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

Veronika Thost* and Julian Dolby

IBM Research
veronika.thost@ibm.com, dolby@us.ibm.com

1 Introduction

The SPARQL query language is probably the most popular technology for querying the Semantic Web and supported by most triple stores and graph databases [6, 7, 3]. Several benchmarks have been developed to evaluate their efficiency (a good overview is provided by the W3C[1]), but correctness tests are not so common. In fact, to the best of our knowledge, the W3C compliance tests[2] are the only test suite publicly available and commonly applied [1, 4]. However, these tests mostly contain fairly synthetic queries over similarly artificial example data and, in particular, comprise only few more complex queries 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[3] and rather simple `SELECT` queries, such as the following query Q_{ex}:

```
SELECT ?s ?p ?o ?z { ?s ?p ?o .
  { BIND(?o+1 AS ?z) } UNION { BIND(?o+2 AS ?z) }}
```

Moreover, `UNION` occurs only together with the `BIND` keyword and once with the `FILTER` key, but with none other. Naturally, hand-crafted tests cannot cover all possible combinations of features. But the following example from the DBpedia query log shows that real queries often contain various and nested features.[4]

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

* Supported by DFG within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed) in CRC 912 (HAEC).

[1] https://www.w3.org/wiki/RdfStoreBenchmarking

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

[3] The 2009 tests actually contain some more such queries, but these regard an empty dataset and hence represent rather unrealistic test cases.

[4] We obtained the query from LSQ: http://aksw.github.io/LSQ/.

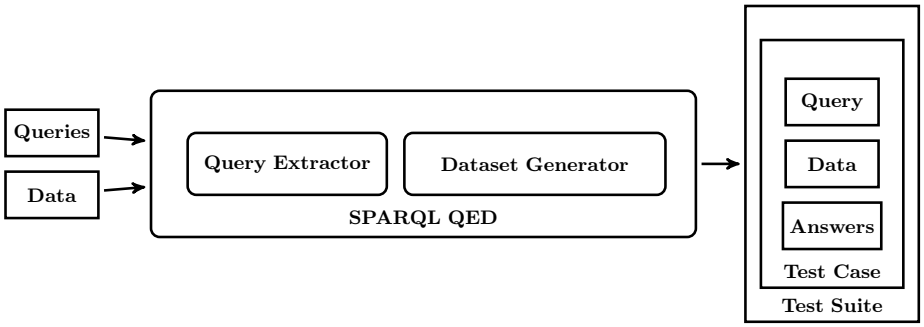

Fig. 1. The approach of SPARQL QED. The input consists of a query log and a SPARQL endpoint providing the data. The application first extracts queries from the log based on their features. Then, a test case is created for each of them by generating a dataset (including some of the input data), and by computing the answers over this data. This test suite then represents the output of the system.

We thus have a considerable gap between the tests and reality. And similar for the data: the test data for Q_{ex} consists of the few triples below, but DBpedia contains over 13 billion triples[5] and neither contains only triples on one property nor does it contain one per subject and property.

```
:s1 :p 1 .  :s2 :p 2 .  :s3 :p 3 .  :s4 :p 4 .
```

As a consequence of this mismatch, many public endpoints do not really comply to the specification and exhibit non-standard behavior.[6]

To complement the existing benchmarks, we present the SPARQL Query Evaluation Dataset generator (QED), which closes the aforementioned gap by generating out-of-the-box datasets for given SPARQL queries over linked data (note that QED similarly works over local data). QED is highly customizable and thus enables users to create correctness tests tailored to their specific SPARQL applications. QED is available at https://github.com/vthost/qed.

2 SPARQL QED

SPARQL QED is a framework for generating datasets for SPARQL query evaluation as outlined in Figure 1. The input consists of two endpoints, one providing the data and the other one log queries over that data. 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, and that they cover various practical use cases. We also verify that the sets of answers included are the correct ones.

[5] http://wiki.dbpedia.org/develop/datasets/dbpedia-version-2016-10
[6] https://www.w3.org/2009/sparql/implementations/

1. Query Extraction The queries are assumed to be given in the LSQ RDF format [5], which captures specific characteristics, such as number of triples, features contained in them (e.g., the `OPTIONAL` or `FILTER` construct), 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 10 test cases with queries that contain `UNION` (and possibly other features); 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.[7] We also have filters that ensure that the extracted queries are diverse and do not contain near-duplicates as generated by bots, for example.

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 a minimal data set of the kind as given for Q_{ex} above). We therefore do not only take a subset of the relevant data that is restricted in size but also ensure that it reflects the variability of the possible answers. For instance, regarding the pattern Q_1`UNION` Q_2, we include four kinds of data (if available): data that satisfies both Q_1 and Q_2, only one, and none of them; and similarly for `OPTIONAL` and `FILTER`.

Alas, we cannot assume that we find all these different kinds of data. In fact, the portion of the provided data matching a query usually only covers a few. To tackle this problem, we generate synthetic data for the remaining cases.

3. Data Generation For those of the above cases where data is not provided but which could arise theoretically,[8] 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 consider each query Q of the different versions of the queries (e.g., for the above pattern, we have queries $Q_1 \& Q_2$, $Q_1 \& !Q_2$, etc.) and represent the answers over the unknown data D in a relational logic formula $Ans(Q, D)$ as proposed in [2]. The resulting equation then simply requires this set to be non-empty and can be solved by Kodkod, yielding a dataset as intended (i.e., satisfying the given query).

4. Answer Generation For all queries and the corresponding data, we include the corresponding answers into the test suite. However, the fact that we rely on a standard endpoint to retrieve the answers must not be ignored if there is no correctness guarantee for the endpoint. For that reason, we establish the correctness of the answers based on the declarative semantics of SPARQL already mentioned above [2], again using Kodkod; note that the semantics has been validated. That is, for each test generated, consisting of sets Q, D, and A, the answers we found, we verify the truth of the assertion $A = Ans(Q, D)$.

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

[8] For some cases, corresponding data cannot exist; for instance, the two subqueries of a may contradict each other and then never be satisfied together.

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

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. This allows testers to reuse existing infrastructure (originally created for the W3C tests) to directly run the tests generated with QED.

3 Future Extensions and Use Cases

We present the initial version of QED together with various exemplary, generated datasets and are interested in community feedback as well as suggestions for improvements. There are several useful extensions we have made out so far:

- **Performance** The complexity of some queries still represents a challenge for the system and leaves room for improvement in efficiency and robustness.
- **Query Features** There are query features such as property paths, aggregation, or complex filters, whose consideration in the tests would certainly benefit applications, but which would need to be handled carefully.
- **Query Forms** We currently focus on the `SELECT` query form, but the standard specifies others too.[10] While our implementation can easily be adapted w.r.t. the `ASK` form, the `CONSTRUCT` form requires a more elaborate extension.
- **Syntax Tests** We concentrate on query evaluation but plan to also provide tests to check whether syntactically (in)correct queries yield exceptions or not, as they are included in the W3C test suite.
- **Application** We plan to do extensive correctness testing of publicly available endpoints in order to demonstrate the usefulness of our system.

Nevertheless, we believe that correctness testing only represents one use case for our datasets. Since QED is highly customizable, it should be applicable in various scenarios by a wider community. We are hence also interested in discussions on other potential applications, such as for developing query-by-example systems, auto-completion add-ons, or query learning systems.

References

1. Aranda, C.B., Hogan, A., Umbrich, J., Vandenbussche, P.: SPARQL web-querying infrastructure: Ready for action? In: ISWC (2013)
2. Bornea, M.A., Dolby, J., Fokoue, A., Kementsietsidis, A., Srinivas, K., Vaziri, M.: An executable specification for SPARQL. In: WISE (2) (2016)
3. Erling, O.: Virtuoso, a hybrid rdbms/graph column store. IEEE Data Eng. Bull. 35(1), 3–8 (2012)
4. Rafes, K., Nauroy, J., Germain, C.: Certifying the interoperability of RDF database systems. In: ESWC (2015)
5. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: the linked SPARQL queries dataset. In: ISWC (2015)
6. Thompson, B.B., Personick, M., Cutcher, M.: The bigdata® RDF graph database. In: Linked Data Management., pp. 193–237 (2014)
7. 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)

[10] https://www.w3.org/TR/sparql11-query/