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

Veronika Thost[1] and Julian Dolby[2]

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

Abstract. In this paper, we present SPAQRL 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 be used in various ways; in particular, they support the development of applications based on SPARQL query answering. For example, 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.

Keywords: SPARQL, data sets, compliance testing

1 Introduction

The SPARQL query language is widely used and probably the most popular technology when it comes to querying the linked data in the Semantic Web. Most triple stores and graph databases support the user-friendly declarative query language [13, 14, 7]. There are several benchmarks targeting the performance of SPARQL query answering [2, 12, 9] (an overview is also provided by the W3C[1]) but, to the best of our knowledge, regarding correctness, the W3C compliance tests[2] are basically the only test suite publicly available and commonly applied [1, 10]. They have been proposed originally in 2001 for SPARQL 1.0[3] and were extended in 2009 regarding the new features of SPARQL 1.1. However, these tests mostly contain seemingly synthetic queries over similarly artificial example data and, especially, comprise only few more complex queries nesting different SPARQL features, which model real user queries more faithfully. A simple text

[1] https://www.w3.org/wiki/RdfStoreBenchmarking
[2] https://www.w3.org/2009/sparql/docs/tests/
[3] https://www.w3.org/2001/sw/DataAccess/tests/

search reveals, for example, that the `UNION` keyword only occurs in nine[4] and rather simple `SELECT` queries, such as the following query Q[5]:

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

Regarding a given dataset, this query retrieves all those bindings of the variables `?s`, `?p`, `?o`, and `?z` to values `s'`, `p'`, `o'`, and `z'` , respectively, such that `s'` and `o'` are related via `p'`, and either `o'`+1=`z'` or `o'`+2=`z'`. Especially, note that, in the W3C tests, `UNION` occurs only together with the `BIND` key 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 features nested in combination.[6]

```
 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') )
  }
```

We thus have a considerable gap between the queries in the tests and those in reality. And this is similar regarding the data: whereas the test data for Q consists of the below four statements, the latest DBpedia dump contains more than 13 billion triples,[7] but it neither contains only triples on one property nor one per subject and property.

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

[4] The 2009 tests actually contain some more such queries, but these regard an empty dataset and hence represent rather unrealistic test cases.
[5] https://www.w3.org/2009/sparql/docs/tests/data-sparql11/bind/bind07.rq
[6] We obtained the query from LSQ: http://aksw.github.io/LSQ/.
[7] http://wiki.dbpedia.org/develop/datasets/dbpedia-version-2016-10

As a consequence of this mismatch, many public endpoints do not really comply to the specification and exhibit non-standard behavior; this is also demonstrated in the official implementation report for SPARQL 1.1[8]. Furthermore, they cannot provide additional features such as comprehensive auto-completion or query suggestions to the users since adequate evaluation data does not exist. This impairs usability and will become more critical in the future if ever more applications consume the linked data available in the Semantic Web. We suppose that enhanced system support would especially increase the adoption of SPARQL by new users. Obviously, we are missing test datasets with real-world queries which, in particular – in contrast to the existing performance benchmarks –, would also have to contain diverse and comprehensive samples of data (i.e., triples) and the corresponding 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 QED similarly works over local data). QED distinguishes given queries according to the different SPARQL features, selects some of them, and creates, for each query, a dataset 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. In Section 2, we propose applications for our data sets and specify corresponding requirements. In Section 3, QED in presented in detail, and Section 4 describes example test suites created from DBpedia and its query log.

2 Motivation

In this section, we outline use cases for QED and specify the corresponding requirements for (or features of) the system. While many of the existing benchmarks targeting performance already rely on real-world queries [2, 12, 9], 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 of other features, for instance: query-by-example approaches [6], where queries are to be learnt from tuples that are among the queries answers; auto-completion add-ons [8, 5]; and query learning systems [3], which suggest queries based on simple user inputs.

Our goal was therefore to keep QED general, in order to allow for as many applications as possible. Specifically, we derived the following requirements:

R1 Originality QED should fill the gaps in existing benchmarks and comprise diverse query and data samples from real applications.

[8] https://www.w3.org/2009/sparql/implementations/

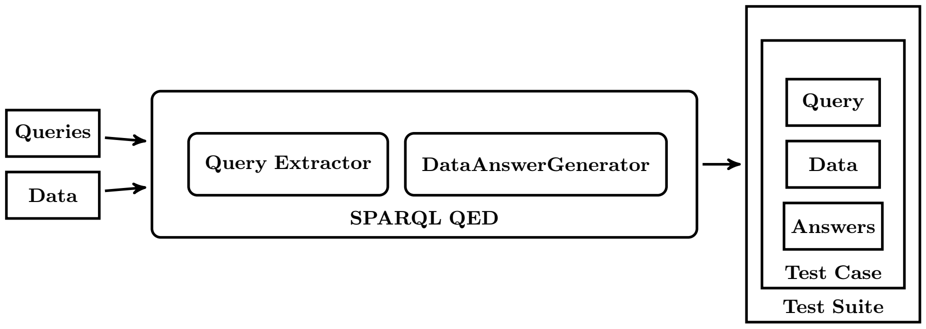

Fig. 1. The architecture of SPARQL QED. The input consists of a query log and a SPARQL endpoint providing the data. The application first extracts queries from the log based on their features. Then, a test case is created for each of them by generating a dataset (taking the input triples into account),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

R2 Reusability QED should be applicable to arbitrary SPARQL queries and corresponding data, and hence be kept as general as possible. Next to that, the system should be easy to handle.

R3 Quality The QED approach is naturally based on existing query logs and linked data. The system design should further be reasonable and follow common Semantic Web practices. Beyond that, QED should be tested adequately.

3 SPARQL QED

To generate test, we need two endpoints to be given: one providing the data, and the other one log queries over that data. Specifically, we rely on queries described in the RDF format proposed in [11], which captures specific characteristics of them; amongst others, the so-called features (e.g., the `OPTIONAL` or `FILTER` construct) the queries contain. The LSQ format also points out queries which yield parse or runtime errors or for which there are no answers, so that we can disregard those. Note that there is a tool for transforming SPARQL queries into this format relatively easily.[9]

As it is shown in Figure 1, QED is made up of two components. The query extractor is configured with sets of features and, as its name suggests, for each of these sets, extracts queries from the query log that contain all the features of the sets (if there are such queries). The DataAnswerGenerator then uses the other endpoint to extract both data that can be used to answer the queries and the corresponding answers over that data.

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

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

of which comes with one file for each, the query, the data, and the result. The SPARQL queries are in text files, whereas the others are in the common TURTLE format.

Besides the above, our implementation additionally ensures that the test cases are diverse and correct, as described in the following.

Variability of the Test Cases

The LSQ format allows us to select diverse queries. However, 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. Nevertheless, we do not want the tests to be too simplistic (i.e., with a minimal data set of the kind as given for Q). 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 data. Consider, for instance, a query containing the pattern `OPTIONAL { ?airport rdfs:label ?name. }`, so that the variable `?name` is bound with answers whose `?airport` is related by the property `rdfs:label`. In this case, we include data of two kinds: about cities with airports that are given the property `rdfs:label` and about cities whose airports do not do so. Second, we consider `FILTER (!bound(?name) || langMatches(lang(?name), 'de'))`, which requires the answers to be such that either the variable `?name` is not bound or it is bound to a German word. Then we include not only data where at least one of the conditions is satisfied but also samples where this is not the case. In the end, we try to multiply the conditions, and try to find data of two $\times$ two kinds.

Alas, we cannot assume that we find all these different kinds of data. In fact, we show in Section 4 that the data matching a query and that is provided with the endpoints usually only covers a few of all possible kinds. To tackle this problem, we generate synthetic data for the remaining cases based on a declarative semantics of SPARQL [4]. Recall that the meaning of a query Q w.r.t. a dataset D is the set of all answers; that is, the set of those bindings of the free variables of Q to values for which there is an extension covering all the variables in the query in such a way that the corresponding subgraph appears in D. We denote the corresponding specification in relational logic proposed in [4] by $\mathfrak{T}(Q, D)$; for instance, we have $\mathfrak{T}(Q_1 \cup Q_2, D) \equiv \mathfrak{T}(Q_1, D) \vee \mathfrak{T}(Q_2, D)$ and $\mathfrak{T}(Q_1 \text{ optional } \mathrm{Q_2, D}) \equiv (\mathfrak{T}(\mathrm{Q_1}) \wedge \mathfrak{T}(\mathrm{Q_2})) \vee \mathfrak{T}(\mathrm{Q_1} \nexists \mathrm{Q_2})$; here, $\cup$ and optional represent `UNION` and `OPTIONAL`, and $\mathfrak{T}(Q_1 \nexists Q_2)$ denotes `FILTER NOT EXISTS` (for the definition of the latter and the other "translation" formulas, see Figures 3 and 4 in [4]). In QED, we use an extension $\mathfrak{T}_C$ of $\mathfrak{T}$, which explicitly considers possible choices in the query through the set C. Specifically, we consider $CVar(Q)$ to be a set containing a variable v_{Q_1} for every `UNION` and `OPTIONAL` subquery Q_1 in the query Q under consideration, $C \in \mathcal{P}(CVar(Q))$, and specify $\mathfrak{T}_C$ as follows:

$$\mathfrak{T}_C(Q, D) \equiv \begin{cases} Q_1 \cup \ Q_2 & \equiv \begin{cases} \mathfrak{T}_C(Q_1) \ v_Q \in C \\ \mathfrak{T}_C(Q_2) \ v_Q \notin C \end{cases} \\ Q_1 \text{ optional } \mathrm{Q_2} \equiv & \begin{cases} (\mathfrak{T}_C(Q_1) \wedge \mathfrak{T}_C(Q_2)) & v_Q \in C \\ \mathfrak{T}_C(Q_1 \nexists Q_2) & v_Q \notin C \end{cases} \end{cases}$$

Regarding the remaining operators, it is defined the same as $\mathfrak{T}$. In QED, we have a given query Q and specifically want to create a dataset D that $(*)$ contains at least one answer for every possible set $C \in \mathcal{P}(CVar(Q))$.

In general, not all possible sets C can always be satisfied and some may clash with others in the dataset. For the first case, a pattern like `Q MINUS Q`, for example, will have no solutions; the second case is a bit more subtle, but can be illustrated as follows:

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

In this example, for a dataset, we either retrieve no answers, if the data does not contain the triple `:s1 :p :s2`; one answer where `?a` is not bound, if the data contains the triple `:s1 :p :s2` but no triple starting with `:s2 :p`; or answers in which `?a` is bound and the data contains the triple `:s1 :p :s2`. For such queries, we assemble multiple datasets greedily: we iteratively collect sets C (i.e., instead of directly considering the entire set $\mathcal{P}(CVar(Q))$, for which a data set satisfying $(*)$ does not exist), and start a new dataset if there is none satisfying $(*)$ w.r.t. the current collection of sets C any more.

Note that, in the current version, our dataset expansion is attempting to generate answers for different sets C, and this means that `FILTER` and `MINUS` each have only one case and hence do not need additional choices. That is because, for each of them, while they have multiple outcomes, only one outcome actually results in answers; this is different for `UNION` and `OPTIONAL`. Once we have collected the constraints (i.e., the collection of sets C) on D, we use our translation $\mathfrak{T}_C$ to relational logic to allow Kodkod[10], a SAT-based constraint solver for extensions of first order logic with relations, to solve it and to construct the dataset we target.

Correctness of the Results

The fact that we rely on the endpoints to retrieve the answers which we include in our test suite must not be ignored if there is no correctness guarantee for the endpoint. For that reason, we have established the correctness of the answers based on the declarative semantics of SPARQL already mentioned above [4]. This semantics has been validated against the W3C compliance tests: it was specified in Kodkod, and then run on the compliance tests. For each test, the answers A given for a query Q over data D were compared to the evaluation of $\mathfrak{T}(Q, D)$ by verifying the truth of the assertion $A =_{RDF} \mathfrak{T}(Q, D)$ using Kodkod; here, $=_{RDF}$ denotes equality under the RDF entailment regime[11]. Since the correctness of these declarative semantics has thus been established, we are able to use the same approach to verify our generated tests. That is, for each test

[10] http://emina.github.io/kodkod/index.html
[11] http://www.w3.org/TR/sparql11-entailment/

generated, consisting of sets A, Q, and D as above, we verify the truth of the assertion $A =_{RDF} \mathfrak{T}(Q, D)$ in the same fashion.

Discussion

It can readily be checked that SPARQL QED follows the guidelines **R1-R3** from Section 2. The originality of the approach is due to the fact that it allows to generate diverse and extensive test cases based on real queries and data. This supports the development of different applications based on SPARQL query answering, and helps to advance the adoption of the Semantic Web technologies. 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. In addition, the (Java) implementation is kept simple and easy to handle. Finally, the system design follows best practices: we rely not only on existing queries and data, but also on the comprehensive approach of the LSQ, 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 produced are correct.

In the initial version, we concentrate on the `SELECT` query form and on evaluation tests. Next to `SELECT`, the standard specifies three other forms, `ASK`, `CONSTRUCT`, and `DESCRIBE`.[12] 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 testing since it returns information that is to be determined by the query service applied; that is, it cannot be tested in a general fashion.

Next to query evaluation tests, the W3C test suite includes further kinds of tests, such as syntax or protocol tests. The former test if syntactically (in)correct queries yield exceptions or not, and we plan to consider them in the future, possibly by integrating a SPARQL parser into QED. The other kinds of tests are however out of the scope of our work, whose goal is to apply given, real-world queries and data to generate tests.

4 Dataset Examples

For demonstrating QED, we focus on DBpedia. Specifically, we use the log queries provided as examples with [11] and the corresponding version of DBpedia; see also that paper for interesting statistics on both the queries and the data.[13]

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

[13] Note that The LSQ format amongst others specifies characteristics such as the number of answers to the query, which is only useful for us if we also consider the DBpedia data that was considered with the generation of the query set. Alterna-

	Qs	CASES			EX. CASES			TRIPLES
FEATURE		min	avg	max	min	avg	max	avg
minus	2	2	2	2	2	2	2	9
orderby	100	1	2	11	0	1	7	5
distinct	100	1	10	11	0	1	3	11
optional	100	2	2	11	0	1	4	3
union	100	11	11	11	0	1	5	10
regex	100	2	9	11	0	1	8	4
limit	100	1	2	11	0	1	6	5

Table 1. Statistics about Test Suite 1

	Qs	CASES			EX. CASES			TRIPLES
FEATURE		min	avg	max	min	avg	max	avg
filter_orderby	100	2	2	11	0	2	7	5
filter_regex	100	2	9	11	0	1	8	4
union_filter	100	8	10	11	0	0	6	1
union_orderby	9	4	8	11	0	2	6	4
union_distinct	100	11	11	11	0	1	3	10
optional_regex	100	11	11	11	0	1	6	3
optional_limit	100	2	8	11	0	2	8	4
union_regex	100	11	11	11	0	0	0	0

Table 2. Statistics about Test Suite 2

A simple test suite can be obtained if we consider queries containing at least one of the features interesting to us. Table 1 gives an overview of the corresponding tests' variability. In the first columns, we show selected features and the number of log queries we have retrieved for them. For each query, we have counted the number of theoretically possible situations ("CASES") modeled in the data (e.g., for a query with an `OPTIONAL` pattern, we have at least two situations; either the `OPTIONAL` part is satisfied or not) – but we usually have more situations since many queries contain several features at once. The minimum/average/maximum are shown in the table. The next three columns (EX. CASES) show the minimum/average/maximum counts of situations that are actually represented in the data we are given. Finally, we count the triples we have retrieved (and included in the test cases) for modeling all situations relevant to a query.

The fact that we do not find lots of queries for all features shows that the variability of the tests strongly depends on the diversity of the input queries, and also on that of the given data. This shows the importance of our dataset extension and of an approach that operates on arbitrary logs, can generate tests for any dataset, and that can be tailored to the SPARQL constructs that are most used in that specific dataset. Note that, for some single features, for example, aggregate features such as `AVG` and `MAX`, for `BIND`, or for `SUBQUERY`, we do not

tively, we could have applied the LSQ framework by ourselves to generate correctly formatted queries based on the current version of DBpedia.

retrieve any queries from the DBpedia log (if we require also the above mentioned absence of errors, a minimal number of answers, and similar basic properties); this is not depicted in the table. It has already been noted in [11] that SPARQL 1.1 features are yet rarely used, and the age of the logs is proposed as one explanation for this. However, we do not consider the missing test cases to be a serious problem, because our approach will naturally create tests for those features that are commonly used in a particular dataset. Thus, running over popular datasets, we will obtain tests for those combinations of features that are generally applied. In this way, QED complements existing tests by covering the sorts of queries we encounter in practice.

As a second example, we consider several arbitrary pairs of features. Table 2 depicts the corresponding statistics (with the same structure as Table 1). Again, we see that we are able to usually find at least 100 queries from the log with the desired features and, as it is to be expected of more-complex queries, we obtain generally slightly more situations than for the first, single-feature test suite.

5 Conclusions

In this paper, we have presented the SPARQL QED framework, which allows to generate test suites for SPARQL query evaluation systems based on real-world queries and data. Beyond standard compliance testing, we have outlined various other SPARQL applications that could make use of the generated tests and that assist users in query formulation. In contrast to the commonly used performance benchmarks, QED thus does not only help to make query answering systems adapt to common usage, and thus to make them more robust, but it also allows to develop and to tune useful add-ons. This is further supported by the fact that QED is highly customizable and, nevertheless, easy to handle. To the best of our knowledge, test suites as the ones generated by QED do not exist yet, and we therefore expect them to impact the adoption of SPARQL and Semantic Web technologies in general.

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 and kinds of tests in the paper. And there are other, rather intricate query features, such as property paths, aggregation, or complex filter conditions, whose consideration would certainly benefit the applications, but which need to be handled carefully. This gives rise to a lot of future work, but it also shows that the approach bears great potential.

References

1. Aranda, C.B., Hogan, A., Umbrich, J., Vandenbussche, P.: SPARQL web-querying infrastructure: Ready for action? In: The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II. pp. 277–293 (2013)

2. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. Int. J. Semantic Web Inf. Syst. 5(2), 1–24 (2009)
3. Bonifati, A., Ciucanu, R., Staworko, S.: Interactive join query inference with JIM. PVLDB 7(13), 1541–1544 (2014)
4. Bornea, M.A., Dolby, J., Fokoue, A., Kementsietsidis, A., Srinivas, K., Vaziri, M.: An executable specification for SPARQL. In: WISE (2). Lecture Notes in Computer Science, vol. 10042, pp. 298–305 (2016)
5. Campinas, S.: Live SPARQL auto-completion. In: Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014. pp. 477–480 (2014)
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. Lehmann, J., Bühmann, L.: Autosparql: Let users query your knowledge base. In: The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I. pp. 63–79 (2011)
9. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.N.: Dbpedia SPARQL benchmark - performance assessment with real queries on real data. In: The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I. pp. 454–469 (2011)
10. Rafes, K., Nauroy, J., Germain, C.: Certifying the interoperability of RDF database systems. In: Proceedings of the 2nd Workshop on Linked Data Quality co-located with 12th Extended Semantic Web Conference (ESWC 2015), Portorož, Slovenia, June 1, 2015. (2015)
11. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.N.: LSQ: the linked SPARQL queries dataset. In: The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II. pp. 261–269 (2015)
12. Saleem, M., Mehmood, Q., Ngomo, A.N.: FEASIBLE: A feature-based SPARQL benchmark generation framework. In: The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I. pp. 52–69 (2015)
13. Thompson, B.B., Personick, M., Cutcher, M.: The bigdata® RDF graph database. In: Linked Data Management., pp. 193–237 (2014)
14. 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)