Usage-Centric Benchmarking of RDF Triple Stores

Mohamed Morsey and **Jens Lehmann** and **Sören Auer** and **Axel-Cyrille Ngonga Ngomo**
AKSW Research Group, University of Leipzig,
Johannisgasse 26, 04103 Leipzig, Germany
{morsey|lehmann|auer|ngonga}@informatik.uni-leipzig.de

Abstract

A central component in many applications is the underlying data management layer. In Data-Web applications, the central component of this layer is the triple store. It is thus evident that finding the most adequate store for the application to develop is of crucial importance for individual projects as well as for data integration on the Data Web in general. In this paper, we propose a generic benchmark creation procedure for SPARQL, which we apply to the DBpedia knowledge base. In contrast to previous approaches, our benchmark is based on queries that were actually issued by humans and applications against existing RDF data not resembling a relational schema. In addition, our approach does not only take the query string but also the features of the queries into consideration during the benchmark generation process. Our generic procedure for benchmark creation is based on query-log mining, SPARQL feature analysis and clustering. After presenting the method underlying our benchmark generation algorithm, we use the generated benchmark to compare the popular triple store implementations Virtuoso, Sesame, Jena-TDB, and BigOWLIM. [1] [2]

Introduction

The RDF data model (Klyne and Carroll 2004) is the main building block of the Semantic Web – it plays a similar role as HTML does for the conventional World Wide Web. The RDF data model resembles directed labeled graphs, in which each labeled edge (called *predicate*) connects a *subject* to an *object*. Such a connection is called an *RDF triple* and nodes in this graph can be *resources*, e.g. a particular book or a person. Every resource is identified by a Uniform Resource Identifier (URI) which is globally unique.A triple expressing that Leipzig is located in Germany could, therefore, be expressed as follows:

```
<http://dbpedia.org/resource/Leipzig>
        <http://dbpedia.org/ontology/country>
                <http://dbpedia.org/resource/Germany>
```

Figure 1: Example of an RDF triple.

SPARQL (SPARQL Protocol and RDF Query Language) is the query language for RDF. A SPARQL processor finds sets of triples in the RDF graph that match to the required pattern. The results of SPARQL queries can be result sets or RDF graphs (Prud'hommeaux and Seaborne 2008). For instance, an example for the query "Who is the spouse of Shakespeare's child?" is shown in Figure 2.

```
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?spouse WHERE {
 dbpedia:William_Shakespeare dbo:child ?child.
 ?child dbp:spouse ?spouse.
}
```

Figure 2: SPARQL query to get the spouse of Shakespeare's child.

SPARQL is based on powerful graph matching that allows binding variables to fragments in the input RDF graph. In addition, operators akin to the relational joins, unions, left outer joins, selections and projections can be used to build more expressive queries (Schmidt et al. 2009). It is evident that the performance of triple stores offering a SPARQL query interface is mission critical for individual projects as well as for data integration on the Web in general. It is consequently of central importance during the implementation of any Data Web application to have a clear picture of the weaknesses and strengths of current triple store implementations.

Existing SPARQL benchmark efforts such as LUBM (Pan et al. 2005), BSBM (Bizer and Schultz 2009) and SP^2 (Schmidt et al. 2009) resemble relational database benchmarks. Especially, the data structures underlying these benchmarks are basically relational data structures, with relatively

[1] This work was supported by grants from the European Union's 7th Framework Programme provided for the project LOD2 (GA no. 257943) and from Eurostars E!4604 SCMS.

[2] This article is an update of (Morsey et al. 2011) with the following changes: (1) We generalised the overall description to a broader audience. (2) The similarity metrics simulation has been improved. (3) New benchmark results based on this similarity metric are presented.

few and homogeneously structured classes. However, RDF knowledge bases are increasingly heterogeneous. Thus, they do not resemble relational structures and are not easily representable as such. Examples of such knowledge bases are curated bio-medical ontologies such as those contained in Bio2RDF (Belleau et al. 2008) as well as knowledge bases extracted from unstructured or semi-structured sources such as DBpedia (Lehmann et al. 2009; Morsey et al. 2012) or LinkedGeoData (Auer, Lehmann, and Hellmann 2009; Stadler et al. 2011). For instance, DBpedia contains thousands of classes and properties. Also, various data types and object references of different types are used in property values. Such knowledge bases *cannot* be easily represented according to the relational data model and hence performance characteristics for loading, querying and updating these knowledge bases might potentially be fundamentally different from knowledge bases resembling relational data structures.

In this article, we propose a generic SPARQL benchmark creation methodology. This methodology is based on a flexible data generation mimicking an input data source, query-log mining, clustering and SPARQL feature analysis. We apply the proposed methodology to datasets of various sizes derived from the DBpedia knowledge base. In contrast to previous benchmarks, we perform measurements on *real* queries that were issued by humans or Data-Web applications against existing RDF data. Moreover, we do not only consider the query string but also the SPARQL features used in each of the queries. In order to obtain a representative set of *prototypical queries* reflecting the typical workload of a SPARQL endpoint, we perform a query analysis and clustering on queries that were sent to the official DBpedia SPARQL endpoint. From the highest-ranked query clusters (in terms of aggregated query frequency), we derive a set of 20 SPARQL query templates, which cover most commonly used SPARQL feature combinations and are used to generate the actual benchmark queries by parametrization. We call the benchmark resulting from this dataset and query generation methodology *DBPSB2* (i.e. DBpedia SPARQL Benchmark version 2). The benchmark methodology and results are also available online[3]. Although we apply this methodology to the DBpedia dataset and its SPARQL query log in this case, the same methodology can be used to obtain application-specific benchmarks for other knowledge bases and query workloads. Since DBPSB2 changes with the data and queries in DBpedia, we envision to update it in yearly increments and publish results on the above website. In general, our methodology follows the four key requirements for domain specific benchmarks as postulated in the Benchmark Handbook (Gray 1991), i.e. it is (1) relevant, thus testing typical operations within the specific domain, (2) portable, i.e. executable on different platforms, (3) scalable, e.g. it is possible to run the benchmark on both small and very large data sets, and (4) it is understandable.

We apply DBPSB2 to assess the performance and scalability of the popular triple stores *Virtuoso* (Erling and Mikhailov 2007), *Sesame* (Broekstra, Kampman, and van Harmelen 2002), *Jena-TDB* (Owens et al. 2008), and *BigOWLIM* (Bishop et al. 2011) and compare our results with those obtained with previous benchmarks. Our experiments reveal that the performance and scalability is by far less homogeneous than other benchmarks indicate. For example, we observed query performance differences of several orders of magnitude much more often than with other RDF benchmarks when looking at the runtimes of individual queries. The main observation in our benchmark is that previously observed differences in performance between different triple stores amplify when they are confronted with actually asked SPARQL queries, i.e. there is now a wider gap in performance compared to essentially relational benchmarks.

The paper is organized as follows: We first show the process of query analysis and clustering in detail. After that, we present our approach to selecting SPARQL features and to query variability. We then assess the four triple stores via DBPSB2 and discuss results. Finally, we conclude with related and future work.

Query Analysis and Clustering

The goal of the query analysis and clustering is to detect prototypical queries that were sent to a SPARQL endpoint based on a query-similarity graph. Several types of similarity measures can been used on SPARQL queries, for example string similarities for comparing the actual query used and graph similarities to compare the query structure. Given that previous work suggest that most triple stores tend to perform well for queries that display certain SPARQL features and less well for others, we carry out the following approach for the benchmark generation: First, we select queries that were executed frequently on the input data source. Second, we strip common syntactic constructs (e.g., namespace prefix definitions) from these query strings in order to increase the conciseness of the query strings. Then, we compute a query similarity graph from the stripped queries by comparing both the features used in the queries and the query strings. Finally, we use a soft graph clustering algorithm for computing clusters on this graph. These clusters are subsequently used to devise the query generation patterns used in the benchmark. In the following, we describe each of the four steps in more detail.

Query Selection For the DBPSB, we use the DBpedia SPARQL query-log which contains all queries posed to the official DBpedia SPARQL endpoint for a three-month period in 2010[4]. For the generation of the current benchmark, we used the log for the period from April to July 2010. Overall, 31.5 million queries were posed to the endpoint within this period. In order to obtain a small number of distinctive queries for benchmarking triple stores, we reduce those queries in the following two ways:

- *Query variations.* Often, the same or slight variations of the same query are posed to the endpoint frequently. A

[3] http://aksw.org/Projects/DBPSB

[4] The DBpedia SPARQL endpoint is available at: http://dbpedia.org/sparql/ and the query log excerpt at: ftp://download.openlinksw.com/support/dbpedia/.

particular cause of this is the renaming of query variables. We solve this issue by renaming all query variables in a consecutive sequence as they appear in the query, i.e., *var0*, *var1*, *var2*, and so on. As a result, distinguishing query constructs such as `REGEX` or `DISTINCT` are a higher influence on the clustering.

- *Query frequency.* We discard queries with a low frequency (below 10) because they do not contribute much to the overall query performance.

The application of both methods to the query log data set at hand reduced the number of queries from 31.5 million to 35,965. This reduction allows our benchmark to capture the essence of the queries posed to DBpedia within the time span covered by the query log and reduces the runtime of the subsequent steps substantially.

String Stripping Every SPARQL query contains substrings that segment it into different clauses. Although these strings are essential during the evaluation of the query, they are a major source of noise when computing query similarity, as they boost the similarity score without the query patterns being similar per se. Therefore, we remove all SPARQL syntax keywords such as `PREFIX`, `SELECT`, `FROM` and `WHERE`. In addition, common prefixes (such as `http://www.w3.org/2000/01/rdf-schema#` for RDF-Schema) are removed as they occur in most queries.

Similarity Computation The goal of the third step is to compute the similarity of the stripped queries. Previous benchmark results have shown that most triple stores perform well for certain SPARQL features (Morsey et al. 2011). Thus, the goal of the similarity computation is to match queries that displayed the same features and to allow the subsequent clustering step to detect the prototypical queries for common sets of property combinations. In addition, our similarity function aims to detect queries that are similar in the order in which the SPARQL features were utilized. Let Q be the set of all queries. We represent each query $q \in Q$ by two properties:

- a binary feature vector $f(q)$ that contains a 1 for each SPARQL feature used by the query and a 0 else and
- the query string $s(q)$.

The similarity $sim(q, q')$ of two queries q and q' is then set to $sim(q, q') = (1+\delta(q, q'))^{-1}$, where the distance function $\delta(q, q')$ is given by

$$\delta(q, q') = \min(||f(q)-f(q')||^2, levenshtein(s(q), s(q'))). \quad (1)$$

Computing the Cartesian product of Q over both components would lead to almost 2.59 billion similarity computations. To reduce the runtime of the benchmark compilation, we use the LIMES framework[5] (Ngonga Ngomo and Auer 2011; Ngonga Ngomo 2011). For the similarity computation, we only consider queries q and q' such that

$$||f(q) - f(q')||^2 \leq \left\lceil \frac{\Delta}{100|Q|} \sum_{x \in Q} |f(x)| \right\rceil \quad (2)$$

and

$$levenshtein(s(q), s(q')) \leq \left\lceil \frac{\Delta}{100|Q|} \sum_{x \in Q} |s(x)| \right\rceil \quad (3)$$

where $|s(q)|$ resp. $|f(q)|$ stands for the length of a query string resp. a query vector. For the work presented herein, we use $\Delta = 2$. The average length of a stripped query string is 143.33 characters, while the feature vector has a constant length of 17. Consequently, only queries that bear a similarity of at least 1/3 with respect the similarity of their strings and 1/2 with respect to their feature vectors are included in the similarity graph. By using this restriction, we are able to reduce the runtime of the similarity computation to approximately 3% of the runtime required by the brute-force approach.

Clustering The final step of our approach is to apply graph clustering to the query similarity graph computed above. The goal of this step is to discover very similar groups queries out of which prototypical queries can be generated. As a given query can obey the patterns of more than one prototypical query, we opt for using the soft clustering approach implemented by the BorderFlow algorithm[6].

BorderFlow (Ngonga Ngomo and Schumacher 2009) implements a seed-based approach to graph clustering. It assumes a weighted graph $G = (V, E, \omega)$ as input, where $\omega : E \to \mathbb{R}$ is the weight function. The default setting of the algorithm (as used in the computation described below) consists of taking all nodes in the input graph as seeds. For each seed v, the algorithm begins with an initial cluster X containing only v. Then, it expands X iteratively by adding nodes from the direct neighborhood of X to X until X is node-maximal with respect to the border flow ratio[7]. The same procedure is repeated over all seeds. As different seeds can lead to the same cluster, identical clusters (i.e., clusters containing exactly the same nodes) that resulted from different seeds are subsequently collapsed to one cluster. The set of collapsed clusters and the mapping between each cluster and its seeds are returned as result. Applying BorderFlow to the input queries led to 622 clusters that contained more than one node, therewith confirming a long-tail distribution of query types across the query log at hand. We picked one query out of each cluster by choosing the query with the highest degree centrality in the cluster.

SPARQL Feature Selection and Query Variability

After the completion of the detection of similar queries and their clustering, our aim is now to select a number of frequently executed queries that cover most SPARQL features and allow us to assess the performance of queries with single

[5] `http://limes.sf.net`

[6] An implementation of the algorithm can be found at `http://borderflow.sf.net`. We used the CUGAR Framework found at `http://cugar-framework.sf.net` for the experiments described herein.

[7] See (Ngonga Ngomo and Schumacher 2009) for more details.

```
SELECT * WHERE {
  { ?v2  a           dbp-owl:Settlement ;
         rdfs:label  %%v%% .
    ?v6  a           dbp-owl:Airport . }
  { ?v6  dbp-owl:city  ?v2 . }
  UNION
  { ?v6  dbp-owl:location  ?v2 . }
  { ?v6  dbp-prop:iata     ?v5 . }
  UNION
  { ?v6  dbp-owl:iataLocationIdentifier  ?v5 . }
  OPTIONAL { ?v6  foaf:homepage  ?v7 . }
  OPTIONAL { ?v6  dbp-prop:nativename  ?v8 . }
}
```

Figure 3: Sample query with placeholder.

```
SELECT DISTINCT ?v WHERE {
  { ?v2  a           dbp-owl:Settlement ;
         rdfs:label  ?v .
    ?v6  a           dbp-owl:Airport . }
  { ?v6  dbp-owl:city  ?v2 . }
  UNION
  { ?v6  dbp-owl:location  ?v2 . }
  { ?v6  dbp-prop:iata     ?v5 . }
  UNION
  { ?v6  dbp-owl:iataLocationIdentifier ?v5 . }
  OPTIONAL { ?v6  foaf:homepage  ?v7 . }
  OPTIONAL { ?v6  dbp-prop:nativename  ?v8 . }
} LIMIT 1000
```

Figure 4: Sample auxiliary query returning potential values a placeholder can assume.

as well as combinations of features. The SPARQL features we consider are:

- the number of triple patterns contained in the query ($|GP|$),
- pattern constructors `UNION` (*UON*), `OPTIONAL` (*OPT*),
- the solution sequences and modifiers `DISTINCT` (*DST*),
- as well as the filter conditions and operators `FILTER` (*FLT*), `LANG` (*LNG*), `REGEX` (*REG*) and `STR` (*STR*).

We pick different numbers of triple patterns in order to include the efficiency of JOIN operations in triple stores. The other features were selected because they frequently occurred in the query log. We rank the clusters by the sum of the frequency of all queries they contain. Thereafter, we select 25 queries as follows: For each of the features, we choose the highest ranked cluster containing queries having this feature. From that particular cluster we select the query with the highest frequency.

In order to convert the selected queries into query templates, we manually select a part of the query to be varied. This is usually an IRI, a literal or a filter condition. In Figure 3 those varying parts are indicated by `%%v%%` or in the case of multiple varying parts `%%vn%%`. We exemplify our approach to replacing varying parts of queries by using Query 9, which results in the query shown in Figure 3. This query selects a specific settlement along with the airport belonging to that settlement as indicated in Figure 3. The variability of this query template was determined by getting a list of all settlements using the query shown in Figure 4. By selecting suitable placeholders, we ensured that the variability is sufficiently high ($\geq$ 1000 per query template). Note that the triple store used for computing the variability was different from the triple store that we later benchmarked in order to avoid potential caching effects.

For the benchmarking we then used the list of thus retrieved concrete values to replace the `%%v%%` placeholders within the query template. This method ensures, that (a) the actually executed queries during the benchmarking differ, but (b) always return results. This change imposed on the original query avoids the effect of simple caching.

Experimental Setup

This section presents the setup we used when applying the DBPSB2 on four triple stores commonly used in Data Web

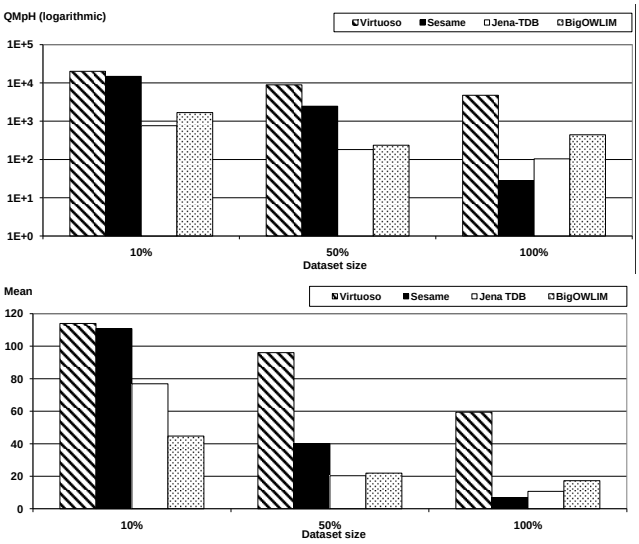

Figure 5: QMpH for all triple stores (top). Geometric mean of QpS (bottom).

applications. The hardware and software setup was exactly the same as in (Morsey et al. 2011). We used a typical server machine with 32GB RAM and an AMD Opteron 6 Core CPU with 2.8 GHz. All triple stores were allowed 8GB of memory. For executing the benchmark, we used DBpedia datasets with different scale factors, i.e. 10%, 50% and 100%. For DBPSB2, we used a warm-up period of 10 minutes, the duration of the hot-run phase to 30 minutes and the time-out and the time-out threshold to 180s. The benchmarking code along with the DBPSB2 queries is freely available[8].

Results

We evaluated the performance of the triple stores with respect to two main metrics: their overall performance on the benchmark and their query-based performance.

The overall performance of the triple stores was measured by computing its query mixes per hour (QMpH) as shown in

[8] `https://akswbenchmark.svn.sourceforge.net/svnroot/akswbenchmark/`

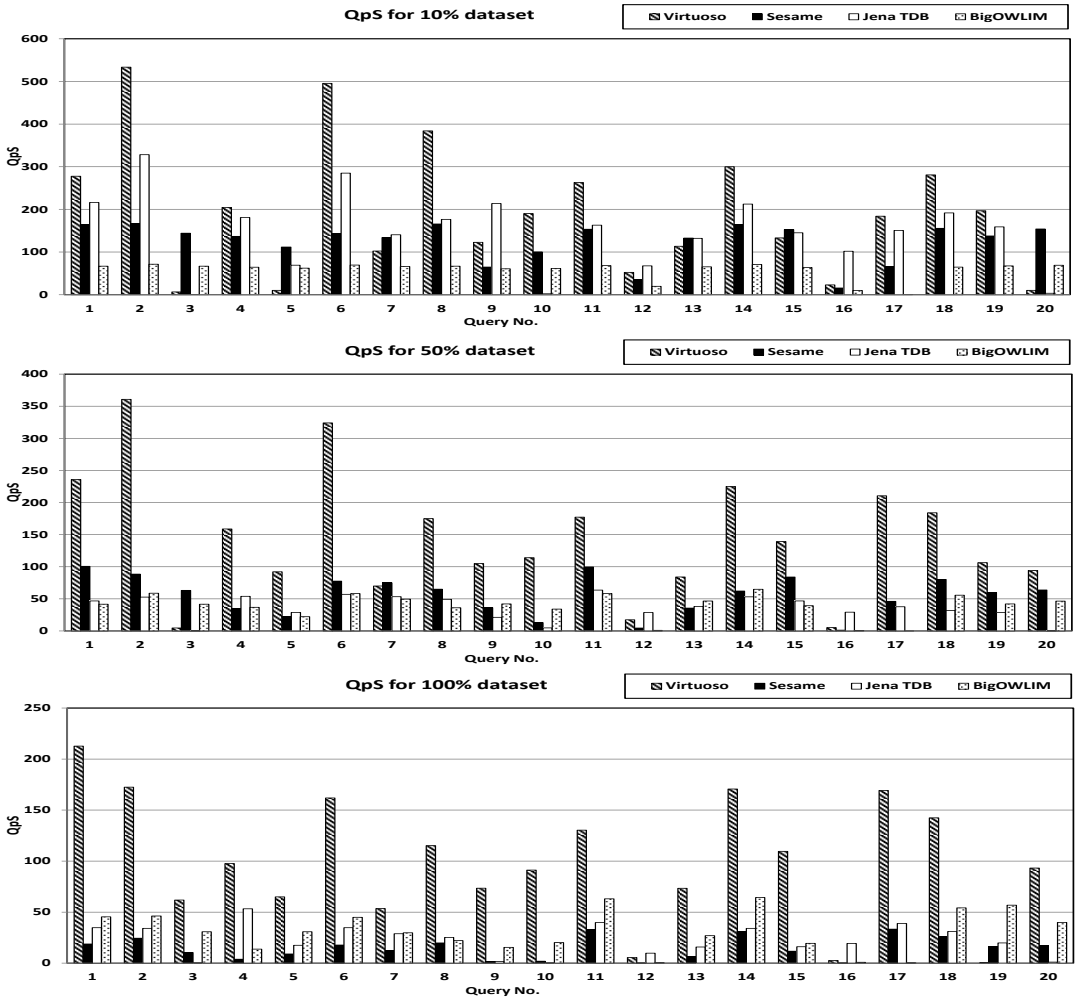

Figure 6: Queries per Second (QpS) for all triple stores for 10%, 50% and 100%.

Figure 5. Note that we used a logarithmic scale in this figure due to the high performance differences we observed. In general, Virtuoso was clearly the fastest triple store, followed by BigOWLIM, Sesame and Jena-TDB. The highest observed ratio in QMpH between the fastest and slowest triple store was 3.2 and it reached more than 1,000 for single queries. The scalability of stores did not vary as much as the overall performance. There was on average a linear decline in query performance with increasing dataset size.

The metric used for query-based performance evaluation is Queries per Second (QpS). QpS is computed by summing up the runtime of each query in each iteration, dividing it by the QMpH value and scaling it to seconds. The QpS results for all triple stores and for the 10%, 50% and 100% datasets are depicted in Figure 6. As the outliers (i.e. queries with very low QpS) affect the mean value of QpS for each store significantly, we also computed the geometric mean of all the QpS timings of queries for each store. The geometric mean for all triple stores is also depicted in Figure 5.

Although Virtuoso performed best overall, it displayed very low QpS rates on Q3, Q5 and Q16. All of these queries require dealing extensively with literals (in contrast to resources). Especially Q16 combined four different SPARQL features (optional,filter,lang and distinct) which seemed to require a significant amount of processing time. BigOWLIM was mainly characterized by a good scalability as it achieves the slowest decrease of its QMpH rates over all the datasets. Still, some queries were also particularly difficult to process for BigOWLIM. Especially Q16 and Q12 which involves three resp. four SPARQL features and a lot of string manipulations were slow to run. Sesame dealt well with most queries for the 10% dataset. The QMpH that it could achieve yet diminishes significantly with the size of the data set. This behavior becomes especially obvious when looking at Q4, Q10 and Q12. Especially Q4 which combines several triple patterns through a UNION leads to a considerable decrease of the runtime. Jena TDB had the most difficulties dealing with the 100% data set. This can be observed especially on Q9, which contained four triple patterns that might have lead to large intermediary results. Especially in the case Jena TDB, we observed that the 8GB RAM were not always sufficient for storing the intermediary results, which led to swapping and a considerable reduction of the overall performance of the system.

	LUBM	SP^{2}Bench	BSBM V2	BSBM V3	DBPSB2
RDF stores tested	DLDB-OWL, Sesame, OWL-JessKB	ARQ, Redland, SDB, Sesame, Virtuoso	Virtuoso, Sesame, Jena-TDB, Jena-SDB	Virtuoso, 4store, BigData, Jena-TDB, BigOWLIM	Virtuoso, Jena-TDB, BigOWLIM, Sesame
Test data	Synthetic	Synthetic	Synthetic	Synthetic	Real
Test queries	Synthetic	Synthetic	Synthetic	Synthetic	Real
Size of tested datasets	0.1M, 0.6M, 1.3M, 2.8M, 6.9M	10k, 50k, 250k, 1M,	1M, 25M, 100M, 5M, 25M	100M, 200M	14M, 75M, 150M
Dist. queries	14	12	12	12	20
Multi-client	–	–	x	x	–
Use case	Universities	DBLP	E-commerce	E-commerce	DBpedia
Classes	43	8	8	8	239(base) +300K(YAGO)
Properties	32	22	51	51	1200

Table 1: Comparison of different RDF benchmarks.

Related work

Several RDF benchmarks were previously developed. The *Lehigh University Benchmark* (LUBM) (Pan et al. 2005) was one of the first RDF benchmarks. LUBM uses an artificial data generator, which generates synthetic data for universities, their departments, their professors, employees, courses and publications. SP^{2}Bench (Schmidt et al. 2009) is another more recent benchmark for RDF stores. Its RDF data is based on the Digital Bibliography & Library Project (DBLP) and includes information about publications and their authors. Another benchmark described in (Owens, Gibbins, and mc schraefel 2008) compares the performance of BigOWLIM and AllegroGraph. The size of its underlying synthetic dataset is 235 million triples, which is sufficiently large. The benchmark measures the performance of a variety of SPARQL constructs for both stores. It also measures the performance of adding data, both using bulk-adding and partitioned-adding. The Berlin SPARQL Benchmark (BSBM) (Bizer and Schultz 2009) is a benchmark for RDF stores, which is applied to various triple stores, such as Sesame, Virtuoso, and Jena-TDB. It is based on an e-commerce use case in which a set of products is provided by a set of vendors and consumers post reviews regarding those products. It tests various SPARQL features on those triple stores by mimicking a real user.

A comparison between benchmarks is shown in Table 1. The main difference between previous benchmarks and ours is that we rely on real data and real user queries, while most of the previous approaches rely on synthetic data. LUBM's main drawback is that is solely relies on plain queries without SPARQL features such as `FILTER` or `REGEX`. In addition, its querying strategy (10 repeats of the same query) allows for caching. SP^{2}Bench relies on synthetic data and a small (25M triples) synthetic dataset for querying. The benchmark described in (Owens, Gibbins, and mc schraefel 2008) does not allow for testing the scalability of the stores, as the size of the data set is fixed. Finally, the BSBM data and queries are artificial and the data schema is very homogeneous and resembles a relational database.

In addition to general purpose RDF benchmarks it is reasonable to develop benchmarks for specific RDF data management aspects. One particular important feature in practical RDF triple store usage scenarios (as was also confirmed by DBPSB) is full-text search on RDF literals. In (Minack, Siberski, and Nejdl 2009) the LUBM benchmark is extended with synthetic scalable fulltext data and corresponding queries for fulltext-related query performance evaluation. RDF stores are benchmarked for basic fulltext queries (classic IR queries) as well as hybrid queries (structured and fulltext queries).

Conclusions and Future Work

We proposed the DBPSB2 benchmark for evaluating the performance of triple stores based on non-artificial data and queries. Our solution was implemented for the DBpedia dataset and tested with 4 different triple stores.The main advantage of our benchmark over previous work is that it uses real RDF data with typical graph characteristics including a large and heterogeneous schema part. By basing the benchmark on queries asked to DBpedia, we intend to spur innovation in triple store performance optimisation towards scenarios which are actually important for end users and applications. We applied query analysis and clustering techniques to obtain a diverse set of queries corresponding to feature combinations of SPARQL queries. Query variability was introduced to render simple caching techniques of triple stores ineffective.

The benchmarking results we obtained reveal that real-world usage scenarios can have substantially different characteristics than the scenarios assumed by prior RDF benchmarks. Our results are more diverse and indicate less homogeneity than what is suggested by other benchmarks. The creativity and inaptness of real users while constructing SPARQL queries is reflected by DBPSB and unveils for a certain triple store and dataset size the most costly SPARQL feature combinations. Several improvements are envisioned in future work to cover a wider spectrum of features in DBPSB2, especially the coverage of more SPARQL 1.1 features (e.g., reasoning and subqueries).

References

Auer, S.; Lehmann, J.; and Hellmann, S. 2009. LinkedGeoData - adding a spatial dimension to the web of data. In *Proc. of 8th International Semantic Web Conference (ISWC)*.

Belleau, F.; Nolin, M.-A.; Tourigny, N.; Rigault, P.; and Morissette, J. 2008. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics* 41(5):706–716.

Bishop, B.; Kiryakov, A.; Ognyanoff, D.; Peikov, I.; Tashev, Z.; and Velkov, R. 2011. Owlim: A family of scalable semantic repositories. *Semantic Web* 2(1):1–10.

Bizer, C., and Schultz, A. 2009. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.* 5(2):1–24.

Broekstra, J.; Kampman, A.; and van Harmelen, F. 2002. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, number 2342 in LNCS, 54–68. Springer.

Erling, O., and Mikhailov, I. 2007. RDF support in the virtuoso DBMS. In Auer, S.; Bizer, C.; Müller, C.; and Zhdanova, A. V., eds., *CSSW*, volume 113 of *LNI*, 59–68. GI.

Gray, J., ed. 1991. *The Benchmark Handbook for Database and Transaction Systems (1st Edition)*. Morgan Kaufmann.

Klyne, G., and Carroll, J. J. 2004. Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation.

Lehmann, J.; Bizer, C.; Kobilarov, G.; Auer, S.; Becker, C.; Cyganiak, R.; and Hellmann, S. 2009. DBpedia - a crystallization point for the web of data. *Journal of Web Semantics* 7(3):154–165.

Minack, E.; Siberski, W.; and Nejdl, W. 2009. Benchmarking fulltext search performance of RDF stores. In *ESWC2009*, 81–95.

Morsey, M.; Lehmann, J.; Auer, S.; and Ngonga Ngomo, A.-C. 2011. Dbpedia sparql benchmark – performance assessment with real queries on real data. In *ISWC 2011*.

Morsey, M.; Lehmann, J.; Auer, S.; Stadler, C.; ; and Hellmann, S. 2012. Dbpedia and the live extraction of structured data from wikipedia. *Program: electronic library and information systems* 46:27.

Ngonga Ngomo, A.-C., and Auer, S. 2011. Limes - a time-efficient approach for large-scale link discovery on the web of data. In *Proceedings of IJCAI*.

Ngonga Ngomo, A.-C., and Schumacher, F. 2009. Borderflow: A local graph clustering algorithm for natural language processing. In *CICLing*, 547–558.

Ngonga Ngomo, A.-C. 2011. A time-efficient hybrid approach to link discovery. In *Proceedings of OM@ISWC*.

Owens, A.; Seaborne, A.; Gibbins, N.; and mc schraefel. 2008. Clustered TDB: A clustered triple store for jena. Technical report, Electronics and Computer Science, University of Southampton.

Owens, A.; Gibbins, N.; and mc schraefel. 2008. Effective benchmarking for rdf stores using synthetic data.

Pan, Z.; Guo, Y.; ; and Heflin, J. 2005. LUBM: A benchmark for OWL knowledge base systems. In *Journal of Web Semantics*, volume 3, 158–182.

Prud'hommeaux, E., and Seaborne, A. 2008. SPARQL query language for RDF. W3C recommendation, W3C.

Schmidt, M.; Hornung, T.; Lausen, G.; and Pinkel, C. 2009. SP2Bench: A SPARQL performance benchmark. In *ICDE*, 222–233. IEEE.

Stadler, C.; Lehmann, J.; Höffner, K.; and Auer, S. 2011. Linkedgeodata: A core for a web of spatial open data. *Semantic Web Journal*.