DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data

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

ABSTRACT

Triple stores are the backbone of increasingly many Data Web applications. It is thus evident that the performance of those stores is mission critical for individual projects as well as for data integration on the Data Web in general. Consequently, it is of central importance during the implementation of any of these applications to have a clear picture of the weaknesses and strengths of current triple store implementations. In this article, we propose a generic SPARQL benchmark creation procedure, which we apply to the DBpedia knowledge base. Previous approaches often compared relational and triple stores and, thus, settled on measuring performance against a relational database which had been converted to RDF by using SQL-like queries. In contrast to those approaches, our benchmark is based on queries that were actually issued by humans and applications against existing RDF data not resembling a relational schema. Our generic procedure for benchmark creation is based on query-log mining, clustering and SPARQL feature analysis. We argue that a pure SPARQL benchmark is more useful to compare existing triple stores and provide results for the popular triple store implementations Virtuoso, Sesame, Jena-TDB, and BigOWLIM. The subsequent comparison of our results with other benchmark results indicates that the performance of triple stores is by far less homogeneous than suggested by previous benchmarks.

1. INTRODUCTION

Triple stores, which use IRIs for entity identification and store information adhering to the RDF data model [8] are the backbone of increasingly many Data Web applications. The RDF data model resembles directed labeled graphs, in which each labeled edge (called *predicate*) connects a *subject* to an *object*. The intended semantics is that the *object* denotes the value of the *subject's* property *predicate*. With the W3C SPARQL standard [16] a vendor-independent query language for the RDF triple data model exists. SPARQL is based on powerful graph matching allowing to bind 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 [17]. 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 [15], BSBM [4] and SP^2 [17] resemble relational database benchmarks. Especially the data structures underlying these benchmarks are basically relational data structures, with relatively few and homogeneously structured classes. However, there 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 [2] as well as knowledge bases extracted from unstructured or semi-structured sources such as DBpedia [9] or LinkedGeoData [1]. DBpedia (version 3.6) for example contains 289,016 classes of which 275 classes belong to the DBpedia ontology. Moreover, it contains 42,016 properties, of which 1335 are DBpedia-specific. Also, various datatypes and object references of different types are used in property values. Such knowledge bases can *not* 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. We evaluate two different methods *data generation* approaches and show how a representative set of resources preserving important dataset characteristics such as indegree and outdegree can be obtained by sampling across classes in the dataset. 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 DB-

VLDB '10, September 13-17, 2010, Singapore

pedia SPARQL endpoint. From the highest-ranked query clusters (in terms of aggregated query frequency), we derive a set of 25 SPARQL query templates, which cover most commonly used SPARQL features and are used to generate the actual benchmark queries by parametrization. We call the benchmark resulting from this dataset and query generation methodology *DBPSB* (i.e. DBpedia SPARQL Benchmark). The benchmark methodology and results are also available online[1]. 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 the DBPSB can change 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 are postulated in the Benchmark Handbook [7], 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 the DBPSB to assess the performance and scalability of the popular triple stores *Virtuoso* [6], *Sesame* [5], *Jena-TDB* [14], and *BigOWLIM* [3] 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. As we explain in more detail later, we believe this is due to the different nature of DBPSB compared to the previous approaches resembling relational databases benchmarks. 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 remainder of the paper is organized as follows. In Section 2, we describe the dataset generation process in detail. We show the process of query analysis and clustering in detail in Section 3. In Section 4, we present our approach to selecting SPARQL features and to query variability. The assessment of four triple stores via the DBPSB is described in Section 5. The results of the experiment are discussed in Section 7. We present related work in Section 8 and conclude our paper in Section 9.

2. DATASET GENERATION

A crucial step in each benchmark is the generation of suitable datasets. Although we describe the dataset generation here with the example of DBpedia, the methodology we pursue is dataset-agnostic. Consequently, it can be applied to arbitrary knowledge bases, thus enabling the generation of domain-specific benchmarks.

The data generation for DBPSB is guided by the following requirements:

- The DBPSB data should resemble the original data (i.e., DBpedia data in our case) as much as possible, in particular the large number of classes, properties, the heterogeneous property value spaces as well as the large taxonomic structures of the category system should be preserved.
- The data generation process should allow to generate knowledge bases of various sizes ranging from a few million to several hundred million or even billion triples.
- Basic network characteristics of different sizes of the network should be similar, in particular the in- and outdegree.
- The data generation process should be easily repeatable with new versions of the considered dataset.

The proposed dataset creation process starts with an input dataset. For the case of DBpedia, it consists of the datasets loaded into the official SPARQL endpoint[2]. Datasets of multiple size of the original data are created by duplicating all triples and changing their namespaces. This procedure can be applied for any scale factors. While simple, this procedure is efficient to execute and fulfills the above requirements.

For generating smaller datasets, we investigated two different methods. The first method (called "rand") consists of selecting an appropriate fraction of all triples of the original dataset randomly. If RDF graphs are considered as small world graphs, removing edges in such graphs should preserve the properties of the original graph. The second method (called "seed") is based on the assumption that a representative set of resources can be obtained by sampling across classes in the dataset. Let x be the desired scale factor in percent, e.g. $x = 10$. The method first selects $x\%$ of the classes in the dataset. For each selected class, 10% of its instances are retrieved and added to a queue. For each element of the queue, its concise bound description (CBD) [18] is retrieved. This can lead to new resources, which are appended at the end of the queue. This process is iterated until the target dataset size, measured in number of triples, is reached.

Since the selection of the appropriate method for generating small datasets is an important issue, we performed a statistical analysis on the generated datasets for DBpedia. The statistical parameters used to judge the datasets are the average indegree, the average outdegree, and the number of nodes, i.e. number of distinct IRIs in the graph. We calculated both the in- and the outdegree for datasets once with literals ignored, and another time with literals taken into consideration, as it gives more insight on the degree of similarity between the dataset of interest and the full DBpedia dataset. The statistics of those datasets are given in Table 1. According to this analysis, the seed method fits our purpose of maintaining basic network characteristics better, as the average in- and outdegree of nodes are closer to the original dataset. For this reason, we selected this method for generating the DBPSB.

3. QUERY ANALYSIS AND CLUSTERING

The goal of the query analysis and clustering is to detect prototypical queries that were sent to the official DBpedia

[1] `http://aksw.org/Projects/DBPSB`

[2] Endpoint: `http://dbpedia.org/sparql`, Loaded datasets: `http://wiki.dbpedia.org/DatasetsLoaded`

Table 1: Statistical analysis of DBPSB datasets.

Dataset	Indegree w/ literals	Outdegree w/ literals	Indegree w/o literals	Outdegree w/o literals	No. of nodes	No. of triples
Full DBpedia	5.45	30.52	3.09	15.57	27,665,352	153,737,776
10% dataset (seed method)	6.54	45.53	3.98	23.05	2,090,714	15,267,418
10% dataset (rand method)	3.82	6.76	2.04	3.41	5,260,753	16,739,055
50% dataset (seed method)	6.79	38.08	3.82	18.64	11,317,362	74,889,154
50% dataset (rand method)	7.09	26.79	3.33	10.73	9,581,470	78,336,781

SPARQL endpoint. To achieve this goal, we follow a four-step approach. First, we select queries that were executed frequently on the input data source. In this case, this step is carried out by selecting frequent queries from the DBpedia SPARQL endpoint log. 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. Third, we compute a query similarity graph from the stripped queries. 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.

3.1 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[3]. 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 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. The renaming of variables also helps in the clustering step, as it leads to less noise when computing the similarity of queries and thus to a higher cluster purity. As a result, distinguishing query constructs such as `REGEX` or `DISTINCT` are a higher influence on the clustering.

- *Query frequency.* We discard queries with 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 just 35,965. This reduction allows our benchmark to capture the essence of the queries posed to DBpedia within the timespan covered by the query log and reduces the runtime of the subsequent steps substantially.

3.2 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 appear in most queries.

3.3 Similarity Computation

The goal of the third step is to compute the similarity of the stripped queries. Computing the Cartesian product of the queries would lead to a quadratic runtime, i.e., almost 1.3 billion similarity computations. To reduce the runtime of the benchmark compilation, we use the LIMES framework [11][4].

The LIMES approach makes use of the interchangeability of similarities and distances. It presupposes a metric space in which the queries are expressed as single points. Instead of aiming to find all pairs of queries such that $sim(q, p) \geq \theta$, LIMES aims to find all pairs of queries such that $d(q, p) \leq \tau$, where sim is a similarity measure and d is the corresponding metric. To achieve this goal, when given a set of n queries, it first computes $\sqrt{n}$ so-called *exemplars*, which are prototypical points in the affine space that subdivide it into regions of high heterogeneity. Then, each query is mapped to the exemplar it is least distant to. The list of queries mapped to an exemplar is then sorted in descending order of distance. The characteristics of metrics spaces (especially the triangle inequality) ensures that the distances from each query q to any other query p obeys the following inequality

$$d(q, e) - d(e, p) \leq d(q, p) \leq d(q, e) + d(e, p), \qquad (1)$$

where e is an exemplar and d is a metric. Consequently,

$$d(q, e) - d(e, p) > \tau \Rightarrow d(q, p) > \tau. \qquad (2)$$

Given that $d(q, e)$ is constant, q must only be compared to the first elements of the list of queries mapped to e that fulfill the inequality above. We can ignore all p that have an index superior to the first query linked to e that does not satisfy the inequality. By these means, the number of similarity computation can be reduced significantly. In this particular use case, we cut down the number of computations to only 16.6% of the Cartesian product without any loss in recall. For the current version of the benchmark, we used the *Levenshtein* string similarity measure and a threshold of 0.9.

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

[4]Available online at: `http://limes.sf.net`

3.4 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[5].

BorderFlow [12] implements a seed-based approach to graph clustering. The default setting for the seeds 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 a function called the border flow ratio. 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 12272 clusters, of which 24% contained only one node, hinting towards a long-tail distribution of query types. To generate the patterns used in the benchmark, we only considered clusters of size 5 and above.

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

- the overall number of triple patterns contained in the query ($|GP|$),
- the graph 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 picked 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 order the query clusters by using the sum of frequencies of queries contained in each cluster to obtain a ranked list of clusters. 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. A full list of selected queries is presented in the Appendix (we omitted namespace declarations and slightly adapted the formatting for brevity). An overview of the features and which of the queries comprise these features is given in Table 2.

[5]An implementation of the algorithm can be found at `http://borderflow.sf.net`

```
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 1: 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 2: Sample auxiliary query returning potential values a placeholder can assume.

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 1 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 1.

This query selects a specific settlement along with the airport belonging to that settlement as indicated in Figure 1. The variability of this query template was determined by getting a list of all settlements using the query shown in Figure 2. By selecting suitable placeholders, we ensured that the variability is sufficiently high (approx. 1000 per query template). The triple store, which we used for computing the variability is different from the triple store, which we later benchmarked in order to avoid potential caching effects.

For the benchmarking we then used the list of thus retrieved concrete values (in this case literals of the settlements) 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.

5. EXPERIMENTAL SETUP

This section presents the setup we used when applying the DBPSB on four triple stores commonly used in Data Web applications. We first describe the triple stores and their configuration, followed by our experimental strategy and finally the obtained results. All experiments were conducted on a typical server machine with an AMD Opteron 6 Core CPU with 2.8 GHz, 32 GB RAM, 3 TB RAID-5 HDD run-

Table 2: SPARQL query templates.

No.	$\|GP\|$	DST	FLT	OPT	UON	LNG	REG	STR
1	1	X						
2	1		X					
3	5			X				
4	4				X			
5	9	X			X			
6	3	X	X					
7	12	X		X				
8	4		X		X			
9	9			X	X			
10	8	X	X	X				
11	9	X		X	X			
12	4	X	X			X		
13	2	X	X				X	
14	3		X		X	X		
15	4		X	X		X		
16	3	X	X	X		X		
17	4	X	X		X	X		
18	2		X		X	X		X
19	2		X		X		X	X
20	12		X	X	X	X		
21	1							
22	2							
23	3							
24	4							
25	5							

ning Linux Kernel 2.6.35-23-server and Java 1.6 installed. The benchmark program and the triple store were run on the same machine to avoid network latency.

5.1 Triple Stores Setup

We carried out our experiments by using the triple stores *Virtuoso* [6], *Sesame* [5], *Jena-TDB* [14], and *BigOWLIM* [3]. The configuration and the version of each triple store were as follows:

1. **Virtuoso** Open-Source Edition version 6.1.2: We set the following memory-related parameters: NumberOfBuffers = 1048576, MaxDirtyBuffers = 786432.

2. **Sesame** Version 2.3.2 with Tomcat 6.0 as HTTP interface: We used the native storage layout and set the spoc, posc, opsc indices in the native storage configuration. We set the Java heap size to 8GB.

3. **Jena-TDB** Version 0.8.7 with Joseki 3.4.3 as HTTP interface: We configured the TDB optimizer to use statistics. This mode is most commonly employed for the TDB optimizer, whereas the other modes are mainly used for investigating the optimizer strategy. We also set the Java heap size to 8GB.

4. **BigOWLIM** Version 3.4, with Tomcat 6.0 as HTTP interface: We set the entity index size to 45,000,000 and enabled the predicate list. The rule set was empty. We set the Java heap size to 8GB.

In summary, we configured all triple stores to use 8GB of memory and used default values otherwise.

5.2 Benchmark Execution

Once the triple stores loaded the DBpedia datasets with different scale factors, i.e. 10%, 50%, 100%, and 200%, the benchmark execution phase begins. It comprises the following stages:

1. **System Restart**: Before running the experiment, the triple store and its associated programs are restarted in order to clear memory caches.

2. **Warm-up Phase**: In order to measure the performance of a triple store under normal operational conditions, a warm-up phase is used. In the warm-up phase, query mixes are posed to the triple store. The queries posed during the warm-up phase are disjoint with the queries posed in the hot-run phase. For DBPSB, we used a warm-up period of 20 minutes.

3. **Hot-run Phase**: During this phase, the benchmark query mixes were sent to the tested store. We kept track of the average execution time of each query as well as the number of query mixes per hour (QMpH). The duration of the hot-run phase in DBPSB was 60 minutes.

Since some benchmark queries did not respond within reasonable time, we specified a 180 second timeout after which a query was aborted and the 180 second maximum query time was used as the runtime for the given query even though no results were returned. The benchmarking code along with the DBPSB queries is freely available[6].

6. 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.

6.1 Overall Performance Measurement

The overall performance of any triple store was measured by computing its query mixes per hour (QMpH) as shown in Figure 4. Please 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 63.5 and it reached more than 10 000 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. Details will be discussed in Section 7.

6.2 Timeout

We tested the queries that each triple store failed to executed withing the 180s timeout and noticed that even much larger timeouts would not have been sufficient most of those queries. We did not exclude the queries completely from the overall assessment, since this would have affected a large number of the queries and adversely penalized stores, which complete queries within the time frame. We penalized failure queries with 180s. A similar penalization technique was performed in the SP2-Benchmark [17]. Virtuoso was the only store, which was able to complete all queries in all dataset sizes within the 180 second timeframe. However, with Sesame and OWLIM only rarely a few particular queries timed out. Jena-TDB had always severe problems with queries 7, 10 and 20 as well as 3, 9, 12 for the larger two datasets (cf. Appendix C).

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

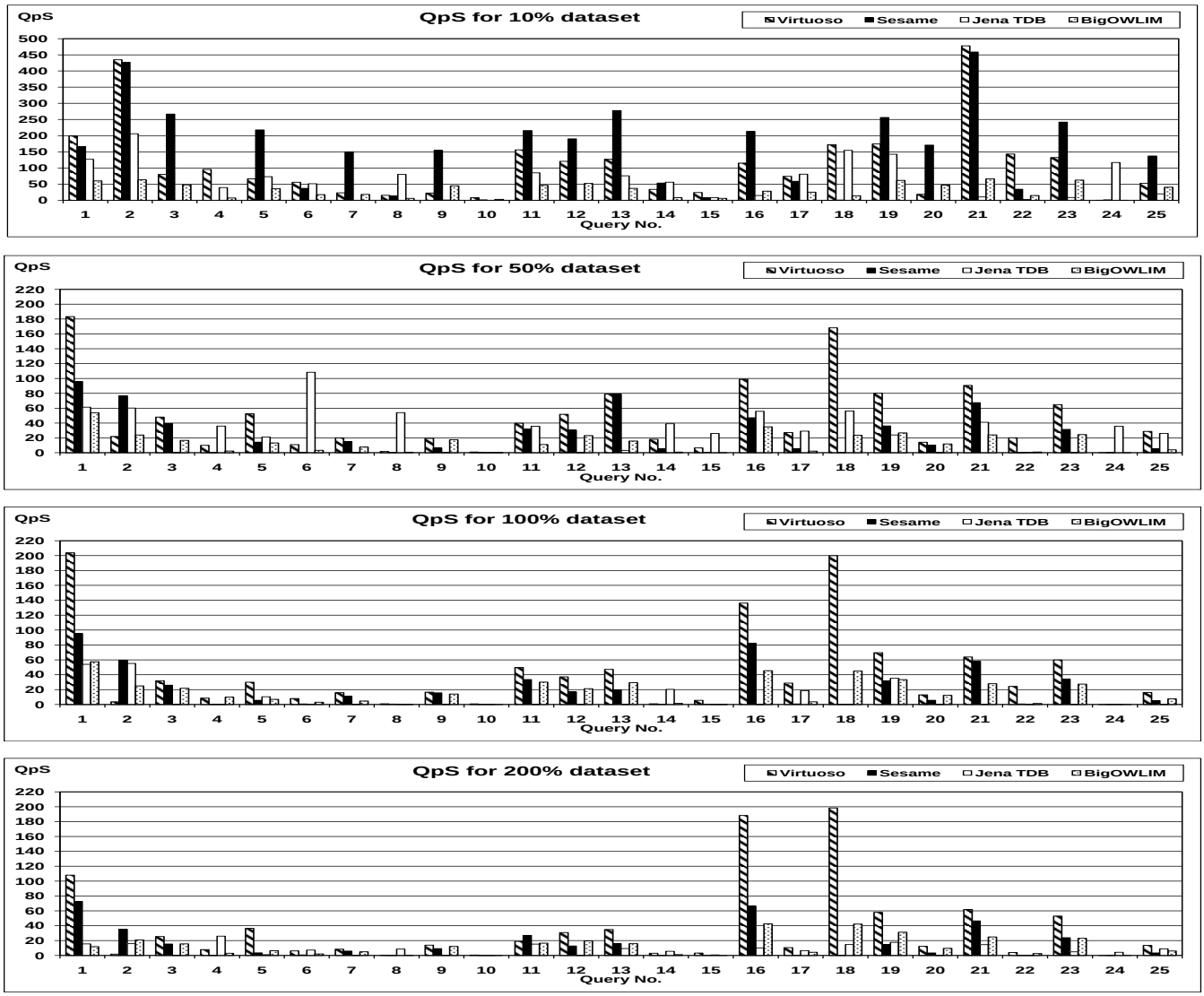

Figure 3: Queries per Second (QpS) for all triple stores for 10%, 50%, 100%, and 200% datasets.

6.3 Query-based Performance

The metric used for query-based performance evaluation is Queries per Second (QpS). The steps required to obtain QpS for each query are as follows:

1. During the hot-run phase, we sum up the runtime of each query during each iteration.
2. This sum is divided by QMpH value and scaled to seconds by dividing through 3600.

The QpS results for all triple stores and for the 10%, 50%, 100%, and 200% datasets are depicted in Figure 3.

The outliers, i.e. queries with very low QpS, will significantly affect the mean value of QpS for each store. So, we additionally calculated the geometric mean of all the QpS timings of queries for each store. The geometric mean for all triple stores is depicted in Figure 5. By reducing the effect of outliers, we obtained additional information from this figure (when compared to Figure 4) as we will describe in the sequel.

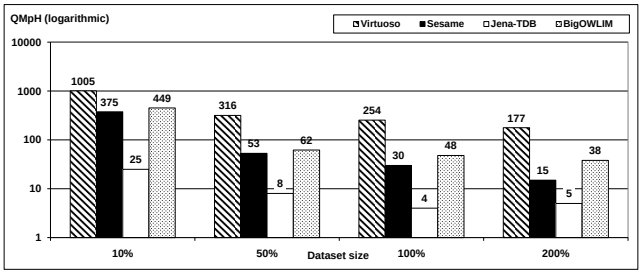

Figure 4: QMpH for all triple stores.

Furthermore, we provide detailed results for each query in Appendix C, which shows queries per second, the geometric mean of query runtime as well as the standard deviation of query runtime. It is useful to observe results in more detail as we will do in the next section.

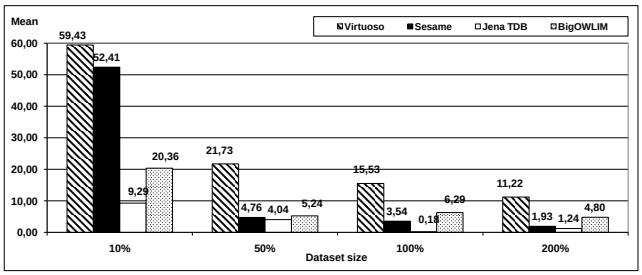

Figure 5: Geometric mean of QpS for all triple stores.

7. DISCUSSION

This section consists of three parts: First, we compare the general performance of the systems under test. Then we look individual queries and the SPARQL features used within those queries in more detail to observe particular strengths and weaknesses of stores. Thereafter, we compare our results with those obtained with previous benchmarks and elucidate some of the main differences between them.

7.1 General Performance

Figure 4 depicts the benchmark results for query mixes per hour for the four systems and dataset sizes. Virtuoso leads the field with a substantial head start of double the performance for the 10% dataset (and even quadruple for other dataset sizes) compared to the second best system (BigOWLIM). While Sesame is able to keep up with BigOWLIM for the smaller two datasets it considerably looses ground for the larger datasets. Jena-TDB can in general not deliver competitive performance with being by a factor 30-50 slower than the fastest system.

If we look at the geometric mean of all QpS results in Figure 5, we observe similar insights. The spreading effect is weakened, since the geometric mean reduces the effect of outliers. Still Virtuoso is the fastest system, although Sesame manages to get pretty close for the 10% dataset. This shows that most, but not all, queries are fast in Sesame for low dataset sizes. For the larger datasets, BigOWLIM is the second best system and shows promising scalability, but it is still by a factor of two slower than Virtuoso.

7.2 Scalability, Individual Queries and SPARQL Features

Our first observation with respect to individual performance of the triple stores is that Virtuoso demonstrates a good scaling factor on the DBSBM. When dataset size changes by factor 5 (from 10% to 50%), the performance of the triple store only degrades by factor 3.12. Further dataset increases (i.e. the doubling to the 100% and 200% datasets) result in only relatively small performance decreases by 20% and respectively 30%.

Virtuoso outperforms Sesame for all datasets. In addition, Sesame does not scale as well as Virtuoso for small dataset sizes, as its performance degrades sevenfold when the dataset size changes from 10% to 50%. However, when the dataset size doubles from the 50% to the 100% dataset and from 100% to 200% the performance degrades by just half.

The performance of Jena-TDB is the lowest of all triple stores and for all dataset sizes. The performance degradation factor of Jena-TDB is not as pronounced as that of Sesame and almost equal to that of Virtuoso when changing from the 10% to the 50% dataset. However, the performance of Jena-TDB only degrades by a factor of 2 for the transition between the 50% and 100% dataset, and reaches 0.8 between the 100% and 200% dataset, leading to a slight increase of its QMpH.

BigOWLIM is the second fastest triple store for all dataset sizes, after Virtuoso. BigOWLIM degrades with a factor of 7.2 in transition from 10% to 50% datasets, but it decreases dramatically to 1.29 with dataset size 100%, and eventually reaches 1.26 with dataset size 200%.

Due to the high diversity in the performance of different SPARQL queries, we also computed the geometric mean of the QpS values of all queries as described in the previous section and illustrated in Figure 5. By using the geometric mean, the resulting values are less prone to be dominated by a few outliers (slow queries) compared to standard QMpH values. This allows for some interesting observations in DBPSB by comparing Figure 4 and 5. For instance, it is evident that Virtuoso has the best QpS values for all dataset sizes.

With respect to Virtuoso, query 10 performs quite poorly. This query involves the features `FILTER`, `DISTINCT`, as well as `OPTIONAL`. Also, the well performing query 1 involves the `DISTINCT` feature. Query 3 involves a `OPTIONAL` resulting in worse performance. Query 2 involving a `FILTER` condition results in the worst performance of all of them. This indicates that using complex `FILTER` in conjunction with additional `OPTIONAL`, and `DISTINCT` adversely affects the overall runtime of the query.

Regarding Sesame, queries 4 and 18 are the slowest queries. Query 4 includes `UNION` along with several free variables, which indicates that using `UNION` with several free variables causes problems for Sesame. Query 18 involves the features `UNION`, `FILTER`, `STR` and `LANG`. Query 15 involves the features `UNION`, `FILTER`, and `LANG`, and its performance is also pretty slow, which leads to the conclusion that introducing this combination of features is difficult for Sesame. Adding the `STR` feature to that feature combination affects the performance dramatically and prevents the query from being successfully executed.

For Jena-TDB, there are several queries that timeout with large dataset sizes, but queries 10 and 20 always timeout. The problem with query 10 is already discussed with Virtuoso. Query 20 contains `FILTER`, `OPTIONAL`, `UNION`, and `LANG`. Query 2 contains `FILTER` only, query 3 contains `OPTIONAL`, and query 4 contains `UNION` only. All of those queries run smoothly with Jena-TDB, which indicates that using the `LANG` feature, along with those features affects the runtime dramatically.

For BigOWLIM, queries 10, and 15 are slow queries. Query 10 was already problematic for Virtuoso, as was query 15 for Sesame.

Query 24 is slow on Virtuoso, Sesame, and BigOWLIM, whereas it is faster on Jena-TDB. This is due to the fact that most of the time this query returns many results. Virtuoso, and BigOWLIM return a bulk of results at once, which takes long time. Jena-TDB just returns the first result as a starting point, and iteratively returns the remaining results via a buffer.

It is interesting to note that BigOWLIM shows in general good performance, but almost never manages to out-

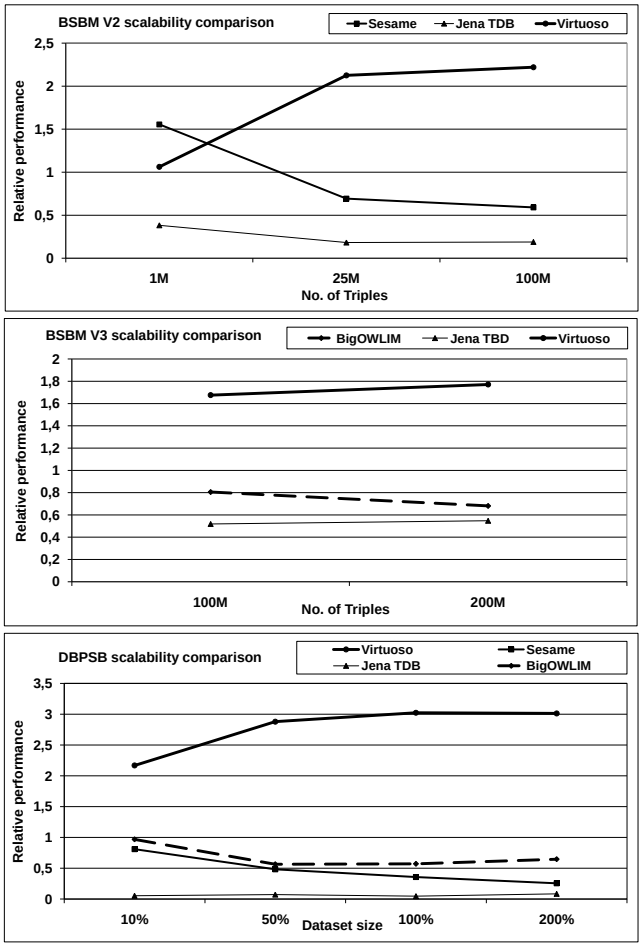

Figure 6: Comparison of triple store scalability between BSBM V2, BSBM V3, DBPSB.

perform any of the other stores. Queries 11, 13, 19, 21 and 25 were performed with relatively similar results across triple stores (cf. column min-max ratio in Table 4) thus indicating that the features of these queries (i.e. UON, REG, FLT) are already relatively well supported. With queries 3, 4, 7, 9, 12, 18, 20 we observed dramatic differences between the different implementations with factors between slowest and fastest store being higher than 1000. It seems that a reason for this could be the poor support for OPT (in queries 3, 7, 9, 20) as well as certain filter conditions such as LNG in some implementations, which demonstrate the need for further optimizations.

7.3 Comparison with Previous Benchmarks

In order to visualize the performance improvement or degradation of a certain triple store compared to its competitors, we calculated the relative performance for each store compared to the average and depicted it for each dataset size in Figure 6. We also performed this calculation for BSBM version 2 and version 3. Overall, the benchmarking results with DBPSB were less homogeneous than the results of previous benchmarks. While with other benchmarks the ratio between fastest and slowest query rarely exceeds a factor of 50, the factor for the DBPSB queries (derived from real DBpedia SPARQL endpoint queries) reaches more than 1 000 in some cases (see Table 4).

As with the other benchmarks, Virtuoso was also fastest in our measurements. However, the performance difference is even higher than reported previously: Virtuoso reaches a factor of 3 in our benchmark compared to 1.8 in BSBM V3. BSBM V2 and our benchmark both show that Sesame is more suited to smaller datasets and does not scale as well as other stores. Jena-TDB is the slowest store in BSBM V3 and DBPSB, but in our case they fall much further behind to the point that Jena-TDB can hardly be used for some of the queries, which are asked to DBpedia. 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.

8. RELATED WORK

Several RDF benchmarks were previously developed. The *Lehigh University Benchmark* (LUBM) [15] 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. This small number of classes limits the variability of data and makes LUMB inherent structure more repetitive. Moreover, the SPARQL queries used for benchmarking in LUBM are all plain queries, i.e. they contain only triple patterns with no other SPARQL features (e.g. `FILTER`, or `REGEX`). LUBM performs each query 10 consecutive times, and then it calculates the average response time of that query. Executing the same query several times without introducing any variation enables query caching, which affects the overall average response time calculated.

SP^{2}Bench [17] 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. It uses the SP^{2}Bench Generator to generate its synthetic test data, which is in its schema heterogeneity even more limited than LUMB. The main advantage of SP^{2}Bench over LUBM is that its test queries include a variety of SPARQL features (such as `FILTER`, and `OPTIONAL`). The main difference between the DBpedia benchmark and SP^{2}Bench is that both test data and queries are synthetic in SP^{2}Bench. In addition, SP^{2}Bench only published results for up to 25M triples, which is relatively small with regard to datasets such as DBpedia and LinkedGeoData.

Another benchmark is described in [13]. It compares the performance of two different triple stores, namely 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 when running in single and in multi-threaded modes. It also measures the performance of adding data, both using bulk-adding and partitioned-adding. The downside of that benchmark is that it compares the performance of only two triple stores. Also the performance of each triple store is not assessed for different dataset sizes, which prevents scalability comparisons.

The Berlin SPARQL Benchmark (BSBM) [4] is a benchmark for RDF stores, which is applied to various triple stores, such as Sesame, Virtuoso, and Jena-TDB. It is based

Table 3: Comparison of different RDF benchmarks.

	LUBM	SP^{2}Bench	BSBM V2	BSBM V3	DBPSB
RDF stores tested	DLDB-OWL, Sesame, OWL-JessKB	ARQ, Redland, SDB, Sesame, Virtuoso	Virtuoso, Sesame, Jena-TDB, Jena-SDB	Virtuoso, 4store, BigOwlim, BigData Jena-TDB	Virtuoso, Sesame, Jena-TDB, BigOWLIM
Test data	Synthetic	Synthetic	Synthetic	Synthetic	Real
Test queries	Synthetic	Synthetic	Synthetic	Synthetic	Real
Size of tested datasets	103K, 646K, 1.3M, 2.8M, 6.9M	10k, 50k, 250k, 1M,	1M, 25M, 100M, 5M, 25M	100M, 200M	14M, 75M, 150M, 300M
Distinct queries	14	12	12	12	25
Multi-client	–	–	x	x	–
Use case	Universities	DBLP	E-commerce	E-commerce	DBpedia
Classes	43	8	8	8	239 (internal) + 300K (YAGO)
Properties	32	22	51	51	1200

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. It tries to mimic a real user operation, i.e. it orders the query in a manner that resembles a real sequence of operations performed by a human user. This is an effective testing strategy. However, BSBM data and queries are artificial and the data schema is very homogeneous and resembles a relational database. This is reasonable for comparing the performance of triple stores with RDBMS, but does not give many insights regarding the specifics of RDF data management.

A comparison between benchmarks is shown in Table 3.

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 [10] 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).

9. CONCLUSIONS AND FUTURE WORK

We proposed the DBPSB 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, namely Virtuoso, Sesame, Jena-TDB, and BigOWLIM. 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. Furthermore, 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 can be envisioned in future work to cover a wider spectrum of features in DBPSB:

- Coverage of more SPARQL 1.1 features, e.g. reasoning and subqueries.
- Inclusion of further triple stores and continuous usage of the most recent DBpedia query logs.
- Testing of SPARQL update performance via DBpedia Live, which is modified several thousand times each day. In particular, an analysis of the dependency of query performance on the dataset update rate could be performed.

Acknowledgments

We thank the people at Openlink software for maintaining the DBpedia SPARQL endpoint, which was a crucial resource for the creation of the DBPSB benchmark. We would also like to thank Barry Bishop from Ontotext for supporting us with the BigOWLIM installation. Peter Boncz and Orri Erling from the LOD2 project team provided some important feedback on certain aspects of DBPSB. This work was supported by a grant from the European Union's 7th Framework Programme provided for the projects LOD2 (GA no. 257943) and LATC (GA no. 256975) and the Eurostars grant E!4604 (SCMS).

10. REFERENCES

[1] Sören Auer, Jens Lehmann, and Sebastian Hellmann. LinkedGeoData - adding a spatial dimension to the web of data. In *Proc. of 8th International Semantic Web Conference (ISWC)*, 2009.

[2] Franois Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics*, 41(5):706–716, 2008.

[3] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. Owlim: A family of scalable semantic repositories. *Semantic Web*, (1):1–10, 2011.

[4] Christian Bizer and Andreas Schultz. "The Berlin SPARQL Benchmark.". *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[5] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In I. Horrocks and J. Hendler, editors, *Proceedings of the First Internation Semantic Web Conference*, number 2342 in Lecture Notes in Computer Science, pages 54–68. Springer Verlag, July 2002.

[6] Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In Sören Auer, Christian Bizer, Claudia Müller, and Anna V. Zhdanova, editors, *CSSW*, volume 113 of *LNI*, pages 59–68. GI, 2007.

[7] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (1st Edition).* Morgan Kaufmann, 1991.

[8] Graham Klyne and Jeremy J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210, February 2004.

[9] Jens Lehmann, Chris Bizer, Georgi Kobilarov, Sren Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - a crystallization point for the web of data. *Journal of Web Semantics*, 7(3):154–165, 2009.

[10] Enrico Minack, Wolf Siberski, , and Wolfgang Nejdl. "Benchmarking fulltext search performance of RDF stores". In *6th Annual European Semantic Web Conference (ESWC2009)*, pages 81–95, June 2009.

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

[12] A.-C. Ngonga Ngomo and F. Schumacher. "borderflow: A local graph clustering algorithm for natural language processing". In *CICLing*, pages 547–558, 2009.

[13] Alisdair Owens, Nick Gibbins, and mc schraefel. Effective benchmarking for rdf stores using synthetic data, May 2008.

[14] Alisdair Owens, Andy Seaborne, Nick Gibbins, and mc schraefel. Clustered TDB: A clustered triple store for jena. Nonpeerreviewed, Electronics and Computer Science, University of Southampton, November 2008.

[15] Zhengxiang Pan, Yuanbo Guo, , and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems.". In *Web Semantics: Science, Services and Agents on the WWW*, volume 3, pages 158–182, 2005.

[16] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[17] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. "SP2Bench: A SPARQL performance benchmark.". In *ICDE*, pages 222–233. IEEE, 2009.

[18] Patrick Stickler. CBD - concise bounded description, 2005. Retrieved February 15, 2011, from `http://www.w3.org/Submission/CBD/`.

APPENDIX

A. PREFIXES

The following prefixes are used throughout the paper:

```
dbpedia:  http://dbpedia.org/resource/
dbp-owl:  http://dbpedia.org/ontology/
dbp-prop: http://dbpedia.org/property/
dbp-yago: http://dbpedia.org/class/yago/
dbp-cat:  http://dbpedia.org/resource/Category/
rdf:      http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:     http://www.w3.org/2000/01/rdf-schema#
owl:      http://www.w3.org/2002/07/owl#
xsd:      http://www.w3.org/2001/XMLSchema#
skos:     http://www.w3.org/2004/02/skos/core#
foaf:     http://xmlns.com/foaf/0.1/
georss:   http://www.georss.org/georss/
```

B. QUERY LIST

```
1  SELECT DISTINCT ?v1 WHERE { %%v%% rdf:type ?v1. }
2  SELECT * WHERE { %%v%% ?v2 ?v1. FILTER(?v2=dbpedia2:
   redirect || ?v2=dbp-prop:redirect) }
3  SELECT ?v4 ?v8 ?v10 WHERE { ?v5 dbp-owl:thumbnail ?v4
   . ?v5 rdf:type dbp-owl:Person. ?v5 rdfs:label %%v%%.
   ?v5 foaf:page ?v8. OPTIONAL { ?v5 foaf:homepage ?v10
   .} }
4  SELECT ?v5 ?v6 ?v9 ?v8 ?v4 WHERE { { %%v%% ?v5 ?v6. ?
   v6 foaf:name ?v8. } UNION { ?v9 ?v5 %%v%%; foaf:name
   ?v4. } }
5  SELECT DISTINCT ?v3 ?v4 ?v5 WHERE { { ?v3 dbp-prop:
   series %%v1%%; foaf:name ?v4; rdfs:comment ?v5; rdf:
   type %%v0%%. } UNION { ?v3 dbp-prop:series ?v8. ?v8
   dbp-prop:redirect %%v1%%. ?v3 foaf:name ?v4; rdfs:
   comment ?v5; rdf:type %%v0%%. } }
6  SELECT DISTINCT ?v3 ?v5 ?v7 WHERE { ?v3 rdf:type dbp-
   yago:Company108058098. ?v3 dbp-prop:numEmployees ?v5
   FILTER (xsd:integer(?v5)>=%%v%%). ?v3 foaf:homepage ?
   v7. }
7  SELECT distinct ?v0 ?v1 ?v2 ?v3 ?v5 ?v6 ?v7 ?v10
   WHERE { ?v0 rdfs:comment ?v1. ?v0 foaf:page %%v%%
   OPTIONAL{?v0 skos:subject ?v6} OPTIONAL{?v0 dbp-prop:
   industry ?v5}OPTIONAL{?v0 dbp-prop:location ?v2}
   OPTIONAL{?v0 dbp-prop:locationCountry ?v3}OPTIONAL{?
   v0 dbp-prop:locationCity ?v9; dbp-prop:manufacturer ?
   v0}OPTIONAL{?v0 dbp-prop:products ?v11; dbp-prop:
   model ?v0}OPTIONAL{?v0 georss:point ?v10}OPTIONAL{?v0
    rdf:type ?v7}}
8  SELECT ?v2 ?v4 WHERE { { ?v2 rdf:type %%v1%%. ?v2 dbp
   -prop:population ?v4. FILTER (xsd:integer(?v4) > %%v0
   %%) } UNION { ?v2 rdf:type %%v1%%. ?v2 dbp-prop:
   populationUrban ?v4. FILTER (xsd:integer(?v4) > %%v0
   %%) } }
9  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.} }
10 SELECT DISTINCT ?v0 { ?v3 foaf:page ?v0. ?v3 rdf:type
    dbp-owl:SoccerPlayer . ?v3 dbp-prop:position ?v6 . ?
   v3 dbp-prop:clubs ?v8. ?v8 dbp-owl:capacity ?v1 . ?v3
    dbp-owl:birthPlace ?v5 . ?v5 ?v4 ?v2. OPTIONAL {?v3
   dbp-owl:number ?v9.} Filter (?v4 = dbp-prop:
   populationEstimate || ?v4 = dbp-prop:populationCensus
    || ?v4 = dbp-prop:statPop ) Filter (xsd:integer(?v2)
    > %%v1%% ) . Filter (xsd:integer(?v9) < %%v0%%  ) .
   FILTER (?v6 = 'Goalkeeper'@en || ?v6 = dbpedia:
   Goalkeeper_%28association_football%29 || ?v6 =
   dbpedia:Goalkeeper_%28football%29) }
11 SELECT distinct ?v3 ?v4 ?v2 WHERE { {%%v%% dbp-prop:
   subsid ?v3 OPTIONAL{?v2 %%v%% dbp-prop:parent}
   OPTIONAL{%%v%% dbp-prop:divisions ?v4}} UNION {?v2 %%
   v%% dbp-prop:parent OPTIONAL{%%v%% dbp-prop:subsid ?
   v3} OPTIONAL{%%v%% dbp-prop:divisions ?v4}} UNION {%%
   v%% dbp-prop:divisions ?v4 OPTIONAL{%%v%% dbp-prop:
   subsid ?v3} OPTIONAL{?v2 %%v%% dbp-prop:parent}} }
12 SELECT DISTINCT ?v5 WHERE { ?v2 rdf:type dbp-owl:
   Person . ?v2 dbp-owl:nationality ?v4 . ?v4 rdfs:label
    ?v5 . ?v2 rdfs:label %%v%% . FILTER (lang(?v5) = 'en
   ') }
13 SELECT DISTINCT ?v2 ?v3 WHERE { ?v2 rdf:type %%v%% ;
   rdfs:label ?v3 . FILTER regex(?v3, 'pes', 'i') }
14 SELECT * WHERE {{ %%v%% rdfs:comment ?v0. FILTER (
   lang(?v0) = 'en')} UNION {%%v%% foaf:depiction ?v1}
   UNION {%%v%% foaf:homepage ?v2}}
15 SELECT ?v6 ?v8 ?v10 ?v4 WHERE { ?v4 skos:subject %%v
   %% . ?v4 foaf:name ?v6 . OPTIONAL { ?v4 rdfs:comment
   ?v8 . FILTER (LANG(?v8) = 'en') . } OPTIONAL { ?v4
   rdfs:comment ?v10 . FILTER (LANG(?v10) = 'de') . } }
16 SELECT DISTINCT ?v7 ?v4 ?v6 ?v5 WHERE { %%v%% ?v4 ?v5
    . OPTIONAL {?v5 rdfs:label ?v6} . FILTER(langMatches
   (lang(?v6),'EN')||(! langMatches(lang(?v6),'*'))) .
   FILTER(langMatches(lang(?v5),'EN')||(! langMatches(
   lang(?v5),'*'))) . OPTIONAL {?v4 rdfs:label ?v7}}
17 SELECT DISTINCT ?v2 ?v3 { {?v2 skos:subject %%v%%.}
   UNION {?v2 skos:subject dbp-cat:Prefectures_in_France
    .} UNION {?v2 skos:subject dbp-cat:
   German_state_capitals .} ?v2 rdfs:label ?v3. FILTER (
   lang(?v3)='fr') }
18 SELECT ?v3 ?v4 ?v5 WHERE { { %%v%% ?v3 ?v4. FILTER (
   (STR(?v3) = 'http://www.w3.org/2000/01/rdf-schema#
   label' && lang(?v4) = 'en') || (STR(?v3) = 'http://
   dbpedia.org/ontology/abstract' && lang(?v4) = 'en')
   || (STR(?v3) = 'http://www.w3.org/2000/01/rdf-schema#
   comment' && lang(?v4) = 'en') || (STR(?v3) != 'http
   ://dbpedia.org/ontology/abstract' && STR(?v3) != '
   http://www.w3.org/2000/01/rdf-schema#comment' && STR
   (?v3) != 'http://www.w3.org/2000/01/rdf-schema#label'
   ) ) } UNION { ?v5 ?v3 %%v%% FILTER ( STR(?v3) = 'http
   ://dbpedia.org/ontology/owner' || STR(?v3) = 'http://
   dbpedia.org/property/redirect' ) } }
19 SELECT ?v1 WHERE { { ?v1 rdfs:label %%v%% } UNION { ?
   v1 rdfs:label %%v%% }. FILTER(regex(str(?v1),'http://
   dbpedia.org/resource/') || regex(str(?v1),'http://
   dbpedia.org/ontology/') || regex(str(?v1),'http://www
   .w3.org/2002/07/owl') || regex(str(?v1),'http://www.
   w3.org/2001/XMLSchema') || regex(str(?v1),'http://www
   .w3.org/2000/01/rdf-schema') || regex(str(?v1),'http
   ://www.w3.org/1999/02/22-rdf-syntax-ns')) }
20 SELECT * WHERE { ?v6 a dbp-owl:PopulatedPlace; dbp-
   owl:abstract ?v1; rdfs:label ?v2; geo:lat ?v3; geo:
   long ?v4. {?v6 rdfs:label %%v%%.} UNION { ?v5 dbp-
   prop:redirect ?v6; rdfs:label %%v%%. } OPTIONAL { ?v6
    foaf:depiction ?v8 } OPTIONAL { ?v6 foaf:homepage ?
   v10 } OPTIONAL { ?v6 dbp-owl:populationTotal ?v12 }
   OPTIONAL { ?v6 dbp-owl:thumbnail ?v14 } FILTER (
   langMatches( lang(?v1), 'de') && langMatches( lang(?
   v2), 'de') )}
21 SELECT * WHERE { %%v%%  dbp-prop:redirect ?v0 . }
22 SELECT ?v2 WHERE { ?v3 foaf:homepage ?v2 . ?v3 rdf:
   type %%v%% . }
23 SELECT ?v4 WHERE { ?v2 rdf:type dbp-owl:Person . ?v2
   rdfs:label %%v%% . ?v2 foaf:page ?v4 . }
24 SELECT * where { ?v1 a dbp-owl:Organisation . ?v2 dbp
   -owl:foundationPlace %%v0%% . ?v4 dbp-owl:developer ?
   v2 . ?v4 a %%v1%% . }
25 SELECT ?v0 ?v1 ?v2 ?v3 where { ?v6 rdf:type %%v%%. ?
   v6 dbp-prop:name ?v0. ?v6 dbp-prop:pages ?v1. ?v6 dbp
   -prop:isbn ?v2. ?v6 dbp-prop:author ?v3.}
```

Listing 1: The 25 DBPSB benchmark queries.

C. DETAILED BENCHMARKING RESULTS

The following table contains the detailed benchmarking results for Virtuoso, Sesame, Jena-TDB and BigOWLIM including Queries per second (QpS), geometric mean of query runtime in milliseconds (GM), and standard deviation of query runtime in milliseconds (SD) for all dataset sizes. The second column lists the variability of a query, i.e. the number of variations as explained in Section 4. The last column is the ratio of highest and lowest QpS value in a row. Higher values indicate a high difference in performance between stores.

Table 4: Queries per second (QpS), geometric mean of query runtime in milliseconds (GM), and standard deviation of query runtime in milliseconds (SD), for the 10% dataset, 50% dataset, 100% dataset and 200% dataset (from top to bottom). X indicates that the query always timed out.

10%		Virtuoso			Sesame			Jena-TDB			BigOWLIM			Min-max
Query	Variability	QpS	GM	SD	QpS	GM	SD	QpS	GM	SD	QpS	GM	SD	ratio
1	335382	199.1	4	8.3	166.3	2.3	64	126.6	3.9	17.5	60.1	16.1	5.8	3.1
2	58	434.8	2.2	1	426.9	2.3	0.1	205.8	4.2	3.8	63.4	15.5	5.2	6.9
3	138450	79.8	12.3	2.9	266.6	3.1	3.4	0.1	1734	35579	47.8	19.1	13.2	2518.0
4	>1M	95.3	8.2	11.5	0.2	4541	1181	39.2	19	21.5	7.2	86	133.6	542.7
5	2676	66.4	14.9	2	217.6	3.7	6.5	72.8	1.7	9.9	36.1	23.8	20.6	6.0
6	242	55.6	17.2	5.6	36.8	20.7	84.8	51.1	15.3	14.7	17.6	54	30.1	3.2
7	149026	22.9	37.5	53.5	149.9	5.4	5.8	0.2	2126.7	4612	18.2	44.1	50.8	737.5
8	15202	15.5	41	40.6	13.3	42.4	110.6	80	9.3	11	5.9	109.4	119	13.5
9	950	22.6	44	4.5	154.6	5.1	5.9	0.3	2463	2333	44.6	21.1	12.3	487.3
10	391	8	124.2	5.5	1.3	735.8	353.7	0	11013	47438	2.8	356.2	13.1	180.5
11	221	156.1	5.8	6.6	215.7	3.8	4.8	84.9	9.5	8.2	46.1	20.3	10.6	4.7
12	36242	120.6	8.1	2.3	190.2	3.8	6.1	0	10321	40525	52	17.5	15.5	7312.5
13	682	126.9	7.8	2.4	277.6	3	3.3	75.8	6.3	34.9	36.8	23.3	28.5	7.5
14	148365	33.6	13.5	250.1	53.3	8.6	49.8	56	13.5	14.1	8.2	59.4	270.4	6.8
15	32983	23.9	11.1	236.1	8.4	13.8	957.2	8.1	22.8	329.3	5.9	35.3	409.8	4.1
16	335438	115	8.4	3.2	213.3	4.2	3.2	14.7	50.2	50.8	28.2	28.2	53	14.5
17	96706	74	10.4	64.3	58.8	7.4	56.5	81	10.2	7.8	25.2	27.7	68.7	3.2
18	338801	172.1	5.6	3.3	0.5	1925	310.1	154.4	6	3.3	14	54.7	56.6	335.6
19	882407	174.2	5.2	4.2	256.3	3.5	2.7	142.4	5.7	5.4	61.5	16.2	2.2	4.2
20	34102	18.3	53.9	7.4	170.7	4.9	5.5	0	48953	50410	46.9	20.2	11.8	12510.7
21	58	477.9	2.1	0.2	458.6	2.2	0.1	10.3	4.4	469	66	15.1	0.7	46.6
22	1593	143.1	5.3	17.9	34	7.1	120.9	3	118.9	426	14.7	22.4	571.8	47.6
23	1014	132	7.3	3.2	241.8	3.2	4.6	8.8	79.2	84.7	62.6	15.7	4.9	27.5
24	310	0.3	1270.3	5108	1.3	273.7	1320	116.8	6.8	6.8	0.2	4140	8459	755.5
25	613	52.7	16.9	18.3	137.1	4.8	9.7	20	14.7	163.8	40.5	20.1	59.1	6.9

50%		Virtuoso			Sesame			Jena-TDB			BigOWLIM			Min-max
Query	Variability	QpS	GM	SD	QpS	GM	SD	QpS	GM	SD	QpS	GM	SD	ratio
1	>1M	183.3	4.1	11.4	96	8.6	6	61.3	15.8	4.9	53.9	18	5.6	3.4
2	408286	21.9	45	7.2	76.5	9.1	23.7	60.2	14.6	7.1	23.9	39.1	14.7	3.5
3	329367	48.1	19.3	8.8	39	19	18	0	19740	42661	16.7	42.9	81.6	1939.2
4	>1M	10	56.7	90	0	25433	30115	35.9	26	10.4	2.4	332.8	313.1	1057.5
5	7459	52.6	17.2	17.4	14.2	40.3	125.2	21.2	32.7	54	13.2	59	58.7	4.0
6	976	10.9	81.3	43.8	0.4	198.5	16554	108.3	7.1	9.7	3.2	232.5	368	284.8
7	>1M	20.1	45.8	29.8	14.9	23.6	243.6	0	57343	82095	7.7	108.3	102.6	2721.9
8	76181	1.6	198.8	467	0.1	368.7	34504	54	14.7	11.7	0.7	587.9	1530	401.7
9	15423	19.3	50.3	15.2	6.1	34.3	885.1	0.4	1226	3024	17.5	44.7	47.9	52.8
10	3178	1	972	72.5	0.1	4685	33167	0	1024	83235	0.3	3608	378.6	46.3
11	1786	39.7	13.5	46.4	31.6	27.5	18.2	35.6	27.5	6.2	11.1	41.7	375.4	3.6
12	73033	51.8	17.4	11	30.4	23.6	49.3	0	52958	56392	23.1	34.9	39.1	3549.1
13	1655	79	10.9	7.3	79.1	11.5	5.8	3.1	22.3	860	15.7	55.8	49.6	25.3
14	827814	18.1	26.3	137.5	4.9	81.6	409.4	39.1	24.4	8.5	0.8	304.9	3000	47.5
15	147464	6.4	30	686.8	0.6	87.3	8404	26	30.5	28.7	0.2	448.5	23850	150.8
16	>1M	99.1	9.7	3.6	47	15.4	21.6	55.8	15.9	8.6	34.9	27.3	9	2.8
17	156887	27.1	32.7	27.3	5.4	75.3	468.8	29.1	32.8	12.1	2.2	166.3	983.6	13.2
18	>1M	168.2	5.3	4.1	0.1	8329	24448	56.2	17.6	3	23.6	37.9	23.9	1974.9
19	>1M	80.1	10.7	8.9	36.1	25.2	11.1	24	39.5	14.2	26.7	33.4	17.8	3.3
20	237618	14.2	68.7	16.1	10.2	34.6	437	X	180000	0	11.9	65.6	57.2	2551.1
21	408286	90.6	7.4	14.5	67.5	11.9	13.9	40.9	21.5	16.8	24	38.9	14.2	3.8
22	2674	20.1	11.1	375.3	71.4	24778	41.4	1427	1691.6	163.6	2896	6.1	15.9	78.5
23	1057	64.7	13.4	14.5	21.3	34.6	97.4	2592	5407	2.4	35.9	19.1	18.2	376.9
24	3190	0.1	6318	8330	1486	9360	0.4	22.9	20	39.3	24475	50796	0.1	1521.6
25	983	28.8	30	34.1	37.9	930.7	57.4	30.9	32.7	16	115	898	7.6	7.4

100%		Virtuoso			Sesame			Jena-TDB			BigOWLIM			Min-max
Query	Variability	QpS	GM	SD	QpS	GM	SD	QpS	GM	SD	QpS	GM	SD	ratio
1	>1M	203.8	4	11	95.2	9	4	54.0	18	2	57.5	17	2	3.8
2	>1M	3.6	276	8	59.5	14.3	10	55.2	16	7	25.1	34	37	16.5
3	357364	31.9	26	28	26	34	18	0	95987	82722	21.8	43	15	4427.3
4	>1M	8.5	69	94	0	42931	40507	0.1	3501	9083	9.9	53	132	515.8
5	9940	30	24	50	5.7	131	141	10.4	86	55	7	89	186	5.3
6	1609	8	120	34	0.8	360	3556	0.3	103	7182	3.1	242	346	28.9
7	>1M	15.9	54	36	10.9	72	61	0	34119	19106	4.7	163	216	623.0
8	110027	1	210	915	0.6	364	2716	0.1	743	24127	0.3	976	4074	16.0
9	22639	16.6	58	16	15.6	51	54	X	180000	0	14.1	67	22	2979.2
10	5802	0.9	1057	1349	0.1	15994	11274	X	180000	0	0.2	6249	582	159.2
11	1881	49.7	10	46	33.6	25	16	0.1	4108	9050	30	30	15	579.2
12	91291	37.2	24	12	17	46	53	X	180000	0	21.2	43	21	6700.7
13	3321	47.2	15	18	19.7	34	43	0.1	7870	8524	29.3	29	18	510.4
14	>1M	1	166	2286	0.3	971	4291	20.3	43	26	1.6	117	1058	60.1
15	262307	5.6	32	965	0.5	781	3582	0	40680	78455	0.4	393	1033	404.1
16	>1M	136.2	7	0.5	82.2	11	5	0.2	162	11266	45.4	21	7	772.7
17	253577	28.8	32	19	0.8	367	2420	18.7	49	27	3.6	81	638	37.1
18	>1M	199.9	4	2	0	20247	43187	0.2	108	12640	45.1	21	5	6361.9
19	>1M	69.6	12	7.9	31.9	27	16	35.2	24	19	33.4	29	7	2.2
20	>1M	13	74	24	5.8	147	106	X	180000	0	12.3	74	34	2333.0
21	>1M	63.9	12	13	57.8	14	10	0.1	12234	21420	27.9	33	17	1198.4
22	3190	24.4	10	306	0.8	497	1845	0	1810	36356	1.3	218	1615	696.9
23	1624	59.9	14	8	34.3	26	13	0	1299	25956	27.4	35	8	1322.1
24	3191	0.1	7015	8819	0.1	2445	15490	0.1	273	25217	0	36091	60802	8.4
25	1624	16.1	36	250	5.2	96	294	0.1	804	18749	7.7	90	137	211.7

200%		Virtuoso			Sesame			Jena-TDB			BigOWLIM			Min-max
Query	Variability	QpS	GM	SD	QpS	GM	SD	QpS	GM	SD	QpS	GM	SD	ratio
1	>1M	108.1	229.9	44.4	72.7	84.4	39.2	15.6	36.6	423.5	11.9	47.7	13.4	9.1
2	>1M	1.8	1.8	0.1	35.2	39.9	27.4	15.9	16.6	6.4	20.9	24.5	11.2	19.1
3	357364	25.5	36.6	21.1	15.4	17.7	9.1	0	0	0	15.6	16.7	8.2	2194.3
4	>1M	7.9	12.1	46.1	0	0	0	26	34.4	33.6	3	4.1	9	2618.7
5	14197	36.5	47.4	19.8	3.6	4.7	2.7	1	3	4.3	6.5	8.7	10.4	36.9
6	1609	6.2	7.6	6.1	0.7	1.4	2	7.3	15.1	47	2.1	2.6	1.3	10.5
7	>1M	8.4	11.7	9.3	5.9	6.6	2.4	X	0	0	5	7	5.3	1516.7
8	110027	0.4	2.8	42.9	0.1	0.4	6.7	8.6	13.9	19.9	0.3	1	15.7	104.7
9	22639	13.9	15.1	6	9.3	9.5	3.2	0	0	0	12.2	13.6	5.1	2013.3
10	5802	0.5	0.5	0	0	0	0	0	0	0	0.2	0.2	0	92.0
11	3762	19	73	57.5	27	30.5	17.2	15.3	15.6	2.7	16.6	19.2	7.9	1.8
12	91291	30.8	36.1	31.8	12.8	15.7	8.1	0	0	0	19.6	20.4	5.4	4816.3
13	3321	34.9	48.6	51.8	15.9	18.5	13.2	9.3	11.1	9.5	15.9	18.2	10.3	3.8
14	>1M	3	11.4	29.5	0.1	0.4	2.8	5.6	6.5	2.9	1.1	3.4	6.5	61.1
15	524614	3.2	36.9	51.5	0.4	1.4	4.5	0.7	1.2	1.6	0.1	0.8	8.8	24.2
16	>1M	188.3	189.8	24.1	66.6	74.9	40.3	10.2	10.6	3.1	42.6	44	10.6	18.5
17	507154	10.5	19.4	12.2	0.3	0.8	1.8	6.6	6.8	1.7	4.3	6.8	6.7	35.8
18	>1M	198.3	207.3	35.8	0	0	0	14.7	17.5	7.9	42.2	43.1	8.3	12761.2
19	>1M	58	69	62.5	14.8	15.3	4.3	17.6	19.6	13	31.4	32.7	8.9	3.9
20	>1M	12.3	12.9	4.4	3.4	4.5	17.5	0	0	0	9.7	11.5	8.2	1964.6
21	>1M	61.9	73.7	77	46.4	50	20.8	14.6	15.2	4.9	25	25.7	6.3	4.2
22	60388	4	74.3	80.5	0.1	2.1	66.2	0.2	0.4	0.6	2.6	5.9	11.5	48.0
23	6380	53	60.1	33.3	23.8	26.9	16.7	4.9	6.9	5.9	23.1	23.6	5.5	10.8
24	3191	0.1	0.1	0	0.1	0.2	0.2	4.3	6.4	4.8	0	0	0.1	324.0
25	1624	13.5	19	15.2	3.5	5.2	7.8	8.8	9	1.6	6.1	8.6	9.5	3.9