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, DBpedia [3], 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 [4] and analytic queries in the biomedical field can involve more than 50 joins per query [9]. 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 columns [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].

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 also be exploited for join cardinality estimation.

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 synopsis 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) [22] 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 as the 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 [22]) 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 allows 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 (s, p, o) ∈ (I ∪ B) × (I ∪ B ∪ 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 '?i' 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 an example SPARQL query (Q) and its query graph QG 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 ⊆ (I∪L∪V) × (I∪V) × (I∪L∪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)1 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 Gsb is an RDF graph describing a set of node shapes S and a set of property shapes P, such that targetfs : S→I and targetgs : P→I are injective functions mapping each node shape sj ∈ S and each property shape pi ∈ P to the IRI of a target class and a target predicate in G respectively, and φ : S→2P is a surjective function assigning to each node shape sj a subset Pi ⊆ 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:property 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.
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:

**Definition 4.1 (Join Ordering).** Given a set of triple patterns \( T = \{t_1, t_2, \ldots, t_n\} \subseteq (\text{I} \cup \text{J} \cup \text{V}) \times (\text{U} \cup \text{V}) \times (\text{I} \cup \text{J} \cup \text{V}) \), the join order \( O \) for \( T \) is defined as a total ordering \( O \) of \( T \) so that for every \( t_i, t_j \in T \), either \( t_i <_O t_j \) or \( t_j <_O t_i \).

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. In line with the related work \([9]\), 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 \( T = \{t_1, t_2, \ldots, t_n\} \), apply a cardinality estimation function \( f : \text{} \times \text{} \rightarrow \mathbb{N} \) such that for every pair of triple patterns \( (t_1, t_2) \in T \), \( f(t_1, t_2) = |t_1 \times t_2| \).

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 of \( f((t_1 \times t_2) \times t_3) \) assuming \( t_1 \times t_2 \times t_3 \). Then, given such estimates, an optimal query plan minimizes the total number of operations to compute, i.e., the execution cost \( \text{Cost}(T, O) \) 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 \( T = \{t_1, t_2, \ldots, t_n\} \) and a join cardinality estimation function \( f \), find the join order \( O \) obtained as \( \text{arg min}_O \text{Cost}(T, O) \).
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.

We consider three possible types of joins between two triple patterns based on the position of the common variable: 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.

6.3 Join Ordering

Given an RDF graph $G$, its shapes statistics graph ($G_{sh}$), and global statistics graph ($G_{glob}$), 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. Its complexity is cubic to the number of triple patterns in the query, i.e., $O(n^3)$.

Given our example query $Q$, and the cardinalities of its triple patterns $T = \{tp_1, tp_2, ..., tp_n\}$ estimated with both global and shape statistics, Tables 2a and 2b show the join ordering computed only using global statistics ($O_{glob}$) and via shapes statistics ($O_{sh}$), respectively. There is a significant difference between the estimated and true join cardinalities and their final total cost. The estimated join cardinalities for $O_{sh}$ are much closer to the true cardinalities of the query than the estimates for $O_{glob}$ with two exceptions for $tp_5$ and $tp_8$ where shapes statistics largely overestimate their cardinalities due to skewed distribution of data.

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 and GraphDB) 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: Nowadays, constraints languages are having widespread application to validate RDF graphs [21]. We assume the availability of SHACL shapes graph with the dataset and provide a Shapes Annotator to extend it with statistics of the graph. For cases where they are not present, the SHACLGEN library is commonly used to generate shapes graphs and we also use it in our case (e.g., for YAGO-4). 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 8031 property shapes) it took 62 minutes. We implemented our join ordering algorithm in Java using Jena5. The source code is available on our website6.

We loaded all three datasets and their relevant SHACL shapes graphs into Jena TDBs7. 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 ontoexplain feature to obtain the query plans. For Characteristic Sets [19] approach, we generated characteristic sets of each

Table 1: Cardinality estimation of triple patterns

- $\text{card}(tp_{SS} \bowtie tp_{SS}) = \frac{\text{card}_{S} \times \text{card}_{S}}{\text{max}(DSC, DSC)}$ (1)
- $\text{card}(tp_{SO} \bowtie tp_{SO}) = \frac{\text{card}_{S} \times \text{card}_{O}}{\text{max}(DSC, DOC)}$ (2)
- $\text{card}(tp_{OO} \bowtie tp_{OO}) = \frac{\text{card}_{O} \times \text{card}_{O}}{\text{max}(DOC, DOC)}$ (3)

Algorithm 1 Join Ordering

Input: $Q$, $G_{sh}$, $G_{glob}$
Output: Join order $O$ of $Q$

1: $p = []; r = []; \text{queue} = \text{queue}\_\text{init}()$
2: $\text{cost} = 0; \text{card} = 0; \text{queue} = \text{queue}\_\text{init}()$
3: $\text{tp}i = \text{getTPs}(Q)$
4: $\text{tp}i = \text{getCandidateShapes}(Q, G_{sh}, G_{glob})$
5: $\text{tp}i = \text{computeCardinalities}(\text{tp}i)$
6: $\text{sort}(\text{asc}, \text{tp}i.\text{cardinality})$
7: $\text{p.add}(\text{tp}i); \text{r.addAll}(\text{tp}i.\text{tp}es);$
8: $\text{cost} = \text{tp}i.\text{cardinality}$
9: $\text{queue.add}(\text{tp}i.\text{index})$
10: for $\text{tp}i, \text{cost} \in \text{queue}$ do
11:  $\text{index} = \text{tp}i.\text{index}$; $\text{cost}_\text{local} = \text{cost}$
12:  while $\text{queue}.\text{isEmpty}$ do
13:    $\text{tp}i = \text{queue}.\text{get}()``
14:    $\text{p.add}(\text{tp}i.\text{index})$
15:    $\text{for} \text{tp}j, \text{c} \in \text{queue}$ do
16:      $\text{c} = 0$
17:      if $\text{tp}j \bowtie \text{tp}i$ then
18:        $\text{c} = f(\text{tp}j, \text{tp}i)$
19:      else $\text{c} = \text{tp}j.\text{cost}_\text{local}$
20:      if $\text{c} < \text{cost}_\text{local}$ then
21:        $\text{cost}_\text{local} = \text{c}$; $\text{index} = \text{tp}j.\text{index}$; $\text{card} = \text{c}$
22:    $\text{cost} = \text{cost}_\text{local}$
23:    $\text{r.remove}(\text{tp}j.\text{get}())$
24:  $\text{O} = \text{queue}.\text{poll}()$
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_C), the estimated join cardinality (E_J_Card) and the true join cardinality (T_Card) for the ordered triple patterns of example query Q computed over LUBM dataset.

<table>
<thead>
<tr>
<th>Triple Pattern (TP)</th>
<th>DSC</th>
<th>DOC</th>
<th>E_C</th>
<th>E_J_Card</th>
<th>T_Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: rdfs:type Professor</td>
<td>30,000</td>
<td>18,852</td>
<td>85,008</td>
<td>85,006</td>
<td>85,006</td>
</tr>
<tr>
<td>2: a: name 78</td>
<td>10,064</td>
<td>1,070,580</td>
<td>8,579</td>
<td>255,148</td>
<td>255,148</td>
</tr>
<tr>
<td>3: a: teacherOf 3C</td>
<td>509,417</td>
<td>539,417</td>
<td>922</td>
<td>1,027,929</td>
<td>1,027,929</td>
</tr>
<tr>
<td>4: c: advisor a</td>
<td>1,045,410</td>
<td>1,045,410</td>
<td>5,045,410</td>
<td>5,045,410</td>
<td>5,045,410</td>
</tr>
<tr>
<td>5: rdfs:type GraduateCourse</td>
<td>912,591</td>
<td>1,082,591</td>
<td>1,027,929</td>
<td>1,027,929</td>
<td>1,027,929</td>
</tr>
<tr>
<td>6: rdfs:type GraduateStudent</td>
<td>2,128,580</td>
<td>2,128,580</td>
<td>1,027,929</td>
<td>1,027,929</td>
<td>1,027,929</td>
</tr>
<tr>
<td>7: degreeFrom 7U</td>
<td>1,439,470</td>
<td>1,439,470</td>
<td>757</td>
<td>638,419</td>
<td>638,419</td>
</tr>
<tr>
<td>8: takesCourse c</td>
<td>1,740,467</td>
<td>1,740,467</td>
<td>3,747</td>
<td>2,964,994</td>
<td>2,964,994</td>
</tr>
<tr>
<td>9: rdfs:type GraduateStudent</td>
<td>1,259,606</td>
<td>1,259,606</td>
<td>1,259,606</td>
<td>1,259,606</td>
<td>1,259,606</td>
</tr>
</tbody>
</table>

Table 3: Size and characteristic of the datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LUBM</th>
<th>WATDIV-S</th>
<th>WATDIV-L</th>
<th>YAGO-4</th>
</tr>
</thead>
<tbody>
<tr>
<td># of triples</td>
<td>814</td>
<td>814</td>
<td>814</td>
<td>814</td>
</tr>
<tr>
<td># of distinct objects</td>
<td>12 M</td>
<td>9 M</td>
<td>9 M</td>
<td>12 M</td>
</tr>
<tr>
<td># of distinct subjects</td>
<td>10 M</td>
<td>5 M</td>
<td>5 M</td>
<td>5 M</td>
</tr>
<tr>
<td># of distinct RDF type triples</td>
<td>1 M</td>
<td>1 M</td>
<td>1 M</td>
<td>1 M</td>
</tr>
<tr>
<td># of distinct RDF type objects</td>
<td>39</td>
<td>46</td>
<td>39</td>
<td>8,912</td>
</tr>
</tbody>
</table>

Table 3: Size and characteristic of the datasets
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 and feasible 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.

Acknowledgments. This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, the EU’s H2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 838216, and the Poul Due Jensen Foundation.

REFERENCES

A EXPERIMENTS ON WATDIV

Figure 5a shows the query runtime analysis for query plans proposed using 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 WATDIV-S (100M). The SS approach were effective for F1, S2, S3, and S5 queries having at least one type defined triple pattern. The query runtime for complex queries (C1, C2, C3) proposed by GS approach performs better than Jena, GDB, CS, and is competitive with SumRDF approach. The query plans proposed by the CS approach are not optimal for complex queries as compared to other approaches. For snowflake queries, the plans proposed by GS and SS are competitive to GDB and SumRDF for all except F1 and F4. The CS approach could not propose optimal query plans for F1, F2, and F4. All approaches identify plans with comparable good performance for star queries. Similar to LUBM and YAGO-4 results, the plans proposed by Jena are often suboptimal and non-deterministic for most of the WatDiv queries.

Likewise, Figure 5b shows the query runtime for WATDIV-L (1 Billion), where the results are analogous to WATDIV-S (100M). However, we could not compare it with the CS and SumRDF approach as the preprocessing steps (to compute characteristics sets and SumRDF’s summary) are computationally expensive and very time-consuming.

In Figure 6a, we analyze the q-error values for SS approach, GS approach, GraphDB (GDB), Characteristic Sets (CS), and SumRDF approach on WATDIV-S queries. For GS approach, 5 queries have q-error less than 15, 8 queries have q-error less than 250, and only 2 queries have q-error value greater than 250. The SS approach improved the q-error value for F1, S2, S3, and S5 as these queries contain at least one type defined triple pattern. The SumRDF approach proposed q-error with less errors for most of the queries. However, this does not have any impact on the performance of the proposed query plans. Similarly, Figure 6b shows the q-error analysis on WATDIV-L queries. The q-error values proposed by SS and GS have less error as compared to GDB.

Finally, Figure 7a and 7b present the analysis between actual and true costs of query plans proposed by SS and GS on WATDIV-S and WATDIV-L datasets. The cost estimated by SS approach is almost equivalent to the actual cost for queries having at least one type defined triple pattern, i.e., F1, S2, S3, and S5. This shows the effectiveness of SS approach for type defined queries in very large datasets.

Figure 5: Query runtime on WatDiv datasets

Figure 6: Q-error analysis on WatDiv datasets

Figure 7: Cost analysis on WatDiv datasets