Optimizing SPARQL Queries using Shape Statistics

Anonymous Author(s)

ABSTRACT

With the growing popularity of storing data in native RDF, we witness more and more diverse use cases with complex SPARQL queries. As a consequence, query optimization – and in particular cardinality estimation and join ordering – becomes even more crucial. Classical methods exploit global statistics covering the entire RDF graph as a whole, which naturally fails to correctly capture correlations that are very common in RDF datasets, which then leads to erroneous cardinality estimations and suboptimal query execution plans. The alternative of trying to capture correlations in a fine-granular manner, on the other hand, results in very costly preprocessing steps to create these statistics. Hence, in this paper we propose *shapes statistics*, which extend the recent SHACL standard with statistic information to capture the correlation between classes and properties. Our extensive experiments on synthetic and real data show that shapes statistics can be generated and managed with only little overhead without disadvantages in query runtime while leading to noticeable improvements in cardinality estimation.

1 INTRODUCTION

Driven by diverse movements, such as Linked Open Government Data, Open Street Map, biological data [22], DBpedia[1], and YAGO [21], more and more data is being published in RDF [7] capturing a multitude of diverse information. Along with the growing popularity, increasingly complex queries formulated in SPARQL [6] are being executed over such data to answer business and research questions. Query logs of the public DBpedia SPARQL endpoint, for instance, contain SPARQL queries with up to 10 joins [3] and analytic queries in the biomedical field can involve more than 50 joins per query [22]. Therefore, the need for high-performance SPARQL query processing is now more pressing than ever.

Existing approaches for query optimization in RDF stores often adapt techniques from relational databases modeling an RDF dataset as a single large table with three column [5, 16] (one column for each of the components of an RDF triple: subject, predicate, and object). Nevertheless, accurate cardinality estimation is at the heart of any query optimizer that does not rely on heuristics but instead uses a cost model to find the best query execution plan for a given query. Cardinality estimation then relies on the availability of statistics describing the characteristics of the data to estimate the sizes of intermediate results produced while query execution. However, general statistics typically result in highly imprecise estimations since they are mostly gathered on the RDF graph as a whole, in contrast to the relational case where it is possible to create such statistics with higher precision since data is separated into multiple tables [15]. Furthermore, assuming independence when joining parts of SPARQL queries (triple patterns) leads to erroneous estimations [9] as co-occurrences of certain predicates are highly correlated [19].

[1] https://dbpedia.org/sparql

Hence, exploiting more fine-grained statistics capturing correlations among RDF triples leads to more accurate join cardinality estimations [19]. However, creating such statistics comes at the price of a very time and resource-intensive preprocessing step. On the other hand, the alternative of online, query-dependent, sampling [20] results in overheads during query optimization. Instead, what we propose in this paper is to better exploit the information that is often provided along with an RDF dataset: SHACL (Shapes Constraint Language) [14] constraints, which is a recent standard for validating RDF datasets that are becoming more and more popular. SHACL defines so-called shapes describing the relationships between entities of a specific class, their properties, and their connections to other classes of entities. Although they are currently only used for validation purposes, we show in this paper that by slightly extending them with basic statistics, they can be exploited for join cardinality estimation as well.

In summary, this paper makes the following contributions. First, we extend the SHACL definition to capture statistical information to replace the need for creating complex (and expensive) statistics over RDF datasets. To the best of our knowledge, this is the first proposal of this kind. Second, we introduce an algorithm to enhance SHACL shapes with statistical information and to exploit these statistics for join cardinality estimation and query optimization. Third, we study the impact of our approach using both synthetic (LUBM [10], WatDiv [2]) and real (YAGO-4 [21]) datasets, demonstrating that shapes statistics can provide higher precision for query optimization with only a little overhead.

This paper is structured as follows. While Sections 2 and 3 discuss related work and introduce preliminaries, Section 4 formally defines the problem. Section 5 then describes our proposed extension of the SHACL standards, and Section 6 presents techniques to exploit the additional information for cardinality estimation and query optimization. Section 7 discusses the results of our extensive experimental study, and Section 8 concludes the paper with an outlook to future work.

2 RELATED WORK

Cardinality estimation has been studied extensively in the context of relational databases [20]. For SPARQL queries, existing techniques adapt relational approaches [13, 24] and focus mostly on specific type of queries [19]. Usually, these approaches construct different kinds of single or multidimensional synopses over databases that can be used to estimate cardinalities [23]. While algorithms designed to generate synopsis for unlabelled graphs are not applicable here (as the edges in RDF graphs are labeled), consequently approaches to generate RDF summaries either produce very large summaries [23], have very high computational complexities, or they are unable to preserve the RDF schema while constructing the summaries [23]. Therefore, the most promising approaches aim at using statistics computed directly from edge label frequencies. In particular, RDF-3X proposes a histogram-based technique for cardinality estimation based on edge label frequencies. This technique was later extended by exploiting the statistical information of Characteristic Sets [19], which compute frequencies of sets of predicates sharing the same subject to estimate the cardinalities. This approach shows high

performance for star-shaped queries while it suffers from significant underestimation due to the independence assumption in the general case [20]. This approach was extended as Characteristic Pairs [18] to overcome this limitation, but it could only support multi-chain star queries. Moreover, extracting Characteristic Sets from large heterogeneous graphs is computationally expensive. SumRDF [23] is another cardinality estimation approach based on a graph summarization. It fails to handle large queries due to a prohibitive computation cost, and it is costly to construct such summaries over large RDF graphs [20].

A recent benchmark, G-CARE [20], analyzed the performance of existing cardinality estimation techniques for subgraph matching. This analysis revealed that the techniques based on sampling and designed for online aggregation outperform the cardinality estimation techniques for RDF graphs. *This calls for a more in-depth study on how to perform cardinality estimation for SPARQL query optimization appropriately.*

In a recent work, Shape Expressions (ShEx) [4] have been used to reorder triple patterns to enable SPARQL query optimization [1], i.e., it estimates an order of execution for the triple patterns based on some heuristic inference on which triples are more selective. For instance, if a shape definition says that every instructor has one or more courses, but every course has exactly one instructor, it infers that the cardinality of courses is at least the same cardinality of instructors and probably larger. Hence, this optimization procedure is not based on actual data.

Therefore, contrary to existing works, we aim at exploiting fine-grained statistics based on shapes to produce more precise cardinality estimations for query planning. This will allow us to overcome the limitations of existing methods that only use the global-statistics. [11]. To this end, instead of creating large expensive summaries and characteristic sets over the RDF graphs to estimate the cardinalities, we exploit SHACL shapes constraints (which are as expressive as ShEx [4]) and annotate the *Node* and *Property Shapes* with the statistics of the input RDF graph. Compared to other solutions, it requires a lightweight preprocessing and retains the structure of original RDF and SHACL shapes graphs. Moreover, *this allow us to study more closely the effect of more fine-grained statistics, and more accurate cardinality estimation for the task of SPARQL query optimization.*

3 PRELIMINARIES

RDF Graphs: RDF graphs model entities and their relationships in the form of triples consisting of SPO <*subjects, predicates, objects*>. We present a simplified example of an RDF graph G based on the LUBM [10] dataset in Figure 1, where oval and rectangular shapes represent IRIs and literal nodes, respectively. An RDF graph is formally defined as:

Definition 3.1 (RDF Graph). Given pairwise disjoint sets of IRIs I, blank nodes B, and literals L, an RDF Graph G is a finite set of RDF triples $\langle s, p, o\rangle \in (I \cup B) \times I \times (I \cup B \cup L)$.

SPARQL: SPARQL [6] is a standard query language for RDF. A SPARQL query consists of a finite set of triple patterns (known as basic graph pattern, BGP) and some conditions that have to be met in order for data to be selected and returned from an RDF graph. Each SPO position in a triple pattern can be concrete (i.e., bound) or a variable (i.e., unbound). The variable names in a SPARQL query are prefixed by a '?' symbol, e.g., ?X. To answer a BGP, we require *a mapping between variables to values in an RDF graph*, all the resulting triples existing in the RDF graph obtained by replacing the variables with values are answers to

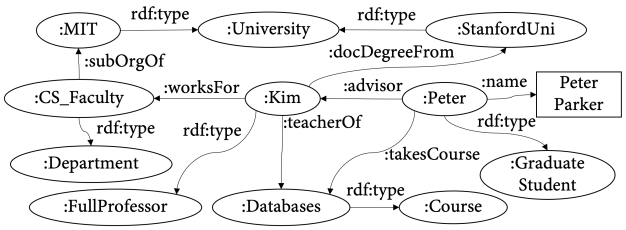


Figure 1: An RDF Graph G

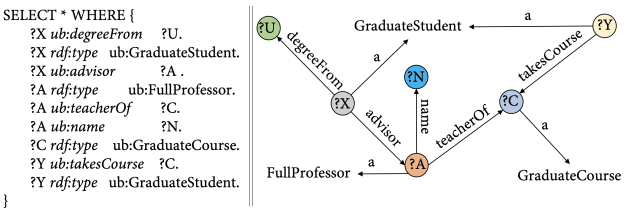


Figure 2: Query Q and its Graph Q_G

the BGP. Figure 2 shows an example SPARQL query (Q) and its query graph Q_G on the graph of Figure 1. A BGP is defined as:

Definition 3.2 (BGP). Given a set of IRIs I, literals L, and variables V, a BGP is defined as $T \subseteq (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$, whose elements are called triple patterns.

Shapes Graphs: Several schema languages have been proposed for RDF in the past, where the most common are RDF Schema (RDFS[2]) and OWL [17]. RDFS is primarily used to infer implicit facts, and OWL is an extension of RDF and RDFS to represent ontologies. The declarative Shapes Constraint Language (SHACL) [14] became a W3C standard recently. SHACL schema provides high-level information about the structure and contents of an RDF graph. It allows to define and validate structural constraints over RDF graphs. SHACL models the data in two components: the *data graph* and the *shape graph*. The *data graph* contains the actual data to be validated, while the *shape graph* contains the constraints against which resources in the *data graph* are validated. These constraints are modeled as node and property shapes, which consist of attributes encoding the constraints. The node shapes constraints are applicable on nodes that are instances of a specific type in the *data graph* while the property shapes constraints are applicable to predicates associated with nodes of specific types. We define a SHACL shapes graph as follows:

Definition 3.3 (SHACL Shapes Graph). A SHACL shapes graph G_{sh} is an RDF graph describing a set of node shapes S and a set of property shapes P, such that $target_S : S \mapsto I$ and $target_P : P \mapsto I$ are injective functions mapping each node shape $s_i \in S$ and each property shape $p_i \in P$ to the IRI of a target class and a target predicate in G respectively, and $\phi : S \mapsto 2^P$ is a surjective function assigning to each node shape s_i a subset $P_i \subseteq P$ of property shapes.

For example in Figure 3, node shape constraints are applicable on node `ub:GraduateStudent` and its property shapes constraints are applicable on predicates like `takesCourse`, and `advisor`. This information is declared with attributes *sh:targetClass* for node shapes and *sh:path* for property shapes. Note that the attributes in the dark shaded boxes are part of our extension of the SHACL definition, explained in Section 5.

[2] https://www.w3.org/TR/rdf-schema/

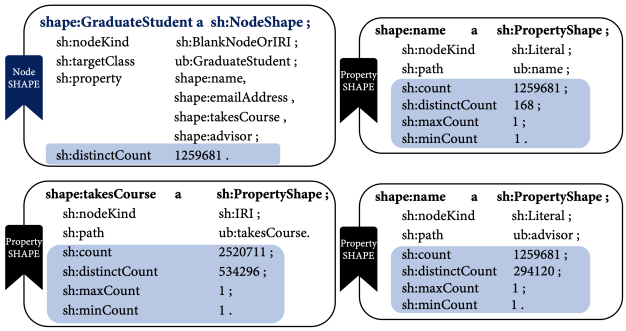


Figure 3: SHACL Shapes Graph

The Shapes Expression (ShEx [4]) language also serves a similar purpose as SHACL to validate RDF graphs. Nonetheless, the two formulations diverge mostly at the syntactic level [12], and our approach can be extended to work using ShEx or other constraints languages as well without the loss of generality.

4 PROBLEM FORMULATION

Given an input query Q, a query optimizer has the goal to find a query plan expected to answer Q in the minimum amount of time [15]. Constructing a SPARQL query plan includes finding a join ordering between triple patterns of its BGPs. In this paper, we focus on the join ordering of BGPs defined as follows:

Definition 4.1 (Join Ordering). Given a set of triple patterns $T=\{tp_1, tp_2, ..., tp_n\} \subseteq (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$, the join order O for BGPs is defined as a total ordering O of T so that for every $t_i, t_j \in T$ either $t_i \prec_O t_j$ or $t_j \prec_O t_i$.

To find an optimal plan, a query optimizer needs to explore the search space of semantically equivalent join orders and choose the optimal (cheapest) plan according to some cost function. It is crucial to accurately estimate the join cardinality between triple patterns of a given query to construct a query plan with an efficient join ordering [9]. In line with the related work [20], we neglect other cost factors and focus on join cardinality as the most dominant cost factor to find a join ordering. We formally define the problem of estimating join cardinalities as follows:

Problem 1 (Join Cardinality Estimation). *Given a set of triple patterns $T=\{tp_1, tp_2, ..., tp_n\}$, apply a cardinality estimation function $\bar{J} : T \times T \mapsto \mathbb{N}$ such that for every pair of triple patterns $(tp_i, tp_j) \in T$, $\bar{J}(tp_i, tp_j) \approx |tp_i \bowtie tp_j|$.*

We extend the above estimation problem also to the case of joining a triple pattern with the intermediate results of prior join operations, e.g., to estimate the total cardinality $\bar{J}((tp_i \bowtie tp_j), tp_k) \approx |(tp_i \bowtie tp_j) \bowtie tp_k|$. Then, given such estimates, an optimal query plan minimizes the total number of operations to compute, i.e., the execution costs $Cost(T, O)$ of the order O for the set T. In practice, this total join cost is obtained by summing up the intermediate cardinalities of each join operation in their respective join order. Hence, we formalize the problem of join order optimization as follows:

Problem 2 (Join Order Optimization). *Given a set of triple patterns $T=\{tp_1, tp_2, ..., tp_n\}$ and a join cardinality estimation function $\bar{J}$, find the join order O obtained as* $\arg\min_O Cost(T, O)$.

5 EXTENDING SHACL WITH STATISTICS

To compute more accurate join cardinality estimations (Problem 1), we capture the correlations between RDF triples by extending SHACL's node and property shapes with fine-grained statistics of the RDF graph. We denote these statistics as *shapes statistics*. These include the total triple count (*sh:count*), minimum (*sh:minCount*) and maximum (*sh:maxCount*) number of triples for each instance, and the number of distinct objects for property instantiations (*sh:distinctCount*). The attributes shown in the dark shaded boxes in Figure 3 are the annotated statistical attributes of their respective node and property shapes. These statistics are computed by executing analytical SPARQL queries over the RDF graph. For instance, to compute the number of instances of *GraduateStudent* in the dataset, i.e, the value of attribute *sh:count* of node shape *GraduateStudent*, the annotator issues the SPARQL query: `SELECT COUNT(*) WHERE {?x a ub:GraduateStudent}`.

Along with *shapes statistics*, we also define *global statistics* by extending VOID[3] statistics with more precise statistics of RDF properties, i.e., the distinct subject count (DSC) and distinct object count (DOC) of each property of the RDF graph.

6 QUERY PLANNING

In this section, we present our approach to exploit global and shapes statistics to obtain more accurate join cardinality estimates (Problem 1). These estimates, in turn, are used for join order optimization (Problem 2).

6.1 Cardinality Estimation of Triple Patterns

A SPARQL query contains joins between multiple triple patterns. Hence, the first step is to estimate how many triples match every triple pattern individually. The quality of this estimation depends on the granularity of the available statistics. We exploit the statistical information contained in the extended SHACL shapes graph (Section 5) to obtain this estimate. Hence, for each triple pattern, we obtain their corresponding node or property shapes using the values of the *sh:targetClass* and *sh:path* attributes.

First, all triples of the type *<?x, a, [Class]>* (i.e., instances with *rdf:type [Class]*) are mapped to the node shape having that class as the value of the attribute *sh:targetClass*. Then, triples having variable *?x* as a subject are also assigned to that node shape. The triple predicate determines instead its corresponding candidate property shapes, i.e., those with a matching value for *sh:path*. For example, given triples tp_1=*<?x, rdf:type, ub:GraduateStudent>* and tp_2=*<?x, ub:name, ?n>*, the subject *?x* is assigned to node *shape:GraduateStudent*, while the predicate in tp_2 matches *shape:name* (Figure 3, top left and top right).

Once the candidate shapes for all the triple patterns are identified, their statistical information combined with the distinct subject and object count (DSC & DOC) from the *global statistics* are used in combination with the formulas shown in Table 1 to compute their expected cardinality. These formulas, inspired by a previous work [11], cover all possible types of triple patterns. The term c_X in the formulas denotes the count of X in the RDF graph; $c_{triples}$ denotes the count of all triples and $c_{objects}$ the count of all objects. Similarly, c_{X_Y} represents the count of X having Y. This can be used, for instance, to derive that there are ~ 85K triples matching *<?x, rdf:type, ub:FullProfessor>* (Table 2a). While both global and shapes statistics can be used to estimate the cardinality of triple patterns using these formulas, they can

[3] Vocabulary of Interlinked Datasets: https://www.w3.org/TR/void/

lead to different estimated cardinalities. When the query does not contain any type-defined triple, only global statistics are used.

Triple Pattern	Cardinality	Triple Pattern	Cardinality
?s ?p obj	$\frac{c_{triples}}{c_{objects}}$	?s ?p ?o	$c_{triples}$
subj ?p obj	$\frac{c_{triples}}{c_{distSubj} \times c_{distObj}}$	subj ?p ?o	$\frac{c_{triples}}{c_{distSubj}}$
?s pred obj	$\frac{c_{pred}}{c_{pred_{obj}}}$	?s pred ?o	c_{pred}
subj pred obj	$\frac{c_{pred}}{c_{distSubj} \times c_{distObj}}$	sub pred ?o	$\frac{c_{pred}}{c_{pred_{sub}}}$
?s rdf:type obj	$c_{entities_{rdf:type_{obj}}}$	?s rdf:type ?o	$c_{rdf:type}$
subj rdf:type obj	1 *or* 0	subj rdf:type ?o	$\frac{c_{rdf:type}}{c_{rdf:type_{sub}}}$

Table 1: Cardinality estimation of triple patterns

6.2 Cardinality Estimation of Joins

The join operation is performed on a common variable between two triple patterns. We consider three possible types of joins between two triple patterns based on the position of the common variable, namely: Subject-Subject (SS), Subject-Object (SO), and Object-Object (OO). If there is no common variable between two triple patterns, the join will result in a Cartesian product. Inspired by related work [8], we estimate the SS, SO, and OO join cardinalities using the formulas stated in Equations 1, 2, and 3. Note that DSC_i and DOC_i in the formulas represent the distinct subject and object count of triple pattern i respectively.

$$\widehat{card}\ (tp_i \bowtie_{SS} tp_j) = \frac{card_i \times card_j}{max\ (\ DSC_i\ ,\ DSC_j)} \quad (1)$$

$$\widehat{card}\ (tp_i \bowtie_{SO} tp_j) = \frac{card_i \times card_j}{max\ (\ DSC_i\ ,\ DOC_j)} \quad (2)$$

$$\widehat{card}\ (tp_i \bowtie_{OO} tp_j) = \frac{card_i \times card_j}{max\ (\ DOC_i\ ,\ DOC_j)} \quad (3)$$

6.3 Join Ordering

Given an RDF graph G, its shapes statistics graph (G_{sh}), and global statistics graph (G_{gs}), we propose an algorithm to compute the join ordering for an input query Q (Algorithm 1). In the first step, the triple patterns of Q are sorted in ascending order of their estimated cardinalities using only global statistics. The algorithm starts with the triple pattern having the least cardinality and then estimates its join cardinality with the rest of the triple patterns using the formulas from Section 6.2. The algorithm iterates over all the triple patterns and chooses a triple pattern with the least estimated join cardinality (size of intermediate result) given the triple already selected. This produces a first join ordering based on global statistics. In the second step, shapes statistics are taken into account, and both the estimated cardinalities and the join ordering proposed in the first step are revised using these shapes specific fine-grained statistics. The algorithm also computes the cost of each join ordering by adding the estimated join cardinalities in each iteration.

Given our example query Q, and the cardinalities of its triple patterns T={$tp_1, tp_2, ..., tp_9$} estimated with both global and shape statistics, Tables 2a and 2b show the join ordering using only global statistics (O_{gs}) and the ordering computed via shapes statistics (O_{ss}), respectively. There is a significant difference between the estimated and true join cardinalities and their final total cost. The estimated join cardinalities for O_{ss} are much closer to the true cardinalities of the query than the estimates for O_{gs}.

Algorithm 1 Join Ordering

Input: Q, G, G_{sh}, G_{gs}
Output: Join order O of Q

$p \leftarrow [\,]$; $r \leftarrow [\,]$; ▷ p: processed, r: remaining
$cost \leftarrow 0$; $card \leftarrow 0$; $queue \leftarrow queue.init()$;
$tps \leftarrow getTPs(Q)$;
$tps_{\triangle} \leftarrow getCandidateShapes(Q, G, G_{sh})$;
$tps' \leftarrow computeCardinalities(G_{gs}$ or $tps_{\triangle})$; ▷ Table 1
$sort(asc, tps'.cardinality)$;
$p.add(tps'_0)$;
$r.addAll(tps' - tps'_0)$;
$cost = tps'_0.cardinality$;
$queue.add(tps'_0.index)$;
for $tp_i \in tps'$ **do** ▷ i > 0
  $index = tp_i.index$; $cost_{local} = cost$;
  $queue' = queue$;
  while $!queue'.isEmpty$ **do**
    $tp_a = queue'.poll()$;
    for $tp_b \in r$ **do**
      $c = 0$;
      if $tp_a \bowtie_T tp_b$ **then** ▷ $T \in \{SS, SO, OS, OO\}$
        $c = \bar{J}(tp_a, tp_b)$; ▷ $\bar{J} : T \times T \mapsto \mathbb{N}$ (Prob 1)
      else $c = cp(tp_a, tp_b)$; ▷ Cartesian Product
      if $c < cost_{local}$ **then**
        $cost_{local} = c$; $index = tp_b.index$; $card = c$;
    $cost$+=$cost_{local}$;
  $queue.add(index)$; $p.add(tps'.get(index))$;
  $r.remove(tps'.get(index))$;
$O \leftarrow queue.poll()$;

7 EXPERIMENTAL EVALUATION

We investigated the performance of query plans proposed using our algorithm (with global and shapes statistics) compared to the plans proposed by two state-of-the-art query engines (Apache Jena ARQ[4] and GraphDB[5]) as well as two state-of-the-art RDF cardinality estimation approaches (Characteristic Sets [19] and SumRDF [23]). All experiments are performed on a single machine with Ubuntu 18.04, having 16 cores and 256GB RAM.

Datasets: We used LUBM [10], WatDiv [2], and YAGO-4 [21] to study various query plans on different datasets and sizes (Table 3). In particular, we used LUBM-500, two variants of WatDiv datasets (WATDIV-S (Small) with ~108.9 M triples and WATDIV-L (Large) with 1 billion triples), and for YAGO-4 we used the subset containing instances that have an English Wikipedia article.

Implementation: To generate SHACL shapes graphs for the synthetic datasets we used the SHACLGEN[6] library. All shapes are then extended with the required statistics using our *Shapes Annotator* (implemented in Java). The SHACL shapes graph for LUBM, for instance, is 45 KB, and the size of extended shapes is 68 KB. The time required to extend the SHACL shapes depends on the number of its nodes and property shapes. The process of extending LUBM shapes graph took 16 minutes, WATDIV-S took 8 minutes, and for YAGO-4 (which consists of 8888 nodes and 80831 property shapes) it took 62 minutes. We implemented our join ordering algorithm in Java using Jena v.3.15.0. The source code and all the related resources are available on GitHub[7].

We loaded all three datasets and their relevant SHACL shapes graphs into Jena TDBs[8]. We used our join ordering algorithm to construct query plans using global and shapes statistics. For Jena, we used its ARQ query engine to obtain the query plans. For GraphDB, we loaded all datasets in GraphDB and used its *onto:explain* feature to obtain the query plans. For the Characteristic Sets [19] approach, we generated characteristic sets of each dataset and used Extended Characteristic Sets [18] (also known as Characteristic Pairs) to optimize query plans for non-star type

[4]https://jena.apache.org/documentation/query/index.html
[5]https://graphdb.ontotext.com
[6]https://pypi.org/project/shaclgen/
[7]https://kworkr.github.io/repo/
[8]https://jena.apache.org/documentation/tdb/

	Triple Pattern (TP)	DSC	DOC	E_{TP} Card	$E_{\bowtie}$ Card	$T_{\bowtie}$ Card
1:	?A *a* ub:FullProfessor	85, 006	85, 006	85, 006		
2:	?A *ub:name* ?N	10, 696, 541	1, 480	10, 696, 541	85, 006	85, 006
3:	?A *ub:teacherOf* ?C	359, 795	1, 079, 580	1, 079, 580	8, 579	255, 148
4:	?C *ub:advisor* ?A	2, 052, 228	299, 177	2, 052, 228	1, 646	2, 055, 430
5:	?X *a* ub:GraduateCourse	539, 467	539, 467	539, 467	822	1, 027, 909
6:	?X *a* ub:GraduateStudent	1, 259, 681	1, 259, 681	1, 259, 681	504	630, 419
7:	?X *ub:degreeFrom* ?U	1, 619, 476	1, 000	2, 337, 985	575	630, 419
8:	?Y *ub:takesCourse* ?C	5, 220, 814	1, 074, 409	14, 405, 077	7, 674	2, 964, 894
9:	?Y *a* ub:GraduateStudent	1, 259, 681	1, 259, 681	1, 259, 681	1, 851	2, 964, 894
					$\sum$ =106, 657	$\sum$ =10, 614, 119

(a) Join ordering using Global Statistics (O_{gs})

	Triple Pattern (TP)	DSC	DOC	E_{TP} Card	$E_{\bowtie}$ Card	$T_{\bowtie}$ Card
1:	?A *a* ub:FullProfessor	85, 006	85, 006	85, 006		
2:	?A *ub:name* ?N	85, 006	10	85, 006	85, 006	85, 006
3:	?A *ub:teacherOf* ?C	85, 006	255148	255, 148	85, 006	255, 148
4:	?C *a ub*:GraduateCourse	539, 467	539, 467	539, 467	255, 148	255, 148
5:	?X *ub:advisor* ?A	2, 052, 228	299, 177	2, 052, 228	1, 750, 207	127, 523
6:	?X *a* ub:GraduateStudent	1, 259, 681	1, 259, 681	1, 259, 681	1, 074, 297	1, 027, 909
7:	?X *ub:degreeFrom* ?U	1, 259, 681	1, 000	1, 259, 681	659, 416	630, 419
8:	?Y *ub:takesCourse* ?C	5, 220, 814	1, 074, 409	5, 220, 814	8, 841, 082	2, 964, 894
9:	?Y *a* ub:GraduateStudent	1, 259, 681	1, 259, 681	1, 259, 681	2, 133, 181	2, 964, 894
					$\sum$ =14, 883, 343	$\sum$ =8, 310, 941

(b) Join ordering using Shapes Statistics (O_{ss})

Table 2: This table shows the statistics (distinct subject count (DSC) and distinct object count (DOC)) of each triple pattern, the estimated cardinality of each triple pattern (E_{TP}), the estimated join cardinality ($E_{\bowtie}$ Card) and the true join cardinality ($T_{\bowtie}$ Card) for the ordered triple patterns of example query Q .

	LUBM	WATDIV-S	WATDIV-L	YAGO-4
# of triples	91 M	108 M	1,092 M	210 M
# of distinct objects	12 M	9 M	92 M	126 M
# of distinct subjects	10 M	5 M	52 M	5 M
# of distinct RDF type triples	1 M	25 M	13 M	17 M
# of distinct RDF type objects	39	46	39	8,912

Table 3: Size and characteristic of the datasets

queries. Generating Characteristic Sets for large RDF graphs is computationally expensive. For instance, it took 6.2 hours to generate Characteristic Sets for LUBM, 1.2 hours for WATDIV-S, and 8.2 hours for YAGO-4.

For SumRDF [23], we generated the summaries of each dataset and adapted our join ordering algorithm to exploit their estimates. Similar to Characteristic Sets, the generated summaries require a few GBs of memory and their generation time depends on the size and heterogeneity of the dataset, e.g., it took 4.5 minutes to generate the summary for the LUBM, 14 minutes for WATDIV-S, and 4.3 hours for YAGO-4. We use the same size of LUBM and WatDiv datasets as used in SumRDF [23]. Hence, we used the same parameters to generate their summaries. It is suggested that a reasonable default size for the target SumRDF's summary should be in the order of tens of thousands [23]. Therefore, for YAGO-4, to generate the summary in a reasonable amount of time, we chose 100K as the target size of the summary.

All query plans obtained using these approaches are executed 10x in Jena TDB and each query is interrupted after a timeout of 10 minutes. Since for some approaches the order in which triples are stated in the query matters we shuffle the triple patterns in the BGPs randomly in each iteration before proceeding with query optimization. As the query planning time is always less than 20 milliseconds for all approaches and queries, in the following we focus on analyzing the precision of the cardinality estimation and the resulting query performance.

Queries: We distinguish complex (C), snowflake (F), and star (S) queries. LUBM provides 14 default queries that have relatively simple structures. Therefore, we selected queries Q2, Q4, Q8, Q9, Q12 and then created a few additional queries for each category C, F, and S. The WatDiv benchmark includes 3 C, 7 S, and 5 F queries. For YAGO-4 there are no available standard queries or query logs available for benchmarking. Therefore, we have handcrafted 13 queries following the C, F, and S graph patterns from the WatDiv Benchmark. These queries are available on GitHub[9].

Query Runtime: Due to space constraints, we only report our findings on LUBM and YAGO-4, results on WatDiv datasets are discussed in the appendix of the extended version of this document[10]. These experiments offer analogous insights to those obtained from the other datasets. Figure 4a shows the query runtime analysis for query plans proposed using the SS approach *(plans constructed by our join ordering algorithm using shapes statistics)*, GS approach *(plans constructed by our join ordering algorithm using global statistics)*, Jena, GraphDB (GDB), Characteristic Sets (CS), and SumRDF on LUBM queries. The query runtime shows that: (i) the plans proposed by the SS approach are more efficient than those obtained with GS for queries having at least one type-defined triple pattern, (ii) the plans proposed by the GS approach are competitive in comparison to the plans of GDB, CS, and SumRDF, (iii) the CS approach is not well suited for large snowflake queries (e.g., F1, F2, & F5), and (iv) the plans proposed by Jena are often suboptimal and non-deterministic (shown in the size of the error bars) as it is based on a heuristics-based query optimizer that takes into account the given order of triple patterns in the input query.

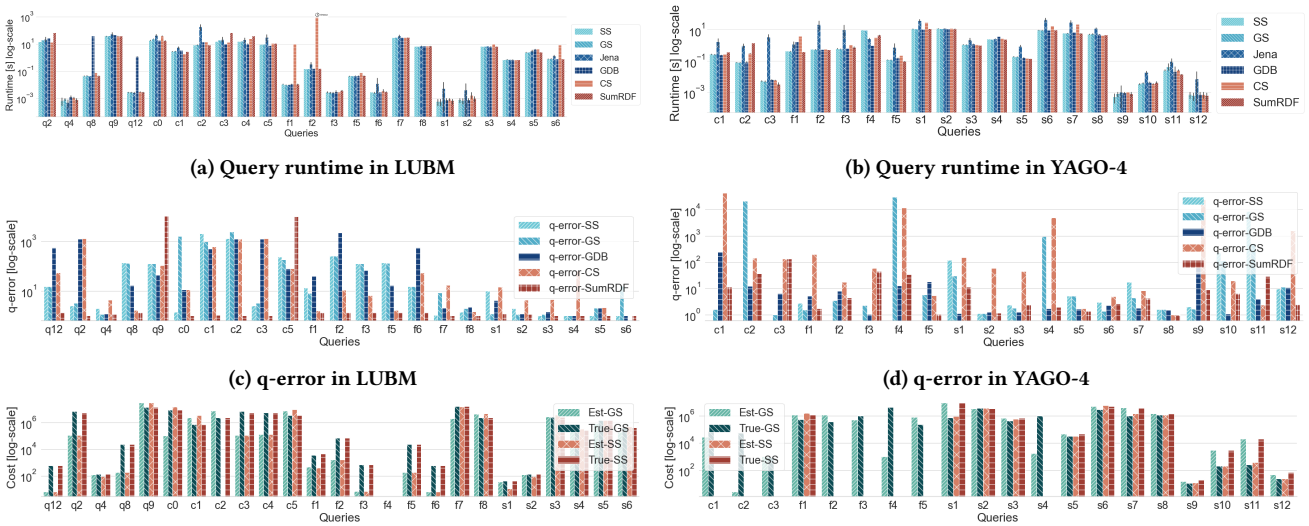
Similarly, Figure 4b shows the query runtime for queries on YAGO-4. The query runtime for complex queries (C1, C2, C3) using SS and GS are competitive to the plans proposed by GDB, CS, and SumRDF. Snowflake queries provide interesting insights where each approach behaves differently for every single query. For instance, CS could not find the optimal query plan for queries F1, F3, F4, F5, and SS and GS could not find the most efficient query plan for query F4 due to underestimation of the join cardinalities. However, GDB and SumRDF found almost optimal query plans for all snowflake queries except F1 (GraphDB) and F4 (SumRDF). For star queries, almost all approaches identify plans with comparable good performances. Similar to LUBM, the plans proposed by Jena are rarely the most efficient.

In contrast to query runtime, we also report the q-error, which is used to measure the precision of the final query result cardinality estimates [19]. It quantifies the ratio between the estimated ($\hat{c}$) and true result cardinality (c) and is computed as the ratio $max(max(1, c)/max(1, \hat{c}), max(1, \hat{c})/max(1, c))$. We analyze the q-error values for SS, GS, GDB, CS, and SumRDF. Figure 4c shows the q-error analysis for LUBM queries. For SS, 15 queries have q-errors lower than 15, 8 queries have q-errors lowever than 250, and only 3 queries have q-errors greater than 250. For GS, 14 queries have q-errors lowever than 15, 8 queries have q-errors lower than 250, and only 4 queries have q-errors greater than 250. Overall, the q-errors for GS and SS are competitive to GDB and CS with few exceptions. However, overall the q-error is very low for SumRDF except queries Q9 and C5. Figure 4d shows the q-error analysis for YAGO-4. For GS and SS, 14 queries have q-errors lower than 15, 2 queries have q-errors lowever than 250, and only 4 queries have q-errors greater than 250. Similar to LUBM, the q-errors of GS and SS are competitive with GDB, CS, and SumRDF with few exceptions.

Finally, Figure 4e and 4f present the analysis between actual and true costs of query plans produced by SS and GS on the LUBM and YAGO-4 datasets. For LUBM, the cost estimated by SS is closer to the actual cost for Q4, Q9, C0, C1, C5, F7, F8, and all

[9]https://kworkr.github.io/repo/

[10]https://github.com/kworkr/repo/blob/master/fullVersion.pdf



(a) Query runtime in LUBM

(b) Query runtime in YAGO-4

(c) q-error in LUBM

(d) q-error in YAGO-4

(e) Cost in LUBM

(f) Cost in YAGO-4

Figure 4: Query runtime, q-error, and cost analysis on LUBM and YAGO-4

star queries. However, for YAGO-4, the cost estimated by SS is closer to the true cost for almost all queries except C2, F4, and S4.

Summary: Our results showed that, with only a few exceptions, the query plans proposed using SS and GS are competitive with the other tested approaches on both the synthetic and real data. Overall, the results revealed that our approach is efficient for all examined types of SPARQL queries while requiring only very little overhead to extend SHACL graphs with statistics, which is more efficient than generating extensive summaries or Characteristic Sets. On average, our approach finds the best query plans for 75% cases on both datasets. For the remaining cases, our approach proposes query plans having an overhead from 14% to 30% on average query runtime w.r.t. the best query plan. Our approach requires 2-4x less preprocessing time, this implies 2 to 6 hours less preprocessing time in our experiments, and 2 orders of magnitude less space.

8 CONCLUSION AND FUTURE WORK

In this paper, we have presented an alternative approach to cardinality estimation for SPARQL query optimization. In particular, we have proposed novel light-weight statistics to capture the correlation in RDF graphs, a cardinality estimation approach, and a join ordering algorithm. We have performed extensive experiments on synthetic and real data to show our approach's effectiveness against two SPARQL query engines and two state-of-the-art RDF cardinality estimators. The results revealed that our approach is efficient in terms of both the preprocessing steps to generate statistics and the cardinality estimation to optimize query plans. Going forward, we plan to integrate our approach with one of the state-of-the-art query engines and enable the support of additional SPARQL query operators.

REFERENCES

[1] Abdullah Abbas, Pierre Genevès, Cécile Roisin, and Nabil Layaïda. 2018. Selectivity Estimation for SPARQL Triple Patterns with Shape Expressions. In *ICWE*. Springer, 195–209.

[2] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management.. In *ISWC*. 197–212.

[3] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. 2011. An Empirical Study of Real-World SPARQL Queries. *CoRR* abs/1103.5043 (2011).

[4] Iovka Boneva, José Emilio Labra Gayo, Samuel Hym, Eric G. Prud'hommeaux, Harold R. Solbrig, and Slawek Staworko. 2014. Validating RDF with Shape Expressions. *CoRR* abs/1404.1270 (2014).

[5] Jeen Broekstra, Arjohn Kampman, and Frank V. Harmelen. 2002. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC*. Springer, 54–68.

[6] WWW Consortium et al. 2013. SPARQL 1.1 overview.

[7] WWW Consortium et al. 2014. RDF 1.1 concepts and abstract syntax. (2014).

[8] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. 2000. *Database system implementation*. Vol. 672. Prentice Hall Upper Saddle River, NJ:.

[9] Andrey Gubichev and Thomas Neumann. 2014. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*. 439–450.

[10] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics* 3 (2005), 158–182.

[11] Stefan Hagedorn, Katja Hose, Kai-Uwe Sattler, and Jürgen Umbrich. 2014. Resource Planning for SPARQL Query Execution on Data Sharing Platforms. In *International Workshop (COLD) co-located with the 13th ISWC)*, Vol. 1264.

[12] Aidan Hogan. 2020. Shape Constraints and Expressions. In *The Web of Data*. Springer, Cham, 449–513.

[13] Hai Huang and Chengfei Liu. 2011. Estimating selectivity for joined RDF triple patterns. In *CIKM*. ACM, Glasgow, United Kingdom, 1435–1444.

[14] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes constraint language (SHACL). *W3C Candidate Recommendation* 11, 8 (2017).

[15] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[16] Brian McBride. 2002. Jena: A semantic web toolkit. *IEEE* 6, 6 (2002), 55–59.

[17] Deborah L McGuinness, Frank Van Harmelen, et al. 2004. OWL web ontology language overview. *W3C recommendation* (2004).

[18] Marios Meimaris, George Papastefanatos, Nikos Mamoulis, and Ioannis Anagnostopoulos. 2017. Extended characteristic sets: graph indexing for SPARQL query optimization. In *ICDE*. IEEE, 497–508.

[19] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*. IEEE Computer Society, Hannover, Germany, 984–994.

[20] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *ACM SIGMOD*. 1099–1114.

[21] Thomas Pellissier Tanon, Gerhard Weikum, Fabian Suchanek, et al. 2020. Yago 4: A reason-able knowledge base. In *ESWC*. 583–596.

[22] Satya Sanket Sahoo. 2010. Semantic Provenance: Modeling, Querying, and Application in Scientific Discovery. (2010).

[23] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *WWW*. ACM, Lyon, France, 1043–1052.

[24] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*. 595–604.