SPARQL Benchmarking with Automatically Generated OLAP Queries

Xin Wang[1], Steffen Staab[1,2], and Thanassis Tiropanis[1]

[1] Web and Internet Science Group
University of Southampton, UK
[2] Institute for Web Science and Technology
University of Koblenz-Landau, Germany

Abstract. The growing use of data analytics on Linked Data requires SPARQL engines to efficiently execute Online Analytical Processing (OLAP) queries. While SPARQL 1.1 provides appropriate basic constructs, corresponding optimization of SPARQL engines is still in its infancy and further development lacks benchmarks that mimick the data distributions found in Link Data. In fact, existing work on OLAP benchmarking for SPARQL has usually adopted queries and data from relational databases, which may not well represent Linked Data. We map typical OLAP operations to SPARQL and propose a tool named ASPG to automatically generate OLAP queries from real-world Linked Data, which can be used to construct analytic benchmarks for SPARQL engines. We present such a benchmark called DBOB as an example which consists of queries generated using DBpedia. We apply this benchmark to a replication of DBpedia and present the result.

Keywords: OLAP, Linked Data, benchmarking, query generation, SPARQL, DBpedia

1 Introduction

Linked Data principles foster the provisioning and integration of a large amount of heterogeneous distributed datasets [2]. SPARQL 1.1 [11] has introduced aggregations that enable users to do basic analytics. Though limited, SPARQL 1.1 is expressive enough to implement Online Analytical Processing (OLAP) which is an approach to analysing and reporting multidimensional statistics from different perspectives and levels of granularity refer to the datasets [5,3].

OLAP contains a rich set of combinations of analytical operations which generate a high workload on SPARQL engines that target the support of analytics queries. In fact, the scalability of SPARQL engines to execute OLAP queries is still rather limited owing further development and optimization. Such optimization and comparison of best developments critically depend on benchmarks that can measure the performance of SPARQL engines on analytic tasks from various perspectives. Several OLAP benchmarks for SPARQL have been proposed. For

example Kämpgen and Harth [12] convert queries and data from the Star Schema Benchmark (SSB) to SPARQL and Linked Data using the RDF Data Cube Vocabulary [6]. Since SSB is based on a relational database scenario, its data do not necessarily resemble typical Linked Data structures. Another example, the BowlognaBench [8], uses data and queries based on the Bowlogna Ontology [7]. Similar to SSB for SPARQL, BowlognaBench covers a specific scenario which may not represent the heterogeneity and structure of Linked Data.

Görlitz et al. [9] propose a SPARQL query generator called SPLODGE to lessen the dependency on fixed queries. Following the same direction we present a tool called Analytical SPARQL Generator (ASPG) that generates OLAP queries in SPARQL which can be used to construct analytic benchmarks. ASPG takes a RDF graph as input and selects triples by semi-random walk. Selected triples are parametrised to generate basic graph patterns (BGPs) which are then extended with aggregations that resemble OLAP operations. Queries produced by ASPG are guaranteed to return results from the given RDF graph since they are generated from triples in the RDF graph. We construct an analytical benchmark based on DBpedia, referred to as DBpedia OLAP Benchmark (DBOB), using ASPG generated queries complemented with manually constructed queries. We evaluate Virtuoso[3] using this benchmark and present the results.

The remaining sections of this paper are organised as follows: technical details of ASPG are described in Section 2; queries and dataset of DBOB are presented in Section 3; experiment settings and evaluation result of DBOB are given in Section 4, and conclusions are given in Section 6. Due to page limit, a complete list of DBOB queries is given in https://github.com/xgfd/ASPG.

2 Generating OLAP Queries from Linked Data

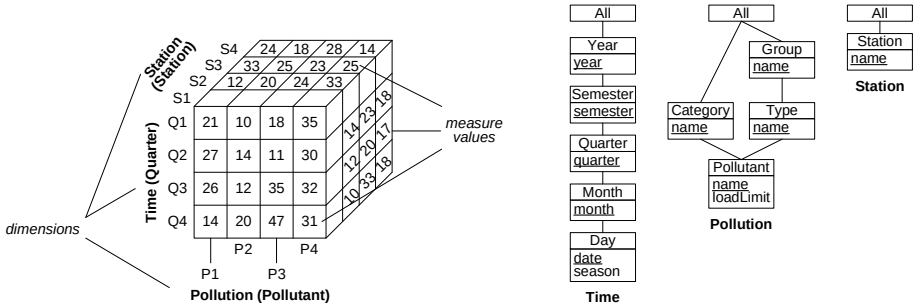
In this section we discuss the correspondence between typical OLAP operations and SPARQL components, and provide details of generating SPARQL queries from arbitrary RDF graphs that resemble typical OLAP operations.

SPARQL queries consist of basic graph patterns (BGPs) which can be viewed as graphs with variable nodes. When evaluating a BGP against a RDF graph, results are returned if and only if the BGP matches a sub-graph of the given RDF. Consequently, given an arbitrary RDF graph, we can construct BGPs that are guaranteed to return results by parametrising sub-graphs in the RDF. By controlling the structure of sub-graphs we can obtain BGPs that consist of chains or star-shaped triple patterns of arbitrary lengths. We simulate typical OLAP operations by summarising along properties (using GROUP BY) with randomly selected aggregate operations (e.g. SUM, COUNT, AVG etc.). In particular we discuss the challenges to generate queries from RDF graphs that are too large to fit in a single store and describe RDF summarising and sampling techniques to resolve those issues.

[3] http://virtuoso.openlinksw.com/

2.1 Background of OLAP Operations

OLAP queries operate on a multidimensional data model that is referred to as an OLAP cube. Each data point in the cube is associated with two types of attributes, dimensions and measures. Dimensions identify data points and are usually organised hierarchically. Measures represents associated values of a data point and are usually operands of aggregations. An example of OLAP cube is shown in Figure 1. There is no clear distinction between dimensions and measures. Any set of attributes that uniquely identifies a data point can be viewed as dimensions, and the remaining attributes are measures.



(a) (b)

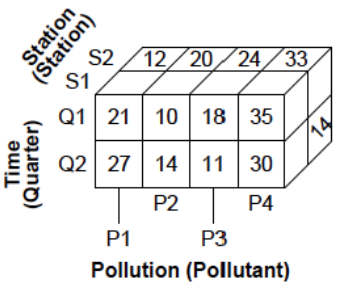
Fig. 1: A three-dimensional cube having dimensions **Time**, **Pollution**, and **Station**, and a measure **concentration**. Dimension hierarchies are shown on the right [4].

Typical OLAP operations defined on cubes include:

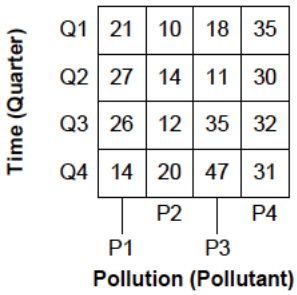
- Dice: Selecting a subset of an OLAP cube (Figure 2a).
- Slice: Slice is a specific case of dice picking a rectangular subset of a cube by choosing a single value for one of its dimensions (Figure 2b).
- Roll-up: Aggregating data by climbing up the hierarchy of a dimension (Figure 2c).
- Drill-down: Aggregating data at a lower level of the dimension hierarchy (Figure 2d). Drill-down is the reverse operation of roll-up.

In this paper we do not take into account operations that involve multiple OLAP cubes, such as drill-across [4], since multiple RDF graphs can be merged into one graph by taking their union.

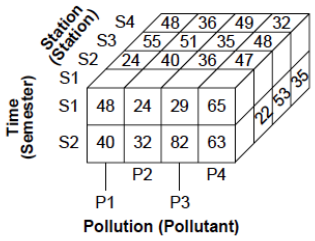
Kämpgen et al. [13] describe an approach to map OLAP queries into SPARQL queries with the RDF Data Cube (QB) vocabulary [6]. Since many Linked Data and SPARQL queries do not use QB, we examine the semantics of the above OLAP operations and propose a mapping between OLAP and SPARQL queries that are not limited to specific vocabularies.



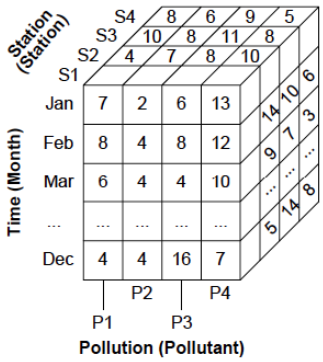
(a) Dice on **Station** = **'S1'** or **'S2'** and **Time.Quarter** = **'Q1'** or **'Q2'**



(b) Slice on **Station** for **StationId** = **'S1'**



(c) Roll-up to the **Semester** level



(d) Drill-down to the **Month** level

Fig. 2: OLAP operations [4].

2.2 Generating Dice and Slice Queries in SPARQL

Dice and Slice select a subset of an OLAP cube while in SPARQL the same functionality is achieved by BGPs.

An OLAP data point and its attributes (dimensions and measures) correspond to a subject and its properties in a RDF graph[4]. Dice selects multiple data points in an OLAP cube, whereas in SPARQL it is analogous to a BGP with optional constraints on object values (using FILTER), as shown below:

Query 1:

```
SELECT ?P ?Q ?S ?concentration
WHERE
{ ?point  :Pollution    ?P ; # FILTER(?P = "P1" || ?P = "P2")
```

[4] Mapping an OLAP data point to a subject is just one intuitive approach. An OLAP data point can be mapped to any RDF term.

```
          :Time          ?Q ; # FILTER(?Q = "Q1" || ?Q = "Q2")
          :Station       ?S ; # FILTER(?S = "S1" || ?S = "S2")
          :concentration :?concentration
}
```

Unlike in relational databases (where dimensions are usually keys that are distinguished from measures), we argue that dimensions and measures are indistinguishable in RDF and SPARQL. Thus any BGPs correspond to a valid Dice operation (with Slice as a special case) in OLAP. There may be difficulties to aggregate on certain types of values, since most aggregations in SPARQL are arithmetic. Meanwhile it is always possible to convert an arbitrary type to a numeric. For example a literal can be converted to its length (i.e. STRLEN), and a resource can be converted to its number of occurrences (i.e. COUNT). Queries generated with the above modifications may not be meaningful in a practical sense, they serve the purpose as far as benchmarks are concerned. In the rest of this paper we interchangeably use Dice query and BGP when no confusion is caused.

2.3 Generating Roll-Up and Drill-Down Queries in SPARQL

Roll-up and drill-down group measure values at a specific dimension level and aggregate values in each group using a given aggregate function. Without losing generality, we focus on the mapping of roll-up since drill-down is the reverse operation. In SPARQL Roll-up is achieved by GROUPing BY some variables (i.e. dimensions) in a query and aggregate on other variables (i.e. measures).

Given a Dice query (a BGP basically) that selects entities at the specified dimension levels (i.e. there is a triple pattern matching each of the specified dimension levels), simply GROUPing BY the specified dimension levels and applying an aggregate function on measure values (i.e. any variable object not appeared in GROUP BY) would simulate Roll-up in SPARQL. Taking Query 1 as an example, if we would like to know the concentration of each pollutant at each station averaged over all time points, we would GROUP BY variable *?P* and *?S* and apply AVG on variable *?concentration*, as shown in the query below:

Query 2:

```
  SELECT ?P ?S (AVG(?concentration) AS ?avgCon)
  WHERE
{ ?point  :Pollution     ?P ;
          :Time          ?Q ;
          :Station       ?S ;
          :concentration :?concentration
} GROUP BY ?P ?S
```

It is worth noticing that GROUPing BY all variables in a BGP does not change the result of the BGP[5]. Thus in SPARQL a Dice query can be trivially

[5] It is enough to GROUP BY a subset of all variables that uniquely identifies an entity. Variables excluded from GROUP BY can be selected using the SAMPLE aggregation.

extended to a Roll-up query by appending a GROUP BY all variables clause at the end of its BGP.

A more involved case is when we are interested in dimension levels that do not explicitly appear in a Dice query. For example, in Query 2 instead of asking for concentration per pollutant, we may ask for the same measure per Category in the Pollution hierarchy (shown in Figure 1b). Depending on whether the hierarchy (dimension instance as in [4]) is explicitly stated in the RDF graph being queried, we use two different techniques to generate Roll-up queries.

Dimension hierarchy is explicit Assuming the hierarchy is stated as triples, e.g. in the form

P_i *:rollupTo* C_j

where P_i is an instance of Pollutant, C_j is an instance of Category and :rollupTo states that its object is one level above its subject in the dimension hierarchy, we can add the triple pattern

?P a Pollutant; :rollupTo ?C. ?C a Category.

to Query 2 and GROUP BY *?C* (and *?S*) instead of *?P*.

Dimension hierarchy is absent In this case values can be manually categorised in SPARQL using an IF expression

rdfTerm IF (boolean cond, rdfTerm expr1, rdfTerm expr2)

where the whole expression evaluates to the value of $expr1$ when *cond* evaluates to *true*, otherwise $expr2$. By nesting IF expressions, we can define a surjective (only) function

$$cat : rdfTerm \rightarrow rdfTerm$$

that maps a value to a category defined by users. For example, assuming both $P1, 2$ belong to $C1$, we can express *cat* in SPARQL as

$$cat(?P) := IF(?P = P1||?P = P2, C1, Other),$$

and convert Query 2 to the following query[6]

Query 3:

```
  SELECT ?C ?S (AVG(?concentration) AS ?avgCon)
  WHERE
{ ?point  :Pollution     ?P ;
          :Time          ?Q ;
          :Station       ?S ;
          :concentration :?concentration
} GROUP BY (cat(?P) AS ?C) ?S
```

[6] SPARQL 1.1 doesn't have the ability to define new functions, and therefore *cat* should be considered as a macro in Query 3.

This technique is more useful to categorise numerics (or elements of totally ordered sets) into different ranges. For example, we can define a *cat* to group numbers into ranges as

$$cat(x) := IF(x <= low, "Low", IF(x <= high, "Medium", "High"))$$

where *low* and *high* are numbers.

Given a BGP (i.e. a Dice query), ASPG adopts a naive heuristic to extend it to a Roll-up query: 1) It randomly selects a subset of all variables of the BGP as dimensions, and the remains as measures; 2) All dimensions are used in a GROUP BY clause; 3) If a measure is known to be numerical, it is aggregated using one of the set functions COUNT, MAX, MIN, AVG, SUM, GroupConcat. Otherwise, this measure is firstly converted to a literal with STR and then to an integer with STRLEN, and aggregated using a set function. This procedure is listed below:

```
queryGen(BGP)
    D, M ∈ vars(BGP)
    GroupBy ← "GROUP BY"
    for d ∈ D
        GroupBy ← concat(GroupBy, d)

    SELECT ← "SELECT"
    for m ∈ M
        AGG ∈ {COUNT, MAX, MIN, AVG, SUM, GroupConcat}
        if m is numerical
            SELECT ← concat(SELECT, AGG(m))
        else
            SELECT ← concat(SELECT, AGG(STRLEN(STR(m))))

    query ← concat(SELECT, BGP, GroupBy)
    return query
```

2.4 Generating Basic Graph Patterns

We generate BGPs by replacing nodes in RDF graphs with variables. A RDF graph (or a BGP) can be decomposed into star-shaped or chain-shaped sub graph patterns. Considering a triple (or a triple pattern) as an undirected edge between subject and object, we define the degree of a node as the number of edges connecting to this node. A star-shaped graph pattern has one and only one central node with a degree greater than 1 and all other nodes of degree 1. A chain only has nodes whose degree are no more than 2. We generate a sub-graph from a RDF graph by repeating two steps: 1) select one node in the RDF graph as root, 2) add an edge connected to the root to the sub-graph. A star-shaped graph pattern is generated by selecting the same node as root in every iteration, while a chain is generated as selecting as root the other node

in the last added edge in each iteration. We generate a mix of stars and chains by controlling a branching probability of whether to select a different root in each step, as shown in the pseudo code below, where *RDF* is a RDF graph, T is a termination predicate function mapping a BGP to a boolean, p is branching probability, and *parametrise* maps a non-property IRI to a variable:

```
BGPGen(RDF, T, p)
    BGP ← {}
    root ∈ IRIs(RDF)
    while (!T(BGP))
        E ← getTriples(root)
        e ∈ E
        BGP ← parametrise(e) ∪ BGP
        if (random() < p)
            root ← root
        else
            root ← getObject(e)
    return BGP
```

The termination function is used to control the length of generated BGPs. In this paper we define the termination condition to result in true if either the BGP reaches 10 triple patterns or the longest path in the BGP reaches 5.

The above algorithm guarantees a non-empty result set when evaluating the generated BGP against the source RDF, but there is no guarantee about the size of the result. To avoid BGPs whose result size is too small for aggregation, we evaluate generated BGPs and filter out those whose result size is less than a threshold. This safe guard is not always necessary. Later we present a set of queries generated from DBpedia and none of the BGPs falls below a threshold of 100,000.

Generating BGP with large or remote RDF graphs

When using the above method, one may encounter difficulties when the RDF graph cannot be use as a direct input to *BGPGen*. For example, the graph may be too large to be traversed or it is only available as a SPARQL endpoint. In order to deal with such cases, we use techniques that combines ontology and triple sampling to convert large RDF graphs into smaller ones. We describe our techniques using DBpedia as an example, but the techniques can be applied to any graph.

To generate a BGP we need to know the connection between nodes. Such information is often captured in an ontology-like structure of a RDF graph that gather all instance level properties to their classes. For simplicity we still use ontology to refer to such structure. We can issue a SPARQL CONSTRUCT query to recover the ontology (assuming all instances in the RDF belong to some classes, i.e. all *rdf:type* are explicit). However, to construct the whole ontology in one query is likely to end with a time out. Instead, we first retrieve all classes,

and then use a script to collect properties between any two classes using the query template below:

Query 4:

```
CONSTRUCT
{ dbo:$1 ?p dbo:$2 }
SELECT DISTINCT ?p
WHERE { [ a dbo:$1] ?p [ a dbo:$2]. }
```

where *$1* and *$2* are replaced by class names (e.g. Person, Event etc.). The ontology is the union of all graphs returned by Query 4. The ontology can be used as the input graph (i.e. the parameter G) in the BGP generation algorithm with some extra care taken. Since all nodes in the ontology are classes, they should all be replaced with variables in generated BGPs. In addition, when following a reflexive property, a new variable should be used as root. For example, *dbo:Person* has a reflexive property *foaf:knows*. When this property is included in a BGP, its subject and object should be two different variables.

Using the ontology instead of the original RDF graph significantly reduces the complexity of BGP generation. However it does not always guarantee that the generated queries have results against the original graphs. For example in DBpedia both *Athlete* and *Artist* are sub-classes of *Person*, an instance of either *Athlete* or *Artist* may also has a *rdf:type* property pointing to *Person*. As a result properties of both *Athlete* and *Artist* are gathered at *Person*. There is a chance that an *Athlete* property and an *Artist* are connected to the same node in a BGP, which may not match any triple in the original graph. This issue can be relieved by gathering properties only to the lowest class of an instance, however doing that in SPARQL is quite cumbersome[7].

When the above method is not applicable (e.g. generating BGPs from DBpedia), we employ triple sampling as an alternative approach to extract subsets of RDF graphs. By repetitively sampling sub-graphs of simple shapes, a more complex and larger sub-graph can be constructed. For example, in ASPG we sample DBpedia using triple chains of length 5, as show in Query 5.

Query 5: Chained triple sampling

```
CONSTRUCT
{
  ?s  ?p1 ?n1. ?n1 ?p2 ?n2.
  ?n2 ?p3 ?n3. ?n3 ?p4 ?n4.
  ?n4 ?p5 ?e.
}
WHERE
{
  ?s a dbo:$1. ?e a dbo:$2.
```

[7] It requires to calculate the position of an item in a linked list and to identify the maximum item in a set. Refer to https://git.io/vwP0t for more details.

```
    ?s  ?p1 ?n1. ?n1 ?p2 ?n2.
    ?n2 ?p3 ?n3. ?n3 ?p4 ?n4.
    ?n4 ?p5 ?e.
}
```

where *$1* and *$2* are replaced by class names. It is left to users to decide how many and what class pairs are used. For example, in the construction of DBOB we use the top 50 classes that have most instances, and it turns out that triple chains sampled by Query 5 intertwine with each other. The result graph is significantly smaller than DBpedia while its structure is rich enough to generate complex queries.

In addition we may also want to identify properties whose ranges are numerics, even it is always possible to convert an arbitrary type to a numeric in SPARQL. Such information enables us to identify variables of numerics to which aggregate functions can be directly applied.

2.5 Complexity Analysis

We examine the time complexity of aggregate functions used in ASPG, namely GROUP BY and set functions COUNT, MAX, MIN, AVG, SUM, GroupConcat (excluding SAMPLE).

GROUP BY can be realised by the application of a higher-order 'map' function on a constant time lower-order function and each set function can be mapped to a higher-order 'fold' function on a constant time arithmetic function. All aggregations used in ASPG have $O(n)$ time complexity, where n is the size of query result regardless of the grouping of the result. We exclude SAMPLE from ASPG since it is a $O(1)$ operation.

We conclude that the time complexity of aggregating on a BGP is linear in the number of aggregate functions and independent of the grouping. In other words, the time complexity of a query (generated by ASPG) can be characterised by its BGP and its number of aggregate functions.

3 DBOB: A Benchmark Constructed with ASPG

In order to evaluate ASPG, we construct an OLAP benchmark named DBOB from DBpedia. DBOB contains 12 queries, of which Q1-3 are real-world queries from online analysis and Q4-12 are generated with ASPG.

Query 4-12 are generated with the following steps:

1. Retrieving the top 50 classes from DBpedia having most instances (count only direct instances).
2. Sampling from the DBpedia SPARQL endpoint using chains of length 5 whose endpoints are drawn from instances of the 50 classes.
3. Generating OLAP queries from the RDF graph gained from step 2.

4. Evaluating the query against DBpedia and filtering out those whose result size is less than 60,000 (most queries have a result size of more than 100,000).

As stated in Section 2.5 as far as time complexity is concerned, queries can be described by their BGPs and numbers of aggregate functions. We roughly measure the complexity of a BGP by its number of triple patterns[8]. Due to the page limit we summarise these queries with the aforementioned characteristics and compare them to OLAP4LD-SSB queries [12] in Table 1. For the complete list of DBOB queries please refer to https://github.com/xgfd/ASPG.

DBOB	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
# of triple patterns	10	5	4	4	6	4	4	8	7	2	7	7
# of group by-s	1	1	3	1	1	1	1	1	1	1	1	1
# of set functions	2	3	3	4	5	4	3	8	7	1	7	7
OLAP4LD-SSB												
# of triple patterns	6	6	7	9	8	8	10	10	8	9	11	13
# of group by-s	0	0	0	1	1	1	1	1	1	1	1	1
# of set functions	1	1	1	1	1	1	1	1	1	1	1	1

Table 1: Comparison of DBOB and OLAP4LD-SSB queries.

Comparing to OLAP4LD-SSB, the number of triple patterns of ASPG queries vary a lot, as a result of random sampling. In addition, since ASPG does not focus on the semantic of queries, it can simply add as many aggregate functions as required. The ability of providing triple patterns and aggregate functions on demand makes ASPG a very flexible tool for benchmarking (although its queries may not make much sense to humans).

4 Evaluation

We run the benchmark on a DBPedia 3.9 endpoint with the following settings: 4*2.9GHz CPU, 16G memory, Ubuntu 14.04.4, Virtuoso opensource 7.1.0.

[8] The complexity of a BGP is not only affected by the number of triple patterns (which represents the number of joins), but also by the number of intermediate results in each join. However the later requires detailed statistics to estimate which are not always available.

We use the BSBM query driver[9] to execute all queries with 0 warm up and 20 runs.

The evaluation result is shown in Table 2, where QET stands for query execution time in seconds, # is the query result size before aggregation, #T is the number of triple patterns, and #A is the number of aggregate functions. We also calculate the correlation between QET and the number of triple patterns, result size and the number of aggregate functions respectively.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**
QET	0.6524	1.1897	1.3082	0.0991	0.7189	0.3141	2.9013	0.0776	1.4366	19.3319	2.7358	1.7249
#	600	39401	NA	95515	59134	65260	548454	120764	258799	81187172	175540	4958966
Correl.	**QET, #**		**QET, #T**		**QET, #A**							
	0.99		0.07		0.38							

Table 2: DBOB evaluation result.

Most queries are finished in no more than 3 seconds. This may due to that queries with aggregation usually do not need to materialise all intermediate results. In addition we see the correlation between QET and the number of triple patterns is quite low. It is not surprising since QET of BGPs is mainly affected by the number of intermediate results which is not captured by only the number of triple patterns. At the same time the number of aggregate functions shows a higher impact on QET. One possible reason could be the high number of aggregate functions in ASPG queries. Alternatively as the contribution to QET from aggregation is liner to result size, the relatively higher impact from aggregation may just be a side effect of the high correlation between the result size and QET. It may be worth measuring only the execution time of aggregation, however such measure is usually difficult to obtain from outside of query engines.

5 Related Work

We divide related work into two categories: SPARQL query generators and SPARQL benchmarks.

5.1 Related Query Generators

ASPG generates queries from a RDF graph, which is similar to SPLODGE [9]. SPLODGE exploits query characteristics (e.g. join type, query type, variable pattern) and constructs queries from a federated RDF graph. While ASPG focuses on simulating OLAP queries, SPLODGE aims to generate queries for federated benchmarks. Both decompose queries into star-shaped or chained triple

[9] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BenchmarkRules/index.html#datagenerator

patterns. ASPG queries are generated by replacing nodes in a sub-RDF-graph with variables, while SPLODGE queries are generated from linked predicates (i.e. a pair of predicates sharing a common node). SPLODGE queries are not guaranteed to have results, but statistics are used to increase the chance.

FEASIBLE [16] represents a different approach to generate benchmark queries. Instead of generating queries from a RDF graph, it takes existing queries (from query logs) as prototypes and generates similar queries. Comparing to ASPG and SPLODGE, FEASIBLE queries are usually more close to real-world queries.

5.2 Related Benchmarks

To the best of our knowledge only two existing benchmarks are based in an OLAP scenario, namely BowlognaBench [8] and OLAP4LD [12]. We also review a few popular non-OLAP benchmarks.

- Lehigh University Benchmark (LUBM) [10] is designed with focus on inference and reasoning capabilities of RDF engines.
- SP^{2}Bench [17] has a focus of testing the performance of a variety of SPARQL features.
- The Berlin SPARQL Benchmark (BSBM) [1] mimics a e-commerce scenario and its dataset resembles a relational database.
- DBpedia SPARQL Benchmark (DBPSB) [14] uses (a sub set of) DBpedia as testing data and most used DBpedia queries as testing queries.
- BowlognaBench models an OLAP use case around the Bowlogna Ontology [7] and implements queries such as TopK, Max, Min, Path etc.
- OLAP4LD converts dataset and queries of the Star Schema Benchmark [15] into RDF and SPARQL. It resembles OLAP queries in relational databases.

We compare DBOB with aforementioned benchmarks in Figure 3.

	LUMB	**SP2Bench**	**BSBM**	**DBPSB**	**OLAP4LD**	**Bowlogna**	**DBOB**
Dataset	Synthetic	Synthetic	Synthetic	Real	Synthetic	Synthetic	Real
Queries	Synthetic	Synthetic	Synthetic	Real	Synthetic	Synthetic	Mix
Dataset classes	43	8	8	239	7	76	239
Dataset properties	32	22	51	1200	28	36	1200

Table 3: Comparison of DBOB and existing benchmarks, adapted from [14].

6 Conclusions and Future Plan

In this paper we present ASPG that can be used to generate Dice, Slice, Roll-up and Drill-down queries in SPARQL. By exploiting ontologies and triple sampling

techniques, ASPG is able to generate queries from large RDF graphs or graphs available as SPARQL endpoints. We further construct a benchmark called DBOB with ASPG and DBpedia to evaluate processing time of OLAP SPARQL queries.

Queries generated by ASPG usually have more complex BGPs compared to real-world queries. Perhaps human users are more likely to issue simple queries and combine their results afterwards, due to the lack of convenient query builders and constraints on query complexity from SPARQL endpoints. We argue that as far as query processing time is concerned, generated queries may give more insight on the performance of SPARQL engines than simple real-world queries. In addition, it is likely that the increasing demand of SPARQL analytics will foster better tools that enable users to generate complex queries. The Roll-up generation heuristic used by ASPG may contribute to the creation of such tools.

Currently ASPG queries only consist of one BGP and randomly selected aggregate functions, while real-world queries may also employ FILTERs and sub-queries (e.g. Q2 and Q3 of DBOB). As a result ASPG queries only represent some basic analytical needs. A future plan is to extend ASPG to generate multiple BGPs and sub queries that covers a broader range of analysis operations.

References

1. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. International Journal On Semantic Web and Information Systems (IJSWIS) - Special Issue on Scalability and Performance of Semantic Web Systems 5(2), 1–24 (2009)
2. Capadisli, S., Auer, S., Riedl, R.: Linked Statistical Data Analysis. Semantic Web (2013), http://km.aifb.kit.edu/sites/swc/2013/submissions/swc2013{_}submission{_}7.pdf
3. Chaudhuri, S., Dayal, U.: An overview of data warehousing and OLAP technology. ACM SIGMOD Record 26(1), 65–74 (1997)
4. Ciferri, C., Ciferri, R., Gómez, L., Schneider, M., Vaisman, A., Zimányi, E.: Cube Algebra: A Generic User-Centric Model and Query Language for OLAP Cubes. Int. J. Data Warehous. Min. 9(2), 39–65 (2013)
5. Codd, E.F., Codd, S.B., Salley, C.T.: Providing OLAP (on-line Analytical Processing) to User-analysts: An IT Mandate. Codd and Date 32, 3–5 (1993), http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem{_}dwh/lit/Cod93.pdf
6. Cyganiak, R., Reynolds, D., Tennison, J.: The RDF Data Cube Vocabulary http://www.w3.org/TR/vocab-data-cube/
7. Demartini, G., Enchev, I.: The Bowlogna Ontology : Fostering Open Curricula and Agile Knowledge Bases for Europe ' s Higher Education Landscape 0, 1–11 (2012)
8. Demartini, G., Enchev, I., Wylot, M., Gapany, J., Cudré-Mauroux, P.: BowlognaBench-Benchmarking RDF Analytics. Data-Driven Process Discovery and Analysis 116, 82–102 (2011)
9. Görlitz, O., Thimm, M., Staab, S.: SPLODGE: Systematic Generation of SPARQL Benchmark Queries for Linked Open Data. The Semantic WebISWC 2012 (2012), http://www.springerlink.com/index/L516V572110884N1.pdf
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics 3(2-3), 158–182 (2005), http://linkinghub.elsevier.com/retrieve/pii/S1570826805000132
11. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language (2013)

12. Kämpgen, B., Harth, A.: No Size Fits All Running the Star Schema Benchmark with SPARQL and RDF Aggregate Views. In: ESWC 2013 (2013), http://people.aifb.kit.edu/bka/ssb-benchmark/{#}rdbms-q1.1--1
13. Kämpgen, B., O'Riain, S., Harth, A.: Interacting with statistical linked data via OLAP operations. LAPIS 2012 913(Ild 2012), 36–49 (2012)
14. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL benchmark - Performance assessment with real queries on real data. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 7031 LNCS, pp. 454–469 (2011)
15. Neil, P.O., Neil, B.O., Chen, X.: Star Schema Benchmark - Revision 3. Tech. rep., UMass/Boston (2009), http://www.cs.umb.edu/{~}poneil/StarSchemaB.pdf
16. Saleem, M., Mehmood, Q., Ngomo, A.C.N.: FEASIBLE: A feature-based SPARQL benchmark generation framework. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 9366, 52–69 (2015)
17. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL performance benchmark. In: proceedings of the International Conference on Data Engineering. pp. 222–233. IEEE (2009), http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=4812405