How to generate query parameters in RDF benchmarks?

Andrey Gubichev [#1], Renzo Angles [†2], Peter Boncz [*3]

[#] *TU Munich, Germany*
[1] gubichev@in.tum.de

[†] *Universidad de Talca, Chile*
VU University Amsterdam, Netherlands
[2] *rangles@utalca.cl*

[*] *CWI, Netherlands*
[3] p.boncz@cwi.nl

***Abstract*—In this paper we consider the problem of generating parameters for queries in RDF benchmarks. We show that uniform random sampling of the substitution parameters is not well suited for RDF benchmarks, since it results in unpredictable runtime behavior of queries. We formulate a formal problem of parameter generation to ensure stable and statistically significant benchmark results.**

I. Introduction

A typical benchmark consists of two parts: (i) the data generator that creates syntactic dataset of the desired scale, and (ii) the workload generator that issues queries against the generated data based on the pre-defined *query templates*. A query template is an expression in the query language (e.g., SPARQL) with *substitution parameters* that have to be replaced with real bindings by the workload generator, for example:

```
select * where {
       ?person sn:firstName %name .
       ?person sn:livesIn %country.
}
```

Here, the workload generator would produce a number of different pairs of bindings (say, 100) for the `%name` and `%country` parameters, then execute the resulting queries and report an aggregate value of the observed runtime distribution per query (usually, the average runtime per query template). This aggregated score serves two audiences: First, the users can evaluate how fit a specific system is for their use-case (choosing, for example, between systems that are good in complex analytical processing and those that have the highest throughput for lookup queries). Second, the database architects can use the score to analyze their systems' handling of certain technical challenges, like handling multiple interesting orders or sparse foreign key joins [5].

In order for the aggregate runtime to be a useful measurement of the system's performance, the selection of parameters should guarantee the following properties of the queries:

- P1: the query runtime has a bounded variance: the average runtime should correspond to the behavior of the majority of the queries
- P2: the runtime distribution is stable: a different sample of 100 parameter bindings should result in an identical runtime distribution
- P3: the query plan for all the parameters is the same: this ensures that a specific query tests the system's behavior under the well-chosen technical difficulty (e.g., handling voluminous joins or proper cardinality estimation for subqueries etc.)

The standard way to get the parameter bindings for `%name` and `%country` is to sample the values (uniformly, at random) from all the possible names in the dataset. This is, for example, how the classical TPC-H benchmark creates its workload; since the TPC-H data is generated with simple uniform distribution of values, the uniform sample of parameters guarantees the properties **P1-P3**.

However, this technique does not work for benchmarks that use real-world datasets (YAGO or DBpedia) or generate datasets with real-world distribution and correlations (LDBC benchmark, which is based on S3G2 generator [7]). In our example above, assuming that names and countries are correlated, the behavior of the query changes significantly depending on the selection of its parameters: if the `%name` is *Li*, and the `%country` is *China*, the query is an unselective join that tests the power of indexes. However, if we select *John* and *China* as parameters, the query becomes a very selective join that favors the ability of skipping unrelevant parts of the index.

We note that the parameter generation problem is a general problem in that it does not depend on the data model. For concreteness though, in this paper we concentrate on RDF benchmarks.

In this paper we make two contributions: first, we show that the straightforward approach of generating parameter bindings uniformly at random for benchmarking RDF systems fails to deliver predictable and stable results; second, we define the problem of generating parameter bindings that would overcome these hurdles.

II. EXAMPLES

In this section we illustrate the statement that uniform selection of parameters leads to unpredictable behavior of the queries, thus making interpretation of the results very difficult.

We use two RDF benchmarks, Berlin SPARQL benchmark (BSBM) [1] and LDBC Social Network benchmark [2]. For BSBM, we consider queries from the Business Intelligence (BSBM-BI) use case. For both BSBM and LDBC, the corresponding data generators were used to create datasets with ca. 100 Million triples each. We employ Virtuoso 7 (Column Store) as a SPARQL query engine, and run our experiments on a commodity server with the following specifications: Dual Intel X5570 Quad-Core-CPU, 64 Gb RAM, 1 Tb SAS-HD, Redhat Enterprise Linux with 2.5.37 kernel.

E1: Runtime distribution has high variance When drawing parameters uniformly at random, we encounter a very skewed runtime distribution even for queries over uniformly distributed syntactic data. In BSBM-BI benchmark, for example, the runtime of the Query 4 (*finds the feature with the highest ratio between price with that feature and price without that feature*) has the variance of $674 \cdot 10^6$. This is caused by the fact that the parameter of the query is the product type: depending on how generic the type is (how high it is in the type hierarchy), the amount of data touched by the query differs greatly.

This issue becomes even more important for the LDBC benchmark, where the data generator seeks to mimic some of the properties of the real-world data: the generated data has correlations and skewed data distributions. In this case, naturally, the randomly generated parameter bindings result in a very skewed runtime distribution.

Moreover, the distribution of runtime is far from normal: the Kolmogorov-Smirnov test that measures the distance between the runtime distribution of BSBM-BI Query 2 (*finds the top 10 products most similar to a specific product*) and the normal distribution, results in the distance of 0.89 (with p-value of 10^{-21}). This value indicates that the observed runtime distribution is extremely non-uniform.

E2: Sampling is not stable A single query in the benchmark is typically being executed several times with different randomly chosen parameter bindings. It is therefore interesting to see how the reported average time changes when we draw a different sample of parameters. In order to study this, we take Query 2 of the LDBC benchmark that *finds the newest 20 posts of the user's friends*. We sample 4 independent groups of parameter bindings (100 bindings in each group), run the query with these parameters and report the aggregated runtime numbers within individual groups (q_{10} and q_{90} are the 10th and the 90th percentiles, respectively).

Time	Group 1	Group 2	Group 3	Group 4
q_{10}	0.14 s	0.07 s	0.08 s	0.09 s
Median	1.33 s	0.75 s	0.78 s	1.04 s
q_{90}	4.18 s	3.41 s	3.63 s	3.07 s
Average	1.80 s	1.33 s	1.53 s	1.30 s

We see that uniform at random generation of query parameters in fact produces unstable results: if we were to run 4 workloads of the same query with 100 different parameters in each workload, the deviation in reported average runtime would be up to 40%, with even stronger deviation on the level of percentiles and median runtime (up to 100%).

We observe similar behaviour of the BSBM-BI Query 2: when it is executed with different groups of 100 random parameter binding each, the mean runtime difference is up to 15% between the groups, and the median value can vary up to 25%.

E3: Average runtime is not representative In addition to being far from uniform (**E1**), the query runtime distribution can also be "clustered": depending on the parameter binding, the query runs either extremely fast or surprisingly slow, and the average across the runtimes does not correspond to any actual query performance. We consider Query 4 of the BSBM BI workload that *finds the feature with the highest ratio between price with that feature and price without that feature*, depending on the input parameter *ProductType*. The product types in BSBM form a hierarchy: the higher the type is, the more general it is and therefore the more products with this type exist. A short summary of statistical properties of the runtime distribution is given in the table below (where q_{95} is the 95th percentile):

Min	Median	Mean	q_{95}	Max
59 ms	354 ms	3.6 s	17.6 s	259 s

Depending on the parameter selection, the query finishes in either 300 ms to 400 ms, or in more than 17 seconds, with almost no query in between those two groups. This way, the arithmetic mean is over 10 times larger than the median. Moreover, there is no actual query with the runtime close to the mean: all of them are either much faster, or significantly slower.

E4: Different plans for different parameters Finally, the uniformly generated parameter bindings can lead to completely different plans for the same query template. It happens because the cardinalities of the subqueries naturally depend on the parameter bindings, and sometimes on the combination of the parameters. For instance, the optimal plan for the LDBC Query 3 (*finds the friends within two steps that have been to countries X and Y*) can start either with finding all the friends within two steps from the given person, or from all the people that have been to countries X and Y: if X and Y are Finland and Zimbabwe, there are supposedly very few people that have been to both, but if X and Y are USA and Canada, this intersection is very large.

We note that the plan variability is not a bad property *per se*: indeed, this query forces the query optimizer to accurately estimate the cardinalities of subqueries depending on input parameters. However, the generated parameters should be sampled independently from two different classes (countries that are rarely and frequently visited together), to allow a fair and complete comparison of different query optimization strategies.

III. Generating Parameter Bindings

Here we define the formal problem of generating the parameter binding for RDF benchmark. In order to compare two query plans (e.g., optimal plans for the same query with different parameter bindings), we use the classical cost function that takes into account the sum of intermediate results produced during the plan's execution [8]. It is formally defined as follows:

$$C_{\text{out}}(T) = \begin{cases} 0 & \text{if } T \text{ is a scan} \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

where $T_1 \bowtie T_2$ is a self-join in the triple store, or a join between two clustered tables in clustered-property storage; the cost function is oblivious to the underlying storage model.

In our experiments, the cost function C_{out} of the query strongly correlates with its running time (ca.85% Pearson correlation coefficient); therefore, if two queries have the same optimal logical plans (with regards to C_{out}), they are expected to have very similar running time.

We consider query Q (with parameters $p_1, \ldots, p_n$) against the RDF dataset D. Every parameter p_i has the domain P_i, and the domain of all the parameters is $P = P_1 \times \ldots \times P_n$. Now, the formal problem of finding the parameter binding is formulated as follows:

PARAMETERS FOR RDF BENCHMARKS: Split P into subsets $S_1, \ldots, S_k$ such that for every S_i holds:

- *a*: $\forall(p_1, \ldots, p_n) \in S_i$ the query Q has the same optimal query plan w.r.t.C_{out}
- *b*: $\forall(p_1, \ldots, p_n) \in S_i$ the cost C_{out} of the optimal plan for Q is the same
- *c*: query plan for parameters from $S_k, k \neq i$, is different from the query plan for S_i

Intuitively speaking, we want to cluster the parameters domain into disjoint classes, such that for every class the query Q has the same optimal plan with identical cost for all the bindings (conditions *a* and *b*), and these optimal plans are different across different parameter classes (condition *c*).

Since the cost function correlates with running time, queries with identical optimal plans w.r.t.C_{out} are expected to have close runtime, so the properties P1-P3 hold within each set of parameters S_i. Then, the workload generator can produce separate parameter bindings by sampling them from every parameter class independently, thus effectively splitting the query into several cases. For example, BSBM-BI Query 4 would turn into two queries, Q4a (where type parameter denote a very specific product's type) and Q4b (with parameter being a generic type of many products). Alternatively, the benchmark authors can decide to tune the workload generator such that it does not generate parameters from the certain class S_j (if, for example, the total number of distinct classes is too high). Reporting aggregated runtime only within these automatically identified parameter classes will make the results more comprehensible for both users and database architects.

Note that discovery of these sets of parameters is far from trivial. In particular, even if we are given the set of parameters S_j from the candidate solution, checking that it satisfies the condition *a* would require finding the optimal join order for the query Q for all binding from S_j, i.e. it boils down to solving multiple $\mathcal{NP}$-hard join ordering problems. We can, therefore, only aim at a heuristic for it, which is the subject of our future work.

IV. Related work

A number of RDF benchmarks has been proposed in recent years [3], [4], [6], [9], [11]; the problem of finding the parameter domains is relevant for all of them.

One step beyond the simple uniform random sample was done for the TPC-DS benchmark, where parameters can be drawn from the given "step-shaped" distribution [10], [12]. Our work is aimed at generalizing this line of research for complex distributions (in both real-world and generated data) and for correlated data.

V. Conclusions

We introduce a general problem of generating the parameters for RDF benchmarks. We demonstrated that conventional uniform sampling is not providing stable and comprehensive results. We consider the formal problem of clustering the parameters domain into classes that yield the same query plan with regards to the sum of intermediate results; the solution of this clustering problem will ensure comparable and stable results of benchmark queries.

Acknowledgment

This work is supported by the EU project LDBC (within the FP7 framework).

References

[1] Berlin SPARQL Benchmark. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark.

[2] LDBC Benchmark. http://ldbc.eu:8090/display/TUC/Interactive+Workload.

[3] LUBM. http://swat.cse.lehigh.edu/projects/lubm.

[4] C. Bizer and A. Schultz. The Berlin Sparql Benchmark. *International Journal On Semantic Web and Information Systems*, 2009.

[5] P. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*, 2013.

[6] G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudré-Mauroux. BowlognaBench – Benchmarking RDF Analytics. In K. Aberer, E. Damiani, and T. Dillon, editors, *Data-Driven Process Discovery and Analysis*, volume 116 of *Lecture Notes in Business Information Processing*, pages 82–102. Springer Berlin Heidelberg, 2012.

[7] P. Minh Duc, P. A. Boncz, and O. Erling. S3g2: A Scalable Structure-Correlated Social Graph Generator. In *Proceedings of TPC Technology Conference on Performance Evaluation & Benchmarking 2012*, 2012.

[8] G. Moerkotte. Building Query Compilers. http://pi3.informatik.uni-mannheim.de/ moer/querycompiler.pdf.

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

[10] M. Poess and J. M. Stephens, Jr. Generating thousand benchmark queries in seconds. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 1045–1053. VLDB Endowment, 2004.

[11] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. In *ICDE*, pages 222–233, 2009.

[12] J. M. Stephens and M. Poess. Mudd: a multi-dimensional data generator. *SIGSOFT Softw. Eng. Notes*, 29(1):104–109, Jan. 2004.