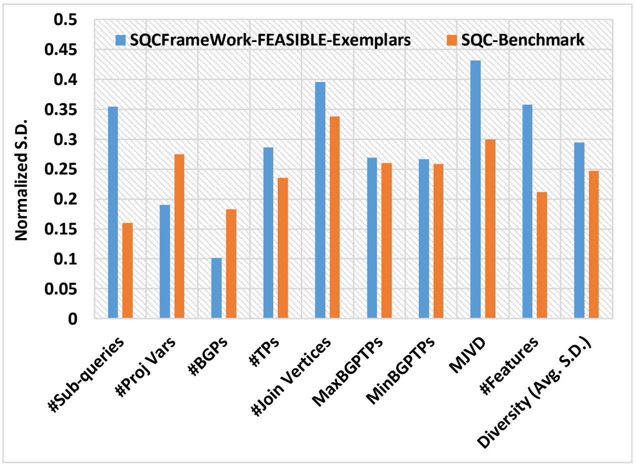


Figure 3: SQC-Bench Vs. SQCFramework: Diversity analysis across the selected features. #Proj Vars = Number of projection variables,#TPs = Number of triple patterns, MJVD = Mean Join Vertex Degree

these solvers can rule out supporting a containment check instead of indicating the actual performance on containment checks.

In comparison to QC-Bench, our results are rather surprising: the solvers (i.e., SPARQL-Algebra, TreeSolver, AFMU) that were able to handle the majority of the QC-Bench test cases are not able to handle more than 5 SQCFramework's generated test-cases. Consequently, this means that the SPARQL query containment that occurs in reality in query logs is different from the query containments presented in QC-Bench: In practice, non-isomorphic containments turn out to be much more scarce.

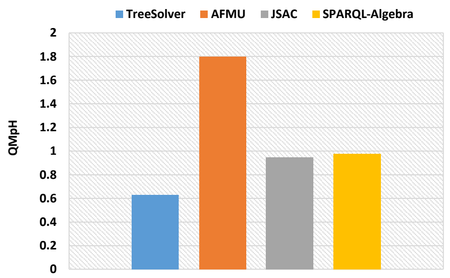


Figure 4: Query Mixes per Hour (QMpH).

Solver	#TT	#HT	#CT	#TO
TreeSolver	1192	5	5	2
AFMU	1192	5	5	12
SPARQL-Algebra	1192	0	0	0
JSAC	1192	1192	1192	0

Table 2: Containment solvers comparison. #TT = total test cases, #HT = number of handled test cases, #CT = number of correct test cases, #To = number of timed out test cases.

6 CONCLUSION

In this paper we presented SQCFramework, a SPARQL containment benchmark generation framework. Our framework allows users to generate customized benchmarks suited for a particular use case, which is important to test the query containment solvers pertaining to a given application. SQCFramework allows the generation of benchmarks using different approaches. We compared the quality of the benchmarks generated using these approaches in terms of their composite error and diversity score. It turns out that DBSCAN+KMeans++ generates the most representative and FEASIBLE-Exemplars generates the most diverse benchmarks compared to other methods used in our framework. We also compared existing SPARQL query containment solvers using the benchmarks generated by our framework. JSAC accuracy is better than other selected solvers. In addition, JSAC is able to check the containment test in a reasonable amount of time. In future work, we will generate benchmarks from other LSQ datasets. We will also consider adding other clustering methods into the framework. A further milestone will be to identify other query features (e.g., the result size of the query) that should be considered while generating the query containment benchmarks.

6.1 Acknowledgements

This work has been supported by the project HOBBIT (GA no. 688227), LIMBO (no. 19F2029I) and OPAL (no. 19F2028A) as well as by Science Foundation Ireland (SFI) under Grant No. SFI/12/RC/2289.

REFERENCES

[1] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. 2012. SPARQL query containment under RDFS entailment regime. In *International Joint Conference on Automated Reasoning*. Springer, 134–148.
[2] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. 2013. Evaluating and benchmarking SPARQL query containment solvers. In *International Semantic Web Conference*. Springer, 408–423.
[3] Melisachew Wudage Chekol and Giuseppe Pirrò. 2016. Containment of expressive SPARQL navigational queries. In *International Semantic Web Conference*. Springer, 86–101.
[4] Chandra Chekuri and Anand Rajaraman. 2000. Conjunctive query containment revisited. *Theoretical Computer Science* 239, 2 (2000), 211–229.
[5] Claus et al. 2017. JSAC Query Containment Solver. https://github.com/AKSW/jena-sparql-api/tree/develop/jena-sparql-api-query-containment. (2017). [Online; accessed 16-May-2017].
[6] Pierre Geneves, Nabil Layaïda, and Alan Schmitt. 2007. Efficient static analysis of XML paths and types. *Acm Sigplan Notices* 42, 6 (2007), 342–351.
[7] Egor V Kostylev, Juan L Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with property paths. In *International Semantic Web Conference*. 3–18.
[8] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. 2013. Static analysis and optimization of semantic web queries. *TODS* 38, 4 (2013), 25.
[9] Todd Millstein, Alon Halevy, and Marc Friedman. 2003. Query containment for data integration systems. *J. Comput. System Sci.* 66, 1 (2003), 20–39.
[10] Reinhard Pichler and Sebastian Skritek. 2014. Containment and equivalence of well-designed SPARQL. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 39–50.
[11] Muhammad Saleem, Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: The Linked SPARQL Queries Dataset. In *ISWC*. 261–269.
[12] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. FEASIBLE: a feature-based SPARQL benchmark generation framework. In *ISWC*. Springer, 52–69.
[13] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. 2014. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In *ESWC*.
[14] Yoshinori Tanabe, Koichi Takahashi, Mitsuharu Yamamoto, Akihiko Tozawa, and Masami Hagiya. 2005. A decision procedure for the alternation-free two-way modal μ-calculus. In *ARATRM*. Springer, 277–291.
[15] Melisachew Wudage, Jérôme Euzenat, Pierre Genevès, and Nabil Layaıda. 2012. SPARQL query containment under SHI axioms. In *Proceedings 26th AAAI Conference on Artificial Intelligence*. 10–16.