Towards a Top-K SPARQL Query Benchmark Generator

Shima Zahmatkesh, Emanuele Della Valle, Daniele Dell'Aglio, and Alessandro Bozzon

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico of Milano
P.za L. Da Vinci, 32. I-20133 Milano - Italy
shima.zahmatkesh@polimi.it, emanuele.dellavalle@polimi.it,
daniele.dellaglio@polimi.it, a.bozzon@tudelft.nl

Abstract. The research on optimization of top-k SPARQL query would largely benefit from the establishment of a benchmark that allows comparing different approaches. For such a benchmark to be meaningful, at least two requirements should hold: 1) the benchmark should resemble reality as much as possible, and 2) it should stress the features of the top-k SPARQL queries both from a syntactic and performance perspective. In this paper we propose Top-k DBPSB: an extension of the DBpedia SPARQL benchmark (DBPSB), a benchmark known to resemble reality, with the capabilities required to compare SPARQL engines on top-k queries.

Keywords: Top-k Query, SPARQL Benchmank, SPARQL engines

1 Introduction

Top-k queries – queries returning the top k results ordered according to a user-defined scoring function – are gaining attention in the Database [7] and Semantic Web communities [8, 13, 4, 3]. Order is an important property that can be exploited to speed up query processing, but state-of-the-art SPARQL engines such as Virtuoso [5], Sesame [2], and Jena-TDB [10], do no exploit order for query optimisation purposes. Top-k SPARQL queries are managed with a *materialize-then-sort* processing schema that computes all the matching solutions (e.g., thousands) even if only a limited number k (e.g., ten) are requested.

Recent works [8, 13, 4, 3] have shown that an efficient *split-and-interleave* processing schema [7] could be adopted to improve the performance of top-k SPARQL queries. To the best of our knowledge, a consistent comparison of those works does not exist. As often occurs, the main cause for this fragmentation resides in the partial lack of a SPARQL benchmark covering top-k SPARQL queries. To foster the work on top-k query processing within the Semantic Web community, we believe that it is the right time to define a top-k SPARQL benchmark.

Following well known principles of benchmarking [6], we can formulate our research question as follows: *is it possible to set up a benchmark for top-k SPARQL*

queries, which resembles reality as much as possible and stresses the features of top-k queries both from a syntactic (i.e., queries should contain rank-related clauses) and performance (i.e., the query mix should insist on characteristics of top-k queries which stress SPARQL engine) perspective?

The remainder of the paper is organised as follows. Section 2 reviews recent works on SPARQL query benchmarks and, in particular, the DBpedia SPARQL Benchmark (DBPSB) [9] that already provides a satisfactory answer the first half of our question. Section 3 defines some relevant concepts. Section 4 reports on our investigations on the second half of the question. The benchmark proposed in this paper, namely Top-k DBpedia SPARQL Benchmark (Top-k DBPSB), uses the same dataset, performance metrics, and test driver of DBPSB, but it extends the *query variability* feature of DBPSB by proposing an algorithm to automatically create top-k queries from the 25 auxiliary queries of the DBPSB and its datasets. Section 5 provides experimental evidence that Top-k DBPSB generates queries able to stress SPARQL Engines. Finally, Section 6 concludes and casts some light on future works.

2 Background

Benchmarking a given **database** is a process of performing well defined tests on that particular database management system for the purpose of evaluating its performance. The response time and the throughput are the two main criteria on which the performance of a database can be measured. According to [6] a benchmark to be useful must be:

- Relevant: It must measure the peak performance and price/performance of systems when performing typical operations within that problem domain.
- Portable: It should be easy to implement the benchmark on many different systems and architectures.
- Scalable: The benchmark should apply to small and large computer systems. It should be possible to scale the benchmark up to larger systems, and to parallel computer systems as computer performance and architecture evolve.
- Simple: The benchmark must be understandable, otherwise it will lack credibility.

In the last decade, the Semantic Web community perceived the need for benchmarks to compare the performance of SPARQL engines.

The Berlin SPARQL Benchmark (BSBM) [1] is one of the earliest benchmarks for SPARQL engines. It is based on an e-commerce use case. The queries contain various SPARQL features. It was often criticised for being too *relational* and not to stress enough RDF and SPARQL charateristics.

SP2Bench [12] is another recent benchmark for SPARQL engines. Its RDF data is based on the DBLP Computer Science Bibliography. Using SP2Bench Generator, SP2Bench generates synthetic data. The benchmark queries vary in common characteristics like selectivity, query and output size, and different types of joins and contain SPARQL features such as FILTER, and OPTIONAL. Also

SP2Bench was criticised to lack the typical heterogeneity SPARQL engines are supposed to have to master.

DBpedia SPARQL Benchmark (DBPSB)[9] was the first benchmark to apply a novel methodology to create heterogeneous datasets of different sizes derived from the DBpedia. The DBPSB methodology follows the four key requirements for domain specific benchmarks [6] reported above. It is relevant, portable, scalable, and understandable.

DBPSB **datasets** (namely 10%, 50%, 100%, and 200%) are generated from DBpedia dataset. The authors use triples duplication to generate the 200% dataset and seed method to generate the 10% and 50% dataset.

DBPSB proposes 25 **template queries** obtained by analysing and clustering three months of DBpedia query logs. In details, the authors removed common prefix from queries, they identified query segments, they built a query similarity graph and applied graph clustering on such a graph. From those query clusters, they selected 25 frequently executed query templates which cover most of the SPARQL features. In order to generate the benchmark queries from the query templates, they defined an *Auxiliary Query* for each query template so to collect a list of values to replace the placeholders in the templates with.

The DBPSB execution consists of the following steps: system restart, warm-up phase, and hot-run phase. The performance of each RDF store was measured by evaluating two measures: query mixes per hour (QMpH), and Queries per Second (QpS).

3 Definitions

First of all, let us recall some basic concepts of SPARQL. We denote IRIs with I, Literals with L, Variables with V, Triple Patters with T, Graph Patterns with P, mappings with μ, and the variables occurring in P with $var(P)$. Readers that need more details are invited to check them out in [11].

In addition we need some definition from SPARQL Rank [8]. Top-k queries in SPARQL 1.1 can be formulated using a *select expression*[1], and a pair of `ORDER BY` and `LIMIT` clauses. In particular, a *scoring function* is a selection expression defined over a set on n *ranking criteria*. A ranking criterion $b(?x_1, \ldots, ?x_m)$ is a function over a set of m variables $?x_i \in var(P)$. A ranking criterion b can be the result of the evaluation of any built-in function of query variables which must have a numerical value or be computable in a finite time. We define as max_b and min_b the application-specific maximal and minimal possible value for the ranking criterion b.

A *scoring function* on P is an expression $\mathcal{F}(b_1, \ldots, b_n)$ defined over the set B of n ranking criteria. The evaluation of a scoring function $\mathcal{F}$ on a mapping μ, indicated by $\mathcal{F}[\mu]$, is the value of the function when all of the $b_i[\mu]$, where $\forall i = 1, ..., n$, are evaluated. It is typical in ranking queries that the scoring function

[1] For more information on projection functions consult http://www.w3.org/TR/sparql11-query/#selectExpressions

$\mathcal{F}$ is assumed to be monotonic, i.e., a $\mathcal{F}$ for which holds $\mathcal{F}(b_1[\mu_1], \ldots, b_n[\mu_1]) \geq \mathcal{F}(b_1[\mu_2], \ldots, b_n[\mu_2])$ when $\forall i : b_i[\mu_1] \geq b_i[\mu_2]$.

For the scope of this work, ranking criterion b consists of a single variable $?x_i \in var(P)$ and the scoring function is a weighted sum of normalized ranking criteria. For this definition, we indicate $w_i \in [0..1]$ a weight and $norm(b_i)$ as a normalization function in the form of $\frac{b_i - min_{b_i}}{max_{b_i} - min_{b_i}}$. So, the scoring function is defined in the following form:

$$\sum_{i=0}^{n} w_i \cdot norm(b_i)$$

Last, but not least, let us introduce few more definitions specific to our process for top-k SPARQL query generation.

Definition 1: *Rankable Data Property* is a RDF property whose range is in xsd:int, xsd:long, xsd:float, xsd:integer, xsd:decimal, or xsd:double (which can be all casted in xsd:double) or xsd:dateTime, and xsd:date (which can be casted in xsd:dateTime), or xsd:time, or xsd:duration.

Definition 2: *Rankable Triple Pattern* is a triple pattern that has a Rankable Data Property in the property position of the pattern.

Definition 3: When a variable, in the object position of a Rankable Triple Pattern, appears in a scoring criteria of the scoring function, we call it *Scoring Variable* and we call *Rankable Variable* the one appearing in the subject position.

For example, the property dbpprop:releaseDate in Query 4 in Figure 1 is a Rankable Data Property because its range is xsd:int and the triple pattern (?var6 dbpprop:releaseDate ?o1) is a rankable triple pattern. Variable ?o1 is a scoring variable and variable ?var6 is a ranking variable.

4 Top-k Query Generation

Now that all basic concept are defined, we can proceed to illustrate how Top-k DBPSB extends DBPSB to generate top-k SPARQL queries from the Auxiliary queries of DBPSB. Figure 1 shows an overview of the process explained with a running example. It takes as input the auxiliary queries and one of the dataset of DBPSB. The process consists of four steps: 1) finding rankable variables, 2) computing maximum and minimum values for each rankable variable, 3) generating scoring functions, and 4) generating top-k queries.

Finding Rankable variables. In order to create SPARQL top-k query templates from DBPSB ones, we look for all variables, in each auxiliary query of each BDPSB query template, which could be used as part of a ranking criterion in a scoring function.

For each DBPSB auxiliary query, we first check if the variables in the query fit the definition of scoring variable. For instance, the set of variables in the DBPSB auxiliary query in Query 1 of Figure 1 is equal to $V_q = \{?var, ?var0, ?var1,$

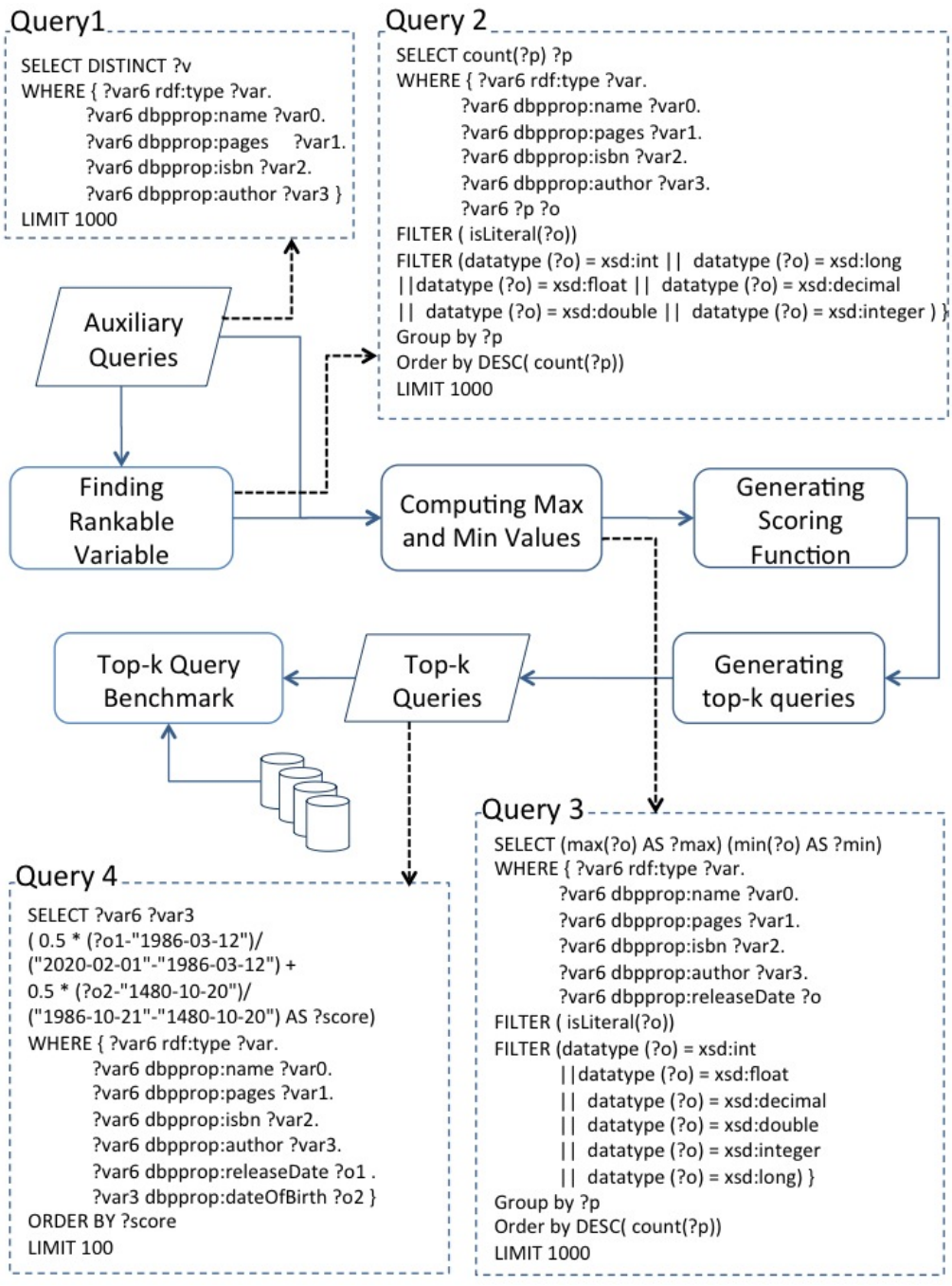

Fig. 1: Conceptual Model

?$var2$, ?$var3$, ?$var6$}. The only *scoring variable* in V_q is ?$var1$ that matches the number of pages[2]

To find additional rankable variable we consider all variables in the query and trying to find rankable triple pattern related to them. For example the Query 2 of Figure 1 shows the extension of Auxiliary Query 1 for variable ?$var6$. This query checks if ?$var6$ is a rankable variable by looking for rankable predicates using the triple pattern (?$var6$, ?p, ?o), the `FILTER` clauses under it and computing the number of results for each predicate so to identify a list of rankable predicates ordered by the number of results they match.

Computation of Max and Min Values. As we defined the scoring function as a weighted sum of normalized ranking criteria, to generate the scoring function, for each rankable variable, we need to compute its maximum and minimum value. So, for each DBPSB auxiliary query and each rankable triple pattern identified in the previous step, we generate a query to find those values. Query 3 of Figure 1 shows an example query which find the maximum and minimum possible value for ?o in the auxiliary query 1 and triple pattern (?$var6$, $dbpprop:releaseDate$, ?o). The maximum value of ?o is "2020-02-01" and the minimum is "1986-03-12".

Top-k Query Generation. At this point, we can create the top-k SPARQL query templates deciding the number of rankable variables[3], the list of weights whose sum must be equal to 1, the number of scoring variables[4], and the value of the `LIMIT` clause.

For instance, for the Query 1 of Figure 1, a valid top-k query can ask for the top 100 pairs of books and authors ordered by (a weighted sum of the normalized) date of birth of the author and (the normalized) date of publication of the book, so to have first the longest books of the most longest-lived authors. Query 4 of Figure 1 shows such a query as generated by the Top-K DBSBM. In order to obtain an executable query, for each scoring variable that appears in scoring function Top-K DBSBM add the related rankable triple pattern to the body of the query. For example it adds the two rankable triple pattern (?$var6$, $dbpprop:releaseDate$?$o1$) and (?$var3$, $dbpprop:dateOfBirth$, ?$o2$) to Query 4. Moreover, Top-K DBSBM also adds the rankable variables ?$var6$ and ?$var3$ in the `SELECT` clause. Last but not least, it puts the `LIMIT` clause randomly choosing between 10 or 100, or 1000 [5].

Formal Description of Top-k Query Generation Algorithm 1 presents the pseudo code of the algorithm informally presented in Figure 1.

---

[2] Strictly speaking, DBpedia dataset contains books which use page numbers written as a string, thus also ?$var1$ is not rankable, but Query 1 is the best suited for presenting out running example.

[3] In this preliminary version of Top-k DBSBM we decided to keep this number between 1 and 3. In future version, we intend to expose this parameter to tune the benchmark

[4] Also the number of scoring variables is kept between 1 and 3, but we intend to expose this parameter in future versions on Top-K DBSBM

[5] The implementation code is available online at *https* : *//bitbucket.org/sh*$_z$*ahmatkesh/top* − *k* − *dbpsb*

Algorithm 1 Top-k query generation pseudo code

```
1: procedure TopKQueryGeneration(Q)
2:     for all q ∈ Q do
3:         R_q ← RN(q)
4:         for all v_i ∈ V_q do
5:             if v_i is rankable variable then
6:                 RT_q ← RT_q ∪ RTP(v_i)
7:             else
8:                 RT_q ← RT_q ∪ RTPS(q, v_i)
9:             end if
10:        end for
11:        for all rt_i ∈ RT_q do
12:            min ← MIN(q, rt_i)
13:            max ← MAX(q, rt_i)
14:            SV_q ← SV_q ∪ {(rt_i, min, max)}
15:        end for
16:        for all st ∈ ST do
17:            W ← SFW(st)
18:            score ← SF(SV_q, st, W_i)
19:            l ← Limit(L)
20:            qt ← TOPKQT(q, score, l)
21:        end for
22:    end for
23: end procedure
```

For each query q in the Set of all the DBPSB query templates (Q) Top-K DBSBM execute the following procedure to generate the top-k queries. Function $RN(q)$, given a query $q \in Q$, returns the number of results of the query and put it in R_q. For each variable v_i in V_q which is the set of all variables of a query $q \in Q$, if v_i is a rankable variable, function $RTP(v_i)$ returns the rankable triple pattern and adds it to the RT_q which is the set of all the rankable triple patterns of a query $q \in Q$ (line 6). If v_i is not a rankable variable, function $RTPS(q, v_i)$, given a query $q \in Q$ and variable v_i, creates a new query and returns one or more rankable triple patterns which have variable v_i in their subject positions. The returned rankable triple patterns are added to RT_q (line 8).

After finding all the rankable triple pattern, in the next step Top-K DBSBM finds the maximum and minimum value matched by ranking variable which is in the object position of rankable triple pattern. Function $MIN(q, rt)$, given a query $q \in Q$ and a rankable triple pattern rt, creates a new query to return maximum value matched by ranking variable which is in the object position of the rt (line 12). Function $MAX(q, rt)$, given a query $q \in Q$ and a rankable triple pattern rt, creates a new query to return minimum value matched by ranking variable which is in the object position of the rt (line 13). Finally we add the rankable triple pattern and its maximum and minimum value to the set SV_q (line 14).

$ST = \{(i\ rankablevariable, j\ rankablesubject) : 1 \leq i, j \leq 3\ and\ i \leq j\}$ is a set of all the possible combination of the rankable variable and scoring variable for defining a scoring function. As we said already, for the scope of this work we decided that the number of scoring variables and rankable variable ranges from 1 to 3. For each possible combination st, the function $SFW(st)$ returns a set of randomly generated Weights (line 17). In the next step function $SF(SV_q, st, W)$, given a set of scoring variables SV_q, a possible combination of the rankable variable and of the scoring variable st and set of weights W, returns the generated scoring function in *score* (line 18). Function $Limit(L)$ selects randomly a limit from set $L = \{1, 10, 100, 1000\}$ (line 19), and finally, function $TOPKQT(q, score, l)$, given a query $q \in Q$, the scoring function formula *score*, and the Limit l, returns the generated top-k SPARQL query (line 20).

5 Preliminary Evaluation

In this section, we provide evidence that the query variability feature of Top-k DBPSB positively answers the second part of the research question we presented in Section 1. In order to do so, we have to show that the queries, which Top-k DBPSB generates, stresses the features that distinguish top-k queries from traditional SPARQL queries.

For this reason, we operationalise our research question, formulating three hypothesis:

H.1 The more the number of the Rankable Variables, the longer the average execution time.
This hypothesis is supported by the intuition that ordering one type of objects for one or more criteria (e.g., a city by its population and year of foundation) appears easier than ordering multiple objects by multiple criteria (e.g., the example in Figure 1).

H.2 The more the number of the Scoring Variables in the scoring function, the longer the average execution time.
This hypothesis is supported by the intuition that ordering one type of objects for one criterion appears easier easier than ordering one object by multiple criteria.

H.3 The value of the `LIMIT` clause has not any significant impact on the average execution time.
This hypothesis is supported by the intuition that SPARQL engines using materialization-then-sort schema cannot benefit from the `LIMIT` clause as those adopting a split-and-interleave one.

As experimental environment, we use an Intel i7 @ 1.8 GHz with 4 GB memory and an hdd disk. The operating system is Mac OS X Lion 10.7.5 and Java 1.6.0_51 is installed on the machine. It is worth to note that the experimental setting is adequate, since we are not interested in the absolute latency, but in the impact on the latency of the number of Rankable Variables (H.1), Scoring Variables (H.2) and the value of the `LIMIT` clause.

We carry out our experiments by using the SPARQL engines Virtuoso, and Jena-TDB. The configuration and the version of each SPARQL engine are as follows:

- Virtuoso Open-Source Edition version 6.1.6: we set the following memory related parameters in file named "'virtouso.ini"'. NumberOfBuffers = 340000, MaxDirtyBuffers = 250000 and set the maximum execution time equals to 400000 sec.
- Jena-TDB Version 2.10.1: We use the default configuration.

In order to perform our evaluation, we loaded DBpedia dataset with the scale factors of 10% on all the mentioned SPARQL engines, and we use the DBpedia SPARQL Benchmark test driver modifying it so to also use Top-k DBPSB top-k queries. We configure the test driver to execute a warm-up phases for 10 minutes and the main execution phases for 30 minutes in which the queries are executed in sequence. During the execution some of the queries exceed the maximum execution time and we omit them before calculating the average execution time.

The summary of the results are shown in Table 1. In the second row of the table the following abbreviations are used: J for Jena, V for Virtuoso, and T for Total. The sign $\surd$ is used for cases compatible with our hypothesis and the sign $\times$ is used for the incompatible cases.

	H.1 – Scoring Variable Number			H.2 – Ranking Variable Number			H.3 – Limit		
Query ID	J	V	T	J	V	T	J	V	T
1				√	√	√	√	√	√
3				√	×	×	√	√	√
4				√	√	√	√	√	√
6				√	√	√	√	√	√
7	√	×	×	√	×	×	√	×	×
8				√	√	√	√	√	√
9	×	√	×	√	√	√	√	√	√
10	√	×	×	√	√	√	√	×	×
11	√	√	√	√	√	√	√	×	×
12	×	√	×	√	√	√	√	√	√
13				√	√	√	√	×	×
14				√	√	√	√	√	√
15				√	√	√	√	√	√
16				√	√	√	√	√	√
17				√	√	√	√	√	√
18				√	√	√	√	√	√
19				√	×	×	√	×	×
20				√	×	×	√	×	×
22				√	√	√	√	√	√
23				√	√	√	√	√	√
25	√	×	×	√	×	×	√	×	×
Total	√	√	×	√	√	√	√	√	√

Table 1: Summery of experimental evaluation

First of all, it has to be noted that the queries, which Top-k DBPSB generates, are adequate to test the hypothesis H.2 and H.3, whereas only the queries 7, 9, 10, 11, 12 and 25 can be use in validating H.1. The other queries have only

one rankable variable and all their scoring variables are related to it. This may depend on our choice of using DBpedia 10% as dataset; in our future work we intend to further investigate this issue.

Let us now elaborate on the results that concerns **hypothesis H.1**. According to hypothesis H.1, if we increase the number of rankable variables we should expect to observe an increasing execution time. Aggregating the results by query (i.e., a $\times$ can be obtain for a query only if both engines have a $\times$ for such a query), only query 11 confirms H.1 while, aggregating by engine (i.e., we give a $\times$ to an engine if 2/3 of the queries confirm the hypothesis), the hypothesis is confirmed. Wrapping up, we can conclude that H.1 is only a rough approximation of what actually stresses a SPARQL engine that has to execute a top-k query. Further investigation is needed to refine the hypothesis and better qualify the cases that stress the engines.

Let us now analyse the results concerning **hypothesis H.2**. Most of the queries' behaviour are compatible with our hypothesis since increasing the number of scoring variables increases the average execution time. Aggregation by SPARQL engine, the results indicate that Jena is compatible with our hypothesis in all cases whereas in Virtuoso only 5 cases do not confirm it. A bit worse are the results obtained aggregating by query: the execution time of queries 3, 7, 19, 20, and 25 are not compatible with our hypothesis in Virtuoso. These queries need more investigation to find the reason of such behaviour in Virtuoso.

Last but not least, let us consider the results about **hypothesis H.3**: the results shows that in general most of the queries' behaviour are compatible with our hypothesis. Aggregating by query, the results show that only queries 7, 10, 11, 13, 19, 20 and 25 do not confirm out hypothesis, whereas, aggregating by SPARQL engine, the results indicate that in Jena our hypothesis is true in all cases, while Virtuoso is able to take advantage of the limit clause for queries 7, 10, 11, 13, 19, 20 and 25.

6 Conclusions and Discussion

Top-k queries are a category of queries which is attracting a growing attention. The aim of top-k query is retrieving only the top k results ordered by a given ranking function. The current implementations of SPARQL engines do not support an efficient evaluation of top-k queries, as this class of queries is managed with a materialize-then-sort processing schema that computes all the matching solutions even if only a limited number k are requested.

In this work, we presented Top-k DBPSB, an extension of DBPSB that adds to the process described in [9] the possibility to automatically generate top-k queries. Top-k DBPSB satisfies the requirement of resembling the reality extending DBPSB which automatically derives datasets and queries from DBpedia and it query logs. Moreover, we provide experimental evidence that Top-k DBPSB also satisfies the requirement to stress the distinguish features of top-k queries.

The Top-k DBPSB proposed in this paper uses the same dataset, performance metrics, and test driver of DBPSB. The innovative part of this work consists in an

algorithm to create the top-k queries from the Auxiliary Queries of the DBPSB and its datasets.

Top-k DBPSB first creates a scoring function for each query combining (with randomly generated weights) one or more ranking criteria. Each criterion is obtained by normalising the values matching the variable in the object position of a ranking predicate. Top-k DBPSB can boost query variability by selecting different combinations of scoring variables from different rankable variables (i.e., the variable in the subject position of a ranking predicate). Then Top-k DBPSB adds to the query template the ranking predicates of the selected scoring variables. Finally, Top-k DBPSB adds `ORDER BY` and `LIMIT` clauses.

In order to support the hypothesis that we positively answered to the research question presented in Section 1, we experimentally showed using Jena, and Virtuoso SPARQL engines that the query variability provided by Top-k DBPSB stresses the SPARQL engines. To this end, we formulated three hypothesis and we empirically demonstrated that when the number of scoring variables increases the average execution time also does (hypothesis H.2) and that the average execution time is independent from the value used in the `LIMIT` clause (hypothesis H.3). Counterintuitively, hypothesis H.1 (i.e., increasing the number of rankable variable increases the average execution time) is not confirmed by our experiments.

The presented work is only a starting point to develop a comprehensive Top-k SPARQL query benchmarking, and it leaves several opportunities for further enhancement. In the following we present some of the possible extension points that we found during our research:

- Using complex type of scoring function in the top-k query generator.
- Using complex form of ranking criteria such as aggregated ranking criteria which use an aggregation function such as average, count, sum, maximum, and so on to compute the value of specific ranking criterion, and rankable criteria which produced by inference.
- More investigation on the queries that not follow our hypothesis, to formulate better hypothesis about what stresses a SPARQL engine that has to evaluate top-k queries.

References

1. C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
2. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In I. Horrocks and J. A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2002.
3. J. P. Cedeno. *A Framework for Top-K Queries over Weighted RDF Graphs*. PhD thesis, Arizona State University, 2010.
4. J. Cheng, Z. Ma, and L. Yan. f-sparql: a flexible extension of sparql. In *Database and Expert Systems Applications*, pages 487–494. Springer, 2010.

5. O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In S. Auer, C. Bizer, C. Müller, and A. V. Zhdanova, editors, *CSSW*, volume 113 of *LNI*, pages 59–68. GI, 2007.
6. J. Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.
7. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.
8. S. Magliacane, A. Bozzon, and E. Della Valle. Efficient execution of top-k sparql queries. In *The Semantic Web–ISWC 2012*, pages 344–360. Springer, 2012.
9. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. Dbpedia sparql benchmark - performance assessment with real queries on real data. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7031 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.
10. A. Owens, A. Seaborne, N. Gibbins, and mc schraefel. Clustered tdb: A clustered triple store for jena. Technical report, Electronics and Computer Science, University of Southampton, 2008.
11. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *The Semantic Web-ISWC 2006*, pages 30–43. Springer, 2006.
12. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. In Y. E. Ioannidis, D. L. Lee, and R. T. Ng, editors, *ICDE*, pages 222–233. IEEE, 2009.
13. A. Wagner, T. T. Duc, G. Ladwig, A. Harth, and R. Studer. Top-k linked data query processing. In *The Semantic Web: Research and Applications*, pages 56–71. Springer, 2012.