QED: Out-of-the-Box Datasets for SPARQL Query Evaluation

Veronika Thost[1(✉)] and Julian Dolby[2]

[1] MIT-IBM-Watson AI Lab, IBM Research, Cambridge, MA, USA
veronika.thost@ibm.com
[2] IBM Research, Yorktown Heights, NY, USA
dolby@us.ibm.com

Abstract. In this paper, we present SPARQL QED, a system generating out-of-the-box datasets for SPARQL queries over linked data. QED distinguishes the queries according to the different SPARQL features and creates, for each query, a small but exhaustive dataset comprising linked data and the query answers over this data. These datasets can support the development of applications based on SPARQL query answering in various ways. For instance, they may serve as SPARQL compliance tests or can be used for learning in query-by-example systems. We ensure that the created datasets are diverse and cover various practical use cases and, of course, that the sets of answers included are the correct ones. Example tests generated based on queries and data from DBpedia have shown bugs in Jena and Virtuoso.

Keywords: SPARQL datasets · Compliance tests · Benchmark

1 Introduction

The SPARQL query language is widely used and probably the most popular technology when it comes to querying linked data. Most triple stores and graph databases support the user-friendly declarative query language [7,22,23]. There are several benchmarks targeting the performance of SPARQL query answering, amongst others [3,14,18,19]. But, to the best of our knowledge, for SPARQL compliance testing, the W3C compliance tests[1] are the only test suite publicly available and commonly applied [2,16]. They have been proposed originally in 2001 for SPARQL 1.0 and were extended in 2009 regarding the new features of SPARQL 1.1. Correctness tests differ from benchmarks targeting scalability: the queries cover the various SPARQL features, the data is rather small – so that SPARQL engine developers can easily trace bugs and analyze the processing –, and the correct answers to the queries are included.

However, the W3C tests mostly contain synthetic queries over similarly artificial example data and, especially, comprise only few more complex queries

[1] https://www.w3.org/2009/sparql/docs/tests/.

P. Hitzler et al. (Eds.): ESWC 2019, LNCS 11503, pp. 491–506, 2019.
https://doi.org/10.1007/978-3-030-21348-0_32

nesting different SPARQL features, which model real user queries more faithfully. A simple text search reveals, for example, that the `UNION` keyword only occurs in nine[2] rather simple `SELECT` queries, such as the following query Q.[3]

```
SELECT * WHERE {
 ?city a <http://dbpedia.org/ontology/Place>; rdfs:label 'Gomeciego'@en.
 ?airport a <http://dbpedia.org/ontology/Airport>.
 {?airport <http://dbpedia.org/ontology/city> ?city} UNION
 {?airport <http://dbpedia.org/ontology/location> ?city} UNION
 {?airport <http://dbpedia.org/property/cityServed> ?city.} UNION
 {?airport <http://dbpedia.org/ontology/city> ?city. }
 OPTIONAL { ?airport foaf:homepage ?airport_home. }
 OPTIONAL { ?airport rdfs:label ?name. }
 FILTER ( !bound(?name) || langMatches( lang(?name), 'de') ) }
```

Fig. 1. A SPARQL query from DBpedia.

```
SELECT DISTINCT * WHERE { { ?s :p ?o } UNION { ?s :q ?o } }
```

For a given dataset, this query retrieves all those tuples (`s`, `o`) for which the data contains either the triple `s :p o`, or `s :q o`, or both `s :p o` and `s :q o`. We assume the reader to be familiar with SPARQL. The concept of matching query patterns and triples for obtaining answers, which are *solution mappings* (also *solutions*) of the query variables to terms, is defined formally in the SPARQL specification[4]. In the W3C tests, `UNION` occurs only together with the `GRAPH` or `OPTIONAL` key and once with `FILTER`, but with none other. Naturally, handcrafted tests cannot cover all possible combinations of features. But an example from the DBpedia query log depicted in Fig. 1 shows that real queries often contain various nested features in combination.[5] We thus have a considerable gap between the queries in the tests and those in reality. And this is similar with the data. While the test data for Q consists of the below three triples, the latest DBpedia dump contains more than 13 billion triples[6]. These triples usually cover various *situations* modeled in queries (i.e., distinct and sometimes only partial instantiations of the query patterns).

```
:x1 :p "abc". :x1 :q "abc". :x2 :p "abc".
```

Here, we only have triples on the single literal `"abc"`, but not the situation that the pattern `?s :q ?o` is satisfied alone.

[2] The 2009 tests actually contain some more such queries, but these consider an empty dataset and hence are rather unrealistic.
[3] For readability, we generally drop prefix declarations.
[4] https://www.w3.org/TR/sparql11-query/.
[5] We obtained the query from LSQ: http://aksw.github.io/LSQ/.
[6] http://wiki.dbpedia.org/develop/datasets/dbpedia-version-2016-10.

As a consequence of this mismatch, it is likely that many endpoints that pass the W3C compliance tests exhibit non-standard behavior in practice. In fact, later in the paper, we show that this is the case even for endpoints like Jena and Virtuoso, which are heavily used in practice. More realistic compliance tests could also serve as evaluation datasets for developing add-ons for auto-completion or query suggestion based on examples (i.e., answer samples), and hence enhance the support for users; observe that standard benchmarks do not fit these tasks. In short, we are missing test datasets which – in contrast to the existing performance benchmarks – contain real-world queries, diverse and comprehensive samples of data (i.e., triples), and the answers to the queries.

In this paper, we present the SPARQL Query Evaluation Dataset generator (QED), which closes the aforementioned gap by generating out-of-the-box datasets for SPARQL queries over linked data (note that the approach similarly works for local RDF data). QED distinguishes queries given by the user according to the different SPARQL features, selects some of them, and creates a test case for each query, a dataset[7] comprising linked data and the query answers over this data. Thereby, it is ensured that the created datasets are small, but diverse, and cover various practical use cases and, of course, that the sets of answers included are the correct ones. QED is available at https://github.com/vthost/qed.

The paper is structured as follows. Section 2, describes applications for our datasets and corresponding requirements. In Sect. 3, QED is presented in detail, and Sect. 4 describes example datasets generated based on DBpedia and Wikidata, including a test that revealed a bug in both Jena and Virtuoso.

2 Motivation

In this section, we outline use cases for QED and specify the corresponding requirements for (or features of) the system. While some of the existing benchmarks targeting performance already rely on real-world queries [14,18], the W3C test suite does not do so. By integrating not only real queries, but also linked data and the corresponding query answers over this data, we hence augment – or rather complement – the two kinds of existing benchmarks. In particular, we open the door for various applications. First of all, the generated tests may serve as SPARQL compliance tests. But they can also support the development and optimization of other systems, for instance: query-by-example approaches [6,8,12,15], where queries are to be learnt from a small set of tuples of URIs that are or, especially, are not among the queries' answers; and auto-completion add-ons [11]. These approaches need test data that is *relevant* for the corresponding queries; that is, data containing positive and negative examples (i.e., answers and approximate answers) instead of arbitrary triples. Our goal was

[7] The term "datasets" usually denotes the sets of data included in the test cases. However, we may also use it for the entire test cases consisting of a query, data, and answers (e.g., in the acronym QED); this is to emphasize that they can be applied for multiple purposes, besides correctness testing, and should be clear from context.

therefore to keep QED modular, parameterizable, and extensible, in order to allow for as many applications as possible. Specifically, we derived the following requirements.

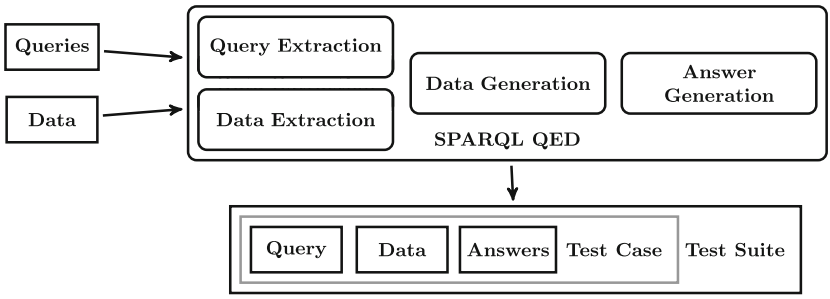

Fig. 2. The architecture of SPARQL QED. The input consists of a query set and the data, or the address of an endpoint providing it. The application first extracts queries from the set based on their features. Then, a test case is created for each of them by creating a dataset comprising both data from the input and generated, synthetic data, and by computing the answers over this data. This test suite is the output.

R1 Originality. QED should fill the gaps in existing benchmarks and produce datasets comprising diverse query and data samples from real applications, and corresponding sets of answers. The queries should cover different SPARQL features: the solution modifiers, keywords associated to algebra operators, and the aggregation operators. The data should be relevant for the corresponding query, cover various situations, and be of reasonable size.

R2 Reusability. QED should be applicable to arbitrary SPARQL queries and corresponding data, and hence be kept as general as possible. It should also be modular and open for and adaptable to various use cases. Next to that, the system should be easy to handle.

R3 Quality. The QED approach is based on existing query logs and linked data and hence integrates existing resources. The system design should further follow common practices (e.g., use Semantic Web data formats). Beyond that, the correctness of the produced datasets should be formally verified.

These aspects capture the scope of QED we envision. Regarding **R1**, note that the quality and variety of the generated datasets (e.g., the coverage of SPARQL features) strongly depends on the input, especially on the queries. However, we will propose means to filter the queries and generate data to augment the one given, and further discuss the topic later, in Sect. 3.

3 SPARQL QED

SPARQL QED is a framework for generating datasets for SPARQL query evaluation as outlined in Fig. 2. The input consists of an endpoint providing the data,

and queries over that data (e.g., from a log). QED distinguishes the given queries according to different SPARQL features, selects some of them, and creates, for each query, a dataset comprising comprehensive samples of the linked data (i.e., triples) and the query answers over this data. Thereby, it is ensured that the created datasets are small, but diverse, that they cover various practical use cases and, of course, that the sets of answers included are the correct ones.

1. Query Extraction. The queries are assumed to be given in the LSQ RDF format [17], which contains the SPARQL queries themselves, but also captures specific characteristics, such as number of triples, features contained in them (e.g., the `OPTIONAL` or `FILTER`), and number of answers. This allows query extraction to be configurable accordingly. For instance, the configuration $(2, 1, \{\{\texttt{OPTIONAL}, \texttt{FILTER}\}, \{\texttt{UNION}\}\}, 10)$ makes QED construct a test suite containing 10 test cases with queries that contain both `OPTIONAL` and `FILTER` (and possibly other features), and 10 test cases with queries that contain `UNION`; and all of the queries contain at least two triples and have at least one answer. Note that there is a tool for transforming SPARQL queries into this format relatively easily.[8] We also have filters that ensure that the extracted queries are diverse and do not contain near-duplicates as generated by bots, for example. In fact, it turned out that the majority of such queries can be recognized by comparing the beginnings of the textual representations of the queries for equality.

The LSQ format capturing different query characteristics allows us to extract relevant sets of queries from possibly large query logs. However, in the end, we are only interested in the SPARQL queries. That is, if the user has a SPARQL query set of reasonable size for which tests should be created, they can also use this set and hence skip the query extraction phase. Note that this option also leaves the user the possibility to apply different query extraction approaches proposed in the literature, such as [19], or to use the queries from other benchmarks.

2. Data Extraction. The sheer number of triples given as input often makes it impossible to include all the relevant data from the endpoint since the test cases should be of reasonable size. On the other hand, the tests should not be too simplistic (i.e., with data covering only some of the situations modeled in the query, such as the one given for Q in the introduction). We therefore do not only take a subset of the relevant data that is restricted in size (i.e., the maximal size is specified by the user) but also ensure that it reflects the variability of the possible answers. More specifically, we extract data describing different situations (if available) depending on the shape of the query and covering the possible types of solutions described in the SPARQL specification:

- Q_1 `UNION` Q_2: data leading to (1) *compatible* solutions to subqueries Q_1 and Q_2 (i.e., solutions that map shared variables to the same term), (2) solutions to Q_1, and to (3) solutions to Q_2.
- Q_1 `OPTIONAL` Q_2: data leading to solutions as in (1)–(3), but we additionally ensure in (3) that the solutions to Q_2 are incompatible to the ones from (2). Q_1 `MINUS` Q_2 and Q_1 `FILTER(NOT)EXISTS` Q_2 are treated in the same way.

[8] http://aksw.github.io/LSQ/.

```
SELECT ?name ?description_en ?musician WHERE {
 ?musician skos:subject cat:German_musicians .
 ?musician foaf:name ?name .
 OPTIONAL {
  ?musician rdfs:comment ?description_en .
  FILTER (LANG(?description_en) = 'en') . } }
```

```
dbpedia:Laith_Al-Deen
rdfs:comment  "Laith Al-Deen, born February 20, 1972 in Karlsruhe,
Germany, is a German language pop musician."@en ;
skos:subject  ns3:German_musicians ;
foaf:name     "Laith Al-Deen"@de .

dbpedia:Kettcar  rdfs:comment  "Kettcar is an indie pop music band
based in Hamburg, Germany."@en ;
skos:subject  ns3:German_musicians ;
foaf:name     "Kettcar"@en .

dbpedia:Lou_Bega skos:subject  ns2:German_musicians ;
foaf:name     "Lou Bega"@en , "Lou Bega"@de ;
rdfs:comment  "Lou Bega je německý zpěvák latinskoamerické hudby, jenž
se proslavil zejména díky písni Mambo No. 5."@cs ,
"Lou Bega ist ein deutscher Latin-Pop-Sänger."@de ,
"David Lubega (born April 13, 1975), also known as Lou Bega ... ."@en .
```

```
<http://gen/0> <http://www.w3.org/2004/02/skos/core#subject>
<http://dbpedia.org/resource/Category:German_musicians> .
<http://gen/0> <http://xmlns.com/foaf/0.1/name> <http://gen/1> .
```

Fig. 3. Example DBpedia query Q_{ex}, data D_{ex}, and additionally generated data D_{gen}.

– Q_1 `FILTER` E: data leading to (1) solutions to Q_1 and for which E is true, and (2) solutions to Q_1 and for which E is false.

If Q_1 or Q_2 again contain such patterns, then we consider all combinations. The extraction is done using SPARQL `CONSTRUCT` queries describing the situations.

Example 1. For query Q_{ex} in Fig. 3, abstracted Q_1 `OPTIONAL` $\{Q_2$ `FILTER` $E\}$, we look for data about German musicians covering all the following situations.

– There is a comment in English
(i.e., there are compatible solutions to Q_1 and Q_2 yielding that E is true; they are merged into a solution to Q_{ex}).
– There is a comment that is not in English
(i.e., there are compatible solutions to Q_1 and Q_2 yielding that E is false; the former represents a solution to Q_{ex}).

– There is no comment – then the language filter is irrelevant for the result (i.e., there is a solution to Q_1 but none or no compatible one to Q_2 `FILTER` E; the former represents a solution to Q_{ex}).

The data we have extracted from DBpedia is also shown in Fig. 3. Observe that we did not retrieve data for the last case; there is no musician without comments. On the other hand, our consideration of various situations makes that the data may also contain approximate answers; for instance, comments in different languages.

Again, the data does not have to be extracted from an endpoint, the user also can provide it directly. Nevertheless, independent of where the data comes from, and as the example also demonstrates, we cannot assume that we have all the different kinds of data. In fact, the portion of the provided data matching a query usually only covers a few of the possible situation; in Sect. 4, we show this in more detail. To tackle this problem, we generate synthetic data for the remaining cases and integrate it with the real data.

3. Data Generation. For those of the above cases where data is not provided but which could arise theoretically, we generate synthetic data. We use Kodkod[9], a SAT-based constraint solver for extensions of first-order logic with relations to construct the dataset we target. In a nutshell, we encode every situation we consider for a given query into a formula Q in relational logic and represent the answers over the unknown data D in a relational logic formula $Ans(Q, D)$ as proposed in [5]. Our final formula is then the equation $Ans(Q, D) = \neg\emptyset$, requiring the answer set to be non-empty, and can be solved by Kodkod, yielding a dataset D as intended (i.e., satisfying Q); $\neg$ reads "not" and $\emptyset$ denotes the empty set.

Example 2. For query Q_{ex}, we consider amongst others the formula $Q := Q'_1 \wedge \neg(Q'_2 \wedge E')$; $\wedge$ reads "and", and X' denotes the relational logic translation of X. It represents the last case in Example 1 partly, namely that there is no compatible solution to Q_2 `FILTER` E. Our data D_{ex} from DBpedia contains no answers to Q because there is an English comment for all the musicians, and hence $Ans(Q, D_{ex}) = \neg\emptyset$ is not satisfied. If we consider the data to be a variable D_{gen}, then Kodkod can find and generate it as a solution to the equation $Ans(Q, D_{gen}) = \neg\emptyset$. Such data is shown in Fig. 3 (bottom).

The generated data is then added to the real data. Note that, in theory, data generated in this way would be sufficient for testing since the generation systematically considers the relevant test cases based on the queries' features. However, our generator currently does not yet cover all possible features because Kodkod is lacking full datatype support (e.g., for filters on dates, QED does not generate triples with different dates relevant for the filter); hence the real data may add important test cases. Also observe that, if there are many situations for one query, the numbers of triples to be considered can become large, but QED can be configured to generate smaller data files, for single situations. Further, there are

[9] http://emina.github.io/kodkod/index.html.

```
[ a rs:ResultSet ;
  rs:resultVariable "musician" , "description_en" , "name" ;
  rs:size            "5"^^xsd:int ;
  rs:solution [
   rs:binding [ rs:value <http://dbpedia.org/resource/Laith_Al-Deen> ;
    rs:variable "musician"] ;
   rs:binding [ rs:value "Laith Al-Deen, born February 20, ..."@en ;
    rs:variable "description_en"] ;
   rs:binding [ rs:value "Laith Al-Deen"@de ;
    rs:variable "name"] ] ;
  rs:solution [
   rs:binding [ rs:value <http://dbpedia.org/resource/Kettcar> ;
    rs:variable "musician"] ;
  rs:binding [ rs:value "Kettcar is an indie pop music band ...."@en ;
    rs:variable "description_en"] ;
  rs:binding [ rs:value "Kettcar"@en ;
    rs:variable "name"] ] ;
  rs:solution [
   rs:binding [ rs:value <http://dbpedia.org/resource/Lou_Bega> ;
    rs:variable "musician"] ;
   rs:binding [ rs:value "David Lubega (born April 13, 1975), ...."@en ;
    rs:variable "description_en"] ;
   rs:binding [ rs:value "Lou Bega"@de ;
    rs:variable "name"] ] ;
  rs:solution [
   rs:binding [ rs:value <http://dbpedia.org/resource/Lou_Bega> ;
    rs:variable "musician"] ;
   rs:binding [ rs:value "David Lubega (born April 13, 1975), ...."@en ;
    rs:variable "description_en"] ;
   rs:binding [ rs:value "Lou Bega"@en ;
    rs:variable "name"] ] ;
  rs:solution [
   rs:binding [ rs:value <http://gen/0> ;
    rs:variable "musician"] ;
   rs:binding [ rs:value <http://gen/1> ;
    rs:variable "name"] ] ; ] .
```

Fig. 4. The solutions to the example query Q_{ex} over the union of data D_{ex} and D_{gen} from Fig. 3.

special cases where the query (and the corresponding formulas) is unsatisfiable or where we cannot integrate all the generated data because the formulas for different situations contradict each other. We ignore the former cases and then generate no data at all. In the latter cases, we generate multiple datasets greedily: we initially consider the maximal dataset (integrating both the extracted and the generated data) and split it iteratively until there are no contradictions anymore.

Example 3. It is safe to ignore a query Q `MINUS` Q since it has no answers, over any data. The second case is more subtle, but can be illustrated with the query:

```
SELECT ?a WHERE { :s1 :p :s2 . OPTIONAL { :s2 :p ?a . }}
```

For any dataset, it either retrieves no answers (if the data does not contain the triple `:s1 :p :s2`), one answer in which `?a` is not bound (if the data contains `:s1 :p :s2` but no triple starting with `:s2 :p`), or answers where `?a` is bound (the data contains the triple `:s1 :p :s2` and some starting with `:s2 :p`). Here, we generate a dataset that contains the triple `:s1 :p :s2` and one starting with `:s2 :p`, and a dataset including `:s1 :p :s2` but no triple starting with `:s2 :p`.

4. Answer Generation. For all queries and the corresponding data, we include the answers. However, we do not rely on a standard endpoint to retrieve the answers since we do not have correctness guarantees for those. Therefore we again use Kodkod to generate the answers similar as the data. More specifically, for each dataset, so far consisting of a query Q and data D (comprising extracted and generated data), Kodkod can generate a set of answers A such that the equation $A = Ans(Q, D)$ is satisfied. Note that the correctness of the system implementing the declarative semantics has been verified with the W3C tests.

Example 4. Figure 4, for instance, shows that every answer to Q_{ex} over D_{ex} has a binding for each `?name`, `?description_en`, and `?musician`, but the answer to Q_{ex} over D_{gen} is missing the optional binding for `?description_en`.

The created test suite is in the format of the W3C tests: a machine-readable custom format that relies on manifest files specifying the single test cases, each of which comes with one file for each, the query, the data, and the result, as depicted for the example query Q_{ex} in Figs. 3 and 4. The queries are in SPARQL, the others in Turtle format.

Discussion

We regard guidelines **R1**-**R3** from Sect. 2, compare the tests to those of the W3C, and discuss possible extensions.

QED indeed allows to generate diverse and extensive datasets for SPARQL query evaluation based on real queries and data. Since the input queries and data, and also the query features in focus can be selected arbitrarily, and the tests are of reasonable size, the system is highly generic and can be used in various scenarios by a wider community. QED does not only allow the integration of other approaches (e.g., for query extraction); since the Java implementation is kept simple, easy to handle, and open source, it is also further customizable. Finally, the system design follows best practices: we rely not only on existing queries and data, but integrate the LSQ approach and adopt the format of the W3C compliance tests. Note that the latter allows testers to reuse existing infrastructure (originally created for the W3C tests) to directly run the tests generated with QED. We also verify that the query answers are correct.

In the initial version, we concentrate on the `SELECT` query form and on evaluation tests. The standard also specifies the forms `ASK`, `CONSTRUCT`, and `DESCRIBE`.

Since the ASK form only tests whether or not a query pattern has a solution, our implementation for SELECT can easily be adapted. The CONSTRUCT query form, which returns a single RDF graph specified by a graph template, would require a slightly more elaborate extension of QED. The DESCRIBE form is irrelevant for common tests since it returns information that is to be determined by the query service applied and hence cannot be tested in a general fashion.

Further, our approach to make the data reflect several possible situations relevant for the considered query is currently only applied with operator features (e.g., OPTIONAL and UNION). There are other, rather intricate features, such as property paths or complex filter conditions, whose consideration during diversification would certainly benefit applications, but which need to be handled carefully. Note that queries with these features are included, but we do not extract or generate data that additionally includes approximate answers w.r.t. these features.

Next to query evaluation tests, the W3C test suite includes syntax and protocol tests. The former test if syntactically (in)correct queries yield exceptions or not, and we plan to consider them in the future, either by using the parse error property of LSQ or by integrating a SPARQL parser into QED. The others are however out of the scope of our work, whose goal is to generate realistic and extensive datasets for SPARQL query evaluation. Nevertheless, we do not suppose our datasets to represent complete compliance tests covering the whole range of SPARQL. They should rather be considered as additional tests tailored to the queries and data used in practice.

4 Dataset Examples

In this section, we first describe a test case that showed a bug in Jena and Virtuoso and thus demonstrates that the comprehensive consideration of various situations is critical for compliance testing. Then we give an overview of example test suites generated based on queries over and data from DBpedia and Wikidata.

A Test Case

The following query over DBpedia nicely illustrates that there are often various possibilities for situations in the data that lead to answers to given queries.

```
SELECT ?name ?musician ?band WHERE {
 ?musician foaf:name ?name .
 OPTIONAL { ?musician foaf:member ?band . }
 MINUS {
  ?x foaf:homepage "some bad page" .
  OPTIONAL { ?x foaf:member ?band . } } }
```

For this query, QED generated the below two triples (amongst others) describing a situation where the optional parts of the query do not match any data.

```
<http://gen/0> <http://xmlns.com/foaf/0.1/name> <http://gen/1> .
<http://gen/2> <http://xmlns.com/foaf/0.1/homepage>
"some bad page" .
```

Since the domains of mappings from the two corresponding query patterns to the triples are disjoint, the `MINUS` does not remove the binding to the first triple from the set of solutions. However, in our evaluation, both Jena and Virtuoso did not present this binding as a solution.[10]

This example query illustrates the importance of considering all ways of satisfying a query, as discussed in Sect. 3. QED also generated multiple other datasets embodying different ways of satisfying this query, in which the domains of the two sides of the minus are not disjoint. An example is the following.

```
<http://gen/0> <http://xmlns.com/foaf/0.1/homepage>
"some bad page" .
<http://gen/0> <http://xmlns.com/foaf/0.1/member> _:BX5FX3A .
<http://gen/0> <http://xmlns.com/foaf/0.1/name> <http://gen/1> .
```

In this case, both optional clauses bind `?band` to the same value and the systems correctly answered the query. Since SPARQL semantics is complex, our systematic exploration is an effective way of building a robust set of tests in an automated way.

Test Suites

Tables 1 and 2 show statistics about exemplary, generated test suites.[11] Since the DBpedia log queries primarily only cover the most common SPARQL features, we also consider the example queries that come with the Wikidata SPARQL endpoint[12], which have been formulated carefully and are rather diverse [4].

For DBpedia, we filter the DBpedia queries that are provided by the LSQ endpoint as described in Sect. 3.[13] Here, we look for queries that contain at least one of the following features: `FILTER`, `MINUS`, `OFFSET`, `OPTIONAL`, `UNION`, and hence create a relatively small and simple dataset in the style of the W3C tests. In the tables, the queries are grouped by the maximal set of the considered features which they contain. It can be seen that the queries are generally rather small (i.e., they only contain a few triple patterns), that there are usually several possible situations, but that the latter are only partly covered by DBpedia.

[10] In Jena, the bug has been solved by now (see https://issues.apache.org/jira/projects/JENA/issues/JENA-1633).

[11] The datasets can be found in the GitHub repository of QED.

[12] https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples.

[13] The LSQ format also specifies characteristics such as the number of answers to the query, which is only useful if we also consider the DBpedia data considered with the generation of the query set. Alternatively, we could have applied the LSQ framework to generate LSQ-formatted queries for the current DBpedia version.

Table 1. Statistics about the datasets generated for DBpedia; Qs = total number of queries, TP = avg. number of triple patterns, Sit = avg. number of situations, Cov = avg. share of situations covered in source data, Triples1 = avg. number of triples in the extracted data, Triples2 = as Triples1 but including generated data (averages are per query); FI = Filter, MI = Minus, OF = Offset, OP = Optional, UN = Union.

Features	Qs	TP	Sit	Cov (%)	Triples1	Triples2
FI	7	3	2	28	1	8
MI	1	3	3	100	5	15
OF	1	4	1	100	4	11
OP	4	4	12	37	6	14
UN	5	6	18	48	7	14
FI,OP	7	3	14	6	2	10
FI,UN	4	3	10	52	8	13
FI,OF,OP	1	8	4374	0	12	21
FI,OP,UN	1	4	18	0	0	4

Hence, our data generation helps to create test cases with diverse data that are of reasonable size though.

For Wikidata, we consider the example `SELECT` queries given and only ignore those that yield parse exceptions or contain the `SERVICE` feature (i.e., we do not apply the LSQ-based filtering). Table 2 shows that they contain more features and often several in combination. Due to space constraints, we focus on the seven most frequent and interesting features; specifically, we ignore `DISTINCT` and `LIMIT`, which occur in many example queries, and `MINUS`, `VALUES`, and `MIN`, which occur in only one query. In six cases, the generator tool could not cope with the filter condition (i.e., there is no generated data although Cov is $< 100\%$).

Nevertheless, especially the coverage of uncommon features differentiates our datasets from the existing ones, which mostly cover only the most common features. Also the currently still sparse usage of the newest SPARQL features (see, e.g., [4]) is a strong reason for generic frameworks like QED, which enable users to update their datasets if new queries or data become available. Finally, our diverse and realistic examples are special in the light of the most popular existing benchmarks, which usually still rely on artificial queries and/or data.

5 Related Work

The importance of SPARQL can easily be estimated by the attention payed to system development, evaluation, and optimization. There are numerous (generators for) datasets for SPARQL query evaluation. We have described the specifics of compliance tests and the tests proposed by the W3C in the introduction. Recall that the latter are nearly exhaustive in terms of the SPARQL feature combinations covered, but they contain artificial data only partly capturing the situations that could arise in practice.

Table 2. Statistics for the datasets for the Wikidata example queries; BI = Bind, FI = Filter, HA = Having, OP = Optional, SQ = Subquery, UN = Union, RE = Regex.

Features	Qs	TP	Sit	Cov (%)	Triples1	Triples2
	16	2	1	96	4	13
BI	2	4	1	100	8	8
FI	16	3	3	82	16	23
HA	1	1	1	100	2	2
BI,FI	6	3	4	86	15	17
BI,RE	1	1	2	100	4	12
FI,OP	14	3	9	51	19	24
FI,SU	1	1	2	0	0	7
BI,FI,OP	1	6	54	70	206	206
BI,OP,UN	1	3	9	100	26	26
FI,OP,SU	1	10	144	14	75	75
FI,OP,UN	1	7	108	86	430	440
BI,FI,OP,UN	1	8	36	97	165	165
BI,FI,RE,SU	1	4	16	62	43	43
FI,HA,OP,UN	1	7	252	13	191	191

All other benchmarks we are aware of focus on performance issues such as scalability. Most popular are the Lehigh University Benchmark (LUBM) [10], the Ontology Benchmark (UOBM) [13], the Berlin SPARQL Benchmark (BSBM) [3], SP2Bench [20], the DBpedia Sparql Benchmark (DBPSB) [14], the Waterloo Stress Testing Benchmark (WSTB) [1], and the BioBenchmark [24]. LUBM and its extension UOBM both integrate ontologies and hence also allow for evaluating reasoning. However, they only contain conjunctive queries and generators for artificial data; this also holds for WSTB. BSBM and SP2Bench also rely on data generation, but the queries contain some SPARQL features, though only a few of all possible ones. The queries in DBSP and BioBenchmark are similar to the latter but the data is real. Note that, with all of the benchmarks the queries are either fixed or generated based on a rather small set of templates. In a certain way, they are also domain-specific since each of the benchmarks focuses on a particular domain. In contrast, QED allows for generating tests for any domain. Also, apart from LUBM and UOBM, none of the benchmarks contains or considers the answers to the queries.

FEASIBLE [19] is an approach for mining SPARQL queries from logs for benchmarking and based on clustering similar queries and selecting prototypes. It is an alternative to our query extraction and, as outlined in Sect. 3, could be integrated with QED.

SPARQL query federation approaches are evaluated using benchmarks such as SPLODGE [9], FedBench [21], or LargeRDFBench [18]. They differ from the general benchmarks in that the queries span several data sources, the data is considerably larger in size (this often is also the case for the answers), and the evaluation focuses also on factors such as average source selection time and number of endpoint requests. SPLODGE is an approach for generating federated queries, but only conjunctive queries. FedBench comprises real data and queries but especially the latter are simple and cover only few SPARQL features. LargeRDFBench is similar to our approach since it contains several real datasets, including queries covering different SPARQL features. Apart from the number of sources they span and basic graph patterns they contain, the queries are comparable to the ones we extracted from DBpedia (see Sect. 4), in terms of complexity and variety. However, the simple queries are taken from standard benchmarks and hence less interesting; the more complex queries are from the life sciences domain and hand-crafted by the developers – and thus fixed –, whereas QED can be applied to extract new and diverse query sets for arbitrary domains. Also, the data (and answer) sets are much too large for the use cases we target and not tailored to the queries as ours. Note, however, that LargeRDFBench also allows for evaluating the correctness of the results.

6 Conclusions

In this paper, we have presented the SPARQL QED framework, which allows generating datasets for SPARQL query evaluation systems based on real-world queries and data. We have proven its value by generating tests that revealed bugs in Jena and Virtuoso, two popular endpoints. Beyond standard compliance testing, we have outlined other applications that could make use of the generated datasets. In contrast to the commonly used performance benchmarks, QED thus does not only help to make query answering systems adapt to common usage scenarios, and thus to make them more robust, but it also allows to develop and tune useful add-ons, such as query-by-example systems. This is further supported by the fact that QED is highly customizable and easy to handle. To the best of our knowledge, datasets as the ones generated by QED do not exist yet. Nevertheless, the current, initial version of QED only integrates a portion of the general approach behind the system. We have sketched possible extensions regarding the supported query forms, features, and kinds of tests in the paper. Support for ontology-based queries would be also an interesting extension.

Acknowledgments. This work is partly supported by the German Research Foundation (DFG) in the Cluster of Excellence "Center for Advancing Electronics Dresden" in CRC 912.

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, P., et al. (eds.) ISWC 2014, Part I. LNCS, vol. 8796, pp. 197–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_13
2. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.-Y.: SPARQL web-querying infrastructure: ready for action? In: Alani, H., et al. (eds.) ISWC 2013, Part II. LNCS, vol. 8219, pp. 277–293. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41338-4_18
3. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. Int. J. Semantic Web Inf. Syst. **5**(2), 1–24 (2009)
4. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. PVLDB **11**(2), 149–161 (2017)
5. Bornea, M., Dolby, J., Fokoue, A., Kementsietsidis, A., Srinivas, K., Vaziri, M.: An executable specification for SPARQL. In: Cellary, W., Mokbel, M.F., Wang, J., Wang, H., Zhou, R., Zhang, Y. (eds.) WISE 2016. LNCS, vol. 10042, pp. 298–305. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48743-4_24
6. Diaz, G.I., Arenas, M., Benedikt, M.: SPARQLByE: querying RDF data by example. PVLDB **9**(13), 1533–1536 (2016)
7. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng. Bull. **35**(1), 3–8 (2012)
8. Fariha, A., Sarwar, S.M., Meliou, A.: SQuID: semantic similarity-aware query intent discovery. In: SIGMOD 2018, pp. 1745–1748 (2018)
9. Görlitz, O., Thimm, M., Staab, S.: SPLODGE: systematic generation of SPARQL benchmark queries for linked open data. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 116–132. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35176-1_8
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Sem. **3**(2–3), 158–182 (2005)
11. Lehmann, J., Bühmann, L.: AutoSPARQL: let users query your knowledge base. In: Antoniou, G., et al. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 63–79. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21034-1_5
12. Lissandrini, M., Mottin, D., Velegrakis, Y., Palpanas, T.: X2q: your personal example-based graph explorer. In: PVLDB 2018 (2018)
13. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a complete OWL ontology benchmark. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 125–139. Springer, Heidelberg (2006). https://doi.org/10.1007/11762256_12
14. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.-C.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In: Aroyo, L., et al. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_29
15. Potoniec, J.: An on-line learning to query system. In: ISWC 2016 Posters & Demonstrations Track (2016)
16. Rafes, K., Nauroy, J., Germain, C.: Certifying the interoperability of RDF database systems. In: Proceedings of the 2nd Workshop on Linked Data Quality (2015)
17. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.-C.N.: LSQ: the linked SPARQL queries dataset. In: Arenas, M., et al. (eds.) ISWC 2015, Part II. LNCS, vol. 9367, pp. 261–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_15

18. Saleem, M., Hasnainb, A., Ngonga Ngomo, A.C.: LargeRDFBench: a billion triples benchmark for SPARQL endpoint federation. J. Web Sem. (2017)
19. Saleem, M., Mehmood, Q., Ngonga Ngomo, A.-C.: FEASIBLE: a feature-based SPARQL benchmark generation framework. In: Arenas, M., et al. (eds.) ISWC 2015, Part I. LNCS, vol. 9366, pp. 52–69. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_4
20. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^2 Bench: a SPARQL performance benchmark. In: ICDE 2009, pp. 222–233 (2009)
21. Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: FedBench: a benchmark suite for federated semantic data query processing. In: Aroyo, L., et al. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 585–600. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_37
22. Thompson, B.B., Personick, M., Cutcher, M.: The bigdata® RDF graph database. In: Linked Data Management, pp. 193–237 (2014)
23. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D., Ding, L.: Supporting scalable, persistent semantic web applications. IEEE Data Eng. Bull. **26**(4), 33–39 (2003)
24. Wu, H., Fujiwara, T., Yamamoto, Y., Bolleman, J.T., Yamaguchi, A.: Biobenchmark toyama 2012: an evaluation of the performance of triple stores on biological data. J. Biomed. Semant. **5**, 32 (2014)