An Introspective Study of Cardinality Estimation in SPARQL

Anonymous Author(s)

ABSTRACT

Cardinality estimates are essential for finding a good join order to improve query performance. Accurate and fine-grained statistics about the dataset are crucial to compute reliable cardinality estimates. It is challenging to collect fine-grained statistics in Knowledge Graphs due to their heterogeneity. This leads to the use of statistics collected globally for each property without considering their distribution among classes. In general, such global statistics are not enough to compute accurate cardinality estimates during query planning. We propose fine-grained statistics called shapes statistics which consider the correlation between classes and properties. We study how global and shapes statistics impact the quality of cardinality estimates and ultimately the query planning in SPARQL. We use SHACL constraints to store shapes statistics. In order to exploit these statistics, we propose a join ordering technique capable of handling global and shapes statistics. We implemented it in Apache Jena and evaluated it using diverse query workload on WatDiv, LUBM, and YAGO-4 datasets against Jena, GraphDB and Characteristics Sets approach. Results show that in general, our approach outperforms the standard query optimization approach of Apache Jena and Characteristics Sets approach. However, overall the impact of having shapes vs global statistics in query planning is noticeable in cardinality estimation, but not significantly on query plans in terms of query runtime.

1 INTRODUCTION

The Resource Description Framework (RDF) [13] is a versatile data model used to represent data from diverse domains and share it on the Semantic Web [5]. Recently, growing amounts of data got published in RDF format, such as Linked Open Government Data [20, 35], Open Street Map Data [19], biological data [34], and other open-domain knowledge graphs such as DBpedia [6] and YAGO-4 [38]. Given the widespread use of RDF data, triplestores have attracted an increasing attention both from research and industry [41]. At the same time, as more and more data is stored within triplestores, these systems are used to execute increasingly complex queries to answer important business and research questions. For instance, query logs of the DBpedia endpoint[1] contain SPARQL queries with up to 10 joins [3] and analytic queries in the biomedical field can involve more than 50 joins per query [34]. Therefore, the need for high-performance SPARQL query processing is now more pressing than ever. An RDF dataset is a graph stored as collection of triples denoted as SPO (*subject, predicate, object*) usually stored in a triplestore and queried by its standard query language SPARQL [11]. A SPARQL query produces answers by describing a set of triple and how they should be joined together to match desired portions of the RDF graph.

In order to cope with the rapid growth of RDF datasets, a substantial amount of research has been performed to improve storage layouts and query processing of RDF triplestores. Different approaches for query optimization in triplestores adapt techniques from the relational world assuming that the triples are stored in a large single table with three columns (corresponding to the SPO positions) [9, 28]. Such query optimization strategies rely on global statistics such as the frequencies of subjects, objects, and predicates, to estimate the sizes of final and intermediate results produced when evaluating the query. These statistics result in highly imprecise estimations, since they are mostly gathered on the whole RDF graph, contrary to the relational models where it is possible to measure these statistics with higher precision since data is separated into tables [26]. As RDF triples patterns are highly correlated [30], therefore, collecting statistics of the RDF graph at the predicate level by assuming independence between the triples leads to underestimating join cardinalities.

On the other hands, fine-grained statistics capable to capture the correlation among RDF triple are promising to estimate accurate join cardinalities. We exploit the widespread use of Shapes Constraint Language (SHACL [24]) to collect such fine-grained statistics over RDF graphs. We name these fine-grained statistics as Shapes Statistics and the statistics collected globally without considering the correlations among RDF triples as Global Statistics. SHACL is a recent W3C recommendation which is considered as one of the most promising validating schemata for RDF graphs [12]. SHACL shapes are formulated in RDF and describe the relationships between entities of specific classes, their properties, and their connections to other entities. Currently it is only used for validation purposes. While previous work proposes to analyze other constraint languages (i.e., Shape Expression [8]) to inform the query planner [1], that approach does not produce any join cardinality estimate. Some approaches using heuristics [40] are not sufficient to accurately estimate the cardinalities of complex queries involving multiple joins. This results in sub-optimal query planning when evaluating complex queries.

To this end, we study the impact of Global and Shapes Statistics on cardinality estimation by proposing a join ordering technique capable to handle both types of statistics while query planning. In this experimental study, at first, we demonstrate how SHACL definitions can be extended to capture the statistical correlations between RDF triples and replace the need for constructing complex (and expensive) synopses over RDF graphs for cardinality estimation. Then we explain our approach of exploiting such statistical information to propose join ordering for query planning, and lastly, we study the impact of Global and Shapes Statistics on query runtime by experimenting on LUBM [17], WatDiv [2], and YAGO-4 [38] datasets. The results show that having Shapes Statistics at the granular level in RDF graphs are beneficial for query cardinality and cost estimation. However, their impact on query runtime is not noticeable compared to the join ordering proposed using Global Statistics except in few cases.

2 RELATED WORK

Cardinality estimation has been studied extensively in the context of relational databases [32]. The existing techniques of cardinality estimation for SPARQL queries adapt from relational

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

approaches [22, 31, 37] and focus mostly on star queries [30]. Usually, these approaches construct different kinds of single [23, 27] or multidimensional [16] synopses over databases that can be used to estimate cardinalities. Instead, since RDF graphs have labelled edges, it becomes non-trivial to generate RDF synopsis using algorithms for unlabelled graphs [25, 33]. Other works either generate very large RDF summaries [39], they have very high computational complexities [10], or they are unable to preserve the RDF schema while constructing RDF summaries [36].

Recently a benchmark called G-CARE [32] was proposed to evaluate performance of cardinality estimation techniques for specific graph-pattern queries. G-CARE requires the input RDF graph to be converted into a directed labeled graph and input queries into query graphs, it is not designed for straight RDF and SPARQL queries as it also supports relational data. The RDF-3X [31] system proposes to estimate cardinalities by constructing histograms. This system was later extended by exploiting the statistical information of CharacteristicSets [30] which characterizes subjects as a set of properties to estimate the cardinalities. CharacteristicSets approach shows high performance only for star-shaped queries and it suffers from significant underestimation due to the independence assumption [32]. Query Optimization for Large Scale Clustered RDF Data [42] is an approach that uses the concept of characteristic sets [30] to define a novel statistics collected for clusters of triples to better capture the dependencies in the RDF graph. This approach redefines an execution plan based on the logical structures and an algorithm based on a customized cost model used for selecting the optimal execution plan. SumRDF [36] is another cardinality estimation approach based on a graph summarization. It fails to handle large queries due to prohibitive computation cost and also it is costly to construct such summaries over large RDF graphs [32]. Another line of work is where Shape Expressions (ShEx[2]) have been used to compute ranks of SPARQL query triple patterns [1]. These ranks are used to estimate an order of execution of query. However, their method determines the order of triple patterns based uniquely on the constraint relation between shape definition, hence they do not estimate any cardinality. For instance, if a shape definition says that every student has exactly one university they infer that the cardinality of students is at most the same cardinality of university and probably larger. Therefore their optimization procedure is only based on this kind of inference and is not based on the actual data. Moreover, in the case when some triples are not described by a shape, the query will not be optimized.

On the contrary, we present an introspective study to analyze the impact of Global and Shapes Statistics on cardinality estimation and join ordering. To this end, instead of creating large expensive summaries and characteristics sets over the RDF graphs to estimate the cardinalities, we exploit SHACL shapes constraints and annotate the Node and Property Shapes with the statistics of the input RDF graph. It requires very less prepossessing and retains the structure of original RDF and SHACL shapes graphs. These Shapes Statistics are used to estimate join cardinalities and compute join ordering. *It also compensate for missing shape definitions* by leveraging a specific extension we call Metadata Shape (Section 4). Moreover, SHACL and ShEx are equivalent [21], our approach hence can be extended to work with other constraints languages as well.

[2]http://shex.io/shex-semantics/

3 PRELIMINARIES

RDF Graph. 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 [17] dataset in Figure 1, where oval and rectangular shapes represent IRI and literal nodes, respectively. And an edge represents a triple <*S, P, O*>. 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 $\langle s, p, o\rangle \in (I \cup B) \times I \times (I \cup B \cup L)$.

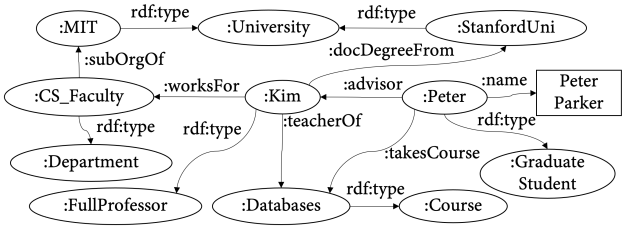


Figure 1: An RDF Graph G

SPARQL. SPARQL [11] is a standard query language for RDF. A SPARQL query consists of a finite set of triple patterns (known as basic graph patterns (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). 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.

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 the BGP. Figure 2 shows a SPARQL query (Q) and its query graph Q_G on LUBM RDF graph (Figure 1). Q asks about graduate students, their advisors who are full professors and teaching graduate courses, the names of the professors, and lastly, the universities they graduated from.

```
SELECT * WHERE {
    ?X ub:degreeFrom     ?U.
    ?X rdf:type  ub:GraduateStudent.
    ?X ub:advisor        ?A .
    ?A rdf:type      ub:FullProfessor.
    ?A ub:teacherOf      ?C.
    ?A ub:name           ?N.
    ?C rdf:type      ub:GraduateCourse.
    ?Y ub:takesCourse    ?C.
    ?Y rdf:type  ub:GraduateStudent.
}
```

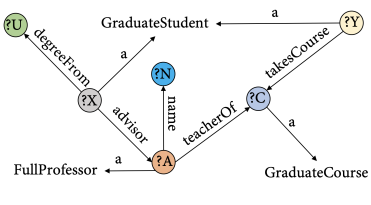


Figure 2: Query Q and its Graph Q_G

Shapes Graph. Several schema languages have been proposed for RDF in the past, where the most common are RDF Schema (RDFS[3]) and OWL[4]. RDFS is primarily used to infer implicit facts and OWL is an extension of RDF and RDFS based

[3]https://www.w3.org/TR/rdf-schema/
[4]https://www.w3.org/TR/owl-features/

on Description Logics [4] to represent ontologies. The declarative Shapes Constraint Language (SHACL)[5] became a W3C standard recently. SHACL schema provides high level information about the structure and contents of an RDF graph [7]. 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 modelled as Node and Property Shapes which consists of certain attributes. Node Shapes constraints are applied on nodes which are instances of a specific type in the *data graph* while the Property Shapes constraints are applied on predicates which are associated to nodes of specific types. For example in Figure 3, Node Shape constraints are applied on node `ub:GraduateStudent` and its Property Shapes constraints are applied on predicates `name`, `takesCourse`, and `advisor`. This information is declared with attributes *sh:targetClass* for Node Shape and *sh:path* for Property Shapes. Note that the attributes in the dark shaded boxes are annotated attributes, and are not the part of core SHACL definition, explained in Section 4. 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 of 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.

Shapes Expression (ShEx[6]) language also serves a similar purpose as SHACL to validate RDF graphs. Nonetheless, the two formulations diverge mostly at the syntactic level [21], and our approach can be extended to work using ShEx or other constraints languages as well without the loss of generality.

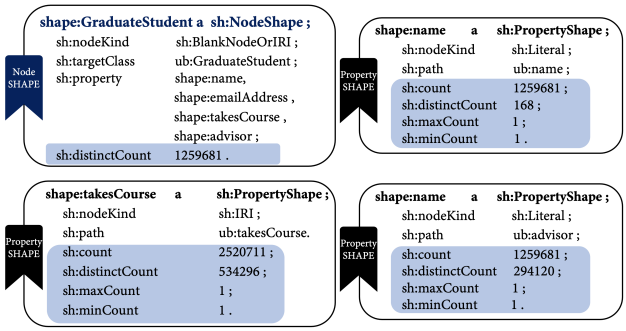


Figure 3: SHACL Shapes Graph

Problem Formulation. Given an input query Q, a query optimizer has the goal to find a query plan expected to return the resultset in the minimum amount of time [26]. A SPARQL query plan is constructed by finding an optimal join ordering between triple patterns of its BGPs. In this paper, we focus on the join ordering of BGPs defined as follows:

Definition 3.4 (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 $\mathcal{B}$ 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. To decide the optimal ordering it is necessary to obtain the size of the resultset of each single join, called join cardinality, i.e., given the number of triples matching tp_1 and the number of triples matching tp_2, compute the number of results of $tp_1 \bowtie tp_2$. Selecting the optimal join ordering ensures that, when executing the query, we compute the minimum amount of operations by producing the smallest amount of intermediate results. Yet, to compute the exact join cardinality most of the time the only solution is to actually execute the operation, which is not a viable option. Therefore, a crucial problem is to accurately estimate the join cardinality between triple patterns of a given query before actually executing it [15]. Moreover, the search space of all possible join orders is often very large. For example, for the input query Q (Figure 2) having nine triple patterns, the size of the whole search space, that is considering all possible orderings of triples, is calculated as 9! = 362880. Therefore, it is quite challenging for query optimizers to explore the huge size of the search space for large SPARQL queries, even by employing dynamic programming. 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 $tp_i, tp_j \in T$, $\bar{J}(tp_i, tp_j) \approx |tp_i \bowtie tp_j|$.*

In line with the related works, we neglect other cost factors and focus on cardinality as the most dominant cost factor to find join ordering. Therefore, with abuse of notation, we extend the above join cardinality estimation problem also to the case of joining a triple pattern with the intermediate results of prior joins operation and so on, 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 an estimate for each join, an optimal query plan minimizes the total amount of comparisons to compute, the $Cost(T, O)$ of the order O for the set T. Given a set of triple patterns T and a join order O the total join cost is obtained by summing the intermediate cardinalities of each join operation in their respective join order. 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* $\arg\min_O Cost(T, O)$.

4 EXTENDING SHACL WITH STATISTICS

We extend Node and Property Shapes of a SHACL shapes graph (G_{sh}) by annotating them with fine-grained statistical information of the RDF graph. We denote these statistics as Shapes Statistics. This includes 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`). For this purpose, we have developed a *Shapes Annotator*[7]. The attributes shown in the dark shaded boxes in SHACL shapes graph of Figure 3 are the annotated statistical attributes of their respective Node and Property Shapes. The *Shapes Annotator* computes these statistics by issuing analytical SPARQL queries over RDF graph. For

[5] https://www.w3.org/TR/shacl/
[6] http://shex.io/shex-semantics/
[7] https://kworkr.github.io/repo/

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}`. Similarly, the *Shapes Annotator* computes the values for other attributes, i.e., the number of distinct/max/min values for the properties *takesCourse*, *name*, and *advisor* of Property Shapes of *GraduateStudent*. These attributes contain values computed over the LUBM dataset. The `sh:count` property in Node Shape *GraduateStudent* shows the cardinality (i.e., ~1.2 million) of `ub:GraduateStudent` class instances in the LUBM dataset.

Additionally, to capture the remaining statistics of the RDF graph such as the total number of triples and distinct subject count (DSC) and distinct object count (DOC) of its properties, we generate Metadata Shape. We also generate the Global Statistics graph (G_{gs}) for the input RDF graph , which serve the same purpose as Vocabulary of Interlinked Datasets (VOID[8]) statistics and include the overall count of classes and properties.

5 CARDINALITY ESTIMATION

The final cardinality of a query is the total number of tuples produced by executing it in any query engine. Join cardinality is the size of intermediate results produced by joining two triple patterns. In order to estimate these cardinalities, the prerequisite is to first compute the selectivity of each triple pattern describing a property.

5.1 Selectivity Estimation of Properties

Given a set of triple patterns T=$\{tp_1, tp_2, ..., tp_n\}$ of a query Q, we decompose each triple pattern tp and find its candidate Node and Property Shapes in the SHACL shapes graph G_{sh}. The subject and object of the tp are evaluated to identify its corresponding candidate Node Shapes. If the subject denote a class or an instance of a class, or the object denote a class, the value of attribute `sh:targetClass` determines its candidate Node Shape. The predicate of a tp is evaluated to determine its corresponding candidate Property Shape using the value of its attribute `sh:path`. For example, given a tp= *<ub:GraduateStudent, ub:name, ?n>*, the subject's IRI correspond to node *shape:GraduateStudent* tracked by the value of its attribute `sh:targetClass`, shown in Figure 3 (top left). Similarly, the predicate's IRI suggests its candidate property *shape:name* tracked by its attribute `sh:path`, shown in Figure 3 (top right).

Once the candidate shapes for T are identified, their statistical information combined with the other required statistics (DSC, DOC) from Metadata Shape are used in our proposed formulas shown in Table 1 to compute the selectivity of triple patterns. These formulas are inspired from [18] and cover all the possible permutations of variables in a triple pattern. The term c_X in the formulas denotes the count of X in the RDF graph such as $c_{triples}$ is count of all the triples and $c_{objects}$ is count of all the objects. Similarly, c_{X_Y} represents the count of X having Y.

Both, the Global and Shapes Statistics can be used to estimate the selectivity of triple patterns using these formulas. However, the use of both statistics result into different values of selectivity with only one exception where the query does not contain any type defined triple. This results into finding only candidate Property Shapes for its triple patterns. In this case, the Shapes Statistics serve the same as the Global Statistics.

[8]https://www.w3.org/TR/void/

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.

5.2 Cardinality Estimation of Joins

The join operation is performed on a common variable or IRI between two triple patterns. There are three possible type of joins in SPARQL queries, namely: Subject-Subject (SS) join in which the triple patterns are joined based on their subjects, Subject-Object (SO) join in which the join is performed on the subject of one triple pattern and the object of other triple pattern, and Object-Object (OO) join where the triple patterns are joined based on their objects. If there is no common variable or IRI between two triple patterns, the join will result in a cartesian product of both triple patterns. We estimate the SS, SO, and OO join cardinalities using the formulas stated in Equation 1, 2, and 3. These formulas are inspired from the corresponding cardinality estimation heuristics for relational databases [14]. Note that DSC_i and DOC_i in the formulas represent the distinct subject and object count of triple i respectively.

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

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

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

Let's take some triple patterns from our example query Q to only extract the graduate students, their advisor who are full professors (along with their names) and their graduation university. At first, the formulas from Table 1 are used to estimate the cardinality of each triple pattern, and then the above equations are used for each pair of join to estimate the join cardinalities between the input triple patterns in the exact same order as stated in Table 2 with their DSC, DOC, estimated Triple Pattern (E_{TP}) cardinality and the estimated join ($E_{\bowtie}$) cardinality.

6 QUERY PLANNING

Here we discuss our join ordering technique which is capable to handle Global and Shapes Statistics to estimate the cardinalities and propose a final join ordering for the query plan of a given query. The cost of a query plan is computed by the sum of all the cardinalities (intermediate results) produced during its execution. The lower the size of the intermediate results, the

	Triple Pattern (TP)	DSC	DOC	E_{TP} Cardinality	$E_{\bowtie}$ Cardinality
1:	**?A** *a* ub:FullProfessor	85,006	85,006	85,006	
2:	**?A** *ub:name* ?N	85,006	10	85,006	85,006
3:	**?X** *ub:advisor* **?A**	2,052,228	299,177	2,052,228	583,105
4:	**?X** *a* ub:GraduateStudent	1,259,681	1,259,681	1,259,681	357,916
5:	**?X** *ub:degreeFrom* ?U	1,259,681	1,000	1,259,681	219,693

Table 2: Cardinality estimation of Joins

faster is the execution time of the query. A query plan having the most optimal join ordering is expected to have the minimum cost. Based on our formulas to calculate the estimated join cardinality of triple patterns (in Section 5.2), the cost of query plan is given by $\sum_1^n f(x) \mid n >= 2$ where n is the number of triple patterns in the input query and $f(x)$ returns the estimated cardinality using Equations 1, 2, or 3.

Join Ordering is a crucial issue in SPARQL query optimization due to high number of triple patterns (tps) and join operations in BGPs. We propose Algorithm 1 to compute the join ordering for an input query Q on RDF graph G having Shapes Statistics in graph (G_{sh}) and Global Statistics in graph (G_{gs}). We observed that the join ordering computed solely using Shapes Statistics is not always proved to be optimal due to high degree of correlation in RDF graphs. We propose an optimal approach where at first we use the cardinalities calculated via Global Statistics to compute an initial join ordering using *computeJoinOrdering* method of Algorithm 1. If Q contains at least one type defined triple, we update the cardinalities of tps using Shapes Statistics (line 3-4) and recompute the join ordering using *computeJoinOrdering* method with both, the global and shapes specific statistics for the given tps of Q.

In *computeJoinOrdering* method, the provided tps of Q with their cardinalities are sorted in an ascending order of their cardinalities (line 6). The *Processed* (p) and *Remaining* (r) lists are maintained to keep track of join ordering. A *queue* is maintained to store the ordered indexes of processed tps. The algorithm starts ordering the tps by selecting a most selective tp (line 10-11) and then iterating over all the remaining triples to estimate the cardinality (line 14 - 20). In case there does not exist any join (SS, SO, OS, OO), it results into a Cartesian product and the algorithm tries to avoid it. A tp with a least expected cost is selected (line 21-22) in each iteration and the overall cost is updated until the *queue* becomes empty. Finally, the *queue*, p, and r lists are updated according to the selected tp index expected to result in a minimum cost and the optimal join order (i.e., having minimum size of intermediate results). The final join ordering O is obtained at the end by polling through the *queue*.

Algorithm 1 Join Ordering

```
   Input: Q, G, G_sh , G_gs
   Output: Join order O of Q
1: p ← [], r ← [], cost ← 0, card ← 0
2: tps ← getTPs(Q)
3: tps_△ ← getCandidateShapes(Q, G, G_sh)
4: tps' ← computeCardinalities(G_gs or tps_△)                ▷ Table 5.1
5: function computeJoinOrdering(tps')
6:     sort(asc, tps'.cardinality)
7:     r.addAll(tps'), queue ← queue.init()
8:     for tp_i ∈ tps' do
       ▷ iteration 0
9:         p.add(tp_i), cost = tp_i.cardinality
10:        queue.add(tp_i.index), r.remove(tp_i)
       ▷ iteration > 0
11:        index = tp_i.index, cost_local = cost
12:        queue' = queue
13:        while !queue'.isEmpty do
14:            tp_a = queue'.poll()
15:            for tp_b ∈ r do
16:                c = 0
17:                if tp_a ⋈_T tp_b then                  ▷ T ∈ {SS, SO, OS, OO}
18:                    c = J̄(tp_a, tp_b)                  ▷ J̄ : T×T ↦ ℕ (Prob 1)
19:                else c = cp(tp_a, tp_b)                ▷ Cartesian Product
20:                if c < cost_local then
21:                    cost_local = c, index = tp_b.index, card = c
22:            cost+=cost_local
23:        queue.add(index) , p.add(tps'.get(index))
24:        r.remove(tps'.get(index))
25:    return O ← queue.poll()
```

We show the join ordering of our example Query Q constructed using Global Statistics in Figure 4 and reordered using Shapes Statistics in Figure 5. Also, we show a comparison between estimated (E) and true (T) join cardinality over each join. The join ordering computed finally using Shapes Statistics (O_{ss}) is more optimal than the one computed solely using Global Statistics (O_{gs}). The estimated final cardinality of O_{ss} (i.e., 2,133,181) is more closer to the actual cardinality (i.e., 2,964,894). On the other hand, the estimated final cardinality of O_{gs} is underestimated up to three orders of magnitude. Similarly, the estimated $cost_E(O_{ss})$ = 106,660 is less than the $cost_E(O_{gs})$ = 8,686,212. The true $cost_T(O_{ss})$ = 10,614,119 is less than the $cost_T(O_{gs})$ = 15,138,493. In terms of query runtime O_{ss} took 14.37 seconds and O_{gs} took 19.97 seconds.

7 EXPERIMENTAL EVALUATION

Here we evaluate the query plans proposed by our join ordering technique using Global and Shapes Statistics against the plans proposed by Apache Jena's ARQ[9] query engine, GraphDB, and CharacteristicSets [30] approach. We used GraphDB's *onto:explain*[10] feature to obtain its query plans. Each query is executed 10x along with shuffling its BGPs randomly in each iteration. All experiments are performed on a single machine with Ubuntu 18.04, having 16 cores, 240GB SSD, and 256GB RAM.

Datasets. We used LUBM [17], WatDiv [2], and YAGO-4 [38] to study various query plans on different datasets. Statistics of these datasets are shown in Table 3. We used two variants of WatDiv datasets, i.e., WATDIV-S (Small) with ~108.9 M triples and WATDIV-L (Large) with 1 billion triples[11]. YAGO-4 is a real English Wikipedia dataset[12] which is a restriction of the Wikipedia flavor to instances that have an English Wikipedia article.

	LUBM	WATDIV-S	WATDIV-L	YAGO-4
# of graphs	1	1	1	1
# 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: Statistics of datasets.

Implementation. We implemented our join ordering technique using Jena v.3.15.0. To generate SHACL shapes graphs, we used Python SHACLGEN[13] library and annotated the generated

[9]https://jena.apache.org/documentation/query/index.html
[10]https://graphdb.ontotext.com/documentation/standard/explain-plan.html
[11]https://hobbitdata.informatik.uni-leipzig.de/intelligent-SPARQL-interface/
[12]https://yago-knowledge.org/downloads/yago-4
[13]https://pypi.org/project/shaclgen/

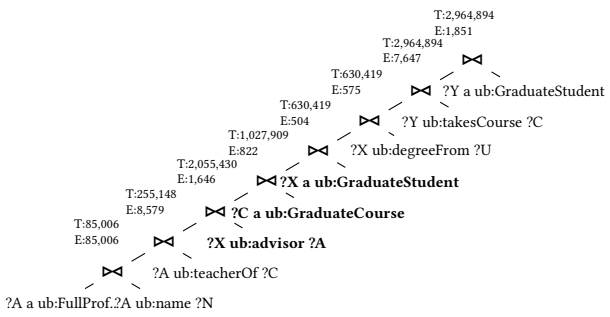


Figure 4: Join ordering (O_{gs}) for Q

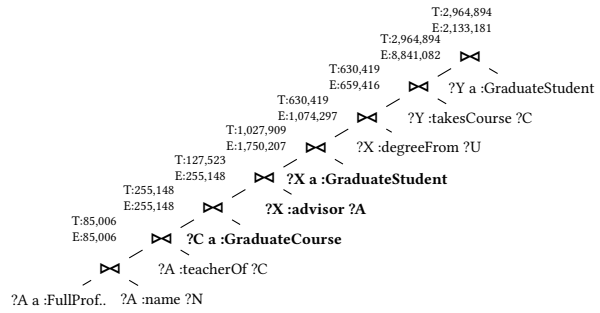


Figure 5: Join ordering (O_{ss}) for Q

SHACL shapes graphs with statistics using our *Shapes Annotator* (implemented in Java). We loaded all the three datasets and their extended SHACL shapes graphs into Jena TDB. For CharacteristicsSets [30] approach, we used ExtendedCharacteristicsSets [29] (also known as Characteristics Paths/Pairs) to find their query plans. The source code and all the related resources are available on GitHub[14].

Queries. We used complex, snowflake, and star type of queries. LUBM provides 14 default queries which are relatively simple, therefore we have chosen some representative queries out of provided 14 queries (i.e., Q2, Q4, Q7, Q8, Q9, Q12) and also handcrafted few additional complex, snowflake, and star queries. The WatDiv benchmark includes 3 complex, 7 star, and 5 snowflake queries. The YAGO-4 is a recently published dataset, it does not have any available standard queries or the query logs. Therefore, we have handcrafted 13 standard queries on YAGO-4 dataset following the complex, snowflake, and star graph patterns of WatDiv Benchmark. We have excluded linear and simple queries having less than two triple patterns for all the datasets. These queries are available in the aforementioned GitHub repository.

Results. Here we evaluate query runtime, q-error, and cost for LUBM and YAGO-4 datasets. The q-error is used to measure the precision of cardinality estimates. The query planning time is negligible as it is never greater than 20 milliseconds for any of our queries. Also, we only discuss the queries having runtime more than one second. Figure 6a shows the runtime in seconds for LUBM queries for query plans proposed by Shapes Statistics (SS), Global Statistics (GS), Jena, GraphDB (GDB), and CharacteristicsSets (CS). In queries Q2, Q7, and Q9, the final plans improvised using SS proved to have better runtime than GS and GDB. The plans by CS approach are competitive to SS for Q7 and Q9 and better than GS and GDB. Jena proposed an optimal query plan for Q7 only as compared to others. For complex queries C1, C2, C4, and C5, the final plans improvised by SS proved to have a slightly better runtime for C2, C4 and C5 than GS. However, GDB proposed more optimal query plans for C4 and C5 as compared to SS and CS. The plans by CS approach are competitive for C1 and C5 only. Jena could not propose a better plan for any of these queries. For snowflake queries F7 and F8, all approaches are competitive to each other in terms of query runtime except Jena and CS. Similarly, for star queries S3, S4, S5, and S6, all approaches are competitive except for CS. Overall, the impact of having fine-grained Shapes Statistics on query runtime is visible in 7 out of 14 various types of queries.

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

Figure 6b shows the runtime in seconds for YAGO-4 queries for query plans proposed by SS, GS, Jena, GDB, and CS. In complex queries C1, C2, and C3, SS could not improvised the plans proposed by GS and overall all the approaches proposed almost the equivalent plans except CS for query C2. For snowflake queries F1, F2, F3, and F4, SS improvised the query plan for query F1, Jena could not propose optimal query plans for F2 and F3 except for F4 where GDB and Jena proposed optimal query plans as compared to SS, GS, and CS. For star queries S1-S8, SS, GS, GDB and CS proposed equivalent query plans for S2, S3, S4, S5, and S8. However, Jena and CS approach could not propose optimal query plans for S1, S6, and S7 as compared to others.

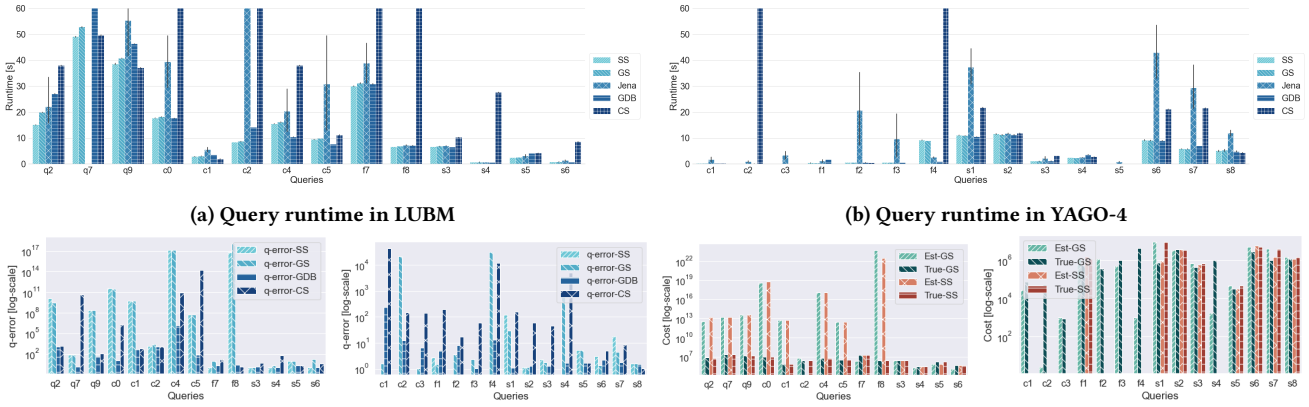
Figure 6c shows the q-error analysis for LUBM queries. The q-error value for GS and SS is relatively higher for Q2, Q9, C1, C4, C5, and F8 as compared to others. However, regardless of high value of q-error, SS and GS plans are efficient in query runtime for these queries. The q-error value for the rest of the queries is competitive to each other and is closer to optimal value. Figure 6d shows the q-error analysis for YAGO-4 queries. For this real dataset, overall GS, SS and GDB plans have relatively low q-error as compared to CS. Specifically, q-error value by SS is optimal for all except queries F4, S1, and S4, and the q-error value by CS is optimal only for S5 and S8. Figure 6e and 6f present the analysis between actual and true costs for query plans produced by SS and GS on LUBM and YAGO-4 datasets, respectively. For LUBM, the cost estimated by SS is more closer to the actual cost for F7, S3, S4, S5, and S6. However, for YAGO-4, the cost estimated by SS is more closer to the true cost for almost all the queries where SS improvised the query plans proposed by GS.

Due to space limitation, the results on WatDiv datasets are discussed and shown in Appendix A of extended version of this paper available here[15].

8 CONCLUSION AND FUTURE WORK

We present an experimental study of cardinality estimation in SPARQL queries on RDF graphs. We collect fine-grained statistics on RDF graphs by extending Shapes Constraint Language (SHACL). We study the impact of having global and fine-grained statistics on cardinality estimation and join ordering in SPARQL queries. We propose a join ordering technique to compute cardinality estimates using various statistics to propose an optimal query plan. We benchmarked our technique using these statistics against the query plans proposed by state-of-the-art query engines (Jena ARQ & GraphDB) and characteristics sets approach on synthetic and real datasets. Our study reveals that having fine-grained statistics over RDF graphs helps to estimate reliable cardinalities but do not significantly impact the quality of query plans (join ordering) except in very few cases. Generally, the query plans proposed by estimating cardinalities using global statistics proved optimal enough to execute the query in an appropriate amount of time. Even the statistics collected using the characteristics sets approach could not propose optimal query plans for all types of queries. We studied that currently, any level of statistics collected over RDF graphs using any state-of-the-art approach is not entirely enough to claim a significant impact on query plans. There is still a need to fill this gap by proposing graph-aware smart statistics.

[15]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 6: Query runtime, q-error, and cost analysis

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*. Springer, 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] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (Eds.). 2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

[5] Tim Berners-Lee, James Hendler, and Ora Lassila. 2001. The semantic web. *Scientific american* 284, 5 (2001), 34–43.

[6] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia-A crystallization point for the Web of Data. *Journal of web semantics* 7, 3 (2009), 154–165.

[7] Iovka Boneva, Jérémie Dusart, Daniel Fernández-Álvarez, and José Emilio Labra Gayo. 2019. Semi Automatic Construction of ShEx and SHACL Schemas. *CoRR* abs/1907.10603 (2019).

[8] 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).

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

[10] Sejla Cebiric, François Goasdoué, and Ioana Manolescu. 2015. Query-Oriented Summarization of RDF Graphs. *Proc. VLDB Endow.* 8, 12 (2015), 2012–2015.

[11] World Wide Web Consortium et al. 2013. SPARQL 1.1 overview.

[12] Julien Corman, Juan L. Reutter, and Ognjen Savkovic. 2018. Semantics and Validation of Recursive SHACL. In *ISWC (1) (Lecture Notes in Computer Science)*, Vol. 11136. Springer, Monterey, CA, USA, 318–336.

[13] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J Carroll, and Brian McBride. 2014. RDF 1.1 concepts and abstract syntax. *W3C recommendation* 25, 02 (2014).

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

[15] Andrey Gubichev and Thomas Neumann. 2014. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*. OpenProceedings.org, Athens, Greece, 439–450.

[16] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. 2005. Selectivity estimators for multidimensional range queries over real attributes. *VLDB J.* 14, 2 (2005), 137–154.

[17] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics* 3, 2-3 (2005), 158–182.

[18] Stefan Hagedorn, Katja Hose, Kai-Uwe Sattler, and Jürgen Umbrich. 2014. Resource Planning for SPARQL Query Execution on Data Sharing Platforms *(CEUR Workshop Proceedings)*, Vol. 1264.

[19] M. Haklay and P. Weber. 2008. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Computing* 7, 4 (2008), 12–18.

[20] J. Hendler, J. Holm, C. Musialek, and G. Thomas. 2012. US Government Linked Open Data: Semantic.data.gov. *IEEE Intelligent Systems* 27, 3 (2012), 25–31.

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

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

[23] Yannis E. Ioannidis. 2003. The History of Histograms (abridged). In *VLDB*. Morgan Kaufmann, Berlin, Germany, 19–30.

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

[25] Kristen LeFevre and Evimaria Terzi. 2010. GraSS: Graph Structure Summarization. In *SDM*. SIAM, Columbus, Ohio, USA, 454–465.

[26] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[27] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 1998. Wavelet-Based Histograms for Selectivity Estimation. In *SIGMOD Conference*. ACM Press, Seattle, Washington, USA, 448–459.

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

[29] Marios Meimaris, George Papastefanatos, Nikos Mamoulis, and Ioannis Anagnostopoulos. 2017. Extended characteristic sets: graph indexing for SPARQL query optimization. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 497–508.

[30] 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.

[31] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment* 1, 1 (2008), 647–659.

[32] 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 *Proceedings of the 2020 ACM SIGMOD*. 1099–1114.

[33] Matteo Riondato, David García-Soriano, and Francesco Bonchi. 2017. Graph summarization with quality guarantees. *Data Min. Knowl. Discov.* 31, 2 (2017), 314–349.

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

[35] N. Shadbolt, K. O'Hara, T. Berners-Lee, N. Gibbins, H. Glaser, W. Hall, and m. c. schraefel. 2012. Linked Open Government Data: Lessons from Data.gov.uk. *IEEE Intelligent Systems* 27, 3 (2012), 16–24.

[36] 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.

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

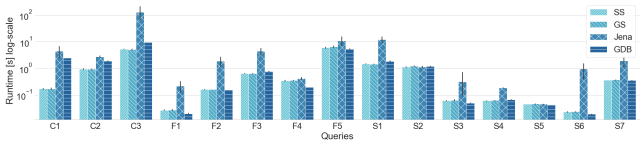
[38] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. 2020. YAGO 4: A Reason-able Knowledge Base. In *ESWC (Lecture Notes in Computer Science)*, Vol. 12123. Springer, Heraklion, Crete, Greece, 583–596.

[39] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. 2008. Efficient aggregation for graph summarization. In *SIGMOD Conference*. ACM, Vancouver, BC, Canada, 567–580.

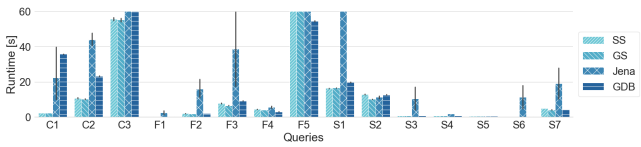
[40] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. 2012. Heuristics-based query optimisation for SPARQL. In *EDBT*. ACM, Berlin, Germany, 324–335.

[41] Dong Wang, Lei Zou, and Dongyan Zhao. 2015. Top-k queries on RDF graphs. *Information Sciences* 316 (2015), 201–217.

[42] Ishaq Zouaghi, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, and Taoufik Aguili. 2020. Query Optimization for Large Scale Clustered RDF Data.. In *DOLAP*. 56–65.



(a) Query runtime in WATDIV 100M



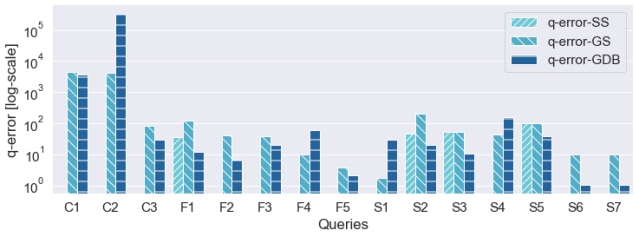
(b) Query runtime in WATDIV 1B

Figure 7: Query runtime using Shapes Statistics, Global Statistics, Jena, and GraphDB (GDB).

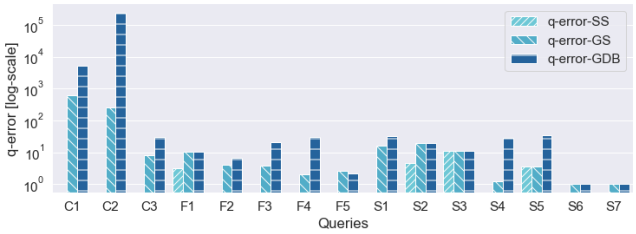
A EXPERIMENTS ON WATDIV

Figures 7a and 7b report the runtime for WatDiv queries on both sizes (i.e., WatDiv 100M and WatDiv 1B). The default query set of WatDiv dataset is quite heterogeneous and only contain RDF type triple patterns in 4 queries, i.e., F1, S2, S3, and S5. Therefore, SS could improvised the query plans proposed by GS for queries F1, S2, S3, and S5 only. For complex queries C1, C2, and C3, GS and SS proposed equivalent query plans which are optimal than Jena and GDB. For snowflake queries F1, F2, F3, F4, and F5, overall GS, SS and GDB proposed almost competitive query plans except Jena which could not propose optimal query plans. For all of the star queries GS, SS, and GDB proposed equivalent query plans. However, the plans proposed by Jena are not competitive to other approaches. As each query is executed 10x, the error-bars shown on Jena's query execution time shows the variation in query plans proposed using heuristics approach.

For small variant of WatDiv (shown in Figure 8a), the q-error value for complex queries proposed by GS and GDB is significantly far from the optimal. However, for snowflake and star queries the q-error value is close to optimal with some variance across queries. In general for the small variant, GDB computes more optimal value for the q-error. On the other hand, for large variant of WatDiv (shown in Figure 8b), GDB computed the q-error with significant errors in comparison to the values computed by GS and SS (for queries F1, S2, S3, S5).



(a) q-error in WATDIV-S



(b) q-error in WATDIV-L

Figure 8: Q-error using Shapes Statistics, Global Statistics, and GraphDB (GDB).