PlanRGCN: Predicting SPARQL Query Performance

Abiram Mohanaraj
Aalborg University
Denmark
abiramm@cs.aau.dk

Matteo Lissandrini
University of Verona
Italy
matteo.lissandrini@univr.it

Katja Hose
TU Wien
Austria
katja.hose@tuwien.ac.at

ABSTRACT

Query Performance Prediction (QPP) is the task of predicting the query runtime performance prior to its execution. While QPP has been studied in relational database systems, it has received little attention for RDF stores, i.e., triplestores that are queried via the SPARQL query language. Existing methods predict the query performance based on the syntactic similarity between a given query and past queries in the query logs. This means that they are not able to generalize to unseen queries with unseen structures or characteristics. We propose a novel GCNN architecture, PlanRGCN, to generalize to unseen queries, fully exploit statistics on the stored KG, and offer more scalable pre-training than the state of the art methods. Furthermore, our architecture is the first to support non-trivial SPARQL operators. In our experiments, we demonstrate both the superior robustness of our prediction method and its practical effect on two downstream tasks: (1) load balancing, achieving a throughput improvement of up to 207% on real-world query logs and (2) execution control, processing up to 70% more queries.

PVLDB Reference Format:
Abiram Mohanaraj, Matteo Lissandrini, and Katja Hose. PlanRGCN: Predicting SPARQL Query Performance. PVLDB, 18(6): 1621 - 1634, 2025.
doi:10.14778/3725688.3725694

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dkw-aau/PlanRGCN.

1 INTRODUCTION

Query Performance Prediction (QPP) is the task of predicting the query runtime prior to its execution [3, 6, 16, 19, 20, 31, 32, 59, 61, 63]. This task can be performed by either internal or external learned database components [37]. Internal components are integrated within a database, such as query optimizers [6, 31, 32, 61, 63]. External components interact with a database without needing integration with its internal APIs, such as load balancers [16, 47], and system resource managers [20, 59]. While the former has access to much more information on the system status and execution, the latter is agnostic to the internals and thus enjoys greater portability. In this work, we focus on the second family of components.

QPP is a well-studied problem for relation databases [2, 13, 22, 23, 28, 31–33, 36, 50, 54–56, 62]. Nevertheless, QPP received much less attention in the context of RDF stores [3, 19, 59]. RDF stores are Database Management Systems (DBMS) that store Knowledge Graphs (KGs) as subject-predicate-object triples [44]. A KG example is depicted in Figure 1, where ⟨:iri1, wdt:child, :iri8⟩ is a triple, modeling the relation wdt:child from entity :iri1 to entity :iri8. SPARQL is the standard language for querying RDF data [17]. SPARQL queries (see for example Figure 2.a) include Triple Patterns (TPs) that are similar to RDF triples but can contain variables [17]. Finite sets of conjunctive TPs, i.e., TPs that may share variables, are called Basic Graph Patterns (BGP) [17]. SPARQL also supports operators like FILTER, OPTIONAL, and property paths [17]. Property paths are fundamental operators to describe paths of unbounded length and extract long-range connections between entities [1, 29].

SPARQL query complexity has increased significantly [9] and real-world query workloads reveal queries containing up to 209 triple patterns [9]. Efficient resource management for executing such complex queries in RDF stores remains a significant open challenge [5, 30, 45]. To achieve better resource management, QPP can be used, for example, to decide how to prioritize query executions in query workloads [16, 47]. For instance, the system could predict varying resource needs for queries and schedule their execution to optimize resource usage, e.g., CPU utilization.

So far, only a few works have explored the QPP problem for RDF stores [3, 18, 19, 59]. Moreover, all existing methods suffer from important design limitations. All existing methods are based on the intuition that new queries will be very similar to queries seen in the past and thus also have similar performance. Hence, they focus on computing similarities between queries, via graph edit distance [43]. Hence, new queries are compared on arrival to the representatives of past queries, and the result of this comparison is used as input for the forecasting model. These approaches suffer from three main limitations: (1) it requires either *human labor* in the selection of representative queries [59] or *computing edit distances* among all pairs of queries in the query log, which can easily reach the order of 10^8 similarity computations; (2) when completely new unseen queries reach the system, these techniques are not able to perform any reasonable prediction because of the feature representation. Moreover, existing approaches limit themselves to textual representations of the SPARQL query instead of using a more precise

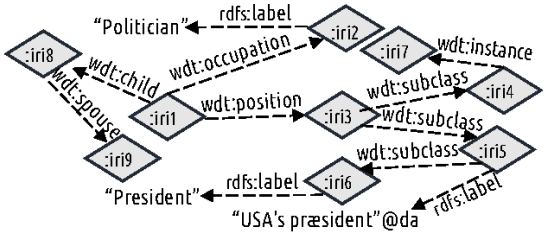


Figure 1: Example of a KG modeled as RDF graph.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725694

representation, e.g., the logical plan. Finally, existing works limit themselves to a subset of SPARQL [17] operators, thus ignoring operators with a large impact on the query execution, e.g., FILTER, regular expressions, and above all property paths.

We overcome these limitations by proposing *PlanRGCN*, which uses a novel representation model for SPARQL queries: the query graph derived by its logical plan (see for example Figure 2). Since *PlanRGCN* models the logical plan, our method is agnostic to system internals, so that the model has to implicitly also learn the choices of the query optimizer. Our experiments on two real-world query logs show that our methods result in a shorter feature construction time, up to 29 times faster. The experiments also show that our method is robust on different query logs compared to existing methods, and it is robust when extending the query graph model to property paths. Our experiments also show the practicality of predictions in a load balancer for an RDF store, resulting in an increased query throughput of 207% and 182%. Lastly, we also investigate our method's inductiveness ability, i.e., the ability to generalize to entities or relations in queries that are not observed in model training, reporting an accuracy of 87.79%.

In summary, our contributions are as follows:

- We propose a novel query plan representation technique for SPARQL QPP within PlanRGCN.
- We show for the first time how to explicitly model SPARQL operators within a RGCN, including some unsupported by existing methods (e.g., property paths).
- We show for the first time an architecture for SPARQL that can generalize beyond training queries.
- PlanRGCN effectively handles large workloads due to its feature construction approach being independent of the query log size.
- We show that PlanRGCN introduces consistent and substantial benefits to load balancing and execution control tasks.

This paper is structured as follows: we first provide an overview of preliminaries including the necessary background and the problem definition (Section 2), followed by a review of related work (Section 3). Subsequently, we give an overview of the framework (Section 4) and describe the details of our proposed method (Section 5) and its application in execution control and load balancing tasks. Lastly, we provide an analysis of the experimental results (Section 6), followed by a conclusion (Section 7).

2 SPARQL PERFORMANCE PREDICTION

We introduce the RDF data model and a SPARQL query language formalization. Then, we define the SPARQL performance prediction.

2.1 SPARQL and RDF

RDF is a graph-based data model, in which triples $\langle s, p, o \rangle$ represent facts or statements as directed labeled edges between entities s and o with relation p [57]. A knowledge graph is defined as follows.

Definition 2.1 (Knowledge Graph). Given a set of blank nodes $\mathcal{B}$, the set of IRIs $\mathcal{I}$ and the set of literals $\mathcal{L}$, an RDF triple is defined as $\langle s, p, o \rangle \in (I \cup B) \times I \times (I \cup B \cup L)$, where $s \in I \cup B$, $p \in I$ and $o \in I \cup B \cup L$. A set of RDF triples is a Knowledge Graph in RDF.

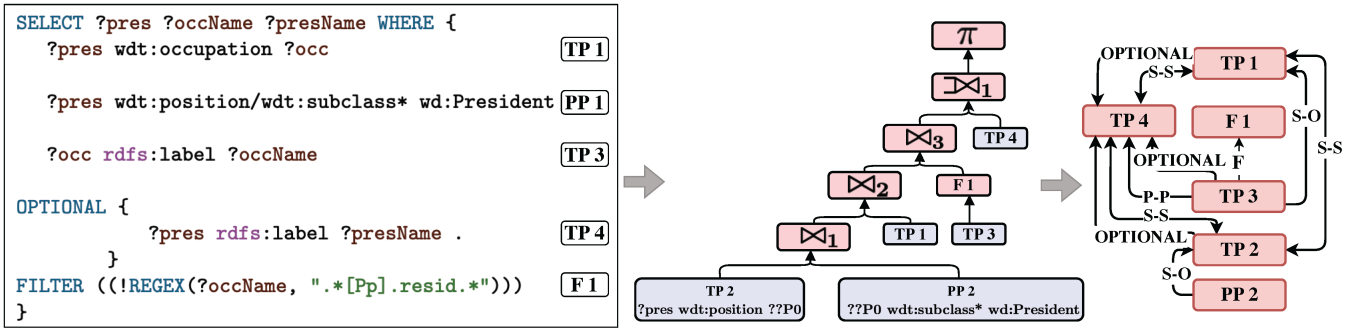
IRIs are identifiers for entities (s and o) and relations (p), blank nodes are anonymous nodes, and literals denote values. SPARQL is the standard query language, and RDF stores are DBMS for RDF data. In Figure 1, nodes with a rhombus shape represent entities, while nodes without a shape denote literals. The labels on the edges are relations. A SPARQL query example is depicted in Figure 2.a. A SPARQL query showcases a set of Triple Patterns (TPs) $\langle s, p, o \rangle$ [44], which can contain variables, combined in one or more Basic Graph Patterns (BGPs). The RDF store computes solutions for variables by matching TPs in the BGP with triples in the KG.

Definition 2.2 (Basic Graph Pattern (BGP)). Given an infinitely countable set of variables $\mathcal{X}$, a Basic Graph Pattern P is defined as a conjunction of a finite set of triple patterns $P = \{ t_1, \ldots, t_n \}$.

Additionally, property paths are arbitrary length routes in the KG and are defined using a special form of triple patterns [17].

Definition 2.3 (Property Paths [17]). A Property Path (PP) is defined as $\langle s, \mathbf{p}, o \rangle$, where $s, o \in (\mathcal{I} \cup \mathcal{X})$ and $\mathbf{p}$ is a Property Path Expressions (PPE). $\mathbf{p}$ is recursively defined as 1) $p \in \mathcal{I}$; 2) given $\mathbf{p_1}$ and $\mathbf{p_2}$, the PPE is either a sequence ($\mathbf{p_1}/\mathbf{p_2}$), a disjunctive path ($\mathbf{p_1}|\mathbf{p_2}$), a negation path ($\hat{}\mathbf{p_1}$), a sequence of zero or more of the same path ($\mathbf{p_1}*$), a sequence of one or more of the same path ($\mathbf{p_1}+$), a path of zero or one occurrence ($\mathbf{p_1}?$), or a fixed sequence between n and m occurrences ($\mathbf{p_1}\{n, m\}$).

FILTER clauses can specify further constraints on the values of variables in a BGP or TP, e.g., Line 9 in Figure 2.a. Furthermore, BGPs can be included through other operators like OPTIONAL, e.g., Lines 6-8 in Figure 2.a, which practically represent outer joins [17].



a. SPARQL Query b. Logical Query Plan c. Query Graph

Figure 2: Query Graph Construction.

Table 1: Comparison table of related work in SPARQL QPP. Feature Type: if the method uses features based on Query Log (QL) or KG. Inductiveness: the adaptability of the method on previously unseen queries. No Human Involvement: if the method does not require manual effort. Δ KG: if the method can handle updates on KG. PP Support: if the method supports property paths.

Method	Feature Type	Inductiveness	No Human Involvement	Δ KG	PP Support
SVM-based QPP [19]	QL	×	✓	×	×
2-Step SVM QPP [18]	QL	×	✓	×	×
NN-based QPP [3]	QL	×	✓	×	×
Hybrid SVM-based QPP [59]	QL	×	×	×	×
PlanRGCN	KG	✓	✓	✓	✓

2.2 Query Evaluation

During query evaluation, query parsing maps the SPARQL query to a logical query plan (see Figure 2.b).

Definition 2.4 (Query Plan). A query plan is a directed graph $p = (\mathcal{N}, \mathcal{E})$ with the node set $\mathcal{N}$ and the edge set $\mathcal{E}$. A node $n \in \mathcal{N}$ represents a logical computation and is annotated with:

- Operator type, e.g., TP, PP, or join
- Operator-specific information, e.g., predicate for TPs

Edges represent dependencies between operators/nodes. A directed edge $e = (n_1, n_2) \in \mathcal{E}$ denotes that computing operator n_2 requires the solutions computed by n_1. For simplicity, we refer to logical operators by their SPARQL operator names, e.g., TP instead of scan.

The logical query plan can be subject to optimization involving operator reordering to minimize evaluation time. In this paper, we remain agnostic to the query optimization step. Hence, we focus on queries right after their logical query plan has been produced.

Property Path Operator. During the query plan generation, PPs with multiple predicates are automatically decomposed into multiple operators each with a single predicate [17]. In particular, PPs with inversion can be transformed by replacing the subject and object positions: $^\wedge\langle s\ \mathbf{p}\ o\rangle \rightarrow \langle o\ \mathbf{p}\ s\rangle$. PPs with sequence can be rewritten into TPs with intermediate joined variables: $\langle s\ p_1\ /\ p_2\ o\rangle \rightarrow \langle s\ p_1\ e_1\rangle\ .\ \langle e_1\ p_2\ o\rangle$. PPs with disjunction can be rewritten to TPs with unions: $\langle s\ p_1\ |\ p_2\ o\rangle \rightarrow \langle s\ p_1\ o\rangle$. UNION $\{\ \langle s\ p_2\ o\rangle\}$. For instance, PP 1 in Figure 2.a is mapped to TP 2 and PP 2 in Figure 2.b. We consider plans where these rewritings have already taken place.

2.3 Problem Definition

Very often, a precise QPP is unnecessary for most external learned DBMS components [16, 47]. That is, it is often sufficient to predict if a query runtime will be fast (e.g., <1 sec), reasonably fast (e.g., <10 secs), or slow (e.g., >10 secs). Methods in query scheduling used coarse-grained runtime intervals of fast and slow queries [47]. In query optimization, the runtime intervals are useful to identify slow-running queries where spending more time on query optimization can be decided upon [42]. Runtime intervals can also be useful for various workload management tasks, e.g., for resource provisioning. Therefore, our QPP objective is to predict the runtime interval for a query within a set of m mutually exclusive time ranges. For example, given the intervals $[0, 1)$, $[1, 10)$, $[10, \infty)$ and a query with a runtime of 5.6 secs, then the associated interval is $[1, 10)$. Formally, we define the problem as a classification task as follows:

Problem 1 (SPARQL Query Performance Classification). *Given KG $\mathcal{G}$, query plan generator $\mathcal{Q}_{gen}$, past query workload $\mathcal{Q} = \{(q_j, t_j)\ |\ j \in [1, n]\}$ where t_j is the runtime for q_j, query runtime intervals $C = \{c_i \mid 0 < i \leq m \wedge c_i = [t_{i\_start}, t_{i\_end})\}$, the problem is to identify a model M that given a new query q_y can predict the query into the correct time interval $c_y \in C$ such that $t_y \in c_y$.*

3 RELATED WORK

In the following, we review the state of the art in query performance prediction for SPARQL queries and discuss its limitations. We also relate the problem of QPP in RDF stores to existing methods for SQL queries. While these methods are not directly applicable to SPARQL, we identify important intuitions that inspired our solution. Query performance prediction has also been studied in Information Retrieval (IR), but the focus there is on predicting result quality rather than response time or resource consumption [7, 15, 41].

SPARQL Query Performance Prediction. The first method to address the QPP problem is SVM-based QPP [19], which considers an encoding of a query's number of operators in the query plan and graph pattern features. The graph pattern features represent query similarity to past queries, based entirely on the BGPs. In a training phase, the queries, modeled as labeled graphs, are used to identify k representative queries through k-Mediod clustering adopting Graph Edit Distance (GED) [43]. The graph pattern features of a query are then an encoding of normalized GEDs to the k queries. This approach was later extended to a 2-step SVM with an SVM time classifier followed by class-specific SVMs [18]. This method suffers from considerable complexity in the feature extraction phase since it requires the computation of the GED across queries for k-Mediod with complexity $O(N^2)$, where N is the size of the query log. *This makes the training phase's time complexity quadratic with the number of past queries, which reaches 10^8 computations for our query logs and easily billions in existing real-world query logs [9, 30].*

Another extension of the mentioned method is hybrid SVM-based QPP [59], which enriches the query model by representing the query plan as a tree and adopting the position in the tree of each operator as another feature. This approach avoids clustering queries and expects a manual selection of representative queries. This approach cannot scale to new query logs and systems, and it is not obvious how to select representative queries from large query logs. More recently, an NN-based method [3] extended SVM-based QPP by using a neural network and extending algebra features.

Table 1 summarizes the limitations of the state of the art, where (1) the methods use features that scale with the query log or require human involvement, (2) they cannot generalize to queries with unseen entities and relations, i.e., they lack inductiveness, and (3) they do not support property paths which are crucial operators for effectively navigating KGs and therefore important for RDF stores [5, 21].

Other methods address the related problem of cardinality estimation. To this end, they use deep learning architectures with KG encodings or embeddings [11, 49]. These methods are also limited in the queries that they support. In particular, they do not support property paths, which are very expensive. Furthermore, they use KG embedding, making them inapplicable to dynamic KGs.

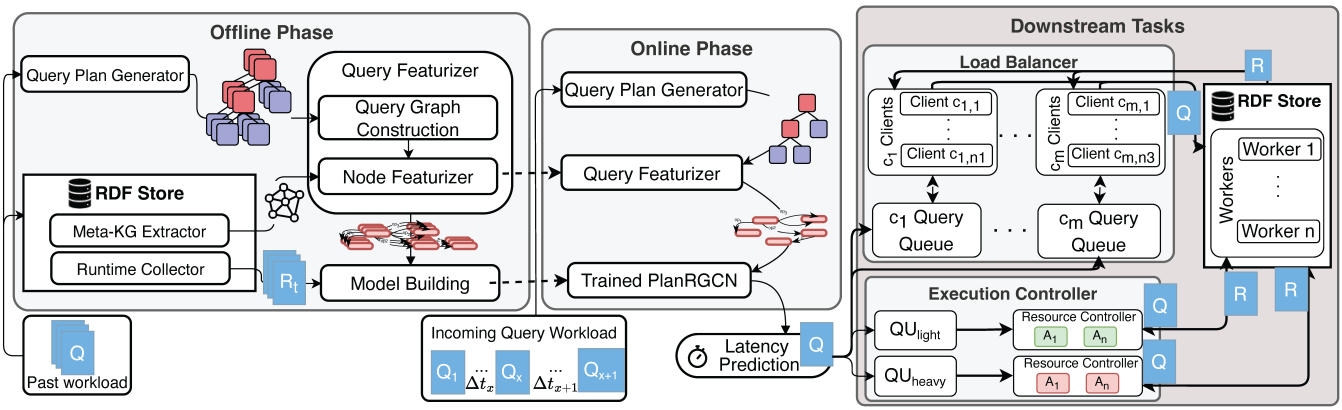


Figure 3: Overview of PlanRGCN with Q for queries, R_t for query runtime, R for query results, and A_i for available resource.

The QPP problem has also been considered for other graph databases, such as property graphs [10, 34, 46]. The methods include QPP for approximate graph analytical queries [34, 46] and graph queries for dynamic system [10]. These methods cannot be applied to RDF stores, because they use algorithm-specific features that cannot easily be extended to RDF stores [34, 46] or use features collected at runtime that are undesirable for downstream tasks [10].

SQL Query Performance Prediction. QPP approaches are divided into access-based methods [2, 13, 23, 54–56] and query plan representation learning approaches [6, 31–33, 36, 62]. Access-based methods include I/O-metric-based methods using linear regression [13], and cost-model-based methods with statistical models [2, 23] or calibration queries for estimating cost parameters [54–56].

Access-based methods are less effective than query plan representation learning methods that capture the structural features of query plans for QPP [33]. The plan-structured deep learning model [33] represents the operators in the query execution plan as nodes in a heterogeneous graph to predict query runtime. GPredictor [62] is a GCN-based architecture for predicting query latencies in a concurrent setting. It models a query workload as a graph of operators, where edges represent commonality among the operators (e.g., parent-child relation in the query plan, shared data access). The database workload characterizer [36] is a transformer model incorporating a plan structure encoder and operator-specific computational encoders to characterize workloads. It is fine-tuned towards specific tasks, including QPP. Query plan representation learning is also used by learned optimizers as a key component in their architecture using convolutional networks [6, 31, 32].

Therefore, query-plan representation learning is currently a predominant trend across existing SQL QPP methods. *Nevertheless, these methods do not directly apply to RDF stores* because they suffer from one or more of the following issues: (1) RDF data inherently differs highly from relational data because of the high heterogeneity and schema-less nature [44]. This means that the schema-based featurization of SQL QPP is unsuitable for RDF stores. (2) Many methods represent the physical query plan, rendering them implementation-dependent. This is suboptimal since RDF stores typically use very disparate graph storage techniques [44]. (3) They typically use query optimizer costs as an important input feature, a feature that has proven to be unreliable for RDF stores [11, 35].

4 FRAMEWORK OVERVIEW

To address the QPP problem, we introduce PlanRGCN: an approach that uses a novel SPARQL logical query plan representation and Relational Graph Convolution Networks (RGCN) to predict query performance. The PlanRGCN framework consists of (1) an offline phase where features are extracted from the KG and the PlanRGCN model is trained and (2) an online phase, where the model performs inference and prediction. The framework is depicted in Figure 3, along with its application in the downstream tasks. For the offline phase the steps are the following:

- The *Runtime Collector* measures and logs the runtimes of a *past query workload*, i.e., queries that previously arrived and computed by the *RDF store*. The other component that we extend the RDF store with is the *Meta-KG Extractor. It collects useful KG statistics for QPP and keeps them updated when the stored KG is updated.* We describe the statistics in Table 2.
- The *Query Plan Generator* generates logical query plans for queries in the past workload. We process the logical query plan (instead of the physical one) to ensure our solution is portable across different systems. Our solution could be further extended to become an internal component if desired. Thus, we use a standard query plan generator that is RDF store-agnostic and free of engine-specific optimizations.
- The *Query Featurizer* transforms the query plan into a featurized query graph, which is our internal representation of the query plan. Despite query plans can be considered as graphs, we notice that this representation is sparse and less suitable for learning. This observation motivates our query graph representation (described in Section 5.1). It entails: (1) query graph construction, and (2) node featurization with representations of performance-related features collected by the *Meta-KG Extractor* for each query graph node (Section 5.2).
- In the *Model Building* component, the runtimes collected by *Runtime Collector* and the featurized query graphs are used to train a PlanRGCN model.

After the offline phase, the trained model from the *Model Building* step is used in the online phase for inference. In the online phase, an incoming query is passed to the *Query Plan Generator*, which outputs the corresponding logical query plan. Next, the plan is converted to a featurized query graph and forwarded through the

trained model for prediction via a forward pass. The prediction can then be used in downstream tasks. Notably, the *Query Plan Generator* and *Query Featurizer* are shared components in the two phases. After the RDF store evaluates an incoming query in the online phase, it is added to the past workload. This means that the offline phase can be repeated periodically upon workload changes.

5 PLANRGCN

In this section, we present the key components of PlanRGCN. We first describe how we address the challenge of representing logical query plans with an RGCN model. Then, we clarify how property paths are supported in our approach. Following this, we discuss the featurization of the query graph with the collected KG statistics. Next, we elaborate on the training strategy. At last, we describe its application in downstream applications.

5.1 Query Graph Construction

We propose to model each logical query plan as a graph and employ a Graph Neural Network (GNN) architecture for representation learning in a dense vector space [60]. Similar to prior SQL QPP methods, we initially considered using query plans that naturally form trees in a GNN. However, we found this suboptimal because: *(1) node features are essential for GNNs, but extracting features for every operator in the query plan is challenging*. For instance, $\bowtie_1$ in Figure 2.b requires features describing how the results of TP 2 and PP 2 operators will be joined. Such features are not easy to collect because the operators depend on the combination of child operators, leading to scalability issues. (2) *Using an unoptimized query plan means we cannot assume fixed operator ordering*. For instance, TP 2 can also be joined with TP 1 before PP 2 (Figure 2.b). Therefore, we must model the potential connections among the operators. The first challenge for the graph representation is to identify which operators in the plan can be annotated with suitable information and design a holistic set of features applicable to nodes representing different operators. The second challenge is to model the operators' connectivity from the query plan.

Thus, we introduce a novel query graph model, where the nodes are operators with performance-dependent features. The directed edges of the query graph represent the operator connectivity that may model other operators in the query plan. We notice that Triple Patterns (TPs), Property Paths (PPs), and FILTER are operators with easily extracted performance-describing features from the KG (described in Section 5.2), making them ideal as query graph nodes. Furthermore, these query graph nodes enable different types of operator connectivity, optionally with other query plan operators, requiring labeled edges to distinguish between the connections. For instance, in Figure 2.b, TP 2 and PP 2 can join on TP 2's object and PP 2's subject. We formally define the query graph as follows.

Definition 5.1 (Query Graph). Given a query plan p, a set of operator types for nodes (O_n), and a set of operator connectivity types (O_e), the query graph is a directed, labeled multigraph $G_{qg} = (\mathcal{V}_{\text{qg}}, \mathcal{E}_{\text{qg}}, \mathcal{R}, \boldsymbol{X})$, where $\mathcal{V}_{\text{qg}}$ are nodes representing operators in p with types in O_n. The connection between query graph nodes n_1 and n_2 possibly through another operator is modeled as an edge (n_1, n_2, o_e), where o_e is the operator connectivity type in O_e and edge modeling is operator order agnostic wrt. the query plans.

Algorithm 1: Query Graph Construction

Input: query plan $p = (\mathcal{N}_p,\ \mathcal{E}_p)$, set of query graph node types (O_{node}), set of query graph edge types (O_{edge})
Output: Query Graph $G_{qg} = (\mathcal{V}_{\text{qg}}, \mathcal{E}_{\text{qg}}, \mathcal{R})$

$\mathcal{V}_{\text{qg}} := \emptyset$; $\mathcal{E}_{\text{qg}} := \emptyset$; Stack S := {root$_p$}; Visited := $\emptyset$;
while *not S.empty()* **do**
  u := S.pop() ;
  if *not* $u \in$ *Visited* **then**
    Visited.add(u);
    if $u_{type} \in O_{node}$ **then** $\mathcal{V}_{\text{qg}}$.add(u) ▷ Node Addition ;
    //Add operators to traverse in a depth-first manner;
    S.push($v_{\text{right\_anc}}$) ;
    if v_{left_anc} *is not empty* **then** S.push($v_{\text{left\_anc}}$) ;
for $i = 0; i <= |\mathcal{V}_{qg}|; i++$ **do** `// Edge Addition`
  for $z = i+1; z <= |\mathcal{V}_{qg}|; z++$ **do**
    foreach $o \in \mathcal{N}_p$ *is ancestor of* $\mathcal{V}^i_{qg}$ *and* $\mathcal{V}^z_{qg}$ **do**
      if *type of* $o \in O_e$ **then**
        $o_e = f_{con}(\mathcal{V}^i_{\text{qg}}, \mathcal{V}^z_{\text{qg}}, \text{o})$;
        $\mathcal{E}_{\text{qg}}$.add($\mathcal{V}^i_{\text{qg}}, \mathcal{V}^z_{\text{qg}}, o_e$);
    if *type of* $\mathcal{V}^z_{qg} \in O_e$ **then** `// Filter edge check`
      $o_e = f_{con}(\mathcal{V}^i_{\text{qg}}, \mathcal{V}^z_{\text{qg}})$;
      $\mathcal{E}_{\text{qg}}$.add($\mathcal{V}^i_{\text{qg}}, \mathcal{V}^z_{\text{qg}}, o_e$);
  if $\mathcal{V}^i_{qg}$ *is not part of any* $e \in \mathcal{E}_{qg}$ **then** `// Catesian product/single operator check`
    $\mathcal{E}_{\text{qg}}$.add($\mathcal{V}^i_{\text{qg}}, \mathcal{V}^i_{\text{qg}}$, cat_prod)
return $\mathcal{V}_{qg}, \mathcal{E}_{qg}$

Finally, given $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}_{\text{qg}}| \times d_n}$ representing the node feature space, the vector $x_i \in \boldsymbol{X}$ encodes the attributes of $n_i \in \mathcal{V}_{\text{qg}}$.

With TP, PP and FILTER as O_n, we consider the following connectivity types (O_e): (1) if two operators n_1 and n_2 can participate in a join, they must share a pivot, i.e., a variable part of both n_1 and n_2 [44]. The connectivity between n_1 and n_2 is represented as X-Y, where X and Y denote the variable positions of the pivot. The relative order of variable positions in the sequence S-P-O determines the edge direction between n_1 and n_2. Specifically, if the pivot position in n_1 comes before n_2's pivot position in the sequence S-P-O, the edge is directed from n_1 to n_2. For example in Figure 4 .3, we observe that TP 2's object is joined on PP 2's subject. Hence, we add the edge (PP 2, TP 2, S-O) If the same pivot position is involved in both operators, bi-directional edges are added between n_1 and n_2, labeled X-X. An example of this is depicted in Figure 4 .2, where TP 2 and TP 1 are joined on subjects, and we add the edges: (TP 2, TP 1, S-S), (TP 1, TP 2, S-S).

(2) if a $\bowtie$ is an ancestor of two operators n_1 and n_2, the connectivity is OPTIONAL. (3) if n_1 is a TP and shares a variable with FILTER operator, we add a FILTER edge from n_1. *Our graph construction ensures that alternative equivalent logical query plans map to the same query graph, which is crucial since we consider unoptimized query plans and thus do not rely on a fixed operator order.*

Algorithm 1 details the query graph construction in node addition and edge addition steps. Figure 4 depicts the partial construction of a query graph using a query plan, with grey arrows

Table 2: Extracted features for query graph node featurization. Examples of extracted features are shown for KG in Figure 1.

Feature	TP pos	Description
Frequency as predicate	p	the number of matching triples with the predicate in RDF triples, e.g., 3 for wdt:subclass
Frequency of entity	p	the number of distinct entities in KG with the predicate, e.g., 4 for wdt:subclass
Frequency of subject	p	the number of distinct subject entities in KG with the predicate, e.g., 2 for wdt:subclass
Frequency of object	p	the number of distinct object entities in KG with the predicate, e.g., 3 for wdt:subclass
Frequency of literals	p	the number of literals in KG with the predicate, e.g., 3 for RDFS:label
Overall entity frequency	$s \vee o$	the number of matching triples with the entity in the KG, e.g., 3 for :iri3
Frequency as subject	s	the number of matching triples with the entity as subject in the KG, e.g., 2 for :iri3
Frequency as object entity	o	the number of matching triples with the entity as object in the KG, e.g., 1 for :iri3
Frequency as literal	o	the number of matching triples with the literal as object in the KG, e.g., 1 for "President"

indicating traversals to identify common ancestor query plan operators. Lines 2-9 describe the node addition step through a depth-first traversal of the query plan, depicted in Figure 4.1; the query plan is traversed, and every operator instance of operator types TP, PP, or FILTER is added to the query graph node set, marked with purple circles. As this step requires a traversal of the query plan, the time complexity is $O(n)$, where n is the number of query plan operators. Lines 10-20 outline the edge addition step, where we iterate each pair of query graph nodes, added in the previous step. We use the function f_{con} to look up previously mentioned edge connectivity types. Edges are added based on the following cases: (1) if a pair of nodes share the same ancestor in the query plan, we add a labeled directed edge between them (Lines 13-15). For instance, in Figure 4.2, TP 2 and TP 4 share the common ancestor $\bowtie_1$, resulting in the edges ($TP2$, $TP4$, S-S), ($TP4$, $TP2$, S-S), ($TP2$, $TP4$, OPTIONAL). (2) if n_2 is a FILTER and involves a variable in n_1, we add an edge from n_1 to n_2 to model the connection (Lines 16-18). *Note that in this case FILTER is in both O_n and O_e, i.e., it is represented by both nodes and edges.* (3) if a node is not part of any edges, e.g., in queries with a single TP or with cartesian products, we add a self-loop on the node, to ensure proper representation in the GNN (Lines 18-19). The worst-case time complexity of edge addition is $O(n^3)$, as it involves iterating each pair of query graph nodes and an extra traversal to identify common ancestors. Thus, the time complexity of Algorithm 1 is $O(n^3)$.

By modeling a subset of the query plan operators as nodes and the operator connections among various operators in the query plan, we, thus, create a dense graph representation suitable for GNNs.

5.2 Node Features

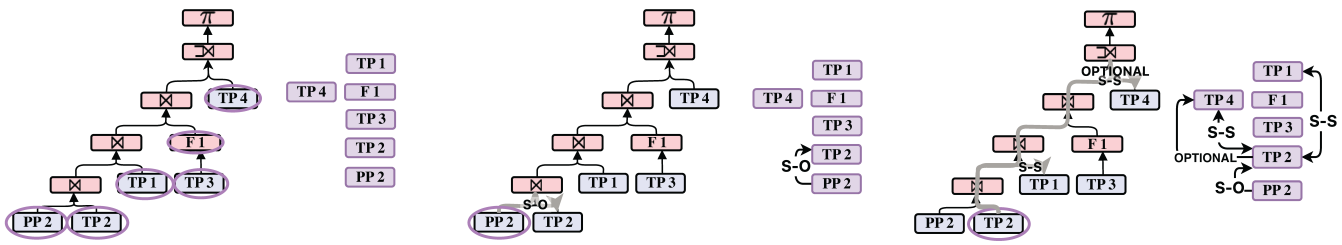
Our node featurization process collects and uses performance-dependent features. Our features include statistics from the knowledge graphs that can be recomputed without the need to retrain the method anytime the data changes.

Knowledge Graph Metadata. The *Meta-KG Extractor* (Figure 3) collects statistics that are data dependent and independent from the query log size. The statistics are used as node features in the query graph. We consider statistics that are easy to collect and maintain during RDF data loading in the system and provide operator selectivity information. Similar to statistics commonly used in query optimization [35, 40, 52], we include statistics on the subjects s, predicates p, and objects o in the KG. Our collected features are described in Table 2, where we categorize them based on their position in the triple pattern (TP pos column), i.e., if they can appear in the s, o, p or both s and o positions. The KG features describe frequency-related information of each relation, entity, and literals.

Feature extraction complexity analysis. To analyze the time complexity of the KG metadata extraction, we consider the relations, entities, and literals, separately. Computing metadata on frequencies involves collecting different frequency statistics in $O(N_{triples})$, where $N_{triples}$ is the size of the KG. For example, to compute the frequency as predicate, we count the triples matching each $p \in R$ that requires a pass over the KG. The time complexity for overall entity frequency and literal frequency similarly requires a pass over the KG. The frequency statistics that require distinct counts, e.g., frequency of subject, also require a pass over the KG where an index is needed to count only distinct information. The index can be a hash table with constant insertion and lookup time complexity, resulting in a time complexity $O(N_{triples})$.

In the current implementation, the KG metadata statistics are extracted through SPARQL queries over the fully loaded KG. *Nevertheless, this information can easily and more efficiently be collected and updated upon insertion and deletion of triples.*

Feature Encoding. We employ a quantile binning-based encoding, similar to histograms used in traditional query optimization since encoding typically yields superior performances. This binning helps group similar characteristics, enabling the model to generalize better by capturing patterns within bins. Given a bin size k



1) Node Addition **2) Edge Addition for PP 2** **3) Edge Addition for TP 2**

Figure 4: Intermediate steps of query graph construction in Figure 2

and a feature map for a particular feature, e.g., a mapping of each unique relation in the KG to the number of RDF triples matching the relation (*frequency as predicate* in Table 2), we first use the values of the feature map to define k-quantile bins, i.e., k intervals that contain an equal number of values from the feature map. The feature corresponds to a k-dimensional 1-hot encoded vector.

Vector Representation. Node features are encoded in a numerical vector $x_i \in \boldsymbol{X}$ computed for each node in the query graph model. Node vectors must have the same dimensionality. As different operators are annotated with different features, we encode the specific features for each query node type in specific positions in the vector, and the remaining positions are encoded as zero. As mentioned earlier, we distinguish between TP, PP, and FILTER as operator node types O_n in Definition 5.1, and we encode the features as follows:

$$x_i = \begin{cases} f_{tp}^{n_{tp}} \oplus [0]^{n_p} \oplus [0]^{n_f}, \text{ if } n_i \text{ is a TP} \\ f_{tp}^{n_{tp}} \oplus f_{pp}^{n_p} \oplus [0]^{n_f}, \text{ if } n_i \text{ is a PP} \\ [0]^{n_{tp}} \oplus [0]^{n_p} \oplus f_f^{n_f}, \text{ if } n_i \text{ is a FILTER} \end{cases}, \quad (1)$$

where $f_{tp} \in \mathcal{R}^{n_{tp}}$, $f_{pp} \in \mathcal{R}^{n_p}$ and $f_f \in \mathcal{R}^{n_f}$ are the feature vectors for TP, PP, and FILTER, respectively. $\oplus$ is vector concatenation. We use f^k to denote a vector f of size l (i.e., $f^l \in \mathcal{R}^l$), while $[0]^l$ is a zero vector of size l. In the following, we describe how we compute the feature vectors for each node type.

Triple pattern (f_{tp}). For nodes representing TPs, the node encoding is $f_{tp} = f_{var} \oplus f_p \oplus f_s \oplus f_o$. We first encode information about the presence of variables and the constants present in subject, predicate, and object position (f_{var}). For instance, the variable encoding for TP 1 in Figure 2 is $[0, 1, 0]$. The rationale behind this encoding is that the presence of variables affects performance. For instance, a TP with all concrete positions typically has a shorter runtime than one with only variables. Additionally, it allows encoding selectivity-based information, such as a TP with a concrete object being more selective than one with a concrete predicate [53].

We then proceed to encode the extracted KG metadata (Table 2) for the triple pattern. If the feature is not present it is encoded as 0, e.g., there is no frequency of predicate if the predicate is variable. The encoded features of p in Table 2 represents the predicate (f_p). For subjects, we consider an entity representation as the encoded features relevant to entities (f_s). For objects, f_o includes a similar entity representation as for subjects, plus a literal encoding. The literal encoding is an encoding of *frequency as literal* in Table 2 and a one-hot encoding of literal data types and language tags.

Table 3: Categorization Filter Functions

Categories	Filter Functions/Operators
Logical	||, &&
Arithmetic	+, *, /, -
Comparison	=, !=, <, >, >=, <=
General	DATATYPE, STR, IRI, LANG, BOUND, IN, NOT IN, isBlank, isIRI, isLiteral
String	STRLEN, SUBSTR, UCASE, LCASE, STRSTARTS, STRENDS, CONTAINS, STRBEFORE, STRAFTER, ENCODE_FOR_URI, CONCAT, LANGMATCHES, REGEX, REPLACE
Time	NOW, YEAR, MONTH, DAY, HOUR, MINUTES, SECONDS, TIMEZONE, TZ

Property path (f_{pp}). As previously mentioned, PlanRGCN is the first QPP approach to consider PP operators for SPARQL queries. We model PPs as an extended version of TPs as they share a similar structure. Complex PPs involving multiple predicates can be decomposed into operators with similar structures as TPs, as mentioned in Section 2. Thus, we encode the same information encoded for TPs but supplement additional features to distinguish PP nodes from TP nodes. We further encode the property paths, i.e., we distinguish the one-or-more operator + from the zero-or-more $*$, and eventually the min and max path lengths specified in the expression for fixed-length property paths. Currently, since no other existing method supports the UNION operator, we have not implemented the support for the UNION in our architecture, and thus for now we do not experiment with queries with disjunctive PPs. Our implementation can trivially be extended to UNION like the OPTIONAL support by adding another edge type in the query graph.

Filter (f_f). We perform a one-hot encoding of the various logical and arithmetic operators and function types appearing in the filter expression. The categorization of the operators and functions is specified in Table 3. The reasoning for this is that different functions have varying performance complexities, e.g., string operation is likely more costly than arithmetic ones.

Inductiveness Support. Our approach supports predictions on queries with unobserved characteristics (IRIs and operator interactions) during training. This is possible because (1) our framework uses KG statistics to featurize query plans, enabling generalization to new queries with unseen entities or relations (Table 2). (2) We employ a GCN model capable of learning operator interactions in query plans, allowing it to generalize to operator combinations that are not seen during training.

5.3 RGCN Model & Model Training

PlanRGCN uses a layered model that consists of L RGCN layers [48], followed by a readout layer and a fully connected layer. The previous layer's output is provided as input to the next layer. An RGCN [48] layer is an architecture designed for handling multi-relational graphs, i.e., graphs with different types of edges. It can be considered as a message-passing framework where a new representation of the nodes is computed in each layer. The forward propagation rule of a node v_i is defined in Equation 2 [48].

$$h_i^{l+1} = \sigma\left(\sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)} + W_O^{(l)} h_i^{(l)}\right) \quad (2)$$

where $h_i^l \in \mathbb{R}^{d^l}$ denotes the hidden state of node v_i in the l-th layer, σ is an element-wise activation function, $\mathcal{N}_i^r$ denotes the neighbors of v_i with relation r, $c_{i,r}$ is a relation-specific normalization constant, while $W_r^{(l)}$ and $W_O^{(l)}$ are weight matrices at layer l.

The input to the PlanRGCN architecture is a query graph, where the input node representations h_i^{l+1} is the node feature matrix $\boldsymbol{X}$. Following the RGCN layers, we apply a *READOUT* node aggregation to get a graph-level representation, where we use the average of the node features. The READOUT values are then forwarded through a fully connected layer with a softmax activation function.

Training the model involves adjusting the weights and bias by minimizing a loss function w.r.t. the ground truth values, i.e., the time interval to which a query belongs in our case, using gradient descent. We employ the categorical cross-entropy as the loss function. The PlanRGCN model is trained based on a training set of query plans and corresponding query runtimes. However, the distribution of query runtimes of real-world query logs is typically highly skewed [9]. We observe, in particular, that a huge portion of queries run in less than one second. To this end, we balance the loss function, so that the errors on time intervals with fewer samples are magnified in contrast to frequent time intervals. We calculate the weights by $w_c = \frac{N}{N_c * D_c}$, where w_c is the weight for class c, N is the total number of queries in the training log, whereas N_c is the number of queries with latency in interval c. D_c is the number of intervals. The loss is then multiplied by the weight w_c.

5.4 Downstream Task

We show PlanRGCN's practical usefulness in execution control and load balancing, depicted as two *Downstream Tasks* in Figure 3. Our optimization objective for the downstream tasks is to maximize throughput, i.e., the rate at which the RDF store successfully evaluates queries. We define execution control as follows:

Task 1 (Execution control). *Given a query workload Q as a sequence of pairs $\{(q_j, t_j)\}$ with t_j being query q_j's arrival time, an RDF store with n_t workers for concurrent query evaluation, the execution control task requires an algorithm $\mathbb{L}_{ec}$ that throttles resources spent on long-running queries. The controlled resources are the max execution time r_{ex} and the concurrent query evaluation slots n_{ec}.*

Similar to existing execution control methods [58], we aim to maximize the query throughput by throttling the resources for *resource-heavy* queries, e.g., queries in the slow interval, thus prioritizing resources for faster queries. To achieve this, we keep two queues: a queue QU_l for short-running queries, and another QU_h to hold long-running queries to be evaluated under limited resources. We predict the runtime interval for each query and use this prediction to assess its placement into the queues. Further, we control the resources used by the queries placed in QU_h by (1) restricting the concurrent query evaluation slots for these queries to n_{ec}. (2) Limiting the execution duration for queries in QU_h, where we interrupt the executions after a timeout r_{ex}. The goal is to free up resources spent on resource-heavy queries.

Task 2 (Load balancing). *Given a query workload Q as the set of pairs $\{(q_j, t_j)\}$, where t_j is the arrival time of SPARQL query q_j, an RDF store with n_t workers for query evaluation, n_t clients, and a time budget t, the load balancing problem requires an algorithm $\mathbb{L}_{lb}$ that decides the scheduling of queries across the RDF store workers using the n_t clients such that the workload throughput is maximized.*

The idea behind our load balancing is to prioritize query executions, where the aim is to schedule queries such that fast-running queries do not end up waiting too long for slow queries to run, by dedicating resources for each query type, i.e., runtime interval in our setting, similar to existing work [47]. Hence, we initialize a list of m queues, where each queue represents a time interval in the set of predefined time intervals, C. Whenever a new query q_i arrives, the load balancer will first use PlanRGCN with a trained model to classify the query into the time interval $c_i \in C$. Given c_i, the query is enqueued in the respective queue. Each of the n_t clients of the system is associated with a queue and by extension a time interval.

In our setting, we deploy more clients than queues, such that queues are shared amongst multiple clients. When a query is in a queue and an associated client is available, i.e., it is not currently awaiting the results of a query, the query is dequeued and processed by the associated client by sending the query to the RDF store. In addition, queues corresponding to fast-running queries should be associated with a larger number of clients so that multiple fast-running queries (and thus also queries with lower resource requirements) can run in parallel. For instance, if we instantiate 4 clients in total and assign 3 clients for fast-running queries and 1 for slow-running queries, the RDF store will process 3 fast-running queries for each slow one, provided the queues are non-empty.

The downstream tasks serve to demonstrate the effectiveness of our runtime interval predictions. Our method demonstrates the promise of runtime interval predictions. Yet, several critical issues require attention, including underutilization, e.g., if only fast-running queries arrive, the resources meant for other intervals are wasted. Yet, our prediction model will now pave the way to experiment with more advanced workload management methods in the future.

6 EXPERIMENTS

We demonstrate PlanRGCN's robustness in QPP across two datasets, including its ability to predict for queries with property paths, and for queries with completely unseen characteristics. Furthermore, we demonstrate the real-world applicability of our QPP method by studying its effect on downstream tasks. We also evaluate alternative runtime intervals. Finally, we report on its efficiency by evaluating the preprocessing and training time and disk space.

6.1 Experimental Setup

Experiments are conducted on a Dell R6415 with 256GB RAM and an AMD 7281 CPU with 16 cores. We use Virtuoso [14] as our reference RDF store. We use Jena ARQ [52] as our query plan generator. The query runtime intervals that we consider in the experiments are $C = \{(0; 1], [1; 10], (10; \infty]\}$. We use RayTune [24] with a 24-hour limit to identify hyperparameters. Our optimal PlanRGCN setting is a bin size of 50 (feature encoding), a learning rate of $1e^{-5}$, and 2 RGCN layers. The layer sizes are 4096 and 1024 for DBpedia and 4096 and 2048 for Wikidata. The input vector size is 485.

Data. We use DBpedia and Wikidata KGs that can contain billions of triples and thousands of distinct predicates. The DBpedia KG is constructed from the source files from October 2016 [12]. We use the same English Wikidata snapshot from 2021 used in WDBench [5]. The statistics of the KGs are shown in Table 4.

Query Log. The query log for DBpedia and Wikidata is split into training, validation, and test sets (see Table 5). Queries are extracted

Table 4: KG statistics

	DBpedia	English Wikidata
# of triples in KG	6,101,406,781	1,253,528,123
# of distinct predicates in KG	167,762	8,604
# of distinct subjects in KG	744,417,316	92,498,623
# of distinct objects in KG	639,693,357	305,419,412

Table 5: Query log distribution

	Training			Validation			Test		
	$(0;1]$	$(1;10]$	$(10;\infty]$	$(0;1]$	$(1;10]$	$(10;\infty]$	$(0;1]$	$(1;10]$	$(10;\infty]$
DBpedia									
With 1 TP	5078	143	581	1673	41	184	1702	107	376
With 2 TP	786	125	627	264	56	214	248	48	233
With > 2 TP	1345	1227	936	465	390	327	407	459	314
With FILTER	2071	1393	1835	727	452	628	677	508	624
With OPTIONAL	430	1048	1146	143	347	393	151	402	389
With PP	3754	54	56	1242	20	26	1250	168	276
Total	**7210**	**1495**	**2144**	**2402**	**487**	**725**	**2358**	**614**	**923**
Wikidata									
With 1 TP	6383	236	236	2147	82	68	2084	83	79
With 2 TP	5777	160	288	1794	66	90	1860	48	110
With > 2 TP	5033	195	447	1766	58	179	1760	68	160
With FILTER	4945	221	101	1631	75	39	1670	68	29
With OPTIONAL	3459	138	305	1173	43	119	1114	43	104
With PP	6576	444	326	2194	158	94	2156	170	124
Total	**17193**	**591**	**971**	**5704**	**206**	**337**	**5702**	**199**	**349**

Table 6: Confusion Matrices for query logs.

DBpedia

	P			NN			SVM			# Total
	$(0;1]$	$(1;10]$	$(10;\infty]$	$(0;1]$	$(1;10]$	$(10;\infty]$	$(0;1]$	$(1;10]$	$(10;\infty]$	
$(0;1]$	90.2	2.1	7.7	96.6	2.5	1.0	**99.7**	0.3	0.0	2358
$(1;10]$	4.9	**80.2**	14.9	4.7	35.3	59.9	50.9	49.1	0.0	614
$(10;\infty]$	2.7	27.3	70.0	10.2	16.5	**73.3**	59.2	40.8	0.0	923

Wikidata

	P			NN			SVM			# Total
	$(0;1]$	$(1;10]$	$(10;\infty]$	$(0;1]$	$(1;10]$	$(10;\infty]$	$(0;1]$	$(1;10]$	$(10;\infty]$	
$(0;1]$	81.2	11.5	7.4	**98.8**	0.8	0.5	95.4	3.4	1.2	5702
$(1;10]$	10.1	**64.3**	25.6	82.4	11.1	6.5	62.3	30.7	7.0	199
$(10;\infty]$	9.7	18.6	**71.6**	42.7	16.6	40.7	42.1	8.3	49.6	349

from LSQ [51], which contains real-world queries submitted against those KGs. We select valid queries involving the operators: TPs, PPs, OPTIONAL, and FILTER. Overall, the real-world query logs feature a limited number of valid queries adopting the property path operator. For Wikidata, we expand the query log with queries from WDBench [5] and queries from a different query log [30] that contained queries that were recorded as timed-out when queried over the public Wikidata service. We obtained 18K queries for DBpedia and 31K for Wikidata.

We ran all queries three times in random order. The average query runtime is used for model training. The total runtime collection time was ~14 days for DBpedia and ~9d for Wikidata. We produce a 60%-20%-20% train/validation/test split by stratified sampling based on 3 classes: $(0;1]$, $(1;10]$, and $(10;\infty]$ on query runtime.

To ensure sufficient representation of complex query types, we generated queries involving PP and unseen properties, as these showed lower cardinality. We design templates for generating PP queries based on property path patterns in conjunction with operators commonly found in query logs [8]. Entities and relations from the training set instantiate the templates. For unseen queries, we create templates with complex structures, such as star and path patterns noted in query logs [9]. The templates are instantiated by extracting entities and relations from the KG that are separate from the training and validation set. To emphasize complexity, we design queries with >2 TPs, incorporating operators like OPTIONAL and FILTER. We also report that < 0.5% of queries have runtimes within 10% of the thresholds in the test sets.

QPP Baseline methods. We compare our approach against state of the art methods, namely SVM-based QPP [19] and NN-based QPP [3]. We used the existing implementation provided by NN-based QPP [3]. Yet, we improved the distance matrix computation in their feature construction step with parallelization. We ensured that our implementation produces the same numbers as the original.

We do not compare against Hybrid SVM-based QPP [59] since we were unable to access their code base nor receive the code from the authors. Furthermore, we note that the method requires hand-selected queries for its feature generation, thus limiting its application to only regular and stable query logs.

6.2 Query Performance Prediction

Since we model our problem as a multiclass classification task, we present our results in confusion matrices. An entry in row i and column j represents the normalized ratio of queries where the actual runtime interval was c_i, but the model predicted interval c_j. The ideal method would show 100 on the diagonal and 0 outside.

Query Performance Prediction. We investigate the performance of our method against the mentioned baseline methods and report the confusion matrices in Table 6 for both datasets.

For the DBpedia dataset, our method demonstrates greater robustness in predictions across various time intervals compared to the best baseline, the NN-based QPP. Specifically, the NN-based QPP achieves 6.4% higher accuracy for queries in the $(0;1]$ time interval and a modest 3.3% improvement in the $(10;\infty]$ interval. In contrast, our method, PlanRGCN, outperforms by 44.9% in the $(1;10]$ interval. Given the negligible difference of 3.3% in $(10;\infty]$, the key distinction between our method and the NN-based QPP is clear: while the NN-based QPP is superior in the $(0;1]$ interval, PlanRGCN's performance boost over the NN-based QPP in the $(1;10]$ interval is three times that of the NN-based QPP's improvement over PlanRGCN in the $(0;1]$ interval.

Additionally, it is crucial to consider the overall model performance levels. For instance, a performance difference between 90.2% and 96.6% is less significant than one between 80.2% and 35.3%. The former range indicates reasonably high performance, whereas the latter suggests poor general performance, further highlighting the robustness of PlanRGCN. The experiment also hints that the NN-based QPP is proficient in predicting course-grained intervals in the $(0;1]$, as the 96.6% are predicting correctly while its mispredictions in $(1;10]$ and $(10;\infty]$ typically fall into the slower intervals. Hence, in predictive quality for DBpedia, there is a trade-off depending on how specific the intervals need to be for the downstream task.

SVM-based QPP is biased towards predicting the $(1;10]$ and $(10;\infty]$ intervals, as reflected by the high values in the $(1;10]$ SVM column in Table 6.DBpedia and thus is not effective. We report a macro F1 of 0.78, precision (PR) of 0.76, and (R) of 0.80 for our method, showcasing the robustness across intervals compared to NN (F1: 0.68, PR: 0.69, R: 0.68) and SVM (F1: 0.45, PR: 0.41, R: 0.50).

For Wikidata, we report an F1 of 0.53, PR of 0.50, and R of 0.72 for our method, showing performance comparable across time intervals to NN (F1: 0.55, PR: 0.64, R: 0.50) and SVM (F1: 0.59, PR: 0.62, R: 0.59). The experimental results in Table 6 demonstrate the superiority of PlanRGCN over the baselines, particularly in the $(1;10]$ and $(10;\infty]$ time intervals. Specifically, PlanRGCN predicts more than twice the number of queries in the $(1;10]$ interval compared to the SVM-based QPP, which is the best-performing baseline, with a performance difference of 33.7%. Moreover, in the $(10;\infty]$ interval,

Table 7: Confusion Matrices on PP queries and unseen queries

		P			NN			SVM			# Total
		$(0;1]$	$(1;10]$	$(10;\infty]$	$(0;1]$	$(1;10]$	$(10;\infty]$	$(0;1]$	$(1;10]$	$(10;\infty]$	
DBpedia (PP)	$(0;1]$	95.8	0.9	3.3	**100.0**	0.0	0.0	82.6	17.4	0.0	552
	$(1;10]$	22.9	4.8	72.3	100.0	0.0	0.0	0.0	**100.0**	0.0	83
	$(10;\infty]$	10.1	0.0	**89.9**	100.0	0.0	0.0	0.7	99.3	0.0	138
Wikidata (PP)	$(0;1]$	54.9	39.9	5.3	**99.4**	0.4	0.2	91.1	8.9	0.0	966
	$(1;10]$	8.5	**77.5**	14.1	87.3	9.9	2.8	53.5	43.7	2.8	71
	$(10;\infty]$	8.2	75.4	**16.4**	73.8	13.1	13.1	73.8	24.6	1.6	61
DBpedia (unseen)	$(0;1]$	**93.9**	4.1	2.0	81.8	18.2	0.0	0.0	100.0	0.0	148
	$(1;10]$	40.0	40.0	20.0	10.0	40.0	50.0	0.0	**100.0**	0.0	20
	$(10;\infty]$	0.0	0.0	**100.0**	21.7	76.5	1.7	0.0	99.1	0.9	115

PlanRGCN correctly predicts 71.6% of the queries, outperforming the SVM-based QPP by 22.1%. In general, the baseline methods struggle to predict runtime intervals. We hypothesize that this is because the baseline methods learn significantly from the query distribution instead of the query features, leading to a bias towards the most frequent time interval, $(0;1]$. This bias is particularly evident in column $(0;1]$ of Wikidata in Table 6, where the baselines mainly predict $(0;1]$, resulting in suboptimal performance in other time intervals. Consequently, our method demonstrates a significant advantage over the baselines by effectively delivering more accurate predictions across different time intervals.

Moreover, the baseline methods exhibit inconsistent performance across different query logs. The NN-based QPP is the superior baseline on DBpedia, while the SVM-based QPP shows superior results on Wikidata. *In conclusion, PlanRGCN proves to be robust in predicting runtime intervals across two real-world query logs.*

Property Path Query Performance Prediction. Table 7 presents the predictive performance on queries in the test set containing PP queries for both datasets. To study the PP operator, we exclude unseen queries — those with relations and entities not observed during training. We first analyze the PP queries in DBpedia (Table 7.DBpedia PP), where our method achieves >90% accuracy in predicting $(0;1]$ and $(10;\infty]$, but it does struggle with the $(1;10]$ queries, predominantly predicting them as $(10;\infty]$. On the other hand, the NN-based QPP, which performed competitively on the overall query log, predicts all PP queries to be in the $(0;1]$ interval, as shown by the $(0;1]$ column in Table 7.DBpedia PP, rendering it ineffective. The SVM-based QPP provides a competitive performance on property path queries. Nevertheless, it still proved ineffective for $(10;\infty]$ with no correct predictions. While our method predicts 13.2% more queries accurately in $(0;1]$, the SVM-based QPP predicts almost twice as many queries in $(1;10]$, pointing to a bias. The $(10;\infty]$ runtime interval is challenging for both baseline methods. However, our method correctly predicts 89.9% of the queries in $(10;\infty]$, underlining the potential of our method.

On the PP queries in Wikidata (Table 7.Wikidata PP), PlanRGCN shows lower accuracy in $(0;1]$. In comparison, the baselines correctly predict most queries in $(0;1]$ but are highly biased to predict queries in $(0;1]$, as seen in Table 6. Thus, the baselines' performance in this interval does not accurately reflect their predictive capabilities. We also observe that the most frequent relations in the 39.6% of queries in $(0;1]$ predicted as $(1;10]$ by PlanRGCN overlap with relations used in queries in $(1;10]$. This results in our methods' mispredictions due to its reliance on predicate statistics (Table 4). On the other intervals, PlanRGCN has a clear advantage with an improvement of 31.8% for $(1;10]$ and a smaller improvement of 6.5% for $(10;\infty]$ compared to the best baseline.

The QPP methods indeed have a harder time predicting the runtime interval for queries with property paths, confirming how challenging it is to predict the effect of the presence of this operator. Yet, with this still being the case, PlanRGCN still performs reasonably compared to the baseline QPPs on the property paths queries. Moreover, our method consistently demonstrates effectiveness across both datasets, as none of the leading baselines on the full query logs (Table 6) excelled in property path queries. For instance, while NN-based QPP is a better baseline on the full DBpedia query log, the SVM outperformed it for PP queries in DBpedia. Similarly for Wikidata, although the SVM-based QPP is the more accurate baseline on the full query log, the NN-based QPP is the more effective baseline on PP queries.

This demonstrates PlanRGCN's capability to extend to queries including property paths to predict runtimes compared to the baselines.

Predictions for Unseen Queries. Our architecture is designed to also predict the performances of unseen queries, i.e., queries with relations and entities that have not been seen during training. Since our method is far superior than the baselines on Wikidata, we report only on unseen queries for DBpedia. We also exclude unseen queries with property paths for fair comparison on specifically the inductiveness aspect alone. (Table 7.DBpedia Unseen) reports the results of unseen queries. Our method demonstrates robust performance in predicting runtime intervals, achieving > 93.9% accuracy for $(0;1]$ and $(10;\infty]$ queries and matching baseline accuracy on $(1;10]$. This consistency across all intervals underscores its robustness for unseen queries. In contrast, the NN-based QPP, while accurate for $(0;1]$, still falls 12.1% short of our method, and struggles to distinguish between $(1;10]$ and $(10;\infty]$, frequently mispredicting them. Notably, it critically mispredicts ~20% of $(10;\infty]$ as $(0;1]$. The SVM-based QPP fails entirely on unseen queries, defaulting to predicting $(1;10]$. Upon closer inspection, we observe that some of the incorrectly predicted queries in $(1;10]$ contain unbounded TPs, i.e., a TP solely consisting of only variables. Since our method relies on relation and entity statistics, this behavior on unbound TPs is expected. *The results clearly show that our method can robustly predict runtime intervals for unseen queries.*

6.3 Downstream Tasks

Workload Setup. To simulate the query arrival, we use an exponential distribution that, given an arrival rate, μ, samples arrival times for each query to generate synthetic and controllable arrival times for all test queries. We use μ of 44 queries/sec, the reported as the arrival rate of the Wikidata endpoint [30]. We use 10 workers and clients and a time budget for the downstream tasks of 2 hours.

Execution control. We consider queries with performance in $(10;\infty]$ as the resource-heavy queries, 95%-quantile of the query log as r_{ex}, and 3 clients for resource-heavy queries (n_{ec}). The 95%-quantile is 51 secs for DBpedia and 1.3 secs for Wikidata. In Table 8, we report the throughput and the "Falsely throttled queries", which are queries that were erroneously throttled and interrupted due to misprediction. The results demonstrate that PlanRGCN processes 69.5% more queries compared to the best baseline on Wikidata (SVM-based QPP). On DBpedia, PlanRGCN processes 4.1% queries

more compared to the best baseline (NN-based), while PlanRGCN produces fewer mispredictions compared to the same.

Load Balancing. We assign 5 clients for $(0; 1]$, 3 for $(1; 10]$, and 2 clients for $(10; \infty]$ intervals. We aim at (1) maximizing throughput and (2) minimizing the query latency, which is the total response time from when the query arrives at the load balancer until its execution completes. This includes waiting time for a worker to become available, and, when applicable, query plan extraction, query graph construction, and inference time. We report that the average PlanRGCN inference time is 0.01 secs.

Additionally, for this experiment, the effect of Head of Line Blocking (HLB) should also be minimized, i.e., a disproportional increase in query latency of fast-running queries compared to their execution time [47] due to waiting in line.

We evaluate our method against a FIFO-based query scheduler to compare it against a standard load balancer. This baseline keeps a single queue of incoming queries and schedules the queries to workers depending on availability. We also consider load balancing with predictions from the NN-based QPP and SVM-based QPP baselines. To further compare against the best achievable predictor, we compare against a load balancer that employs an oracle, which provides the ground truth time interval for a query.

The FIFO load balancer is significantly outperformed by our PlanRGCN-based load balancer. Using PlanRGCN's predictions, the query throughputs shown in Table 9 increase by 183% and 207% compared to when the FIFO load balancer is used on Wikidata and DBpedia, respectively. Additionally, Figure 5 shows that query latencies for queries in the $(0; 1]$ interval are significantly reduced with PlanRGCN, thereby minimizing HLB. *This demonstrates that runtime interval predictions offer significant advantages for the load balancing task compared to a standard FIFO load balancer.*

For the QPP-based load balancers, i.e., our PlanRGCN, the NN-based QPP, and the SVM-based QPP, we observe a similar trend to the predictive performance of the baseline methods. Specifically, the NN-based QPP is the superior baseline on DBpedia, while the SVM-based QPP performs better on Wikidata, highlighting the robustness issues in the baseline methods.

On the DBpedia dataset, our PlanRGCN-based load balancer enables the RDF store to process 20% more queries than the NN-based load balancer. The query latency also shows promising results, with lower latencies for queries in $(0; 1]$ compared to other methods. Although the SVM-based load balancer has lower latencies in $(0; 1]$, this advantage comes at the cost of significantly reduced throughput. On the Wikidata dataset, the RDF store processes 6% fewer queries with our PlanRGCN-based load balancer compared to the SVM-based one, which translates to roughly 4 fewer queries per minute. Despite this reduction in query throughput, we observe a significantly lower effect of HLB for the PlanRGCN-based load balancer (Figure 5). *Overall, compared to the baselines, PlanRGCN exhibits superior robustness in the effect of the runtime interval prediction in the downstream task, providing evidence of its effectiveness.*

The Oracle load balancer reveals that there are still optimization opportunities to explore. Interestingly, the SVM-based load balancer achieves a higher query throughput than the Oracle load balancer, which can be attributed to the issues mentioned in Section 5.4 and motivates the need for an algorithm that also handles

Table 8: Results of execution control, where query/sec is the throughput, and FTQ is the false throttled queries

		query/sec	FTQ		query/sec	FTQ
PlanRGCN	Wikidata	**223.41**	7	DBpedia	**18.27**	43
NN		100.08	7		17.53	45
SVM		131.78	**6**		14.92	**39**

Table 9: Query throughput (query/min.)

	DBpedia	Wikidata
FIFO	6	17
PlanRGCN	18	47
Oracle	**24**	49
SVM-based QPP	7	**50**
NN-based QPP	14	22

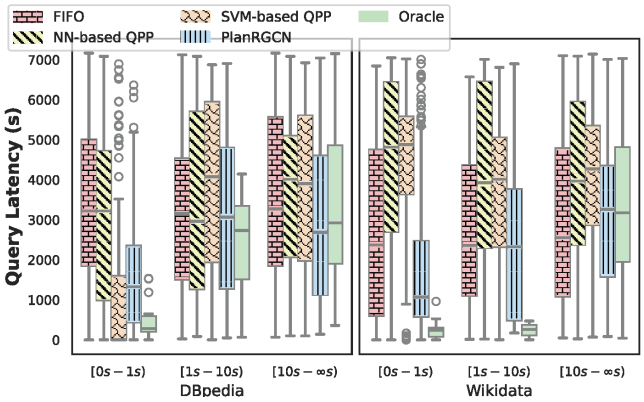


Figure 5: Query Latency by Time Interval

scenarios where query executions diverge from the predictions. Further analysis shows instances where the intervals change in concurrent settings, e.g., 318 queries in $(0; 1]$ have a runtime in $(1; 10]$, highlighting the challenges mentioned in Section 5.4.

We have demonstrated PlanRGCN's potential in two downstream tasks. However, effectively utilizing these predictions in practical applications remains a key consideration that we leave as future works.

6.4 Alternative Time Intervals

We further investigate our model's performance on other time intervals. We first investigate the accuracy of our model when the time intervals become increasingly fine-grained. As the majority of our queries are fast, we add a threshold of 4 msec to separate very fast from fast queries. It would also be instrumental to determine if a query will time out, e.g., to reject the query. Hence, we also consider queries that time out as a category on its own. Table 10 shows the results of the fine-grained interval. Our model can effectively predict the queries timing out, whereas the baselines do not. Our method also performs reasonably on the 4 msec threshold intervals, $(0; 0.004]$ and $(0.004; 1]$, where it struggles on Wikidata $(0.004; 1]$ with a performance of 45.2%. This shows that PlanRGCN can potentially be extended to more fine-grained intervals.

We further evaluated runtime intervals using 50%- and 95%-percentile runtime of the query log as alternative thresholds in Table 11. On Wikidata, PlanRGCN consistently achieved >75% accuracy across the runtime intervals, outperforming both baselines. On DBpedia, PlanRGCN maintains >51% accuracy overall, excelling

Table 10: Fine grained runtime intervals

		P					NN					SVM					# Total
		I1	I2	I3	I4	I5	I1	I2	I3	I4	I5	I1	I2	I3	I4	I5	
Wikidata	I1: (0; 0.004]	70.2	20.8	0.22	5.10	3.69	32.4	66.9	0.57	0.13	0.0	**96.1**	2.1	1.2	0.6	0.0	3139
	I2: (0.004; 1]	25.0	43.8	3.4	18.8	8.9	4.3	**93.7**	0.9	1.0	0.0	83.0	9.0	6.1	2.0	0.0	2563
	I3: (1; 10]	2.0	10.1	15.1	55.3	17.6	0.0	80.9	15.1	4.0	0.0	35.2	27.1	**30.7**	7.0	0.0	199
	I4: (10; 900]	2.4	6.1	0.6	**82.3**	8.5	0.3	43.0	17.1	39.6	0.0	29.0	13.1	8.8	49.1	0.0	321
	I5: Timeout	0.0	9.5	0.0	28.6	**61.9**	0.0	42.9	9.5	42.9	4.8	38.1	4.8	0.0	57.1	0.0	28
DBpedia	I1: (0; 0.004]	**76.2**	20.4	0.0	1.3	2.2	2.2	97.7	0.0	0.2	0.0	40.0	38.2	21.8	0.0	0.0	555
	I2: (0.004; 1]	1.7	92.2	0.6	4.3	1.2	0.1	**95.4**	3.2	1.2	0.0	1.0	5.0	93.6	0.3	0.0	1803
	I3: (1; 10]	0.0	15.1	**68.6**	8.9	7.4	0.0	4.7	35.3	59.9	0.0	0.0	0.2	50.7	49.1	0.0	614
	I4: (10; 900]	0.0	13.1	27.2	44.4	15.4	0.0	9.9	15.4	**74.7**	0.0	0.0	0.3	57.8	41.9	0.0	886
	I5: Timeout	0.0	8.1	0.0	5.4	**86.5**	0.0	16.2	37.8	40.5	5.4	0.0	0.0	81.1	18.9	0.0	37

on slower Q3 queries with 78.2% accuracy. However, the Q2 interval is more difficult for our method. *Still, these results highlight PlanRGCN's robustness and adaptability across datasets and runtime intervals, particularly on challenging long-running queries.*

6.5 Preprocessing and Training Time

We investigate the preprocessing time for our approach and then the model training time. We extract the meta-KG statistics described in Table 2 by sequentially querying the RDF store, resulting in the use of a single CPU. Our method's total feature extraction duration is ~28 min for Wikidata and 1 h 23 min for DBpedia.

For the baselines' feature extraction time, the CPU time for *Query Operator Statistics*, i.e., encoding of a query's number of operators, is ~0.2 min. In comparison, the graph pattern features took significantly longer, at 4772 min for DBpedia and 13608 min for Wikidata. As previously mentioned, we revised this component to heavily exploit parallel execution, where we used 20 CPUs to compute the query distance matrix used in the previously mentioned feature construction. The wall clock times, i.e., the actual duration of the feature construction, for the baseline methods are 4 h 37 min and 13 h 56 min for DBpedia and Wikidata, resulting in 3.3 and 29.4 times more wall clock time compared to PlanRGCN. To further compare against our method under more equal settings, we report the total CPU time: 79 h 31 min and 226 h 48 min for DBpedia and Wikidata, respectively. In CPU time, this means that our method's feature construction time is 58 times and 478 times less computationally intensive than the baselines' feature construction time for DBpedia and Wikidata. For DBpedia, model training time takes 1 h 4 min 23 secs, 23 min 43 secs, and 6 min 1 secs for PlanRGCN, NN-based QPP, and SVM-based QPP, respectively. Similarly for Wikidata, the training time is 1 h 32 min 24 secs, 26 min 34 secs, and 42 min 42 secs. *Despite the shorter baseline training time, our method is still at least 1.9 times faster for DBpedia and 7 times faster for Wikidata in combined preprocessing and training time (wall clock time), even using fewer CPUs. Furthermore, it should be stressed, as mentioned in Section 5.2, that the collection of meta-KG statistics should be implemented in the RDF store and update the information upon KG updates. Therefore, in a continuously evolving system, our approach can adapt to changing workloads, whereas the baseline works need a separate step for constructing the graph pattern features.*

Next, we compare the duration for computing the features and making predictions with the appropriate model, i.e., the mean online inference time. We can report that the mean online inference time is 0.013 secs and 0.011 secs for PlanRGCN on DBpedia and Wikidata, respectively. For NN-based and SVM-based QPP, it is 0.027 secs and 0.022 secs. *Overall, PlanRGCN's inference times are much shorter than the baselines by 50%-52% on the datasets.*

Table 11: 50%- and 95%-percentile runtime intervals on Wikidata (Wiki) and DBpedia (DBped)

		P			NN			SVM			# Total
		I1	I2	I3	I1	I2	I3	I1	I2	I3	
Wiki	I1: (0; 0.004]	75.6	20.4	4.0	30.5	69.0	0.4	**96.7**	1.8	1.5	3147
	I2: (0.004; 1.3]	2.2	91.9	5.9	4.8	**93.1**	2.1	79.7	6.7	13.6	2637
	I3: (1.3; ∞]	0.0	4.8	**95.2**	0.0	56.0	44.0	30.4	10.1	59.5	514
DBped	I1: (0; 0.04]	85.2	12.6	2.2	**89.3**	10.6	0.1	0.6	99.4	0.0	1788
	I2: (0.04; 50.8]	13.3	51.4	35.3	12.9	**85.7**	1.4	0.3	99.7	0.0	1688
	I3: (50.8; ∞]	1.9	19.9	**78.2**	1.4	79.0	19.6	0.0	100.0	0.0	419

6.6 Feature Construction Space

We investigate the disk space used by the collected features of our method compared to the baselines. On the space complexity, our method's features use $O(N_{triples})$, whereas the existing methods use $O(N_{ql}^2)$. In terms of disk space, the existing methods use 229G and 64G for the distance calculation for Wikidata and DBpedia, respectively. On the other hand, our method uses 0.54G and 5.1G to store the features in Table 2 for Wikidata and DBpedia, respectively. *This means that our method uses 424 and 12 times less disk space compared to the baseline methods for the datasets.*

7 CONCLUSION

This paper presents a novel approach for predicting the performance of SPARQL queries based on graph representation learning. Our method significantly diverges from existing techniques by not only considering similarities to past queries but also collecting features natively extracted from the RDF stores' statistics computation. We evaluate our method and show robust performance on two real query logs over real-world large-scale KGs. We further show that our method can support other important previously unsupported SPARQL operators, namely property paths. Our model can be further extended by adding new node and edge types to accommodate additional operators. Importantly, we also show that our model is capable of predicting the performances of queries showcasing entities and relations unseen during training. Furthermore, we demonstrate that the runtime interval predictions produced by PlanRGCN are useful in practice for load balancing and execution control tasks. In the future, we aim to implement support for KG updates [38, 39], more operators, and more advanced load balancing and execution control algorithms, incorporating our runtime interval prediction model. Our results will be an important component for systems enabling KG analytics and exploration [4, 25–27].

ACKNOWLEDGMENTS

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B and the Poul Due Jensens Fond (Grundfos Foundation).

REFERENCES

[1] Christian Aebeloe, Gabriela Montoya, Vinay Setty, and Katja Hose. 2018. Discovering diversified paths in knowledge bases. *Proc. VLDB Endow.* 11, 12 (2018), 2002–2005.

[2] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *ICDE*. 390–401.

[3] Daniel Arturo Casal Amat, Carlos Buil Aranda, and Carlos Valle-Vidal. 2021. A Neural Networks Approach to SPARQL Query Performance Prediction. In *CLEI*. 1–9.

[4] Sihem Amer-Yahia, Jasmina Bogojeska, Roberta Facchinetti, Valeria Franceschi, Aristides Gionis, Katja Hose, Georgia Koutrika, Roger Kouyos, Matteo Lissandrini, Silviu Maniu, Katsiaryna Mirylenka, Davide Mottin, Themis Palpanas, Mattia Rigotti, and Yannis Velegrakis. 2025. Towards Reliable Conversational Data Analytics. In *EDBT*. 962–969.

[5] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. WDBench: A Wikidata Graph Query Benchmark. In *ISWC*. Vol. 13489. 714–731.

[6] Christoph Anneser, Nesime Tatbul, David E. Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *PVLDB* 16, 12 (2023), 3515–3527.

[7] Negar Arabzadeh, Chuan Meng, Mohammad Aliannejadi, and Ebrahim Bagheri. [n.d.]. ECIR'24', Vol. 14612. 381–388.

[8] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. In *WWW'19*. 127–138.

[9] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679.

[10] Zheng Chu, Jiong Yu, and Askar Hamdulla. 2020. A novel deep learning method for query task execution time prediction in graph database. *FGCS* 112 (2020), 534–548.

[11] Angjela Davitkova, Damjan Gjurovski, and Sebastian Michel. 2022. LMKG: Learned Models for Cardinality Estimation in Knowledge Graphs. In *EDBT*. 2:169–2:182.

[12] DBpedia. [n.d.]. Downloads 2016-10. https://downloads.dbpedia.org/wiki-archive/downloads-2016-10.html Last accessed: April 4, 2025.

[13] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *SIGMOD*. 337–348.

[14] Orri Erling and Ivan Mikhailov. 2009. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Management*. 501–519.

[15] Guglielmo Faggioli, Thibault Formal, Simon Lupart, Stefano Marchesin, Stephane Clinchant, Nicola Ferro, and Benjamin Piwowarski. 2023. Towards Query Performance Prediction for Neural Information Retrieval: Challenges and Opportunities. In *SIGIR'23 (ICTIR '23)*. 51–63.

[16] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. 2008. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC*. 13–22.

[17] Steve Harris and Seaborne Andy. 2013. SPARQL 1.1 Query Language. W3C Recommendation 21 March (2013). https://www.w3.org/TR/sparql11-query/ Last accessed: April 4, 2025.

[18] Rakebul Hasan. 2014. Predicting SPARQL Query Performance and Explaining Linked Data. In *ESWC*, Vol. 8465. 795–805.

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

[20] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. 2018. Workload-Aware CPU Performance Scaling for Transactional Database Systems. In *SIGMOD*. 291–306.

[21] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoc. 2015. SPARQL with Property Paths. In *ISWC*, Vol. 9366. 3–18.

[22] Fangpeng Lan, Jinwen Zhang, and Baoning Niu. 2021. Predicting Response Time of Concurrent Queries with Similarity Models. *Big Data Research* 25 (2021), 100207.

[23] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB* 5, 11 (jul 2012), 1555–1566.

[24] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *CoRR* abs/1807.05118 (2018). arXiv:1807.05118

[25] Matteo Lissandrini, Katja Hose, and Torben Bach Pedersen. 2023. Example-Driven Exploratory Analytics over Knowledge Graphs. In *EDBT 2023*. OpenProceedings.org, 105–117.

[26] Matteo Lissandrini, Davide Mottin, Katja Hose, and Torben Bach Pedersen. 2022. Knowledge Graph Exploration Systems: are we lost?. In *CIDR 2022*. www.cidrdb.org.

[27] Matteo Lissandrini, Torben Bach Pedersen, Katja Hose, and Davide Mottin. 2020. Knowledge graph exploration: where are we and where are we going? *SIGWEB Newsl.* 2020, Summer (2020), 4:1–4:8.

[28] Honghao Liu, Zhiyong Peng, Zhe Zhang, Huan Jiang, and Yuwei Peng. 2022. MSP: Learned Query Performance Prediction Using MetaInfo and Structure of Plans. In *APWeb-WAIM*. 3–18.

[29] Katja Losemann and Wim Martens. 2012. The complexity of evaluating path expressions in SPARQL. In *PODS*. 101–112.

[30] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *ISWC*. 376–394.

[31] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *SIGMOD* 51, 1 (jun 2022), 6–13.

[32] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718.

[33] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB* 12, 11 (2019), 1733–1746.

[34] Mohammad Hossein Namaki, Keyvan Sasani, Yinghui Wu, and Assefaw Hadish Gebremedhin. 2017. Performance Prediction for Graph Queries. In *NDA@SIGMOD*. 4:1–4:9.

[35] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDBJ* 19 (2010), 91–113.

[36] Debjyoti Paul, Jie Cao, Feifei Li, and Vivek Srikumar. 2021. Database Workload Characterization with Query Plan Encoders. *PVLDB* 15, 4 (2021), 923–935.

[37] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Eng. Bull.* 42, 2 (2019), 32–46.

[38] Olivier Pelgrin, Luis Galárraga, and Katja Hose. 2021. Towards fully-fledged archiving for RDF datasets. *Semantic Web* 12, 6 (2021), 903–925.

[39] Axel Polleres, Romana Pernisch, Angela Bonifati, Daniele Dell'Aglio, Daniil Dobriy, Stefania Dumbrava, Lorena Etcheverry, Nicolas Ferranti, Katja Hose, Ernesto Jiménez-Ruiz, Matteo Lissandrini, Ansgar Scherp, Riccardo Tommasini, and Johannes Wachs. 2023. How Does Knowledge Evolve in Open Knowledge Graphs? *TGDK* 1, 1 (2023), 11:1–11:59.

[40] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2021. Optimizing SPARQL queries using shape statistics. In *EDBT*. 505–510.

[41] Fiana Raiber and Oren Kurland. 2014. Query-performance prediction: setting the expectations straight. In *SIGIR '14*. 13–22.

[42] Bhashyam Ramesh, Jaiprakash Chimanchode, Naveen Sankaran, and Jitendra Yasaswi. 2018. Optimizer time estimation for SQL queries. In *SSDBM*. 24:1–24:4.

[43] Kaspar Riesen and Horst Bunke. 2009. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vis. Comput.* 27, 7 (2009), 950–959.

[44] Tomer Sagi, Matteo Lissandrini, Torben Bach Pedersen, and Katja Hose. 2022. A design space for RDF data representations. *VLDB J.* 31, 2 (2022), 347–373.

[45] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[46] Keyvan Sasani, Mohammad Hossein Namaki, Yinghui Wu, and Assefaw Hadish Gebremedhin. 2018. Multi-metric Graph Query Performance Prediction. In *DASFAA*, Vol. 10827. 289–306.

[47] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *SIGMOD '23*. 225–237.

[48] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *ESWC*, Vol. 10843. 593–607.

[49] Tim Schwabe and Maribel Acosta. 2023. Cardinality Estimation over Knowledge Graphs with Embeddings and Graph Neural Networks. *CoRR* abs/2303.01140 (2023).

[50] Rekha Singhal and Manoj Nambiar. 2016. Predicting SQL Query Execution Time for Large Data Volume. In *IDEAS*. 378–385.

[51] Claus Stadler, Muhammad Saleem, Qaiser Mehmood, Carlos Buil-Aranda, Michel Dumontier, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2022. LSQ 2.0: A linked dataset of SPARQL query logs. *Semant. Web* (2022), 1–23.

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

[53] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. 2012. Heuristics-based query optimisation for SPARQL. In *EDBT*. 324–335.
[54] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. 2013. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB* 6, 10 (2013), 925–936.
[55] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*. 1081–1092.
[56] Wentao Wu, Xi Wu, Hakan Hacigümüs, and Jeffrey F. Naughton. 2014. Uncertainty Aware Query Execution Time Prediction. *PVLDB* 7, 14 (2014), 1857–1868.
[57] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2019. RDF Data Storage and Query Processing Schemes: A Survey. *ACM Comput. Surv.* 51, 4 (July 2019), 1–36.
[58] Mingyi Zhang, Patrick Martin, Wendy Powley, and Jianjun Chen. 2018. Workload Management in Database Management Systems: A Taxonomy. *IEEE Trans. Knowl. Data Eng.* 30, 7 (2018), 1386–1402.
[59] Wei Emma Zhang, Quan Z. Sheng, Yongrui Qin, Kerry Taylor, and Lina Yao. 2018. Learning-based SPARQL query performance modeling and prediction. *WWW* 21, 4 (2018), 1015–1035.
[60] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2020. Deep learning on graphs: A survey. *TKDE* 34, 1, 249–270.
[61] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2023. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *PVLDB* 17, 4 (2023), 823–835.
[62] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *PVLDB* 13, 9 (2020), 1416–1428.
[63] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *PVLDB* 16, 6 (feb 2023), 1466–1479.