Fast and Accurate Optimizer for Query Processing over Knowledge Graphs

Jingqi Wu, Rong Chen, Yubin Xia
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Shanghai Artificial Intelligence Laboratory

Abstract

This paper presents GPL, a fast and accurate optimizer for query processing over knowledge graphs. GPL is novel in three ways. First, GPL proposes a type-centric approach to enhance the accuracy of cardinality estimation prominently, which naturally embeds the correlation of multiple query conditions into the existing type system of knowledge graphs. Second, to predict execution time accurately, GPL constructs a specialized cost model for graph exploration scheme and tunes the coefficients with target hardware platform and graph data. Third, GPL further uses a budget-aware strategy for plan enumeration with a greedy heuristic to boost the overall performance (i.e., optimization time and execution time) for various workloads. Evaluations with representative knowledge graphs and query benchmarks show that GPL can select optimal plans for 33 of 39 queries and only incurs less than 5% slowdown on average compared to optimal results. In contrast, the state-of-the-art optimizer and manually tuned results will cause 100% and 36% slowdown, respectively.

CCS Concepts:
• **Information systems → Query optimization**.

Keywords: Query optimization; knowledge graphs

ACM Reference Format:
Jingqi Wu, Rong Chen, Yubin Xia. 2021. Fast and Accurate Optimizer for Query Processing over Knowledge Graphs. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472883.3486991

1 Introduction

In recent years, there has been a significant interest in building large-scale graph stores and querying systems for knowledge graphs [8, 13, 18, 24, 25, 28, 36–38, 40, 41]. The query optimizer is used to find the best possible query plan such that the execution time (i.e., query latency) is minimal, which is crucial for query performance. Although query optimization is a well-studied area in the database community, the *schema-free nature* of graph data [27, 34] and the emerging *graph-exploration scheme* for query processing [36, 40] introduce new challenges for query optimization.

In this paper, we first investigate the three main components of the classical query optimizer architecture, as shown in Figure 1, to reveal the major issues and challenges on the accuracy and performance of prior state-of-the-art approaches [18, 32, 40]. For cardinality estimation, the correlation among all query conditions (e.g., variable vertices and/or edges) is crucial to the accuracy. Thus, statistical synopses and estimation based on the assumption of independence would incur significant errors on estimated results. For cost model, it is hard to attach the mature join-centric cost model to emerging exploration-based query processing. Further, the linear combination of computation and communication cost is not competent to predict execution time (wall-clock time) accurately. For plan enumeration, with significant performance improvement by leveraging advanced software and hardware techniques [36] (e.g., graph exploration and RDMA), the cost to enumerate the whole plan space (optimization time) may become a non-trivial part of the overall query time, especially for selective (short-lived) queries.

To remedy these issues, we introduce three key designs and demonstrate the viability and efficacy of them in GPL, a fast and accurate optimizer for query processing over knowledge graphs.

Type-centric estimation. Based on the observation that vertices with the same type commonly have a similar combination of edges, GPL embeds the correlation of query conditions explored so far into the existing type system.[1] GPL first uses the correlation statistics of types and edges as the statistical synopses and then estimates the cardinality (i.e., the number of bindings) for each directed exploration of conditions in the query plan by using two kinds of mechanisms: expansion and pruning.

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486991

[1] The type in RDF (https://www.w3.org/TR/rdf-schema) and the label in property graph (https://neo4j.com/docs/getting-started/current/graphdb-concepts).

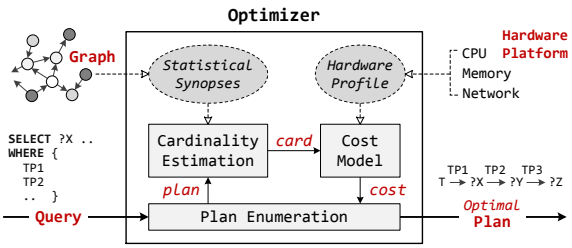

Figure 1. *The classical query optimizer architecture.*

Exploration-oriented cost model. We first construct a specialized cost model aiming at graph exploration with various query conditions, and then design a tool (built-in queries) to one-off tune coefficients of the model on the target hardware platform and graph data. Further, we observe that predicting cost based on the estimation results (i.e., cardinalities) could be seen as coarse-grained query processing over knowledge graphs. Thus, GPL mimics the execution of graph query to accurately predict the execution time by feeding the estimation results to the specialized cost model for exploring query conditions one by one.

Budget-aware enumeration. In response to demand from workload heterogeneity in time and differentiated enumeration strategies, GPL proposes a new budget-aware enumeration scheme to optimize all kinds of graph queries without advance knowledge of execution time. Due to the importance of the enumeration order, GPL adopts a depth-first enumeration with a greedy heuristic to select the next exploration.

We have implemented GPL based on Wukong [36], a state-of-the-art graph store and execution engine for RDF knowledge graphs and SPARQL queries. We evaluated GPL on an 8-node cluster using common SPARQL query benchmarks over a set of synthetic and real-life RDF datasets. The experimental results show that GPL can select optimal plans for 33 of 39 queries and only incurs less than 5% slowdown on average (geometric mean) compared to optimal results. In contrast, state-of-the-art approaches (Trinity.RDF [40]) and manually tuned results (Wukong [36]) will cause a 100% and 36% slowdown, respectively. GPL can further provide a very close prediction on execution time (the median of *q-error* [26] is 1.44). In addition, our budget-aware enumeration scheme can reduce optimization time by 75% (from 0.17ms to 0.04ms), at the expense of a slight increase in execution time (from 0.32ms to 0.38ms).

In summary, the contributions of this paper are:

- An in-depth analysis of three key components of the query optimizer to reveal the main issues and challenges on the accuracy and performance for traditional approaches;
- A new query optimizer with three new techniques that targets the schema-free nature of graph data and the emerging graph-exploration scheme;

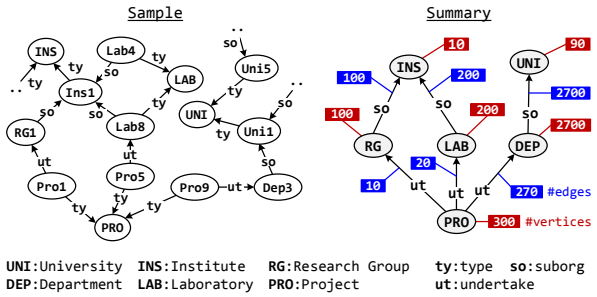

Figure 2. *A sample RDF knowledge graph (G) and its summary. The summary contains the schema (i.e., ontology) and the number of vertices and edges for each type and predicate.*

- A prototype implementation and an evaluation that shows high accuracy and good performance of GPL.

The source code of GPL, including all benchmarks, is available at https://github.com/SJTU-IPADS/wukong.

2 Background

2.1 Graph Model and Query Language

Many graph models and declarative query languages [1, 4, 6, 15, 19, 20, 23, 28, 35] are developed to meet the increasing demands to store and query knowledge graphs with high performance and sufficient expressiveness. In this paper, without loss of generality, we use RDF [2] and SPARQL [1], the representative graph model and query language recommended by W3C, to reveal the major issues of prior approaches and demonstrate the efficacy of our design.

The RDF dataset is a graph (aka RDF knowledge graph) composed of triples, where a triple is formed by $\langle subject, predicate, object \rangle$ and can be regarded as a directed edge (*predicate*) connecting two vertices (from *subject* to *object*).

Figure 2 illustrates a part of the sample graph (G) and the summary of the whole graph. There are three categories of edges linking six types of vertices. SPARQL is the standard query language for RDF datasets. The major part of the SPARQL query is as follows:

$$Q := \text{SELECT RD WHERE GP}$$

where GP is a group of *triple patterns*, and RD is a *result description*. Each triple pattern (TP) is of the form $\langle subject, predicate, object \rangle$, where each of the subject, predicate, and object may denote either a *variable* (e.g., ?X) or a *literal* (e.g., Lab8). RD contains a subset of variables in GP.

Given an RDF graph, the SPARQL query searches on the graph for a set of subgraphs, each of which matches all triple patterns (i.e., query conditions) by binding pattern variables to values in the graph. For example, the query Q in Figure 3 asks for all projects (?Z) that were undertaken (ut) by a laboratory (LAB), which is a sub-organization (so) of an institute (INS). The possible binding over the part of graph G in Figure 2 is only Pro5. The *graph exploration* is proposed as

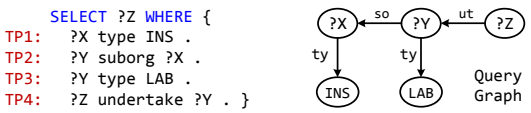

Figure 3. *A sample SPARQL query (Q).*

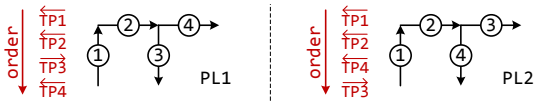

Figure 4. *Two sample plans (PL1 and PL2).*

a new primitive to process SPARQL queries for knowledge graphs [8, 36, 40], which starts from a set of vertices and explores bindings via edges by matching every triple pattern. Note that the execution order of triple patterns (i.e., *plan*) will only impact the performance (much), not the result of the query. Therefore, the RDF system normally relies on the query optimizer to enumerate the valid execution orders to find an *optimal* query plan (i.e., minimal execution time) from *semantically equivalent* plan alternatives, such as PL1 and PL2 in Figure 4. For each plan, the optimizer first estimates the *cardinality* (e.g., the number of intermediate results) based on the *statistical synopses* of RDF data, and then predicts the *cost* (e.g., a relative score) based on the *cost model* for a given hardware setting (see Figure 1).

2.2 Query Optimization

Inspired by traditional optimizers for relational databases, Trinity.RDF [40] proposes the original optimizer for SPARQL queries on graph-exploration systems, which models a query plan as a graph traversal and finds an order of triple patterns explored with minimal cost.[2] The optimizer uses the number of intermediate results for graph exploration (i.e., the number of paths) as the cardinality, and employs dynamic programming for plan enumeration with a proportional cost model.

Specifically, the RDF-specific statistical synopses mainly contain two types of statistics associated with *predicates* for graph exploration. First, the optimizer precomputes the number of distinct subjects ($C_s(p)$), objects ($C_o(p)$), and triples ($C(p)$) for each predicate (p). Note that a specific type T is also considered as a predicate ty_T, so $C(\text{ty}_\text{T})$ and $C_s(\text{ty}_\text{T})$ are the same, namely the number of subjects with the type T. Second, the correlation ($Cor(p_1, p_2, D)$) between predicate pairs (p_1 and p_2) is estimated to denote the number of distinct vertices with both two predicates according to D (the combination of the directions of two predicates). Figure 5 illustrates a part of the statistics on the sample graph (G). so and ut are correlated predicates.

The optimizer assumes the *cardinality* of a query Q as the sum of the cardinality of triple patterns ($\langle \text{TP}_1, \ldots, \text{TP}_n \rangle$)

[2]The traditional scan-join RDF systems, like RDF-3X [32] and TriAD [18], store RDF graph as a set of triples in relational tables and leverage scan-join operations to process SPARQL queries. The query optimizer [18, 32] enumerates plans with different orders of join operations on triple patterns.

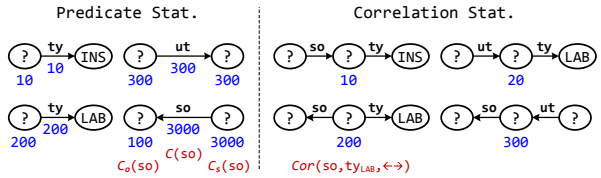

Figure 5. *A part of statistics over the sample graph (G), adopted by predicate-based query optimizer (e.g., Trinity.RDF).*

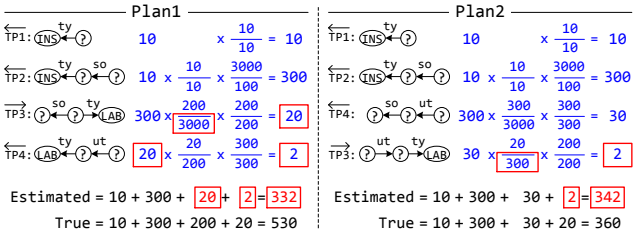

Figure 6. *A case of cardinality estimation for the sample query (Q), adopted by predicate-based query optimizer (e.g., Trinity.RDF [40]). The true cardinality (True) is calculated as the sum of intermediate results for each triple pattern according to the execution order of two sample plans (PL1 and PL2).*

explored in a specific order (*plan*), where the cardinality of triple pattern with the predicate p is estimated as the number of intermediate results ($|R(\text{TP})|$).

$$Card(\text{Q}) = \sum_{i=1}^{n} |R(\text{TP}_i)|$$

Given the number of bindings for subjects $|B(s)|$ in the exploration (assume the direction is from subject to object), $|R(\text{TP})|$ and $|B(o)|$ (the number of bindings for objects) can be formalized as

$$|R(\text{TP})| = |B(s)|\frac{C(p)}{C_s(p)}, \quad |B(o)| = |B(s)|\frac{C_o(p)}{C_s(p)}$$

Further, the number of bindings ($|B(v)|$) is affected by the number of bindings for the correlated triple pattern already explored ($|B'(v)|$). For exploring triples with the predicate p_2 from the triples with the predicate p_1, the number of new bindings ($|B(v)|$) can be estimated as

$$|B(v)| = |B'(v)|\frac{Cor(p_1, p_2, D)}{C(p_1)}$$

For example, as shown in Figure 6, the cardinality of exploring the triple pattern TP3 ($\langle$?Y, type, LAB$\rangle$) in PL1 is 20, where the number of bindings ($|B(s)|$) for ?Y is 300 and the correlation statistic $Cor(\text{so}, \text{ty}_\text{LAB}, \leftarrow\rightarrow)$ is 200. $C(\text{so})$, $C(\text{ty}_\text{LAB})$, and $C_s(\text{ty}_\text{LAB})$ are 3000, 200, and 200 respectively. Consequently, the estimated cardinality of PL1 is 332, the sum of cardinalities for all triple patterns (TP1=10, TP2=300, TP3=20, TP4=2).

The cardinality of every triple pattern is fed into the cost model, which is a linear combination of computation cost and communication cost. The computation cost is estimated as the number of intermediate results ($|R(\text{TP})|$), while the communication cost is estimated as the shipping results ($|B(v)|$).

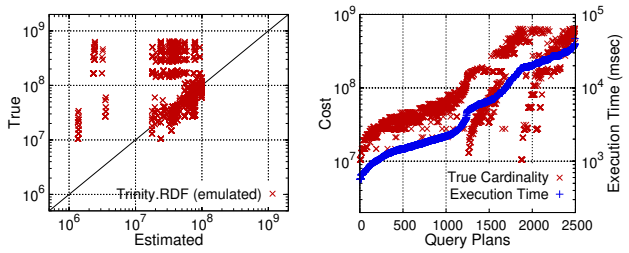

Figure 7. *The comparison of (a) estimated cardinality vs. true cardinality and (b) true cardinality vs. execution time by using all of plans (2496) for Q1 over LUBM-2560. Trinity.RDF's optimizer [40] is not available, and we reimplement it on Wukong [36].*

Finally, a bottom-up dynamic programming (DP) algorithm is used in plan enumeration to determine the order of triple patterns that yields the minimal cost. At each DP step, the optimizer calculates the cost of the partial plan considered so far and prunes if the current branch cannot gain the minimal cost anymore.

3 Analysis of Query Optimization

This section presents an analysis on the accuracy and performance issues of state-of-the-art optimization approaches for query processing over knowledge graphs.

Cardinality estimation. The main drawback of the existing approach is to simply assume the *independence* among triple patterns and only consider *at most* two correlated predicates (like ty_{LAB} and so). However, we observe that *the correlation among all triple patterns is crucial to the accuracy of cardinality estimation.* For example, as shown in Figure 6, the estimated cardinalities of both two plans in Figure 4, PL1 and PL2, are incorrect. Specifically, according to the statistical synopses, there are 3000 vertices (subjects) that have the predicate so, while only 200 of them also have the predicate ty_{LAB}. Thus, for TP3 in PL1, given 300 vertices (?Y) with the predicate so, the number of bindings with the predicate ty_{LAB} is estimated as 20. However, another *dependent* predicate ty_{INS} in this query (TP1) is overlooked. In fact, for the sub-organization (so) of INS, the probability of being a LAB is $\frac{200}{300}$, rather than $\frac{200}{3000}$. PL2 has the same problem but in a different order. As a result, this drawback penalizes different plans for a single query in varying degrees and even reverses the relative results of PL1 and PL2 (see Figure 6). Figure 7(a) illustrates the true and the estimated cardinality for different plans of a query (Q1) on LUBM-2560 [5].[3] The straight diagonal line denotes the ideal results (namely the true cardinality is equal to the estimated cardinality), while the deviation for the existing approach [40] is extremely serious.

Besides, in the extreme, the combination of multiple predicates may lead to an empty result, namely the *contradictory* query. For example, if the predicate (ty_{INS}) in TP1 is replaced

[3]Detailed experimental setup can be found in §5.

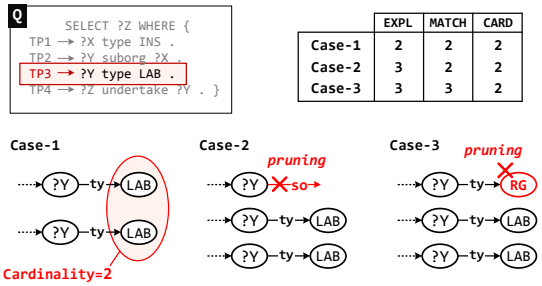

Figure 8. *Three cases of the result for TP3 with the same cardinality but different costs. The red cross denotes the path is pruned due to an unmatched predicate (ty vs. so) or vertex (LAB vs. RG).* EXPL *and* MATCH *denote the number of predicates (edges) and subjects/objects (vertices) that will be explored and matched.* CARD *denotes the number of intermediate results (cardinality).*

by ty_{UNI}, the query will not retrieve any binding. However, the existing approach cannot detect such contradictory queries when the correlation involves more than two predicates, so it would waste time on unnecessary query processing.

More importantly, there is a *fundamental* problem of approaches that estimates the cardinality based on the correlated predicates. A straightforward solution is to maintain more statistics for three or more correlated predicates. However, the size of statistics would rapidly increase beyond the memory capacity, and it is impossible to precompute all the dependencies between multiple predicates due to the schema-free nature of graph data. For datasets with numerous predicates, like DBPSB [3], it fails to maintain the correlation statistics even for just two correlated predicates (see Table 5).

Cost model. The cost model for (structured) relational databases has been well-studied, which thoroughly considers different join operations (e.g., hash and merge join) and the cost of data transferring by network connections. Unfortunately, *it is quite difficult or impossible to attach a join-centric cost model to the exploration-based query processing directly.* The initial cost model [40] briefly uses a linear combination of computation cost (the number of intermediate results) and communication cost (the number of shipping bindings).

To reveal the importance and necessity of an appropriate cost model, Figure 7(b) compares the cost derived from the true cardinality and the execution time for the same plan. For brevity, we run the query (Q1) on LUBM-2560 using a single machine to eliminate communication cost. It is obvious that the cardinality alone is not sufficient to predict the execution time exactly. As an example, Figure 8 illustrates three exploration cases for TP3 with the same cardinality (CARD=2) but different computation costs (exploring and matching predicates and subjects/objects). In fact, there

Table 1. *A comparison of execution and optimization time (in millisecond) for different queries on WSDTS.* #TP *and* #PLAN *denote the number of triple patterns and plans.* EXE *and* PLAN *denote the execution time and the optimization time. The numbers in red denote that the optimization time is comparable to the execution time.*

	#TP	#PLAN	TriAD		Trinity.RDF	
			EXE	PLAN	EXE	PLAN
F1	6	1260	3.93	0.34	0.199	0.131
F2	8	97440	18.07	1.25	0.590	0.438
F3	6	1440	15.57	0.34	0.723	0.025
F4	9	705600	6.95	2.53	0.675	0.861
F5	6	1440	34.12	0.32	0.075	0.040
C1	8	13320	16.17	0.60	1.346	0.143
C2	10	127800	35.77	0.95	2.513	1.402
C3	6	5040	44.59	0.51	39.54	0.173

are still many other types of triple patterns for graph exploration. Hence, it is imperative and challenging to model graph exploration for predicting the cost accurately.

More importantly, existing approaches focus on comparing the relative cost among different plans of one same query, instead of predicting the wall-clock time, which is a more challenging task but also has more value. For example, based on the wall-clock time, the optimizer can make a tradeoff between accuracy (execution time) and performance (optimization time) for various queries.

Plan enumeration. The optimization time (PLAN) has not received much attention since it is negligible compared to the execution time (EXE) in traditional scan-join systems like TriAD [18] (see Table 1). However, with significant performance improvement by leveraging advanced software and hardware techniques [36] (e.g., graph exploration scheme and fast RDMA-enable networks), the optimization time may occupy a notable fraction of the end-to-end query time (e.g., see Trinity.RDF [40] in Table 1).

More importantly, the optimization time has nothing to do with the execution time but with the complexity of the query (i.e., the number of triple patterns). Hence, we face a *dilemma* that *the long-lived query (e.g., C3) demands the best possible plan by thorough optimizing, while the short-lived query (e.g., F4) demands an acceptable plan by efficient optimizing*. Unfortunately, prior work [18, 36, 40] uses a "one size fits all" enumeration scheme to handle all kinds of queries, which would sacrifice the end-to-end performance. To make matters worse, the execution time of graph queries is not polarized (either long-lived or short-lived) and is relative to the optimization time, like C1 and C2. Thus, a rule-based mechanism with simple heuristics is far from enough.

4 Designs

4.1 Type-centric Estimation

Observations. To overcome the *fundamental* problem of traditional approaches that estimate the cardinality based on the correlated predicates, the query optimizer needs to find an efficient way to *embed the lineage of correlated triple patterns explored so far and pass it on to the next graph exploration*. Fortunately, the predicate type is a perfect candidate. W3C has provided a set of unified vocabularies (as part of the RDF standard) to encode the rich semantics, where the predicate type (short for rdf:type) provides a classification of vertices of an RDF graph into different groups. We observe that *vertices with the same type commonly have a similar combination of predicates*. For example, in Figure 2, all institutes (INS) has two predicates: so and ty_{INS}. Table 2 shows the percentage of vertices with a similar combination of predicates as other vertices of its type for three synthetic and real-life datasets [3, 5, 7]. Therefore, we argue that the combination of predicates in triple patterns can be used to deduce the type of bindings, which may have multiple candidates with different probabilities. Further, the combination of the type of current bindings and the predicate in the next triple pattern can also be used to deduce the number and the type of next bindings. Moreover, if the predicate type appears in certain triple patterns of the query, which is common (e.g., Q in Figure 3), it will directly improve the overall accuracy of the cardinality estimation. It is worth noting that this observation could be generalized beyond RDF/SPARQL systems. For example, the property graph also has a similar type system, namely the label of vertex and edge.

Table 2. *The similarity of vertices with types in different datasets.* #P, #T, *and* $\#V_T$ *denote the number of predicates, types, and vertices with at least one type.* Similarity *denotes the percentage of vertices with a similar combination of predicates as other vertices of its type. Note that we consider a different combination of types as a new type.*

Dataset	#P	#T	$\#V_T$	Similarity
LUBM-2560	17	14	52,272,182	96.29%
WSDTS	86	39	10,234,195	72.28%
DBPSB	14,128	54,736	707,641	74.95%

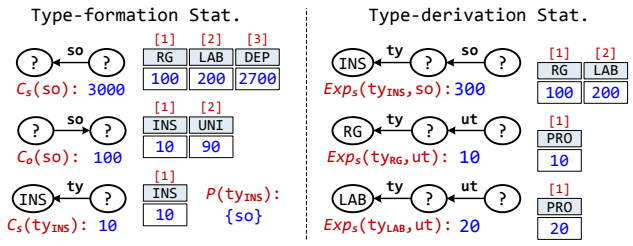

Figure 9. *A part of type-based statistics over sample graph (G).*

Type-based statistical synopses. Based on the above observation, it makes sense that the type-centric approach uses the correlation statistics of types and predicates as the statistical synopses. Two kinds of type-based statistics are precomputed during loading graph data.

Table 3. *A summary of notations in the type-centric estimation.*

Notation	Description
T	Type of bindings
p	Predicate of the triple or the triple pattern
TP_k	The kth triple pattern
$\|B\|$	Number of bindings
$\|R\|$	Cardinality of the triple pattern explored

Type-formation statistics: For each predicate p, GPL precomputes the *type-formation* array for the subjects and objects, namely $C_s(p)$ and $C_o(p)$, which returns the type composition and the number of subjects/objects. For each type (T), besides C_s(T), GPL further collects the combination of predicates (P(T)). In Figure 9, $C_s(so)$ returns an array of the type composition and the number of sub-organizations (subjects). The total number is 3000, including 100 research groups ($\mathsf{ty}_{\mathsf{RG}}$), 200 laboratories ($\mathsf{ty}_{\mathsf{LAB}}$), and 2, 700 departments ($\mathsf{ty}_{\mathsf{DEP}}$). Further, $C_s(\mathsf{ty}_{\mathsf{INS}})$ and $P(\mathsf{ty}_{\mathsf{INS}})$ return the number of institutions (10) and the collection of predicates ($\{so\}$), respectively.

Type-derivation statistics: For each pair of correlated predicate and type, GPL precomputes the *type-derivation* array for exploring triples with the predicate (p) from subjects/objects with the type (T), namely $Exp_s(\mathsf{T}, p)$/$Exp_o(\mathsf{T}, p)$, which returns the type composition and the number of triples. In Figure 9, $Exp_s(\mathsf{ty}_{\mathsf{INS}}, so)$ returns an array of the type composition and the number of triples for all institutes. There are 300 triples in total, including 100 from research groups ($\mathsf{ty}_{\mathsf{RG}}$) and 200 from laboratories ($\mathsf{ty}_{\mathsf{LAB}}$).

Type-centric cardinality estimation. GPL still uses the number of intermediate results of graph exploration (i.e., the number of paths) as the cardinality and estimates the cardinality for each directed exploration of triple patterns in the plan. Table 3 summarizes the notations in our type-centric approach. There are two kinds of graph exploration: *expansion* and *pruning*.

Expansion: The triple pattern is explored from the *known* (i.e., a specific literal/type or a variable has been explored) to the *unknown* (i.e., a variable has not been explored), like TP1 and TP2 in PL1. Thus, given the number and the type of the known bindings ($|B'|$ and T'), the number and the type of the unknown bindings ($|B|$ and T) explored by a certain predicate (p) can be formalized as

$$\sum_{j=1}^{m} |B(T^j)| = |B'(T')| \frac{\sum_{j=1}^{m} Exp_v(T', p)[j]}{C_s(T')}$$

where T^j denotes the jth type of the new bindings, which have m types. v denotes the target of directed exploration, which is either s (subject) or o (object). Hence, $Exp_v(T', p)[j]$ denotes the total number of new bindings with the jth type by exploring triples with the predicate (p) from subjects/objects with the type (T'). For example, as shown in Figure 10, the number and the type of bindings explored by $\overleftarrow{\mathsf{TP1}}$ in PL1 are 10 and $\mathsf{ty}_{\mathsf{INS}}$. According to $Exp_o(\mathsf{ty}_{\mathsf{INS}}, so)$ and $C_s(\mathsf{ty}_{\mathsf{INS}})$ in Figure 9, the intermediate result of TP2 contains two types of bindings, including 100 research groups ($\mathsf{ty}_{\mathsf{RG}}$) and 200 laboratories ($\mathsf{ty}_{\mathsf{LAB}}$).

Suppose the kth triple pattern explores from the correlated bindings already explored by a previous triple pattern ($\mathsf{TP}_{k'}$, $0 \le k' < k$). Based on the above equation, the cardinality of the kth triple pattern ($|R_e(\mathsf{TP}_k)|$), is the sum of bindings explored from all types of previous bindings along with the predicate p_k, which can be formalized as

$$\begin{aligned} |R_e(\mathsf{TP}_k)| &= \sum_{i=1}^{n_{k'}} \left(|B_{k'}(T^i_{k'})| \frac{\sum_{j=1}^{m^i} Exp_v(T^i_{k'}, p_k)[j]}{C_s(T^i_{k'})} \right) \\ &= \sum_{i=1}^{n_k} |B_k(T^i_k)| \end{aligned} \tag{1}$$

where $T^i_{k'}$ denotes the ith type of the bindings explored by the k'th triple pattern. n_k denotes the number of types for the bindings explored by the kth triple pattern, and $n_k = \{m^1 \cup \ldots \cup m^{n_{k'}}\}$. For example, as shown in Figure 10, the cardinality of TP3 in PL2 is 30, the total number of the bindings explored from 100 research groups ($\mathsf{ty}_{\mathsf{RG}}$) and 200 laboratories ($\mathsf{ty}_{\mathsf{LAB}}$) separately.

Further, to estimate the cardinality of the first triple pattern (TP_1), T_0 and $|B_0(T_0)|$ in Equation (1) would be the type and the number of bindings for the exploration point, which could be a literal (e.g., Lab8) or a variable (e.g., ?X). For a literal, the number of bindings is 1, and the type could be retrieved from graph data directly. For a variable, the type and the number of bindings for the exploration point could be retrieved from the type-formation statistics with the predicate of the first triple pattern, but in the opposite direction ($C_{\neg v}(p_1)$). For a special case, the first triple pattern defines the type of bindings (e.g., $\overleftarrow{\mathsf{TP1}}$ in PL1), so that the type and the number of bindings could be retrieved from the type-formation statistics ($C_s(T_1)$).

$$|B_0(T_0)| = \begin{cases} 1 & \text{a literal (e.g., } \overrightarrow{\mathsf{TP1}}\text{: } \langle \mathsf{Lab8}, \mathsf{so}, ?\mathsf{X} \rangle) \\ C_{\neg v}(p_1) & \text{a variable (e.g., } \overrightarrow{\mathsf{TP1}}\text{: } \langle ?\mathsf{Y}, \mathsf{so}, ?\mathsf{X} \rangle) \end{cases}$$

$|R_e(\mathsf{TP}_1)| = C_s(T_1)$ a type (e.g., $\overleftarrow{\mathsf{TP1}}$: $\langle ?\mathsf{Y}, \mathsf{ty}, \mathsf{LAB} \rangle$)

Pruning: The triple pattern is explored from the *known* to the *known* (i.e., a specific literal/type or a variable already explored), like $\overrightarrow{\mathsf{TP3}}$ in PL1. Thus, we first explore the triple pattern as expansion by Equation (1), and then can estimate the cardinality as the product of the explored bindings and the probability of success matching the knowns.

To match a specific literal (e.g., Lab8), the bindings must first match the type of literal (ty_{L}), and then the probability is $\frac{1}{C_s(\mathsf{ty}_{\mathsf{L}})}$. Similarly, to match a variable already explored by the k'th triple pattern ($0 \le k' < k$), the bindings also must first match a certain type of the known bindings ($T^i_{k'}$), and then the probability is $\frac{1}{C_s(T^i_{k'})}$. Finally, for a special case, the triple pattern limits the type of bindings (T_k), thus the cardinality

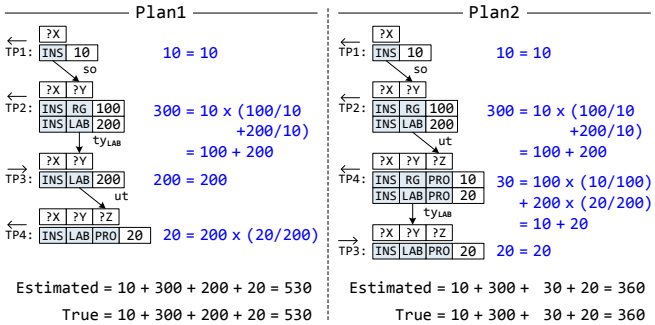

Figure 10. *A case of type-centric cardinality estimation for the sample query (Q). The true cardinality (True) is calculated as the sum of intermediate results for each triple pattern according to the execution order of two sample plans (PL1 and PL2).*

is exactly equal to the number of correlated bindings so far with the given type ($|B_{k-1}(T_k)|$).

$$|R_p(\mathrm{TP}_k)| = \begin{cases} \frac{|B_k(\mathrm{ty_L})|}{C_s(\mathrm{ty_L})} & \text{a literal (e.g., } \overleftarrow{\mathrm{TP}_k}\text{: ⟨Lab8, so, ?X⟩)} \\ \sum_{i=1}^{n_{k'}} \left(\frac{|B_k(T^i_{k'})|}{C_s(T^i_{k'})} \right) & \text{a variable (e.g., } \overrightarrow{\mathrm{TP}_k}\text{: ⟨?Y, so, ?X⟩)} \\ |B_{k-1}(T_k)| & \text{a type (e.g., } \overrightarrow{\mathrm{TP}_k}\text{: ⟨?Y, ty, LAB⟩)} \end{cases}$$

Figure 10 shows the cardinality estimation for the query Q with two plans by using the type-centric approach. In both plans, $\overleftarrow{\mathrm{TP1}}$, $\overleftarrow{\mathrm{TP2}}$, and $\overleftarrow{\mathrm{TP4}}$ are *expansion* triple patterns, while TP3 is a *pruning* triple pattern. Compared to the predicate-based approach, which just considers the correlation between at most two predicates, the type-centric approach can estimate the cardinality more accurately, thanks to embedding the correlations into the types and passing them on through graph exploration. For example, $\overrightarrow{\mathrm{TP3}}$ is severely under-estimated in Figure 6, since the estimation involves the correlation among three predicates, so, $\mathrm{ty_{LAB}}$, and $\mathrm{ty_{INS}}$. In contrast, the estimated cardinality is exactly equal to the true cardinality for both two plans in Figure 10.

No type and multiple types. Due to the schema-free nature of graph data, it is possible that some vertices have no associated types, especially in real-life datasets (e.g., about 87% vertices in DBPSB [3]). Still, based on the observation that *the same type of vertices commonly has a similar combination of predicates*, GPL will assign new *virtual* types to them according to the combination of their predicates. On the other hand, some vertices might have multiple specified types. GPL will also assign new *virtual* types to them according to the combination of their types. The virtual type will be treated as normal, except that the bindings with this virtual type will not be pruned if the target type is contained in the virtual type. Finally, GPL will limit the total number of virtual types to reduce the memory consumption and estimation cost, and assign a generic type GType to all vertices with rare virtual types. The generic type uses the median results of virtual types removed in statistical synopse.

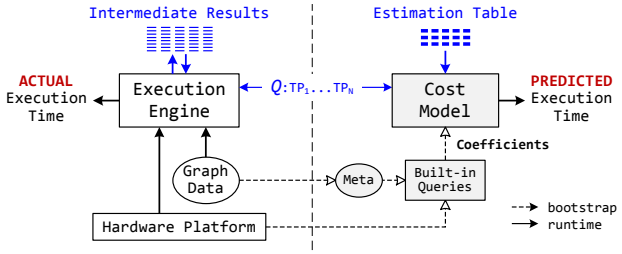

Figure 11. *A comparison of query processing and query planning.*

Variable predicate. In rare cases, the predicate in the triple pattern is a variable, like ⟨Lab8, ?P, ?X⟩. If so, at least one of the subject and the object in the triple pattern is *known* (i.e., a specific literal/type or a variable already explored). Thus, GPL could retrieve all candidates of the variable predicate according to the type of the known (e.g., $P(\mathrm{ty_{INS}})$ in Figure 9), and extend the triple pattern into multiple triple patterns with the known predicate. Finally, GPL would estimate all of them and merge the results.

Contradictory queries. The *contradictory* combination of query conditions (triple patterns) does induce an empty result and only wastes time and resources if without early detection and treatment by the query optimizer. GPL uses the type-centric estimation to detect contradictory queries even when the contradictory correlation involves more than two triple patterns, which are not detectable by traditional predicate-based approaches [8, 18, 32, 40]. For example, Q3 of LUBM [5] is a typical contradictory query, as it retrieves the undergraduate students that have obtained a degree from the university. Previous systems executed the query and retrieved an empty result as expected since the contradictory correlation involves three predicates.[4] Differently, GPL embeds the lineage of correlated triple patterns explored so far into the types and passes them on to the next. Thus, GPL can detect a contradictory query safely (no false positive[5]) when the total number of bindings becomes zero during exploration. To our knowledge, GPL is the first graph query optimizer that can recognize contradictory queries and avoid running them.

4.2 Exploration-oriented Cost Model

Observations. We observe that *predicting cost based on the estimation table could be seen as coarse-grained query processing over knowledge graphs.* First, the type-centric approach provides much more detailed estimations for each exploration (i.e., *estimation table*), which could be seen as a brief summary of intermediate results from actual query processing. Second, the cost model should be extracted from

[4] Q3 contains three triple patterns below: ⟨?Y, rdf:type, ub:University⟩, ⟨?X, ub:undergraduateDegreeFrom, ?Y⟩, ⟨?X, rdf:type, ub:UndergraduateStudent⟩.
[5] Note that GPL assumes the dataset no longer changes after the statistical synopses are collected. Currently, GPL simply disables the optimization for dynamic knowledge graphs.

Algorithm 1: *Pseudo-code of exploring triple patterns.*

Input: `lit`, *literal* in the triple pattern.
`kvar/uvar`, *known/unknown variable* in the triple pattern.
$\overrightarrow{p}$, *predicate* in the triple pattern with direction.
Output: `result`, *results* after exploring the triple pattern.

```
▷ expansion triple pattern (known-to-unknow)
1: Function L2U(lit, p⃗, uvar)                    ▷ literal-to-unknown
2:     ucol = GetCol(uvar)
3:     vals = GetTriples(lit, p⃗)
4:     for val in vals do                                  ▷ explored
5:         row[ucol] = val
6:         result.Append(row)
7: Function K2U(kvar, p⃗, uvar)                   ▷ known-to-unknown
8:     kcol = GetCol(kvar)
9:     ucol = GetCol(uvar)
10:    for row in result do                               ▷ initiated
11:        vals = GetTriples(row[kcol], p⃗)
12:        for val in vals do                              ▷ explored
13:            row[ucol] = val
14:            new.Append(row)
15:    result.Swap(new)
   ▷ pruning triple pattern (known-to-known)
16: Function K2L(kvar, p⃗, lit)                   ▷ known-to-literal
17:    kcol = GetCol(kvar)
18:    for row in result do                               ▷ initiated
19:        vals = GetTriples(row[kcol], p⃗)
20:        for val in vals do                              ▷ explored
21:            if lit == val then
22:                new.Append(row)                          ▷ matched
23:    result.Swap(new)
24: Function K2K(kvar1, p⃗, kvar2)                 ▷ known-to-known
25:    kcol1 = GetCol(kvar1)
26:    kcol2 = GetCol(kvar2)
27:    for row in result do                               ▷ initiated
28:        vals = GetTriples(row[kcol1], p⃗)
29:        for val in vals do                              ▷ explored
30:            if row[kcol2] == val then
31:                new.Append(row)                          ▷ matched
32:    result.Swap(new)
```

graph query processing (i.e., the exploration of triple patterns) with the characteristics of hardware and graph data. Figure 11 illustrates an overview and comparison of query processing and cost prediction. Inspired by symbolic execution [14], we mimic the execution of graph query by feeding the estimation table to a specialized cost model for exploring triple patterns one-by-one. Thus, to predict execution time (wall-clock time) accurately, we should construct a subtle cost model targeting graph exploration and tune the coefficients with target hardware platform and graph data.

Cost model. The emerging graph-exploration scheme for query processing demands a specialized model. The execution of the query will explore triple patterns over graph data one by one according to the plan. Thus, it is imperative to model different kinds of triple patterns carefully. Algorithm 1 lists the pseudo-code of exploring different kinds of triple patterns. For *expansion* triple patterns, both `L2U` and `K2U` iterate over every *known* binding (*initiated*) and explore new bindings (*explored*) by `GetTriples`[6] with a given predicate and direction ($\overrightarrow{p}$). For *pruning* triple patterns, after exploration, both `K2K` and `K2L` further prune bindings by matching them to the *known* (*matched*). Note that the triple pattern with either a literal or a type shares the same function (i.e., `L2U` and `K2L`). For example, the triple patterns in `PL1` will invoke `L2U`, `K2U`, `K2L`, and `K2U` in order.

The cost of a query is the sum of the cost of triple patterns, which is predicted by feeding the number of bindings to the computation and communication models.

Computation: The cost model in the computation part for each triple pattern can be formalized as

$$Cost_{comp}(\mathsf{TP}) = \lambda_I|I| + \lambda_E|E| + \lambda_M|M| + \lambda_C$$

where $|I|$ denotes the number of bindings *initiated* before exploring the triple pattern. $|E|$ and $|M|$ denote the number of bindings *explored* and *matched* respectively, after exploring the triple pattern. The different coefficients (λ_I, λ_E, λ_M, and λ_C) are one-off tuned with target hardware platform and graph data for different kinds of triple patterns.

Following the equations for expansion and pruning triple patterns, for the kth triple pattern, $|I|$, $|E|$, and $|M|$ are estimated by $|B_{k'}(T_{k'})|$, $|B_k(T_k)|$, and $|R_p(\mathsf{TP}_k)|$ respectively. For example, the three parameters are 300, 300, and 200 for $\overrightarrow{\mathsf{TP3}}$ in `PL1` (see Figure 10).

Further, considering parallel query execution of the exploration-based scheme, exploring edges is completely *independent* and can be fully parallelized on multiple threads and machines by evenly partitioning the exploration points. Thus, the computation time decreases almost *linearly* with the number of cores (N) involving query processing.

Communication: For distributed execution on S machines, the cost model in the network communication part for each triple pattern can be formalized as

$$Cost_{comm}(\mathsf{TP}_k) = \begin{cases} (S-1)\cdot(f_w(\frac{|I_k|}{S}) + f_w(\frac{|M_k|}{S})) & \text{migrating exec} \\ (S-1)\cdot f_r(\frac{|E_k|}{|I_k|})\cdot(\frac{|I_k|}{S} + \frac{|E_k|}{S}) & \text{migrating data} \end{cases}$$

where f_w and f_r denote the network cost functions with a parameter of payload size for writing and reading remote data, respectively.$|I_k|$ and $|E_k|$ denote the number of bindings before and after exploring the kth triple pattern, and $|M_k|$ denotes the number of bindings matched eventually.

[6]We assume `GetTriples` and `GetCol` in Algorithm 1 are provided by the underlying graph store [36, 40].

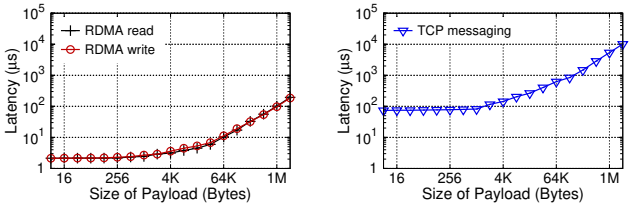

Figure 12. The network cost functions (f_w and f_r) using one-sided RDMA and TCP messaging with the increase of payload.

The communication cost can be represented as the product of the total number and the unit cost of network operations. There are two ways to implement distributed execution: migrating execution (fork-join mode) and migrating data (in-place mode) [36]. Suppose graph data is randomly assigned to S machines and the query will also evenly access graph data (i.e., $\frac{1}{S}$). By migrating execution, the exploration points ($|I_k|$) should be sent to the remote machine hosting the bindings before exploring the triple pattern, and the matched points ($|M_k|$) should be retrieved from the remote machine afterwards. Thus, the local machine will send $\frac{1}{S}$ of bindings to each remote machine for $S-1$ times. By migrating data, the bindings explored should be retrieved from the machine hosting the bindings during exploring the triple pattern, $\frac{|I_k|}{S}$ times for the vertices and $\frac{|E_k|}{S}$ times for the edges. The average size of payload is $\frac{|E_k|}{|I_k|}$.

Profiling coefficients. Unlike prior work [18, 40], Gpl expects to predict execution time (wall-clock time) instead of the relative cost. Thus, we built a tool to one-off tune the coefficients (namely λ_I, λ_E, λ_M, and λ_C) for the computation model directly on the target hardware platform and graph data. In the bootstrap, the graph store will first load, partition, and store graph data, and then precompute type-centric statistical synopses for estimation. During this period, Gpl will collect a representative dataset based on the statistics (e.g., types and predicates) to cover the graph data. To tune the coefficients, Gpl designs several query templates based on different kinds of triple patterns and repeatedly runs them on the target hardware platform with parameters selected from the dataset randomly. Further, for the network functions (f_w and f_r) in the communication model, Gpl measures and stores the latency with the increase of payload sizes (power of 2), as shown in Figure 12. To predict the cost, the actual payload size is rounded to the nearest power of 2. Note that the coefficients of the cost model cannot shared by different datasets and hardware platforms, while the cost of profiling coefficients is low (see §5.6).

4.3 Budget-aware Enumeration

Observations. Prior work [18, 36, 38, 40] has reported that the heterogeneity in graph queries can result in large execution time differences. As shown in Table 1, the performance gap can even reach more than 7,000× (490ms and 0.069ms for Q7 and Q5 on LUBM-2560 accordingly). We observe that *workload heterogeneity in time requires differentiated enumeration strategies*. For long-lived queries (e.g., second-level), the accuracy of optimization is most important, so it is essential to find the best possible plan by exhaustive enumeration. For short-lived queries (e.g., millisecond-level), the optimization time may dominate the end-to-end query time, so it is better to balance the optimization time and execution time by using fast enumeration with an acceptable plan. Fortunately, the new cost model can predict wall-clock time of query processing, which opens an opportunity to directly compare the (actual) optimization time and the (predicted) execution time during query optimization.

Algorithm 2: *Pseudo-code of selecting the best possible plan within the budget of optimization time.*

```
Input: Q, SPARQL query.
       RATIO, ratio of optimization time to execution time.
1: Function Optimize(Q, RATIO)
2:   start = Now()
3:   ⟨selected_plan, least⟩ ← { NULL, MAXTIME }
4:   repeat
5:     plan = Enumerate(Q)
6:     if plan is NULL then break
7:     time = Cost(Estimate(plan))
8:     if time < least then
9:       ⟨selected_plan, least⟩ ← { plan, time }
10:    budget = least × RATIO
11:  until budget > (Now() - start)
12:  return ⟨selected_plan, least⟩
```

Budget-aware strategy. In response to workload heterogeneity, Gpl adopts *budget-aware* enumeration to optimize all kinds of queries without the advance knowledge of the execution and optimization time. Algorithm 2 outlines the enumeration strategy with a user-defined budget for an input query Q. The time budget is the ratio (i.e., RATIO) of the expected optimization time to the predicted execution time. For every valid query plan, Gpl first predicts execution time (cost) based on the cost model (Line 7), and updates the minimal cost (least) and the selected plan (selected_plan) in current enumerated plans (Line 9) if necessary. Further, the time budget could be calculated from the current minimal cost (Line 10). Note that the time budget adapts to the change of current minimal cost so that it can work well for various queries. The optimization terminates when exhausting all plans (Line 6) or the time budget (Line 11).

Moreover, the user could offer an initial ratio (e.g., 5%), and then Gpl refines it adaptively according to the workload. More specifically, Gpl would decrease it if the optimal plan has been found before exhausting the time budget and increase it if not. Gpl could enumerate all plans to find missing optimal plans in the background for sampled queries (e.g., 1%). We leave this as future work.

Gpl adopts a depth-first search (DFS) to enumerate candidate plans, rather than traditional breadth-first approach [18, 28, 32, 40], in order to obtain the *first* predicted execution time as soon as possible. Then Gpl can calculate the time budget earlier. As a result, the enumeration order (i.e., the order of triple patterns) becomes particularly important when not all query plans are estimated, namely short-lived queries. This is because the predicted execution time based on a worse plan may significantly lengthen the time budget and waste more time on query optimization.

Greedy selection. Gpl proposes a *greedy* heuristic to select the next triple pattern for DFS-based plan enumeration, aiming to find a *better* plan (i.e., shorter execution time) earlier. Gpl first estimates the cost of all valid triple patterns as the next exploration and then greedily selects the triple pattern with minimal cost. The predicted costs of all triple patterns will be buffered and reused for predicting other plans.

5 Evaluation

5.1 Experimental Setup

Hardware configuration. All evaluations were conducted on a rack-scale cluster with 8 machines. Each machine has two 12-core processors and 128GB DRAM. We dedicate one processor to run up to 10 worker threads, and use a single thread to generate requests and perform query optimization.

Comparing targets. To compare accuracy and performance of Gpl against state-of-the-art approaches, the following comparing targets are used: 1) TriAD [18] is a state-of-the-art join-centric system, which adopts a well-tuned optimizer for RDF graph and SPARQL queries. We use it to show the efficiency of exploration-based query processing and its query optimization. 2) OPT stands for the optimal query plan with minimal execution time, which was obtained by repeatedly evaluating a query with all of the valid plans, varying from tens to thousands (e.g., Q1 of LUBM has 2,496 plans). We use it to show the accuracy of query optimization. 3) Manual represents the query plans adopted by Wukong's originally-published results [36], which were manually selected by tuning results with simple heuristics.[7] We compare against it to show the necessity of cost-based query optimization. 4) Trinity.RDF [40] is an exploration-based RDF store with an initial query optimizer for graph exploration. We re-implemented it on Wukong [36] since the source code is not available. We compare against it to show the efficacy of our approach in Gpl.

Benchmarks. We use two synthetic and one real-life knowledge graphs, as shown in Table 4. The synthetic datasets include the Leigh University Benchmark (LUBM) [5] and the Waterloo SPARQL Diversity Test Suite (WSDTS) [7], which simulate the knowledge graphs in the education and retail domains, respectively. For LUBM, we generate two datasets with different sizes (up to 1.4 billion triples) and use the queries published in Atre et al. [12], which were widely used by prior work [12, 18, 24, 36, 38, 40, 41]. WSDTS publishes a total of 20 queries in four categories (**L**inear, **S**tar, Snow**F**lake-shaped, and **C**omplex), which allows evaluating Gpl with diverse queries. The real-life dataset is the DBpedia's SPARQL Benchmark (DBPSB) [3] derived from the DBpedia knowledge base. We choose 5 representative queries provided by its official website like prior work [36, 38].

[7] https://github.com/SJTU-IPADS/wukong/tree/master/scripts/sparql_query

Table 4. *A summary of RDF datasets.* #Tr*,* #S*,* #O*, and* #P *denote the number of triples, subjects, objects and predicates, respectively. (†) The size of datasets in raw NT format.*

Dataset	#Tr	#S	#O	#P	(†)Size
LUBM-2560	352 M	55 M	41 M	17	27G
LUBM-10240	1,410 M	222 M	165 M	17	108G
WSDTS	109 M	5.2 M	9.8 M	86	15.6G
DBPSB	15 M	0.3 M	5.2 M	14,128	<2G

5.2 Overall Performance and Accuracy

We first make a comprehensive study on the query performance to compare the prediction accuracy of different optimization approaches. We report the average results of one hundred runs and confirm the query plans selected carefully. Table 5 gives a complete execution time (i.e., query latency) comparison between Gpl and other setups using four different datasets and queries. Note that, for Wukong, we can obtain the optimal plan (OPT) by repeatedly running a query with all of the valid plans, while it failed for TriAD as it is not very stable for repeated execution.

For LUBM, Gpl can select optimal plans for query processing on a single machine (i.e., LUBM-2560) and incur zero performance overhead (vs. OPT), thanks to the type-centric estimation. Further, Gpl can recognize the contradictory query (Q3) during query optimization and avoid actual query processing. Strictly speaking, Gpl can even outperform the optimal results (OPT). On the contrary, the predicate-based approach (Trinity.RDF) cannot select optimal plans for the queries that involve the correlation between more than two predicates (Q1, Q3, and Q7). It incurs about 30% performance slowdown of the average (geometric mean) query latency (up to 240% for Q7). The manually tuned results (Manual) cannot guarantee the optimal results either, due to a relatively large plan space (Q7), resulting in roughly 29% slowdown. For distributed query processing (i.e., LUBM-10240), the cost of network traffic reduces the accuracy of optimization. Fortunately, the impact of Gpl on performance slowdown is trivial (1.6%), compared to Trinity.RDF (79%) and Manual (39%). Because Gpl can precisely estimate the size of data transferred and model the communication cost.

Table 5. *A comparison of optimal and selected performance (in milliseconds) on various datasets and queries. The numbers in blue denote the optimal results, and* GM *denotes geometric mean.* Gpl *can detect and skip the execution of contradictory queries (†).*

	#TP	TriAD	Wukong			
			OPT	Manual	Trinity.RDF	GPL
LUBM-2560: #T=352M and #P=17 (1 machine)						
Q1	6	588	301	523	478	301
Q2	2	148	93	93	93	93
Q3	6	347	255	255	303	†(~~255~~)0
Q4	5	3.0	0.031	0.031	0.031	0.031
Q5	2	2.1	0.023	0.023	0.023	0.023
Q6	4	28.8	0.092	0.092	0.092	0.092
Q7	6	2,317	144	498	490	144
GM		74.5	4.89	6.32	6.38	4.89
LUBM-10240: #T=1,410M and #P=17 (8 machines)						
Q1	6	3,188	72	284	99	72
Q2	2	789	48	48	814	48
Q3	6	1,279	70	128	70	†(~~70~~)0
Q4	5	3.0	0.327	0.327	0.327	0.327
Q5	2	1.8	0.073	0.073	0.073	0.073
Q6	4	98.4	0.299	0.302	0.488	0.299
Q7	6	12,509	208	292	321	232
GM		215.2	6.21	8.65	11.13	6.31
WSDTS: #T=109M and #P=86 (1 machine)						
L1	3	2.78	0.039	0.039	0.039	0.039
L2	3	4.07	0.829	0.829	4.218	0.829
L3	2	1.85	0.035	0.035	0.035	0.035
L4	2	1.74	0.444	0.570	0.566	0.444
L5	3	2.84	1.356	1.356	5.471	5.493
S1	9	12.23	0.044	0.044	0.045	0.044
S2	4	4.12	1.082	1.105	1.082	1.082
S3	4	2.09	0.285	0.346	0.285	†(~~0.285~~)0
S4	4	2.59	0.200	0.200	0.205	0.200
S5	4	3.41	0.373	0.588	1.427	†(~~0.373~~)0
S6	3	1.83	0.047	0.047	0.190	0.047
S7	3	1.38	0.031	0.031	0.031	†(~~0.031~~)0
F1	6	3.93	0.199	0.204	0.199	0.199
F2	8	18.07	0.432	0.933	0.590	0.432
F3	6	15.57	0.440	0.814	0.723	0.440
F4	9	6.95	0.442	1.343	0.675	0.442
F5	6	34.12	0.064	0.077	0.075	0.064
C1	8	16.17	0.801	0.836	1.346	0.861
C2	10	35.77	1.118	1.875	2.513	1.118
C3	6	44.59	31.48	62.44	39.54	31.48
GM		5.76	0.298	0.380	0.467	0.321
DBPSB: #T=15M and #P=14,128 (1 machine)						
D1	2	6.07	1.610	1.724	Failed	1.724
D2	3	2.61	0.023	0.023	Failed	0.023
D3	3	3.03	0.026	0.026	Failed	0.026
D4	5	4.74	0.285	5.444	Failed	0.311
D5	3	2.53	0.176	0.176	Failed	0.191
GM		3.56	0.137	0.251	--	0.144

For WSDTS, Gpl can also select optimal plans for all kinds of queries, except L5 and C1 that deviate from statistical synopses. Thus, compared to OPT, it only causes a trivial slowdown of 8%, even aside from contradictory queries (S3, S5, and S7). However, the slowdown of Trinity.RDF exceeds 57% since it selects sub-optimal plans for 14 of 20 queries. Manual still increases execution time by 28% on average, even with enormous efforts.

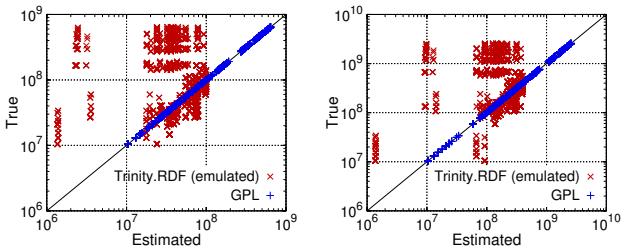

Figure 13. *A comparison of cardinality estimation btw.* Gpl *and Trinity.RDF using Q1 on (a) LUBM-2560 and (b) LUBM-10240.*

For DBPSB, due to large amounts of predicates (14,128) in the real-life dataset, the predicate-based approach (Trinity.RDF) fails to maintain the correlation statistics and find proper query plans, due to lengthy preparation time and large memory consumption. Yet, Gpl still works well with the schema-free graph data and achieves near-optimal performance (5% slowdown). D4 is a typical case that manual optimization may select an extremely poor plan (19× slowdown) due to a relatively large plan space. Hence, the query optimizer is critical to improve the performance and alleviate the burdens on programmers.

Table 6. *The q-error for cardinality estimation. The closer the* q-error *is to 1, and the more accurate the optimizer is*

LUBM-2560	Median	90th	95th	MAX
Trinity.RDF	1.785	12.928	18.755	255.952
Gpl	1.001	1.002	1.002	1.004

5.3 Cardinality Estimation

We compare the estimated cardinality between Trinity.RDF and Gpl using all plans (2496) of Q1 on LUBM datasets. The sum of the intermediate results in each exploration is recorded as the true cardinality. As shown in Figure 13, the estimated cardinality in Trinity.RDF extremely deviates from the ideal result (a straight diagonal line), where the estimated and true cardinality are equal. The main reason is that it assumes the independence among triple patterns and only considers at most two correlated predicates. In contrary, the estimated cardinality of Gpl is very close to the ideal result on both two datasets, which confirms the benefits of the type-centric approach. It can embed the correlation of multiple predicates into the type system of knowledge graphs to support accurate predictions.

We further use *q-error* [26] to measure the accuracy of cardinality estimation across different approaches, which denotes the factor by which an estimated cardinality differs from the true one. For example, *q-error* should be 10 for both the estimates as 5 or 500 if the true is 50. Table 6 shows the median (50th), 90th, 95th and max (100th) percentiles of the *q-error* for the cardinality estimation using all plans of Q1 on LUBM-2560. Gpl can achieve negligible errors thanks to the

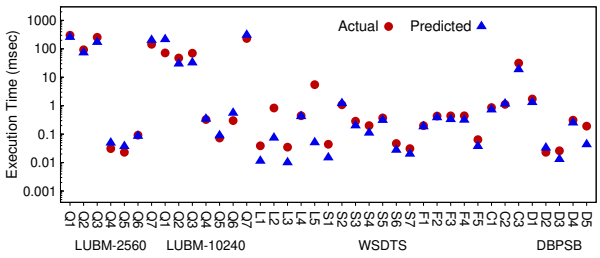

Figure 14. *A comparison between actual and predicted execution time for different datasets.*

type-centric approach. The max *q-error* is still quite close to the optimal value. In contrast, Trinity.RDF produces a very high *q-error* (up to 256× deviation).

5.4 Cost Model

To study the accuracy of the cost model for predicting execution time, we directly compare the predicted and actual execution time for all datasets and queries. Note that the actual execution time is the median of one hundred runs, and the optimization of detecting contradictory query is disabled. As shown in Figure 14, except for selecting optimal plans, Gpl can further provide a close prediction on execution time, where the median of *q-error* is 1.44. On the contrary, it is hard or even impossible to predict execution time by using prior optimizers [18, 40] or manual optimization [24, 36]. Moreover, we observe that the accuracy of the cost model is more sensitive to the type of queries, instead of the actual execution time. For example, the deviation on the linear queries (e.g., L5 in WSDTS) is relatively large.

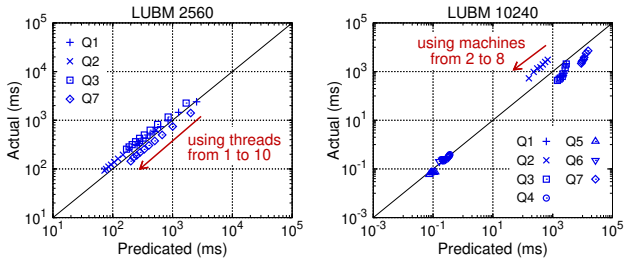

Figure 15. *A comparison btw. actual and predicted execution time for LUBM with different numbers of (a) threads and (b) machines.*

Multi-threads and multi-machines. We further evaluate the accuracy of the cost model when using multiple threads and machines. Figure 15(a) shows the results of long-lived (non-
selective) queries with multithreading. Note that the short-lived (selective) query always uses a single thread per machine in Wukong [36]. Since exploring edges is completely independent and can be fully parallelized on multiple threads, Gpl retains the accuracy with the increase of threads and ensures the median of *q-error* to 1.32. Figure 15(b) further shows the results with different numbers of machines. Gpl can still predict the execution time accurately for distributed execution, thanks to the communication part of the cost model. The median of *q-error* sightly increases to 1.82.

Table 7. *A comparison of the overall query time in milliseconds (TOTAL), consisting of the execution time (EXE) and the optimization time (PLAN), for WSDTS with and without budget-based enumeration. The numbers in red denote the increase of execution time, and the numbers in blue denote the decrease of optimization time.*

	#TP	w/o Budget			w/ Budget		
		EXE	PLAN	TOTAL	EXE	PLAN	TOTAL
WSDTS: #T=109M and #P=86 (1 machine)							
L1	3	0.039	0.049	0.088	0.039	0.026	0.065
L2	3	0.829	0.017	0.846	0.829	0.010	0.839
L3	2	0.035	0.016	0.051	0.035	0.007	0.042
L4	2	0.444	0.024	0.468	0.444	0.024	0.468
L5	3	5.493	0.021	5.514	5.503	0.011	5.514
S1	9	0.044	1.714	1.758	0.044	0.106	0.150
S2	4	1.082	0.060	1.142	1.082	0.060	1.142
S3	4	0.285	0.094	0.379	0.285	0.017	0.302
S4	4	0.200	0.062	0.262	0.200	0.026	0.226
S5	4	0.373	0.054	0.427	0.373	0.015	0.388
S6	3	0.047	0.015	0.062	0.047	0.009	0.056
S7	3	0.031	0.020	0.051	0.031	0.003	0.034
F1	6	0.199	1.395	1.594	0.236	0.072	0.308
F2	8	0.432	1.551	1.983	0.592	0.137	0.729
F3	6	0.440	0.447	0.887	0.566	0.077	0.643
F4	9	0.442	0.716	1.158	0.818	0.155	0.973
F5	6	0.064	0.254	0.318	0.077	0.071	0.148
C1	8	0.861	1.056	1.917	0.861	0.168	1.029
C2	10	1.118	7.832	8.950	7.480	0.090	7.570
C3	6	31.48	2.690	34.17	31.48	2.690	34.17
GM		0.321	0.166	0.725	0.381	0.041	0.470

5.5 Plan Enumeration

Gpl uses budget-aware enumeration to make a tradeoff between accuracy (execution time) and performance (optimization time). As shown in Table 7, Gpl can achieve the best query performance by exhausting all plans (w/o Budget). While using a fixed ratio (5% of predicted execution time), Gpl can further reduce optimization time by close to 75% (from 0.166 to 0.041), at the expense of a slight increase in execution time (about 19%), on average (geometric mean). Consequently, the budget-aware enumeration reduces the overall query latency by 35% (up to 91%), from 0.725 to 0.470. This is mainly because the time budget can be automatically adjusted with predicted execution time so far, so that it works well for all queries.

5.6 Preprocessing Cost

Different to the predicate-based approach (e.g., Trinity.RDF) that focuses on the correlation of pairwise predicates (PxP, see Figure 5), Gpl uses type-centric cardinality estimation

Table 8. *A summary of preprocessing overhead during startup for different datasets using Trinity.RDF and* Gpl. `Stat.Time/Size` *denotes the time to precompute statistical synopses and the size of them.* `Tune.Time` *denotes the time to one-off tune coefficients of cost model. (†) The actual number of types, including* virtual *types, is 1,302 and 194,212 for WSDTS and DBPSB, before using the generic type.*

	LUBM-2560	LUBM-10240	WSDTS	DBPSB
Startup Time	35m	41m	8m	5m
#Predicates	17	17	86	14,128
#Types	14	14	†69	†1,423
Trinity.RDF:				
Stat.Time	206s	684s	20s	Failed
Stat.Size	4KB	7KB	23KB	Failed
Gpl:				
Stat.Time	342s	1,175s	128s	185s
Stat.Size	6KB	13KB	40KB	3MB
Tune.Time	7.3s	15.0s	4.7s	3.0s

that relies on type-based statistical synopses (PxTxP, see Figure 9). As shown in Table 8, the preprocessing overhead of Gpl is higher than that of Trinity.RDF as expected. Fortunately, the preprocessing overhead is negligible compared to the startup time. For example, Gpl takes 432 seconds to precompute 6KB statistical synopses during 35 minutes of startup for LUBM-2560. Further, Gpl introduces a generic type GType to limit the overhead for datasets with massive predicates and types. For DBPSB, Gpl can reduce the number of types from 194,212 to 1,423 and make it practical to generate statistical synopses (185 seconds). On the contrary, Trinity.RDF is failed to support DBPSB.

In addition, Gpl only takes several seconds to one-off profiles coefficients of the cost model during startup (`Tune.Time`), like 7.3 seconds for LUBM-2560. The trivial cost is worthwhile because it enables accurate prediction of execution time. Note that Gpl leverages the generic type GType to make it practical for the dataset with a very large number of types (e.g., DBPSB).

6 Related Work

Several systems [21, 28, 37] are built atop RDBMSs to store knowledge graphs and handle queries with mature, relational optimizers. RDF-3X [32] focuses on join ordering optimization using a bottom-up strategy and uses extra statistics to optimize for frequent paths. TriAD [18] additionally tunes the cost model to be suitable for distributed execution and applies optimizer for both summary and data graph.

There is an increasing interest in using native graph model to store and query knowledge graphs [8, 16, 36, 38, 40]. Trinity.RDF [40] proposes the initial optimizer for exploration-based query processing, and uses a predicate-based scheme to capture the correlation between at most two triple patterns. SPARTex [8] also adopts a correlation method between pairwise predicates in cardinality estimation. Initially, Wukong [36] selects query plans manually by tuning results with some simple heuristics. However, for a query with relatively large plan space, the manually tuned approach may still choose sub-optimal plans and cause lengthy execution time. Our initial attempt [39] demonstrates the potential of type-centric estimation for graph query optimizer with some preliminary results. To our knowledge, Gpl is the first query optimizer for knowledge graphs that proposes exploration-based cost model and budget-aware enumeration. Kaskade [16] can also identify infeasible plans based on some constraints and prune the search space of query plans for enumeration, which may decrease the optimization time. However, its execution engine (Neo4j) still needs to run the query using a feasible view. Differently, Gpl can detect the contradictory query and skip the execution since the result must be empty, even if the query has feasible plans.

Most previous works assumed independence among join or exploration patterns and only have very simple cost models. Leis et al. [26] investigated the quality of industrial-strength optimizers for databases. The results also show that simple assumptions like uniformity and independence in most optimizers are frequently wrong and may lead to sub-optimal plans. Thus, prior work [17, 31] also proposes characteristic sets (a set of predicates of a subject/object) to improve cardinality estimation for star-shaped SPARQL queries, which can be considered a limited version of type-based statistical synopses. There are also several efforts to match one vertex of query at a time, namely joining multiple edges connecting the vertex at once [9, 11]. Further, recent work has developed query optimizers that integrate both binary and multi-way joins [29]. It would be interesting to integrate this idea into our type-centric approach. We leave this as future work.

Hasan and Gandon [22] use machine learning to learn SPARQL query performance from previously executed queries. Papailiou et al. [33] integrate workload-adaptive caching with a DP-based planner [30] to generate superior join execution plans for SPARQL queries. Alotaibi et al. [10] aim to improve the overall query performance for domain-specific knowledge graphs by optimizing graph schemas. Currently, Gpl does not consider the impact of various workloads and datasets, which will be part of our future work.

7 Conclusion

This paper presents Gpl, a fast and accurate optimizer for query processing over knowledge graphs with three key designs, including type-centric cardinality estimation, exploration-oriented cost model, and budget-aware plan enumeration. Our evaluation confirms the efficacy of our new approach compared to the state-of-the-art optimizers and manually tuned results. Further, Gpl can predict execution time accurately and detect contradictory queries without running the queries.

Acknowledgments

We sincerely thank our shepherd Rebecca Taft and the anonymous reviewers for their insightful suggestions. We also thank Youyang Yao for implementing the initial version of Gpl, and Kai Zeng for sharing his experience to design and implement optimizer for Trinity.RDF. This work is supported in part by the National Key Research & Development Program of China (No. 2020AAA0108500) and the National Natural Science Foundation of China (No. 61772335). Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] 2013. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/.

[2] 2014. Resource Description Framework (RDF). https://www.w3.org/RDF/.

[3] 2021. DBpedia's SPARQL Benchmark. http://aksw.org/Projects/DBPSB.

[4] 2021. Neo4j Cypher Query Language. https://neo4j.com/developer/cypher-query-language/.

[5] 2021. SWAT Projects - the Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.

[6] 2021. TigerGraph GSQL Query Language. https://www.tigergraph.com/gsql/.

[7] 2021. Waterloo SPARQL Diversity Test Suite (WSDTS). https://dsg.uwaterloo.ca/watdiv/.

[8] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, and Panos Kalnis. 2017. Combining vertex-centric graph processing with sparql for large-scale rdf data analytics. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3374–3388.

[9] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (2017).

[10] Rana Alotaibi, Chuan Lei, Abdul Quamar, Vasilis Efthymiou, and Fatma Ozcan. 2021. Property Graph Schema Optimization for Domain-Specific Knowledge Graphs. In *Proc. ICDE*. 924–935.

[11] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worstcase Optimal Low Memory Dataflows. *arXiv preprint arXiv:1802.03760* (2018).

[12] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. 2010. Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *Proc. WWW*.

[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proc. Usenix ATC*. 49–60.

[14] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. OSDI*, Vol. 8. 209–224.

[15] Mariano P Consens and Alberto O Mendelzon. 1990. GraphLog: a Visual Formalism for Real Life Recursion. In *Proc. PODS*. 404–416.

[16] J. F. da Trindade, K. Karanasos, C. Curino, S. Madden, and J. Shun. 2020. Kaskade: Graph Views for Efficient Graph Analytics. In *Proc. ICDE*. 193–204.

[17] Andrey Gubichev and Thomas Neumann. 2014. Exploiting the Query Structure for Efficient Join Ordering in SPARQL Queries. In *Proc. EDBT*. 439–450.

[18] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *Proc. SIGMOD*.

[19] Ralf Hartmut Güting. 1994. GraphDB: Modeling and Querying Graphs in Databases. In *Proc. VLDB*, Vol. 94. 12–15.

[20] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. 1994. A Graph-oriented Object Database Model. *IEEE Transactions on Knowledge & Data Engineering* 4 (1994), 572–586.

[21] Stephen Harris and Nigel Shadbolt. 2005. SPARQL Query Processing with Conventional Relational Database Systems. In *Proc. WISE*. 235–244.

[22] Rakebul Hasan and Fabien Gandon. 2014. A Machine Learning Approach to SPARQL Query Performance Prediction. In *Proc. WI-IAT*. 266–273.

[23] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-Time: Query Language and Access Methods for Graph Databases. In *Proc. SIGMOD*. 405–418.

[24] Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. 2019. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *Proc. EuroSys*.

[25] Pradeep Kumar and H. Howie Huang. 2016. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proc. SC*.

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215.

[27] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. 2007. Challenges in Parallel Graph Processing. *PPL* 17, 01 (2007), 5–20.

[28] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. 2016. G-SQL: Fast Query Processing via Graph Exploration. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 900–911.

[29] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-case Optimal Joins. *arXiv preprint arXiv:1903.02076* (2019).

[30] Guido Moerkotte and Thomas Neumann. 2008. Dynamic Programming Strikes Back. In *Proc. SIGMOD*. 539–552.

[31] Thomas Neumann and Guido Moerkotte. 2011. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *Proc. ICDE*. 984–994.

[32] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: A RISC-style Engine for RDF. *Proc. VLDB Endow.* (2008).

[33] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. 2015. Graph-Aware, Workload-Adaptive SPARQL Query Caching. In *Proc. SIGMOD*.

[34] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (Dec. 2017).

[35] Lei Sheng, Z Meral Ozsoyoglu, and Gultekin Ozsoyoglu. 1999. A Graph Query Language and its Query Processing. In *Proc. ICDE*. 572–581.

[36] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration. In *Proc. OSDI*.

[37] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proc. SIGMOD*.

[38] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration. In *Proc. Usenix ATC*.

[39] Youyang Yao, Jiaqi Li, and Rong Chen. 2018. Analysis and Improvement of Optimizer for Query Processing on Graph Store. In *Proc. APSys*. 6.

[40] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. In *Proc. VLDB*.

[41] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *Proc.*

SOSP.