Latest updates: https://dl.acm.org/doi/10.1145/3769810

PDF Download
3769810.pdf
03 February 2026
Total Citations: 0
Total Downloads: 177

Published: 05 December 2025

Citation in BibTeX format

RESEARCH-ARTICLE

NeuSO: Neural Optimizer for Subgraph Queries

LINGLIN YANG, Peking University, Beijing, China

LEI ZOU, Peking University, Beijing, China

CHUNSHAN ZHAO, Peking University, Beijing, China

Open Access Support provided by:

Peking University

Proceedings of the ACM on Management of Data, Volume 3, Issue 6 (December 2025)
https://doi.org/10.1145/3769810
EISSN: 2836-6573

NeuSO: Neural Optimizer for Subgraph Queries

LINGLIN YANG, Peking University, China
LEI ZOU, Peking University, China
CHUNSHAN ZHAO, Peking University, China

Subgraph query is a critical task in graph analysis with a wide range of applications across various domains. Most existing methods rely on heuristic vertex matching orderings, which may significantly degrade enumeration performance for certain queries. While learning-based optimizers have recently gained attention in the context of relational databases, they cannot be directly applied to subgraph queries due to the heterogeneous and schema-flexible nature of graph data, as well as the large number of joins involved in subgraph queries. These complexities often leads to inefficient online performance, making such approaches impractical for real-world graph database systems. To address this challenge, we propose NeuSO, a novel learning-based optimizer for subgraph queries that achieves both high accuracy and efficiency. NeuSO features an efficient query graph encoder and an estimator which are trained using a multi-task framework to estimate both subquery cardinality and execution cost. Based on these estimates, NeuSO employs a top-down plan enumerator to generate high-quality execution plans for subgraph queries. Extensive experiments on multiple datasets demonstrate that NeuSO outperforms existing subgraph query ordering approaches in both performance and efficiency.

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

Additional Key Words and Phrases: Subgraph Query, Query Optimization

ACM Reference Format:
Linglin Yang, Lei Zou, and Chunshan Zhao. 2025. NeuSO: Neural Optimizer for Subgraph Queries. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 345 (December 2025), 28 pages. https://doi.org/10.1145/3769810

1 Introduction

Graph models have gained considerable attention in recent years due to their straightforward and clear representation of relationships between entities. They are widely applied across various fields, including social networks [18], biology [11], and knowledge graphs [32, 55]. To retrieve a user-interested subgraph from the original big data graph, users may invoke subgraph queries on the data graph, which is a fundamental type of graph queries in the field of graph databases [5, 7, 92].

Specifically, given a query graph and a data graph, subgraph query is to find all matches of the query graph in the data graph, which preserves the label constraints of vertices and topology constraints of edges. Typically, the processing of a subgraph query involves three stages: filtering, planning, and enumeration. The filtering stage employs various techniques to extract a subset of vertices in the data graph (i.e., potential matching candidates) for each query vertex, thereby narrowing the search space. Next, in the planning stage, a matching order for the query vertices is

Authors' Contact Information: Linglin Yang, linglinyang@stu.pku.edu.cn, Peking University, Beijing, China; Lei Zou, zoulei@pku.edu.cn, Peking University, Beijing, China; Chunshan Zhao, zhaochunshan@pku.edu.cn, Peking University, Beijing, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2025/12-ART345
https://doi.org/10.1145/3769810

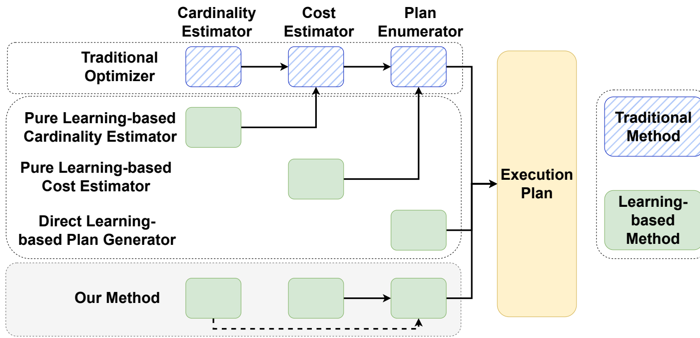


Fig. 1. Optimizer classification. The blue block represents traditional method, while the green block represent learning-based method.

generated. Finally, the matching results are obtained by enumerating according to the specified matching order.

The matching order plays a significant role in execution efficiency. The choice of matching order can result in performance variations spanning several orders of magnitude [37, 67, 88]. While much research has focused on improving the speed of subgraph matching, particularly through various filtering techniques [6, 8, 9, 23] and reducing duplicate computations during enumeration [9, 23, 29], few works have addressed the matching order selection problem effectively. Most existing subgraph matching approaches either depend on heuristic strategies or leverage dynamic programming (DP) techniques to determine the matching order. Nevertheless, these methods often face several critical limitations:

- *Structured-based heuristic methods* [11, 68] are based solely on the structure of the query graph, resulting in the same matching order for different data graphs if the query graph is the same. This can lead to significant performance degradation when applied to biased data graphs.
- *Rule-based heuristic methods* [8, 9, 23, 24, 31] rely on simple rules, such as prioritizing vertices with large degrees or small candidate sizes, without considering more detailed factors. This oversight can easily lead to suboptimal matching orders, which may be several orders of magnitude worse than the optimal solution.
- *Dynamic programming-based methods* [19, 28, 50, 78] are typically limited to small-scale query graphs and can become computationally expensive or even intractable for large query graphs.

1.1 Motivation

Traditional optimizers for relational databases are based on cost estimation and follow a pipeline structure comprising three main components: the cardinality estimator, the cost estimator, and the plan enumerator, as shown in Fig. 1. When a query is received, the plan enumerator generates possible execution plans, then selects the plan with the lowest estimated execution cost for actual execution. One of the most critical factors influencing execution cost is cardinality, which refers to the size of the intermediate results. As a result, the cost estimator often relies on the cardinality estimator to estimate the execution cost for each candidate plan.

With the rise of machine learning, a number of learning-based optimization methods for relational databases have emerged. These methods can be categorized into the following three approaches, as shown in Fig. 1:

- **Pure learning-based cardinality estimator** [25, 34, 40, 58, 81]. These methods focus on using machine learning to estimate cardinalities, while still relying on traditional hand-designed cost

estimators and dynamic programming (DP)-based plan enumeration to generate the execution plan.
- **Pure learning-based cost estimator** [45, 47, 66]. In this kind of approach, machine learning models are used to directly estimate the execution cost for each plan, with a traditional plan enumerator used to determine the optimal plan.
- **Direct learning-based order generator** [14, 46, 80, 84]. These methods bypass the cardinality and cost estimation stages entirely, using machine learning, particularly reinforcement learning, to directly generate the execution plan recursively.

However, these learning-based methods designed for relational databases cannot be directly applied to subgraph queries, as they typically assume a fixed table schema, while graph data is inherently more heterogeneous and schema-flexible. To the best of our knowledge, apart from RLQVO [74], there is no work on learning-based optimization for subgraph queries. RLQVO formulates the matching order generation as a Markov Decision Process (MDP), iteratively assigning scores to vertices and selecting them accordingly. However, RLQVO determines the scores solely from the representation of the query vertex, without considering the overall structure of subqueries. As a result, it occasionally produces suboptimal matching orders, leading to poor performance (Sec. 7).

Another potential approach to matching order optimization relies on cardinality estimation combined with a hand-crafted cost model. Several learning-based methods for subgraph counting (NSIC [43], LSS [89], NeurSC [73], and GNCE [61]) have been proposed, which can be applied to estimate cardinality. However, few studies have integrated these techniques into query optimizers as core components. This is primarily due to several challenges that hinder their practical adoption in graph database systems:

- **High computational cost for online prediction.** For example, NeurSC [73] requires running a Graph Neural Network (GNN) on a filtered graph that can be as large as the original data graph. This makes it inefficient when estimating the cardinalities of numerous subqueries. Moreover, these methods are primarily designed to estimate the cardinality of an entire query graph rather than its subqueries. Consequently, employing these methods for subquery-level cardinality estimation introduces additional latency, further exacerbating the computational overhead.
- **Neglect of Subquery Relationships.** Given a query graph Q, the subgraph query optimizer needs to predict the cardinalities of multiple subqueries of Q for the purpose of plan generation. However, existing subgraph counting approaches only focus on the cardinality estimation of a single subgraph query (Q itself), neglecting the relationships among these various subgraph queries.
- **Expressiveness Limits of GNN.** Existing methods [43, 61, 73, 89] primarily use message-passing neural networks (MPNNs) for graph feature extraction. However, MPNNs are limited in their expressiveness, as they do not surpass the capabilities of the 1-WL (1-dimensional Weisfeiler-Lehman) graph isomorphism test, which constrains the accuracy of these methods.

1.2 Our Solutions

In this paper, we propose **NeuSO**, a novel neural optimizer for subgraph queries (short for **Neu**ral **S**ubgraph Queries **O**ptimizer). Our method does not fit into any of the three existing categories of learning-based optimizers for relational databases as shown in Fig. 1; instead, it is more like a combination of them. Specifically, our approach simultaneously learns both the cardinality and the cost of queries and outputs a matching order under a new plan enumerator. The key insight behind jointly learning cardinality and cost is that both can be viewed as intrinsic properties of subqueries. This dual learning approach improves the robustness and accuracy of subgraph representations.

We also introduce a new metric, *minimum cost* (see Def. 5.4), which refers to the execution cost of the best possible plan for each subquery among all potential plans. Using the minimum cost, we propose a new top-down plan enumerator that significantly reduces the enumeration cost. Additionally, users can leverage the learned cardinality estimates to produce the matching order in traditional optimization settings.

In summary, we make the following contributions in this paper:

- **Multi-task training framework.** Our approach employs a *multi-task* training framework. Since cardinality and execution costs are intrinsic properties of subqueries, we enhance the model's robustness and predictive accuracy by simultaneously training it to predict these two aspects.
- **Novel query graph encoder.** We propose an enhanced GNN-based architecture that equips each vertex with richer neighborhood information compared to traditional MPNNs, thereby increasing the model's expressive power and predictive accuracy. Additionally, we improve the interaction between data graphs and query graphs. By initializing query vertex features with statistical information from filtered data graphs, our query graph encoder achieves both effectiveness and efficiency.
- **Top-down subgraph query plan enumerator.** We design a top-down greedy plan enumeration algorithm that generates high-quality execution plans. This approach minimizes excessive substate exploration typical in dynamic programming and outperforms heuristic-based methods in terms of performance.
- **Extensive experiments on real and synthetic datasets.** Extensive experiments on multiple real-world and synthetic datasets confirm that our method outperforms existing approaches in both efficiency and performance.

2 Preliminaries

We focus on undirected vertex-labeled graphs in this paper. A graph G is represented as a quadruple $G(V, E, L, \Sigma)$, where V is the set of vertices and E is the set of edges connecting vertices in V. Σ is a set of some vertex labels, and L is a function mapping every vertex in V to a unique label from Σ. Sometimes, without causing ambiguity, we denote $G(V, E, L, \Sigma)$ as $G(V)$ or $G(V, E)$ for simplicity.

Tab. 1 lists the notations frequently used in this paper.

Table 1. Frequently used notations.

Notation	Description
$G(V, E, L, \Sigma)$	undirected vertex-labeled graph
Q, q	query graph and subquery
o, Q_i	matching order and the partial query $Q(V_i^o)$
$N_o^-(u)$	the backward neighbors of u in order o
$C(u)$	candidate set of the query vertex u

2.1 Subgraph Query & Subgraph Matching

Subgraph query finds subgraph matches within a larger graph by identifying patterns that meet specific constraints. One of the most widely used matching semantic for subgraph matching is subgraph isomorphism [13, 57, 60, 64], which is defined formally below:

Definition 2.1 (Subgraph Matching (Isomorphism)). Given a query graph $Q(V_Q, E_Q, L_Q)$ and a data graph $G(V_G, E_G, L_G)$, a subgraph matching is an injective function f from V_Q to V_G such that

- (Label constraint) $L_Q(v) = L_G(f(v)), \forall v \in V_Q$;
- (Edge constraint) $e(f(u), f(v)) \in E_G, \forall e(u, v) \in E_Q$.

We also refer to the mapped graph $m = f(Q)$ as a subgraph match of the query Q in the graph G. The subgraph matching problem is to find all such subgraph matchings given a query and a data graph.

While our methods are generally applicable to various subgraph matching semantics, we adopt subgraph isomorphism here because of its practical importance and inherent complexity [13, 57, 60, 64]. Extensions to other semantics (like homomorphism for SPARQL [2] or cyphermorphism for Cypher [20]), are discussed in Sec. 6.3.1.

For simplicity of presentation, we assume that the query graphs are always connected. This does not affect the generality of our methods: for every unconnected query graph, the subgraph matching results are the Cartesian product of the matching results of its connected components with an additional check for injectivity.

2.2 Execution of Subgraph Queries

Executing a subgraph query in a graph database essentially involves finding all matchings of the query graph within the stored data graph [92]. This process can be divided into three stages: filtering, planning, and enumeration [67] as shown in Alg. 1.

Algorithm 1: Subgraph query execution process [67]

```
Input: The data graph G, the query graph Q
Output: The subgraph matching results R = R(Q, G)
1  C ← Generate candidate sets
2  o = (o_1, o_2, ··· , o_n) ← Generate a matching order
3  Enumerate(Q, G, o, ∅, 1)
4  Procedure Enumerate(Q, G, o, M, i)
5      if i = |o| + 1 then
6          R ← R ∪ {M}; return
7      LC(u_{o_i}, M) ← Compute local candidates
8      foreach v ∈ LC(u_{o_i}, M) do
9          if ∄u' ∈ V_Q, s.t. M[u'] = v then
10             Enumerate(Q, G, o, M ∪ {(u_{o_i}, v)}, i + 1)
```

First, the engine analyzes the query graph and applies filters to identify a smaller data graph from which all matching results can be retrieved. Next, an execution plan, typically a matching order, is formulated by the database engine's optimizer. Finally, the executor enumerates all subgraph matchings to obtain the final results according to the matching order.

Filtering. Filtering is an important technique in subgraph matching. It employs various restrictions to quickly exclude vertices and edges in the data graph that are not relevant to the query, thereby constructing a candidate set for each query vertex. Vertices that do not appear in a candidate set can be pruned during execution. The formal definition of a candidate set is provided below:

Definition 2.2 (Candidate Set). Given a query graph Q and a data graph G, the candidate set $C(u)$ of a query vertex $u \in V_Q$ is a subset of vertices in G, such that if there exists a subgraph matching f of Q, then $f(u) \in C(u)$.

Plan Generation (Optimization). An execution plan is generated during the planning process. For subgraph queries, the main difference across different execution plans is the matching order (also known as the join order) defined in Def. 2.3. In this paper, we use matching order and execution plan interchangeably as they essentially mean the same.

Definition 2.3 (Matching Order). For a query graph $Q = Q(\{u_1, u_2, \cdots, u_n\})$, a matching order is a permutation of the query vertices: $o = (u_{o_1}, u_{o_2}, \cdots, u_{o_n})$. Additionally, we denote the prefix i-subquery vertex set as $V_i^o = \{u_{o_1}, \cdots, u_{o_i}\}$.

It is reasonable to require that the matching order preserves prefix connectivity. Otherwise, it will result in the Cartesian product. Formally, along with the matching order, each partial query $Q_i = Q(V_i^o) = Q(\{u_{o_1}, \cdots, u_{o_i}\})$ is a connected subquery, and the next vertex to match, $u_{o_{i+1}}$, is a neighbor of the set V_i^o. We denote the backward neighbors of u as $N_o^-(u)$, which represents the set of neighbor vertices of u with matching orders preceding u.

Enumeration. During the enumeration of the i-th vertex u_{o_i}, the backward neighbors of u_{o_i} already have matches. Therefore, the local candidates for u_{o_i} in the partial matching M can be computed using the partial matching M intersecting the neighbor lists from the matches of $N_o^-(u_{o_i})$ (line 7 of Alg. 1).

2.3 Challenges in Subgraph Query Optimization

In relational databases, the most common cost-based optimizer usually consists of three parts: a plan enumerator, a cost estimator, and a cardinality estimator. When users input a query, the plan enumerator generates several plans for the input query. Then, the cost estimator invokes the cardinality estimator for the cardinality estimation of subqueries, which helps estimate the cost of candidate plans. Then the plan with minimum estimated cost is transferred to the execution engine for execution.

For subgraph queries, there are also some works like the traditional relational optimizers which enumerate the whole join space in a dynamic programming framework [19, 50, 78]. However, the larger number of vertices and edges of graph queries greatly enlarges the plan space, making dynamic programming optimizers impractical for real scenarios. For instance, a query graph with 24 vertices may have up to millions of connected subqueries. Dynamic programming methods would generate one plan for each subquery as an optimal subsolution, which can be extremely time-consuming.

Thus, most of the existing methods [8, 9, 11, 23, 24, 31, 68] rely on greedy heuristic strategies. However, they usually lack the detailed exploration of data characteristics and often lead to local optimum.

3 Overview

Our method, NeuSO, consists of three main components: a query graph encoder, a cardinality & cost predictor, and a plan enumerator.

As illustrated in Fig. 2, the proposed method begins with a filter that generates query-related statistics upon receiving a new query graph. These statistics, along with the query graph, are processed by a GNN-based query graph encoder, which produces vertex-level representations. Subsequently, these vertex representations can be aggregated to derive subquery representations on demand, as required by the plan enumeration phase. A representation for the entire query graph is also computed in this way.

Next, the cardinality & cost predictor leverage these (sub)graph representations to estimate the cardinality and execution cost of the queries. Finally, a plan enumerator module utilizes these estimates to generate join plans by minimizing the cost estimates.

NeuSO differs from existing learning-based optimizers, as illustrated in Fig. 1. After encoding the query graph, NeuSO employs a multi-task learning framework to simultaneously predict both the cardinality and execution cost using the same subquery representation. Multi-task learning has gained significant attention recently and demonstrated effectiveness across various domains, such

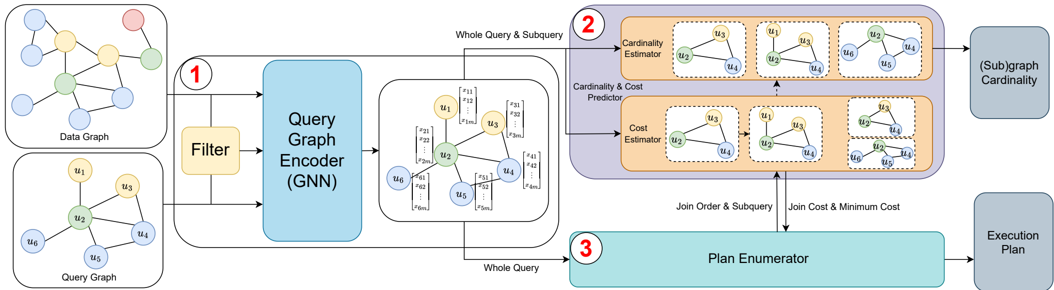


Fig. 2. Overview of NeuSO. (1) GNN-based query graph encoder processes input after feature initialization from filtered data graph statistics to generate representation for (sub)query (Sec. 4); (2) For each (sub)query, the cardinality & cost predictor uses MLP to estimate the (sub)query cardinality and execution cost (Sec. 5.2); (3) A top-down plan enumerator that utilizes the cost estimator to identify the execution plan with the lowest estimated cost (Sec. 5.3).

as NLP [12, 87]. In the context of subgraph query optimization, this approach offers the following four key advantages:

- **Enhanced Representation Learning.** By sharing the same subquery representation, the model can inherently capture the relationships between cardinality and cost. This leads to improved performance through higher-quality subquery representations.
- **Improved Robustness.** The multi-task framework allows the model to leverage supervisory signals from one task to mitigate noise or errors in the other, resulting in greater overall robustness.
- **Increased Interpretability.** It is reasonable that the model produces a large cost estimate when the cardinality estimate is also large. Additionally, the intermediate cardinality and cost estimates provide insights that can help diagnose issues when the model produces suboptimal execution plans.
- **Flexibility to Meet Diverse Requirements.** This multi-task framework is adaptable to different application needs. For instance, users with a well-designed cost estimator can utilize only the learned cardinality estimator, while those with less expertise and seeking an end-to-end solution can rely on the model's learned cost predictor. This flexibility also enables seamless integration with existing components.

4 Query Graph Encoder

In NeuSO, we first utilize a graph neural network to encode the query graph, leveraging query-related information from the data graph (the query graph encoder in Fig. 2) to generate representations for (sub)queries. Details on the initial feature construction and the specific graph neural network used are provided in Sec. 4.1 and Sec. 4.2, respectively. Each query vertex is represented by a high-dimensional embedding after the encoding process.

Subsequently, as discussed in Sec. 4.3, an aggregation pooling derives a representation for the (sub)query graph, serving as the input of the subsequent optimization process.

4.1 Feature Initialization

The design of initial features for an encoder is crucial. For a query graph, the most basic feature of a query vertex is its label. Given the relatively limited number of labels, one-hot encoding is widely adopted in existing subgraph tasks [73, 83]. However, this encoding method presents two significant drawbacks: (1) it results in highly sparse representations that are challenging for subsequent learning processes; (2) it fails to incorporate insights from the topological structures of the data graph.

An embedding-based approach effectively alleviates these issues. To obtain informative embeddings of each label in the data graph G, we pre-trained on a *label-augmented* graph $G_A = G_A(V_G \cup V_L, E_G \cup E_L)$, which adds a vertex l for each label in the label set Σ_G (V_L) and an edge $e(u, L(u))$ for every vertex u and its label $L(u)$ (E_L). Any existing embedding method [15, 21, 56] can be applied to this label-augmented graph, and the resulting embedding $\boldsymbol{x}_l$ for a label vertex l is utilized as the initial feature for the query vertex with label l. In our experiments, we employ ProNE [86], which integrates sparse matrix factorization with embedding propagation, offering a fast and effective solution for downstream tasks.

It should be noted that relying solely on label features is inadequate, as it assigns identical features to vertices with the same label, regardless of whether they belong to the same query graph or different ones. To incorporate more query-specific information, we enhance the initial label features by integrating filtered statistics. As described in Sec. 2.2, the filter stage produces a candidate set $C(u)$ for each query vertex u. Additionally, the candidate edge count $|C(u_1, u_2)|$ can be computed for each query edge $e(u_1, u_2)$ with a very small cost. These statistics are more accurate and query-relevant than those derived directly from the original data graph.

In summary, the initial features of the query graph encoder are constructed for a vertex u and an edge $e(u_1, u_2)$ as follows:

$$\boldsymbol{x}_u^{(0)} = \boldsymbol{x}_{L(u)} \oplus |C(u)|, \tag{1}$$

$$\boldsymbol{x}_{e(u_1,u_2)}^{(0)} = \boldsymbol{x}_{u_1}^{(0)} \oplus \boldsymbol{x}_{u_2}^{(0)} \oplus |C(u_1, u_2)|, \tag{2}$$

where $\oplus$ denotes concatenation.

4.2 TriAT: Triangle Attention Network

4.2.1 Motivation for TriAT. Graph neural networks (GNNs) are widely used to extract features and identify patterns in graph-structured data. Popular GNN such as GCN [35], GraphSAGE [22], GIN [76] and GAT [71] belongs to the message-passing neural network (MPNN) framework. These models iteratively update vertex embeddings as follows:

$$\boldsymbol{x}_u^{(k)} = \gamma^{(k)}\{\boldsymbol{x}_u^{(k-1)}, \phi^{(k)}\{\boldsymbol{x}_v^{(k-1)}, \boldsymbol{x}_{e(u,v)}^{(k-1)} \mid v \in N(u)\}\}, \tag{3}$$

where ϕ aggregates information from neighbors, and γ combines it with the vertex's prior representation $\boldsymbol{x}_u^{(k-1)}$ and edge representation $\boldsymbol{x}_{e(u,v)}^{(k-1)}$, producing the next-layer representation $\boldsymbol{x}_u^{(k)}$.

Despite their success, theoretical studies have shown that the expressive power of MPNNs is limited by the 1-WL test [38, 76], preventing them from distinguishing certain graph topologies. For example, in Fig. 3, two graphs have distinct topologies but cannot be distinguished by MPNNs, as vertices with identical neighborhood label sets receive identical embeddings. For instance, vertices A_1 and A_2 in both graphs gather information from neighbors with the same label distributions, resulting in identical representations under the update rule in Eqn. 3.

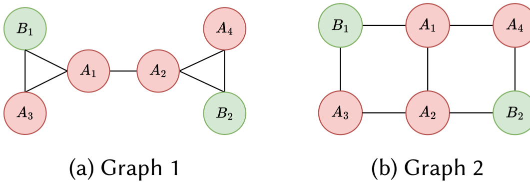


Fig. 3. Two graphs that cannot be distinguished by MPNNs or the 1-WL test. Vertices with identical notations receive the same representation under MPNNs.

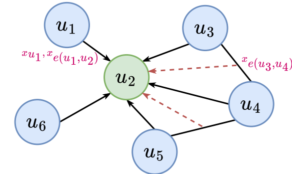


Fig. 4. An example of the update procedure in TriAT (for u_2). Each vertex updates its embedding based on both its 1-hop neighbors and the edges between those neighbors.

Distinguishing between such graphs is crucial for subgraph query optimization, for different query graphs may lead to different optimal execution plans. To overcome this limitation, we propose **TriAT**, which explicitly incorporates triangle structures, enhancing the network's ability to capture higher-order patterns. In Fig. 3a, triangles are formed by two B-label vertices linking to two A-label vertices, while no such triangle exists in Fig. 3b. TriAT leverages these triangle structures to differentiate between graphs that standard MPNNs cannot.

It is worth noting that triangle patterns are ubiquitous in real-world graph structures and are commonly found in subgraph queries. As the simplest cyclic structure, triangles serve as a fundamental building block for many denser and more complex motifs, such as k-cliques. We observed that over 90% of the cyclic queries tested in our experiments contain at least one triangle substructure. Previous research has also highlighted the prevalence of triangle structures in real graph queries [10]. By identifying and leveraging these triangle patterns, we can effectively model complex relationships and higher-order dependencies, making them indispensable for robust graph representation.

4.2.2 TriAT Architecture. TriAT updates the embedding of vertex u by considering both its 1-hop neighbors $N(u)$ and the edges among these neighbors $E_{N(u)} = \{e(v_1, v_2) \mid v_1, v_2 \in N(u)\}$, forming triangles with u. This is illustrated in Fig. 4, where vertex u_2 updates its embedding using both neighbor information (solid black arrows, such as $\{\boldsymbol{x}_{u_1}, \boldsymbol{x}_{e(u_1,u_2)}\}$ for neighbor u_1) and neighbor linkages (brown dotted arrows, such as $\{\boldsymbol{x}_{e(u_3,u_4)}\}$ for the edge $e(u_3, u_4)$).

Formally, TriAT follows the update procedure below:

$$\begin{aligned}\boldsymbol{x}_u^{(k)} = \gamma^{(k)}\{\boldsymbol{x}_u^{(k-1)}, \phi^{(k)}\{\boldsymbol{x}_v^{(k-1)}, \boldsymbol{x}_{e(u,v)}^{(k-1)} \mid v \in N(u)\},\\ \tau^{(k)}\{\boldsymbol{x}_{e(v_1,v_2)}^{(k-1)} \mid e(v_1, v_2) \in E_{N(u)}\}\},\end{aligned} \tag{4}$$

$$\boldsymbol{x}_{e(u,v)}^{(k)} = W_e^{(k)}(\boldsymbol{x}_u^{(k)} \oplus \boldsymbol{x}_v^{(k)}), (k \geq 1) \tag{5}$$

here, $W_e^{(k)}$ is the learnable linear transformation matrix used for updating edge representations. $\phi^{(k)}$ and $\tau^{(k)}$ are attention-based aggregation functions for neighbor vertices and their interconnections:

$$\phi^{(k)}\{\boldsymbol{x}_v^{(k-1)}, \boldsymbol{x}_{e(u,v)}^{(k-1)} \mid v \in N(u)\} = \sum_{v \in N(u)} \alpha_{u,v}^{(k)} \Theta^{(k)} \boldsymbol{x}_v^{(k-1)}, \tag{6}$$

$$\tau^{(k)}\{\boldsymbol{x}_e^{(k-1)} \mid e \in E_{N(u)}\} = \sum_{e \in E_{N(u)}} \beta_{u,e}^{(k)} \Psi^{(k)} \boldsymbol{x}_e^{(k-1)}. \tag{7}$$

$\Theta^{(k)}$ and $\Psi^{(k)}$ are learnable projection matrices. $\alpha_{u,v}^{(k)}$ and $\beta_{u,e}^{(k)}$ are attention coefficients computed as follows:

$$\alpha_{u,v}^{(k)} = \frac{\exp(\mathrm{LR}(\boldsymbol{a}^{(k)}[W_1^{(k)}\boldsymbol{x}_u^{(k-1)} \oplus W_2^{(k)}\boldsymbol{x}_{e(u,v)}^{(k-1)}]))}{\sum_{w \in N(u)} \exp(\mathrm{LR}(\boldsymbol{a}^{(k)}[W_1^{(k)}\boldsymbol{x}_u^{(k-1)} \oplus W_2^{(k)}\boldsymbol{x}_{e(u,w)}^{(k-1)}]))}, \tag{8}$$

$$\beta_{u,e}^{(k)} = \frac{\exp(\mathrm{LR}(\boldsymbol{b}^{(k)}[W_1^{(k)}\boldsymbol{x}_u^{(k-1)} \oplus W_2^{(k)}\boldsymbol{x}_{e(v_1,v_2)}^{(k-1)}]))}{\sum_{e' \in E_{N(u)}} \exp(\mathrm{LR}(\boldsymbol{b}^{(k)}[W_1^{(k)}\boldsymbol{x}_u^{(k-1)} \oplus W_2^{(k)}\boldsymbol{x}_{e'}^{(k-1)}]))}, \tag{9}$$

where $\boldsymbol{a}^{(k)}$ and $\boldsymbol{b}^{(k)}$ are learnable attention weight vectors. $W_1^{(k)}$ and $W_2^{(k)}$ are learnable linear transformation matrices. LR denotes the LeakyReLU nonlinearity, and we set it with a negative slope of 0.2 in our experiments. The attention mechanism in TriAT enables the model to assign varying weights to different neighbors and their connections based on their importance, enhancing its ability to capture complex patterns and dependencies.

For the update function $\gamma^{(k)}$, we sum the inputs and apply a ReLU nonlinearity. Additionally, we incorporate a multi-head mechanism [70] to further enhance TriAT's expressive power.

4.2.3 Expressive Power & Complexity. The following theorems reveal the expressiveness and complexity of TriAT. Specifically, TriAT is more expressive than standard MPNNs and the 1-WL test, as it considers adjacency between neighboring vertices. Its time and space complexities are also characterized. The proof of Theorem 4.1 can be found in the full version of our paper [79].

Theorem 4.1 (TriAT's Expressive Power). *The following expressiveness inclusion relation holds: 1-WL test = MPNN $\prec$ TriAT.*

Theorem 4.2 (TriAT's Complexity). *Let Δ denote the set of all triangles in a graph $Q(V, E)$. Then the time complexity of one layer of TriAT is $O(|E| + |\Delta|)$, and the space complexity is $O(|E|)$.*

Proof of Theorem 4.2. Each layer of TriAT requires gathering information from neighbors (ϕ) and their mutual connections (τ) for every vertex. The total computational complexity for ϕ and τ across all vertices is $O(|E|)$ and $O(|\Delta|)$, respectively. This is because each edge is involved in ϕ through its two endpoints, and contributes to τ only if it is part of a triangle. These two components together yield the overall time complexity of $O(|E| + |\Delta|)$. The space complexity is $O(|E|)$, as TriAT adopts a vertex-update framework that requires storing features for all vertices and edges. □

Although there are other GNNs that are more expressive than MPNNs, such as GNNs based on 2-FWL [48, 85], these methods incur a computational cost of $O(|V|^3)$ per layer and require $O(|V|^2)$ space. This high complexity significantly affects the runtime efficiency of 2-FWL. However, our experiments show that TriAT achieves comparable accuracy to 2-FWL without a substantial drop in performance, while maintaining a much lower computational cost (Sec. 7.3.2). Therefore, TriAT provides a favorable trade-off between efficiency and effectiveness.

4.3 Representation of (sub)Graphs

After applying the graph neural network to the query graph, we obtain an embedding matrix $X \in \mathbb{R}^{n \times m}$, where n is the number of query vertices and m is the vertex embedding dimension of the final layer of the TriAT used. Each row in the matrix X corresponds to the embedding $\boldsymbol{x}_u^T$ of a query vertex u. To get the representation of the (sub)query graph q, we need to apply a pooling layer.

It is important to note that some vertices may have a greater influence on the final result than others. For example, vertices with high degrees impose stricter topological constraints and prioritizing those vertices in matching order leads to lower cardinality. To capture the varying contributions of vertices, we apply a self-attention-based weighted pooling layer that assigns adaptive importance to each vertex in the final (sub)graph representation:

$$\boldsymbol{x}_q = \sum_{u \in V_q} \alpha_u \boldsymbol{x}_u, \tag{10}$$

where, α_u is the attention coefficient for vertex u:

$$\alpha_u = (K_1 \boldsymbol{x}_u) \cdot (K_2 \boldsymbol{x}_u), \tag{11}$$

here, K_1 and K_2 are two trainable parameter matrices, and $\cdot$ denotes the inner product.

5 Neural Graph Query Optimizer

After the query graph encoder, we can obtain representations for every subquery. Next, we use learning-based estimators to predict cardinalities and execution costs and a top-down plan enumerator to generate matching order.

5.1 Reformulation of Optimization

5.1.1 Intuition. As discussed in Sec. 2.2, the execution plan of a graph query involves determining the matching order $o = (u_{o_1}, u_{o_2}, \cdots, u_{o_n})$ for the query graph $Q = Q(\{u_1, u_2, \cdots, u_n\})$. Along with the matching order o, each subquery $Q_i = Q(V_i^o) = Q(\{u_{o_1}, \cdots, u_{o_i}\})$ contains only one more query vertex u_{o_i} than the previous subquery Q_{i-1}. Thus, the matching order can be seen as a state transition path from the initial state (empty query Q_0) to the final state (complete query Q). From this perspective, subgraph query optimization is the process of finding a near-optimal state transition path within the linkage graph, consisting of all connected partial queries.

5.1.2 Formal Reformulation. To start with, we define a lattice structure named **Cardinality-Cost Graph (CCG)** that captures the relationship between subqueries of one query graph G:

Definition 5.1 (Cardinality-Cost Graph, CCG). Given a query graph Q, the CCG of Q is a rooted (weakly) connected acyclic directed graph $CCG_Q(\mathcal{V}_Q, \mathcal{E}_Q, C_Q)$, where:

(1) $\mathcal{V}_Q$ is the set of connected partial subqueries of Q, including the empty subquery ($\emptyset$) and Q itself. Each vertex q in $\mathcal{V}_Q$ represents an intermediate state in the execution, referred to as a *state*.
(2) $\mathcal{E}_Q$ is the edge set, where edges originate from smaller states and terminate at larger states that include one more query vertex. These edges represent *state transitions*.
(3) C_Q is the cardinality and cost function. For each state $q \in \mathcal{V}_Q$, $C_Q(q)$ denotes the actual cardinality of q, while for each state transition $e\langle q_{sub}, q(V_{q_{sub}} \cup \{u\})\rangle$, $C_Q(e)$ indicates the actual execution cost of joining vertex u to the partial intermediate results of q_{sub}. For convenience, we omit the subscript Q when there is no ambiguity.

Given a state q_0 from CCG_Q, we denote its out-neighbors as $\mathcal{N}^{out}(q_0) = \{q \in \mathcal{V}_Q \mid e\langle q_0, q\rangle \in \mathcal{E}_Q\}$. Similarly, we represent the in-neighbors of q_0 as $\mathcal{N}^{in}(q_0)$.

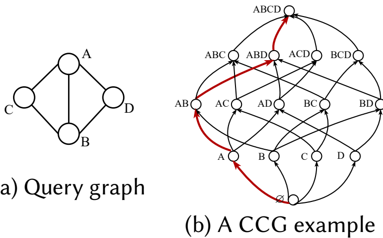
With the help of CCG, we can formulate the optimization as a shortest path problem on the CCG:

Definition 5.2 (Optimization & SP Problem on the CCG). Given a query graph Q, each join order $o = (o_1, \cdots, o_n)$ corresponds to a path P from $\emptyset$ to Q in CCG_Q:

$$P : (\emptyset =)\, q_0 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow \cdots \longrightarrow q_n \,(= Q) \tag{12}$$

with length defined as $l(P) = \sum_{i=1}^{n} C_Q(e\langle q_{i-1}, q_i\rangle)$.

The optimization's goal is to find the shortest path from $\emptyset$ to Q.



(a) Query graph

(b) A CCG example

Fig. 5. An example of a query graph and its corresponding CCG. For simplicity, we omit the cardinalities of states and the transition costs in the CCG.

Algorithm 2: Plan enumeration process

Input: The data graph G, the query graph Q
Output: The execution plan o

1 $o \leftarrow \emptyset, q \leftarrow Q$
2 **while** $q \neq \emptyset$ **do**
3 $\quad q' \leftarrow \arg\min_{q_0 \in \mathcal{N}^{in}(q)} \hat{C}(e\langle q_0, q\rangle) + \hat{\mathcal{M}C}(q_0)$
4 $\quad o.prepend(V_q - V_{q'})$
5 $\quad q \leftarrow q'$
6 **return** o

Example 5.3. Consider the query graph in Fig. 5a. The topology of its corresponding CCG is depicted in Fig. 5b, where each state is annotated with the vertices of its associated subquery. And a possible vertex order (A, B, D, C) is represented in the red path.

It is important to emphasize that the CCG is influenced by the underlying data graph, since the cardinality of each state and the cost of each transition are dependent on the specific data graph being matched.

If the cardinalities of the subqueries and the costs of state transitions are provided, a CCG for the query can be constructed. On this CCG, a shortest path algorithm is applied to determine the optimal execution plan. However, two major challenges arise:

(1) The true cardinalities and costs are not precisely available during the optimization phase prior to execution. Therefore, it is necessary to employ a cardinality estimator and a cost estimator.
(2) For large queries, the CCG may grow excessively large, rendering comprehensive exploration computationally prohibitive. For example, the CCG of a query graph with 24 vertices can encompass millions of states.

Our methods, as detailed in Sec. 5.2 and Sec. 5.3, are designed to address these two issues.

5.2 Learning-based Cardinality & Cost Predictor

To address the first challenge (unavailable precise cardinality and cost during optimization), we leverage the powerful fitting and learning capabilities of neural networks. Specifically, we note that the cardinality is inherently tied to a single subquery, whereas the join cost is determined by the relationship between two adjacent subqueries, as represented by the edges on the CCG.

Cardinality. For any subquery q, we use a multi-layer perceptron (MLP) MLP_{card}, for cardinality estimation. This can be formally expressed as:

$$\hat{C}(q) = MLP_{card}(\boldsymbol{x}_q). \tag{13}$$

Cost. For a state transition $e\langle q_1, q_2\rangle$ on the CCG, where q_2 is a (sub)query that adds a linked vertex u_0 to q_1, we can utilize the above cardinality estimation and calculate the cost by any cost model such as gCBO [78] as below:

$$\hat{C}(e\langle q_1, q_2\rangle) = \hat{C}(q_1) \times \min_{u \in N(u_0) \cap V_{q_1}} \frac{|C(u, u_0)|}{|C(u)|}. \tag{14}$$

This cost estimate reflects the maximum possible number of intermediate results for q_2. Specifically, for each intermediate result f of q_1, the term $\min_{u \in N(u_0) \cap V_{q_1}} \frac{|C(u,u_0)|}{|C(u)|}$ provides an upper bound on the number of matches of q_2 that would be generated from f.

This type of traditional method takes the cardinality estimate as input. However, it often overlooks certain aspects (such as storage cost), and the inherent inaccuracies in input cardinality estimation would propagate to the cost estimates. Therefore, we recommend an end-to-end approach, using neural networks to learn the relationship between the cost and the representation of the state transition in NeuSO:

$$\hat{C}(e\langle q_1, q_2\rangle) = MLP_{cost}(\boldsymbol{x}_{q_1} \oplus \boldsymbol{x}_{q_2}), \tag{15}$$

where $\boldsymbol{x}_{q_1} \oplus \boldsymbol{x}_{q_2}$ represents the concatenation of the embeddings for the start state q_1 and the end state q_2.

5.3 Top-down Cost-based Plan Enumerator

To tackle the second challenge (large CCG), we introduce the concept of the *minimum cost of states* to guide the plan exploration on CCG:

Definition 5.4. The minimum cost of a state q in the CCG is the minimum length of paths from $\emptyset$ to q:

$$\mathcal{MC}(q) = \min_{p\,:\,p_0=\emptyset,\, p_{|V_q|}=q} l(p). \tag{16}$$

Example 5.5. Consider the CCG in Fig. 5. There are 6 different paths from $\emptyset$ to $q_{\{A,B,D\}}$. Suppose the shortest path among these is $P^* = \{\emptyset, q_{\{A\}}, q_{\{A,B\}}, q_{\{A,B,D\}}\}$. Then the minimum cost of $q_{\{A,B,D\}}$ is $l(P^*)$.

In traditional optimizers, the minimum cost of a state q is determined through dynamic programming (DP) enumeration, which is time-consuming. Instead, we use a neural network (MLP) to directly predict the minimum cost:

$$\hat{\mathcal{MC}}(q) = MLP_{mc}(\boldsymbol{x}_q). \tag{17}$$

Based on the estimated minimum cost, we design a top-down greedy search for plan enumeration, as described in Alg. 2.

The critical step is in line 3. A state q can be reached from states $\mathcal{N}^{in}(q)$. For a state q_0 among them, it can be accessed via a path from $\emptyset$ of length $\mathcal{MC}(q_0)$, which is q_0's minimum cost. Therefore, the sum $\hat{C}(e\langle q_0, q\rangle) + \hat{\mathcal{MC}}(q_0)$ gives the estimated length of the shortest path from $\emptyset$ to q through q_0. The shortest path is selected and the additional vertex $V_q - V_{q'}$ is added to the matching order.

By utilizing the enumeration procedure in Alg. 2, we avoid constructing the entire CCG for the query Q. Instead, only promising states and transitions are explored, thereby reducing the cost of plan enumeration.

Example 5.6. Take the CCG of Fig. 5b as an illustration. The enumerator of Alg. 2 first explores all subqueries of size 3. If it determines that subquery $q_{\{ABD\}}$ lies on the optimal path (as guided by the equation in line 3 of Alg. 2), it infers that vertex C is the last to be joined. It then examines $q_{\{AB\}}$, $q_{\{AD\}}$, and $q_{\{BD\}}$, continuing the top-down process until reaching the initial state ($\emptyset$).

5.3.1 Complexity. Let the time complexities of running the models MLP_{cost} and MLP_{mc} be T_{cost} and T_{mc}, respectively. During each selection step (line 3 of Alg. 2), the time cost is $O(|V_Q|(T_{cost} + T_{mc}))$, since the number of possible child states is bounded by $O(|V_Q|)$. As each selection step chooses a single vertex, the while loop repeats $O(|V_Q|)$ times. Therefore, the total complexity of Alg. 2 is $O(|V_Q|^2(T_{cost} + T_{mc}))$. If we assume both T_{cost} and T_{mc} are $O(1)$, then the complexity simplifies to $O(|V_Q|^2)$.

In comparison, traditional dynamic programming (DP) methods typically have a time complexity of $O(|\mathcal{E}_Q|)$, which is often exponential in terms of $|V_Q|$, due to the need to explore all possible states. Consequently, our approach provides a substantial improvement in efficiency, particularly for large graphs.

6 Training Details & Extensions

In Sec. 6.1, we will first introduce the training data collection, followed by the training process in Sec. 6.2. Finally, we show that our approach can also be extended to other matching semantics and directed edge-labeled graphs in Sec. 6.3.

6.1 Training Data Collection

To collect the training data for a data graph G, we first obtain a representative set of query graphs $\mathcal{T}$, which can be selected from user logs or simulated based on query templates. For each query in $\mathcal{T}$, we construct its corresponding complete CCG. We then traverse each state within the CCG, recording (1) the cardinalities of the states, (2) the execution costs (running times) of the transitions, and (3) the minimum cost associated with each state. These data are stored as the training dataset.

However, for large-scale queries with many vertices or those producing substantial intermediate results, exhaustive traversal of all CCG states becomes computationally infeasible. To address this challenge, we employ a partial data collection strategy for such queries. Specifically, for a large or complex query Q, we first generate an optimized matching order o using the optimizer from [78], while other optimization methods can also be employed. Then we explore not only the states q_i (where $i = 1, 2, \ldots, |o|$) but also their adjacent states $N^{out}_{CCG}(q_i)$. In this process, we only collect (1) the state cardinalities and (2) the transition costs.

We categorize queries based on the extent of exploration: queries that are fully explored are labeled as $q.state = Full$, while those partially explored are labeled as $q.state = Partial$. Additionally, to ensure computational efficiency, we impose memory and time thresholds for each transition (3 minutes per transition and a space limit of 100 million embeddings per state in our experiments).

6.2 Training Process

For fully explored query graphs, we utilize cardinalities, costs, and minimum costs during training. In contrast, for partially explored query graphs, we only use cardinalities and costs, as minimum costs are unavailable. In both scenarios, loss coefficients α_i, where $i \in \{card, cost, mc\}$, are used to balance the multiple training objectives (we set α_i as 0.4, 0.3 and 0.3 for $i \in \{card, cost, mc\}$ in our experiments). For these objectives, we apply the L2 loss over the log transformation which imposes higher weights on larger error over the average [16]:

$$\mathcal{L}_q(\hat{y}, y) = |\log \hat{y} - \log y|^2 = |\log(\max\{\hat{y}/y, y/\hat{y}\})|^2. \tag{18}$$

In addition to these three separate losses from each neural predictor, we introduce a supplementary *constraint loss* term to ensure consistency between the single-step cost predictor (MLP_{cost}) and the minimum cost predictor (MLP_{mc}). This constraint loss penalizes any unrealistic minimum cost estimate for a state q_0 that are lower than the minimum single-step cost estimates:

$$\mathcal{L}_c(q_0) = || \max\{0, \min_{q' \in \mathcal{N}^{in}(q_0)} \hat{C}(e\langle q', q_0\rangle) - \hat{MC}(q_0)\}||^2. \tag{19}$$

In our training process, the constraint loss is applied to the explored states Q_e. Note that for fully explored query graphs, the set of explored states Q_e includes all subqueries of Q. A pseudocode summarizing the entire training process is included in the full version of our paper [79].

6.3 Extensions

6.3.1 Extension to Other Matching Semantics. Thanks to the generalization ability of neural networks, NeuSO can be extended to other subgraph matching semantics with minimal changes. Specifically, only two adjustments are needed: (1) adapting the query execution engine to reflect the desired matching semantics, and (2) collecting training data accordingly so that the supervision aligns with the new semantic. The neural model will then learn to approximate the new outputs automatically.

For example, when using NeuSO for subgraph counting and optimization of homomorphism, the execution engine do not need to do the isomorphic check (line 9 of Alg. 1), and the training data of cardinality and cost should be the number of homomorphisms and the homomorphism execution time, respectively.

6.3.2 Extension to Directed Edge-labeled Graphs. Extending NeuSO to directed and edge-labeled graphs is straightforward. The most basic modification involves filtering and enumerating matching results according to the directions and labels of the query edges.

Additionally, to account for the directionality of edges in the query graphs, the out-edges and in-edges should be handled separately in the query graph encoder. Specifically, we modify Eqn. 4

to the following iterative process:

$$\begin{aligned}\boldsymbol{x}_u^{(k)} = \gamma^{(k)}\{\boldsymbol{x}_u^{(k-1)}, \phi_1^{(k)}\{\boldsymbol{x}_v^{(k-1)}, \boldsymbol{x}_{e\langle u,v\rangle}^{(k-1)} \mid v \in N_{out}(u)\}, \\ \phi_2^{(k)}\{\boldsymbol{x}_v^{(k-1)}, \boldsymbol{x}_{e\langle v,u\rangle}^{(k-1)} \mid v \in N_{in}(u)\}, \\ \tau^{(k)}\{\boldsymbol{x}_{e\langle v_1,v_2\rangle}^{(k-1)} \mid e\langle v_1, v_2\rangle \in E_{N(u)}\}\}.\end{aligned} \tag{20}$$

We also incorporate edge label information into the initial edge features, as a modification of Eqn. 2:

$$\boldsymbol{x}_{e\langle u_1,u_2\rangle}^{(0)} = \boldsymbol{x}_{u_1}^{(0)} \oplus \boldsymbol{x}_{u_2}^{(0)} \oplus \boldsymbol{x}_{L(e\langle u_1,u_2\rangle)} \oplus |C\langle u_1, u_2\rangle|, \tag{21}$$

where $\boldsymbol{x}_{L(e)}$ denotes the embedding of the edge label $L(e)$. Existing methods can compute a representation for edge labels, such as RDF2Vec [59] and Ridle [75]. In our experiments (Sec. 7.5), we simply use one-hot encodings based on edge labels, which also yield good performance.

7 Experiments

7.1 Experiments Setup

7.1.1 Datasets. We select six real-world datasets for the experiments, as they are widely used in prior work on subgraph matching [9, 67, 68] and subgraph counting [73, 89]. They cover various domains, including biology (Yeast, HPRD), social networks (DBLP, YouTube), the web (EU2005), and citation networks (Patents). These datasets vary in terms of scale and difficulty (e.g., topology, density), and their statistics are provided in Tab. 2.

Table 2. Datasets statistics

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Sigma\rvert$	average degree
Yeast	3,112	12,519	71	8.0
HPRD	9,460	34,998	307	7.4
DBLP	317,080	1,049,866	15	6.6
EU2005	862,664	16,138,468	40	37.4
YouTube	1,134,890	2,987,624	25	5.3
Patents	3,774,768	16,518,947	20	8.8

7.1.2 Query graphs. We use queries from an influential subgraph matching survey [67]. These queries are generated by randomly extracting connected subgraphs from the data graph, following the approach adopted in previous work [9, 23, 69]. This will ensure every query has at least one embedding in the data graph. The queries vary in the number of vertices, ranging from 4 to 32, with 1800 queries for each dataset. Specifically, each query set with i query vertices, denoted as Q_i ($i = 4, 8, 16, 24, 32$), includes two types of queries: Q_{iD} (dense queries) and Q_{iS} (sparse queries). Following [67], a query is classified as dense if its average degree exceeds 3; otherwise, it is considered sparse. Each category contains 200 queries per dataset, except for the size-4 queries, which are not further subdivided into dense or sparse.

During the data collection phase for training, we fully explore the queries in Q_4 and partially explore those in Q_8, Q_{16}, and Q_{24} (Sec. 6.1). The query sets with 32 query vertices are only used for testing. For the training process, we randomly sample 80% of the queries from Q_4, Q_8, Q_{16}, and Q_{24} as the training set, with the remaining 20% and Q_{32} used for testing.

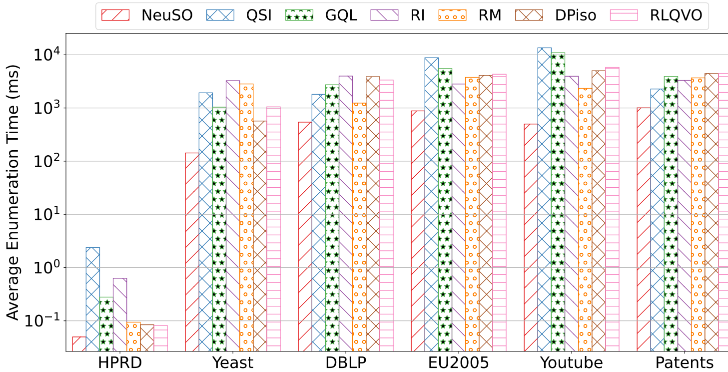


Fig. 6. Average enumeration time comparison.

7.1.3 Compared Methods. We compare our method with several traditional graph query ordering methods: QSI [62], GQL [24], RI [11], RM [68], and DPiso [23]. The first four methods are static ordering methods, while the last one, DPiso, is a dynamic ordering method that selects the next matched query vertex based on the partial match. All of these methods have been shown to be effective in numerous experiments and outperform other not selected methods [67, 88]. We include RLQVO [74] in our comparison and adopt its experimental settings. Additionally, we did not find any other learning-based ordering methods apart from it.

For a fair comparison, we keep the filter and enumeration components of the pipeline execution identical across all methods; the only difference lies in the matching order. We do not include other dynamic methods, such as VEQ [31], as it is almost identical to DPiso in terms of order, except for degree-one vertices, and its ordering requires a special enumerator.

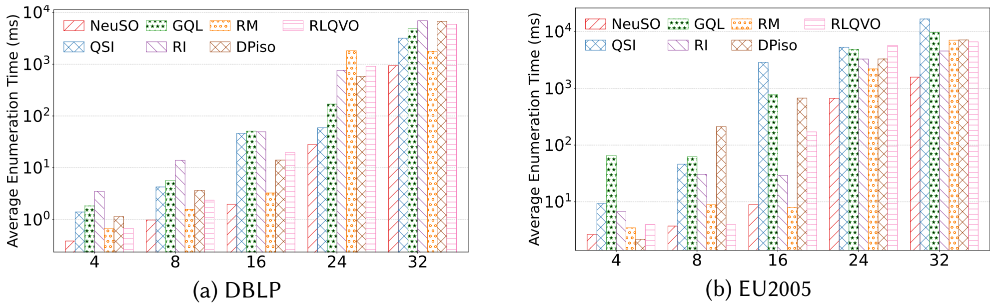
For NeuSO, we implement it in Python using the AdamW optimizer [44]. The initial feature dimension of the query graph encoder is 128, and the number of TriAT layers is 2, with each layer having a dimension of 64. We train the models for 100 epochs with a learning rate of 0.002. A learning rate scheduler is used to decays the learning rate by 20% every 20 steps. The query execution part of our experiments is adapted from the code in [67], which is implemented in C++ and stores the graph using adjacency lists. The codes of NeuSO can be found at https://github.com/fyulingi/NeuSO.

7.1.4 Setup. Our experiments are conducted on a server with 256GB RAM, running Ubuntu 20.04.5 LTS. The server has an Intel Xeon Gold 6326 2.90GHz CPU and an NVIDIA A100-PCIE-40GB GPU. We set a 350-second time threshold for each query. Each experiment is repeated three times, and the mean values are reported.

7.2 Results for Optimization

7.2.1 Enumeration Time Comparison. In this experiment, we use enumeration time as an indicator of the quality of the generated matching orders. To isolate the effect of matching orderings, we ensure that all methods use the same filtering mechanism (GQL) and enumeration method (QSI), eliminating the influence of other factors. This allows us to focus solely on performance differences arising from the various matching order strategies.

Fig. 6 presents the average enumeration time across the six datasets for different methods. Our results show that the matching orders generated by NeuSO improve the enumeration efficiency compared to other methods by a factor of 1.63 to 47.93. This demonstrates that, after training, the model effectively learns optimization-relevant information from the data graph, enabling it to generalize its experience to unseen queries.



(a) DBLP (b) EU2005

Fig. 7. Average enumeration time on DBLP and EU2005. The x-axis represents different query graph sizes V_Q.

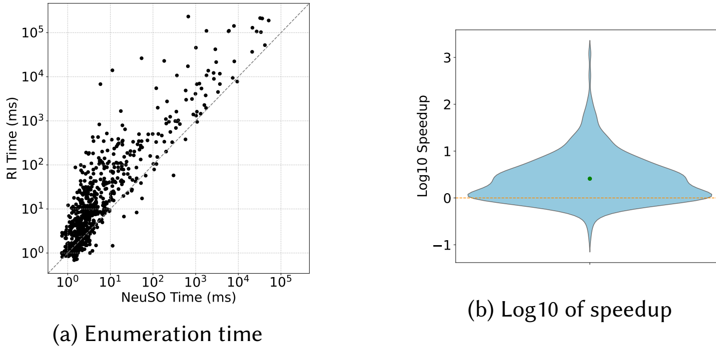
From Fig. 6, we observe that different matching orders result in significant variations in enumeration times. On the relatively simple HPRD dataset, all methods achieve low enumeration times, with the learning-based method RLQVO outperforming the other traditional methods. However, on more complex datasets such as DBLP and YouTube, RLQVO does not consistently outperform other methods and is 4.40 to 11.62 times slower than NeuSO. This performance gap can be attributed to several factors. First, RLQVO is trained using reinforcement learning and may not have encountered certain transition patterns in the entire CCG graph during training. In contrast, NeuSO is trained in a supervised manner using carefully collected training data, enabling it to learn more comprehensive mapping relationships between transition patterns and costs. Second, RLQVO employs GCN as its query graph encoder, which is less expressive and powerful compared to the TriAT encoder used by NeuSO. Additionally, NeuSO utilizes filtered statistics for feature initialization, which are more accurate than the naive statistics derived from the data graph used by RLQVO. Finally, RLQVO selects the next matching vertex by computing scores based on vertex representations, without considering higher-level subqueries, which limits its ability to optimize matching orders effectively.

Fig. 7 shows the enumeration time comparison on DBLP and EU2005 with different query graph sizes. Due to space limitations, we omit results for other datasets, which exhibit similar trends. The results indicate that NeuSO maintains consistently strong performance across different query size.

Note that the average enumeration time can be significantly affected by extreme values, such as those very slow queries. To better understand NeuSO's performance, we compared the enumeration time of each query on EU2005 between NeuSO and the second-best method (RI). The results are shown in a scatter plot in Fig. 8a. We find that NeuSO accelerates queries especially that other methods take a long time to process. Most of these queries are large, with 24 or 32 vertices, which require a significant amount of time to retrieve results (some even take up to 10^5 ms). Fig. 8b present the log10 speedup of NeuSO compared RI on EU2005. In general, the results show a long-tail distribution. The green point means that the average speedup is 2.58, indicating that when other methods choose a slow query plan, NeuSO selects a better execution plan, leading to significant improvements.

7.2.2 End-to-end Running Time Comparison. We designed NeuSO to strike a balance between efficiency and effectiveness. Fig. 9 compares the end-to-end running time for the YouTube and Patents datasets. While heuristic methods achieve faster optimization times, the gains from query optimization achieved by NeuSO outweigh the additional time costs, resulting in superior overall performance.

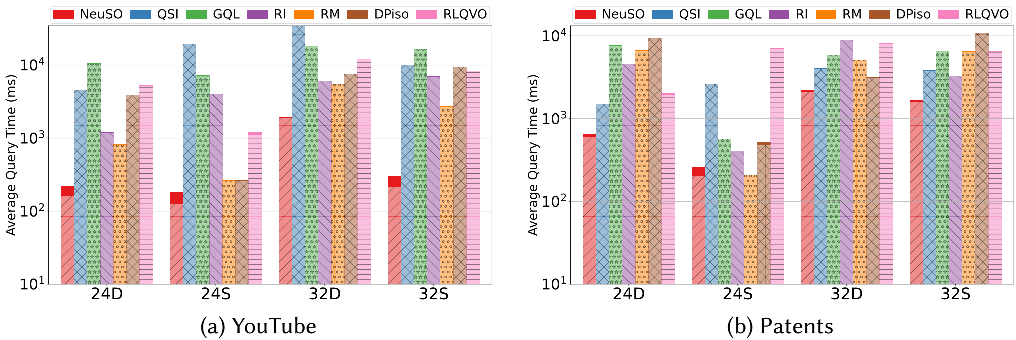
For a query graph Q, NeuSO requires $O(|V_Q|)$ decisions, each of which triggers one deep model invocation. Tab. 3 compares the optimization time of NeuSO and RLQVO. The efficiency advantage of



(a) Enumeration time

(b) Log10 of speedup

Fig. 8. Enumeration time comparison on EU2005 (vs. RI).



(a) YouTube

(b) Patents

Fig. 9. Average end-to-end query time comparison. The bottom bars with hatching represent filter and enumeration time, while the solid bars above represent optimization times.

NeuSO primarily stems from the fact that NeuSO runs only one GNN on the query graph, eliminating the need to re-run the query encoder after each decision, as required by RLQVO.

Table 3. Optimization time comparison (ms)

Methods	Q_4	Q_8	Q_{16}	Q_{24}	Q_{32}
NeuSO	9.49	20.05	36.90	56.63	82.55
RLQVO	12.94	29.35	61.18	93.30	125.48

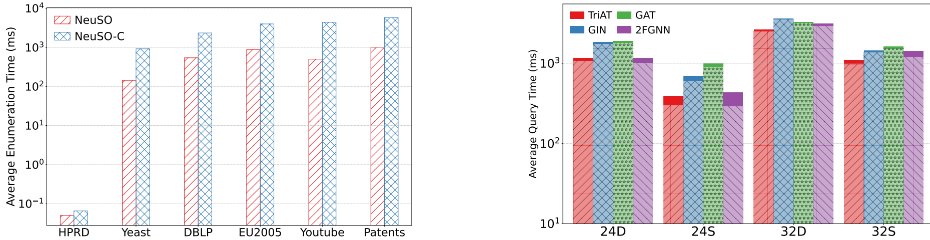
Table 4. Number of unsolved queries

Methods	Yeast	DBLP	EU2005	YouTube	Patents
NeuSO	**1**	28	**37**	**13**	**16**
QSI	6	42	96	41	24
GQL	5	50	72	39	25
RI	2	35	51	31	21
RM	**1**	**27**	44	19	20
DPiso	3	39	64	51	28
RLQVO	9	58	48	45	34

7.2.3 Unsolved Queries Number. The number of unsolved queries is an important metric for evaluating the quality of the matching order. Tab. 4 presents the number of unsolved queries across five datasets, excluding HPRD, where all methods return results within the required time due to the simplicity of the dataset. A query is considered unsolved if it cannot be answered within 350 seconds. On most datasets, NeuSO produces better matching orders, leading to a smaller number of unsolved queries compared to other methods.

7.3 Ablation Study

7.3.1 Ablation on Multi-task Learning. As described in Sec. 3, the multi-task learning framework of NeuSO jointly trains the model to estimate both cardinality and cost. We hypothesize that this design



(a) Ablation study on multi-task learning. (b) Ablation study on different GNN models (EU2005).

Fig. 10. Ablation studies. (a) Enumeration time comparison with NeuSO-C (trained w/o cardinality estimation); (b) End-to-end performance of different GNN variants used in NeuSO. The solid bars on top indicate optimization time.

improves the quality of subquery representations and enhances model robustness by enabling shared feature learning. To validate this hypothesis, we conduct the following ablation study.

As shown in Fig. 10a, compared to NeuSO-C, which is trained solely using subquery cost information (excluding cardinality), NeuSO achieves a speedup in enumeration time ranging from 1.31× to 8.78×. This result demonstrates the effectiveness and advantage of the multi-task learning framework.

7.3.2 Ablation For GNN Model. To evaluate the effectiveness of our proposed TriAT module, we conduct an ablation study by replacing it with several representative GNN architectures, including GIN [76], GAT [71], and 2FGNN (based on 2FWL) [48].

The results on the EU2005 are shown in Fig. 10b. TriAT consistently outperforms GIN and GAT, benefiting from its stronger expressive power. It also achieves performance comparable to the theoretically more powerful model 2FGNN. However, 2FGNN incurs higher computational costs, resulting in weaker end-to-end performance for some large queries. We further observe that 2FGNN occasionally produces suboptimal plans, suggesting that excessive structural awareness does not necessarily translate to better optimization performance. All these results highlight the advantage of TriAT in balancing expressiveness and efficiency for query optimization tasks.

7.4 Results for Cardinality Estimation

NeuSO is trained via a multi-task approach, with cardinality estimation as a byproduct (although it is not required for matching order generation in NeuSO). We evaluate its effectiveness and efficiency compared to other methods.

We selected three state-of-the-art (SOTA) neural methods, LSS [89], NeurSC [73], and GNCE [61], as well as two traditional methods, Alley [33] and Fast [63], for comparison. These methods were chosen because they have significantly outperformed other cardinality estimation techniques. We used the official implementations provided by the authors and conducted experiments following the settings and parameters described in the original papers.

7.4.1 Accuracy. The results of cardinality estimation accuracy are illustrated in Fig. 11, where we only show the results for queries whose cardinalities can be accurately determined within a reasonable time frame. To distinguish between underestimation and overestimation, we plot the logarithmic estimation ratio between predicted and true cardinalities: $\log_{10}(\hat{c}/c)$, where overestimation is depicted above $y = 0$ and underestimation below $y = 0$.

As depicted in Fig. 11, NeuSO consistently outperforms the other three learning-based cardinality estimation methods: LSS, NeurSC, and GNCE. LSS provides stable estimates, with balanced overestimations and underestimations. NeurSC performs better than LSS on the HPRD and Yeast datasets, as filter effectively captures more query-relevant information from the data graph. However, NeurSC performs poorly on larger datasets such as DBLP and EU2005, and tends to underestimate as the

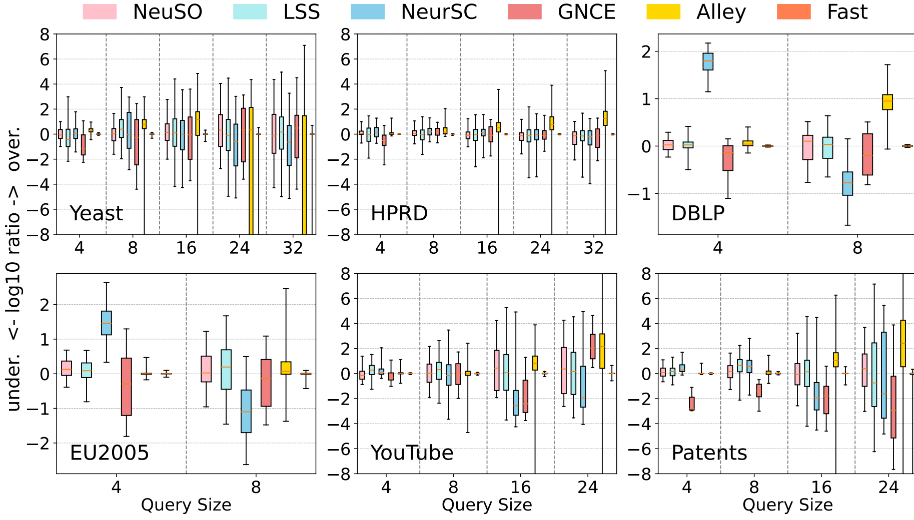


Fig. 11. Cardinality estimation accuracy comparison. The boxplot shows $\log_{10}(\hat{c}/c)$, the base-10 log of the estimation ratio. The lower and upper whiskers denote the min and max values. The bounds of the box represent the 25th and 75th percentiles, with the median indicated by the line inside the box. A denser box closer to 0 indicates higher accuracy.

query size increases. Similarly, GNCE performs poorly on large datasets such as YouTube and Patents.

Alley performs well for small queries (e.g., 4 vertices), but deteriorates significantly as query size grows. This is primarily due to sampling failure: with over 16 vertices, Alley's method struggles to generate valid results, often leading to underestimation.

Fast, the SOTA sampling-based method, improves upon the sampling approach by sampling from the filtered data graph, increasing the likelihood of successful sampling. Our experiments show that Fast achieves the best performance in most cases. However, it also encounters sampling failures for large queries (e.g., size 32 on Yeast and size 24 on Patents). NeuSO performs better than other methods except Fast in most scenarios, leveraging a more expressive GNN and utilizing query-specific information through its filter.

7.4.2 Efficiency. Although Fast achieves higher accuracy than NeuSO, its estimation cost is prohibitively high for query optimizers. Fig. 12 compares the running time for a single estimate (entire query) on the EU2005 and DBLP datasets. For NeuSO and NeurSC, the filtering time is included.

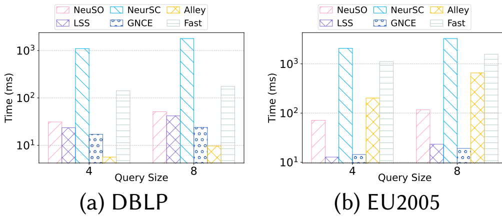


Fig. 12. Average cardinality estimate time.

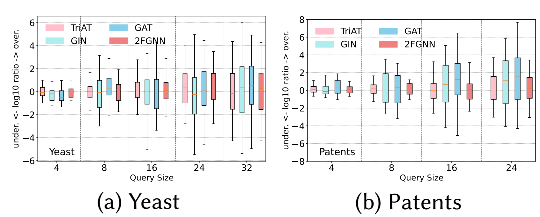
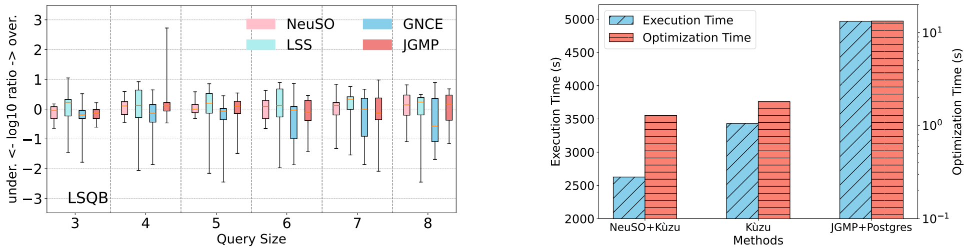


Fig. 13. Model ablation for cardinality estimation.

NeurSC incurs the highest time cost due to its use of a GNN on the filtered graph. Fast likewise requires considerable time for filtering and sampling. In contrast, NeuSO only gathers lightweight



(a) Cardinality estimation accuracy on LSQB. (b) Execution & optimization time comparison on LSQB.

Fig. 14. Experiments on LSQB.

statistics during filtering, leading to improved efficiency. In our experiments, NeuSO achieved a speedup of 1.75× to 222.12× over Fast across various query sizes and datasets. LSS and GNCE apply a GNN directly to the query graph without a filtering stage, and are thus faster per invocation than NeuSO. However, the filtering in NeuSO is performed only once and amortized across multiple model calls during subquery optimization. As a result, when integrated into a query optimizer, NeuSO benefits from amortized filtering costs and reduced redundant computation across subqueries, whereas LSS and GNCE are not designed for such use cases.

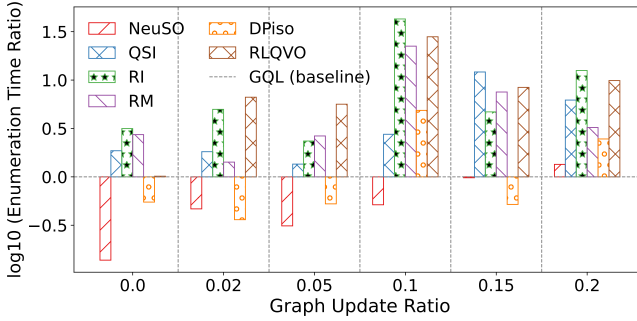
7.4.3 Further Ablation Results for TriAT. As an additional ablation study, we replace our proposed TriAT with other GNN architectures, including GIN [76], GAT [71], and 2FGNN [48], for cardinality estimation task. Fig. 13 presents the experimental results on the Yeast and Patents datasets. As shown in the figure, TriAT consistently outperforms both GIN and GAT, which aligns with its more expressive power. Moreover, while TriAT exhibits comparable accuracy with more expressive 2FGNN, it achieves this with lower computational cost (2FGNN requires up to 8.16 times more computation time than TriAT for query graphs with 32 vertices). This highlights TriAT's favorable trade-off between accuracy and efficiency.

7.5 Experiments on Edge-labeled Graphs

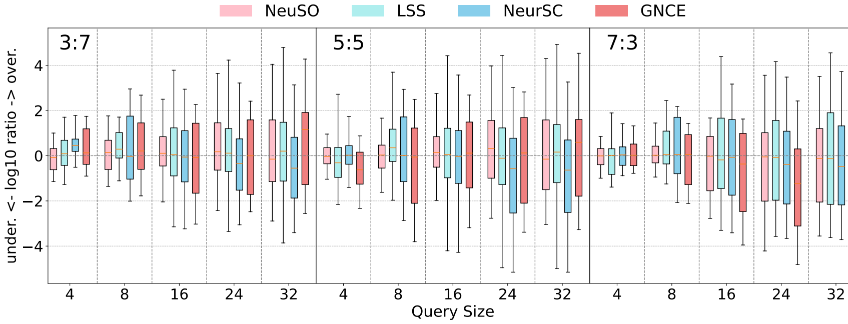
In addition to the datasets containing only vertex labels, we also conduct experiments on the Labeled Subgraph Query Benchmark (LSQB) [49], a directed benchmark featuring both vertex and edge labels based on LDBC SNB [17]. LSQB focuses on complex join queries that are typical in social network analysis. Our experiments were conducted using LSQB with a scale factor of 3, which contains 11.3 million vertices and 66.2 million edges.

Since LSQB includes a schema graph representing possible vertex linkages, certain learning-based optimization methods for relational databases, such as JGMP [58], can also be applied. We therefore include it in our experiments for comparison. As the original LSQB only provides 9 queries, we generate 75 queries for each number of query vertices from $\{3, 4, 5, 6, 7, 8\}$. We randomly split 80% of the generated queries for training and use the remaining 20% for testing. Reported results are based on the test set. More details can be found in the full version of our paper [79].

7.5.1 Cardinality Estimation Accuracy. We compare NeuSO with other subgraph cardinality estimation methods (LSS and GNCE) that support directed and edge-labeled graphs. Methods such as NeurSC, Alley, and Fast are excluded as they do not handle directed edge-labeled graphs. We also include JGMP [58], a state-of-the-art cardinality estimator for relational databases, for comparison. As shown in Fig. 14a, NeuSO achieves a good balance between accuracy and stability, consistent with the results in Sec. 7.4.



(a) Enumeration time comparison under graph updates, where GQL is taken as the baseline method.



(b) Cardinality estimation under varying training workload.

Fig. 15. Robustness (a) and transferability (b) on Yeast.

7.5.2 Results for Optimization. We further evaluate the optimization performance of NeuSO on Kùzu [19], an open-source graph database designed for complex, join-heavy analytical workloads. Since the baseline methods discussed in Sec. 7.2 do not support directed edge-labeled graphs, we exclude them from this experiment. Instead, we inject the query plans generated by NeuSO into Kùzu and compare them with plans produced by Kùzu's built-in DP-based optimizer. Additionally, we consider a relational baseline by replacing PostgreSQL's cardinality estimator with JGMP [58], and executing the queries on PostgreSQL [1]. A uniform timeout of 10 minutes is set for each query across all methods.

Fig. 14b presents the comparison in terms of total execution time (including optimization time). On Kùzu, the execution plans generated by NeuSO run 1.31× faster than those generated by the default optimizer, and the optimization process itself is 1.41× faster than Kùzu's optimizer. Due to differences in storage and execution engines, the relational method is significantly slower. These results demonstrate that NeuSO can generate more efficient execution plans and is applicable in real-world systems.

7.6 Robustness and Transferability

In this section, we evaluate the robustness and transferability of NeuSO under two realistic scenarios: (i) updates to the data graph, and (ii) workload shifts. We test on Yeast dataset as the queries are evenly distributed on various sizes and true cardinality.

7.6.1 Robustness to Graph Updates. Although NeuSO is designed for static graphs, it can tolerate moderate changes in the data graph, as the query encoder receives input statistics that reflect the updated data graph after filtering. We consider three types of updates: vertex label changes, edge

insertions, and deletions, applied in a fixed ratio of 1:1:1, with total update ratios of 0%, 2%, 5%, 10%, 15%, and 20%. As the data graph differs across update ratios, absolute enumeration time is incomparable. Thus, we use GQL as the baseline within each ratio to compute relative enumeration times.

The results are presented in Fig. 15a. When the update ratio is no more than 10%, NeuSO still produces high-quality matching orders. However, when the graph changes significantly (i.e., over 10%), NeuSO becomes slower than GQL and DPiso, suggesting that retraining is necessary.

7.6.2 Workload Transferability. We evaluate the transferability of NeuSO by constructing diverse workloads and comparing its estimation accuracy with other learning-based methods, including LSS, NeurSC, and GNCE. Specifically, we enlarge the query pool to a size of 2000, consisting of 400 queries for each query size in $\{4, 8, 16, 24, 32\}$, whose cardinalities can be computed without exceeding limit time. We designate 500 queries as the test set (100 for each size), and construct three distinct training workloads, each containing 1000 queries drawn from sizes 4, 8, 16, and 24. Queries of size 32 are excluded from training and used solely for evaluating generalization to unseen query sizes. In each workload, the ratio between small queries (sizes 4 and 8) and large queries (sizes 16 and 24) varies among {3:7, 5:5, 7:3}.

The result is shown in Fig. 15b. Existing methods tend to underestimate the cardinalities of large queries in workloads dominated by small queries, and overestimate the cardinalities of small queries in workloads dominated by large queries. In contrast, NeuSO achieves more stable estimation performance across different workload compositions, as it leverages information from subqueries of large queries during training.

8 Related Works & Discussions

8.1 Related Works

8.1.1 Learned Query Optimization for Relational Databases. Learning-based optimizers for relational databases can be categorized into three types: (1) learning-based cardinality estimators combined with traditional cost estimators and DP plan enumerators [25, 34, 40, 58, 81], (2) learning-based cost estimators with traditional DP plan enumerators [45, 47, 66], and (3) direct learning-based query plan generators [14, 46, 80, 84]. Each of these approaches replaces one or more components of the traditional optimizer to improve efficiency and accuracy. There are also some recent works that utilize LLMs for optimization. Most of them focus on hint generation [4, 82] or query rewriting [41, 42, 91].

8.1.2 Sugraph Query Optimizer. Subgraph matching order has been widely studied, and most approaches rely on relatively simple heuristics. For instance, GraphQL [24] processes queries by prioritizing vertices with smaller candidate sets before moving on to those with larger sets. Similarly, CFL [9] and RapidMatch [68] initiate the matching process from the denser regions of the query graph. DPiso [23] and VEQ [31] further enhance the matching process by dynamically adjusting the matching order during execution. RLQVO [74] proposes a matching order method based on reinforcement learning, which models the matching order selection as an MDP. In addition to optimizations for general subgraph queries, several works focus on specific structures, e.g., paths [52, 53, 77].

In practical graph database systems, most modern systems (e.g., Neo4j [28], GraphFlow [50], Kùzu [19], and gStore [78]) employ dynamic programming to determine optimal execution plans. When dealing with complex query graphs, these systems may also resort to greedy heuristic search strategies. Some optimization methods [3, 30, 36] rely on cost models tailored to the specific

execution operators they adopt (such as factorization). These designs are orthogonal to our method, which can be applied independently of such operator-specific assumptions.

8.1.3 Subgraph Counting. Subgraph counting aims to provide cardinality estimates for the entire given query graph. Exact methods compute the count by exhaustively enumerating all subgraph matches in the data graph [26, 27, 54], while approximate methods utilize graph summaries or sampling strategies for estimation. For instance, CharacteristicSets [51] decomposes queries into star-shaped subqueries and estimates them via precomputed statistics; SumRDF [65] summarizes the graph by label and estimates counts on the abstracted structure. Sampling-based approaches further improve efficiency by probabilistically exploring subgraphs [33, 39, 63, 72, 78, 90]. These methods focus on reducing bias and variance through advanced sampling strategies.

Recent advances leverage machine learning to model query-data graph interactions for count prediction [43, 61, 73, 89], offering a paradigm shift from traditional algorithmic designs.

8.2 Discussions

Compared with reinforcement learning-based optimizer RLQVO [74] and other learning-based subgraph counting approaches including LSS [89], NeurSC [73], and GNCE [61], our NeuSO demonstrates three key distinctions:

Enhanced Graph Encoding. While RLQVO adopts GCN [35] and both LSS and NeurSC employ GIN [76], more recent efforts such as GNCE [61] propose advanced encoders like TPN. In comparison, NeuSO introduces TriAT, which provides stronger expressive power than these message-passing neural networks.

Multi-Task Capability. Existing methods exhibit functional limitations: RLQVO focuses solely on matching order generation, while LSS, NeurSC, and GNCE estimate only the full query graph's cardinality. In contrast, NeuSO establishes a unified multi-task framework that simultaneously predicts both cardinality and computational cost for arbitrary subqueries, while maintaining the capability to generate high-performance matching orders.

Efficient Cross-Graph Interaction. All of RLQVO, LSS, and GNCE rely on static statistical features from data graphs (loose statistics on the number of vertices (RLQVO) or pre-learned embeddings on the data graph (LSS and GNCE)). Additionally, RLQVO needs to perform GNN computations iterately during matching order selection. Although NeurSC incorporates dynamic graph filtering, its dual GNN architecture (query graph + filtered data graph) incurs significant overhead. NeuSO innovates through a lightweight interaction mechanism that propagates filtered statistics through a single GNN operating exclusively on the query graph, which is more efficient than NeurSC while maintaining high accuracy.

9 Conclusions

In this paper, we propose NeuSO, a unified and efficient framework for subgraph query optimization that generates high-quality vertex matching orders. NeuSO employs a novel GNN-based architecture on the query graph and leverages a multi-task learning strategy to jointly predict subquery cardinalities and costs. Based on these predictions, we design a new top-down plan enumerator that selects the next vertex accordingly. Experimental results show that NeuSO outperforms existing methods in both enumeration time and end-to-end execution time.

Acknowledgments

This work was supported by The National Key Research and Development Program of China under grant 2023YFB4502303 and NSFC under grant 62532001. Lei Zou is the corresponding author of this work.

References

[1] [n. d.]. PostgreSQL. https://www.postgresql.org/.

[2] [n. d.]. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/.

[3] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Stanley Clark, and Mark H. Chignell. 2021. Answer Graph: Factorization Matters in Large Graphs. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra (Eds.). OpenProceedings.org, 493–498. doi:10.5441/002/EDBT.2021.57

[4] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. 2024. The unreasonable effectiveness of llms for query optimization. *arXiv preprint arXiv:2411.02862* (2024).

[5] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1–39.

[6] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[7] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. 175–188.

[8] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.

[9] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

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

[11] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14 (2013), 1–13.

[12] Rich Caruana. 1997. Multitask learning. *Machine learning* 28 (1997), 41–75.

[13] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-performance dynamic pattern matching over disordered streams. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 220–231.

[14] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A learned optimizer towards generating efficient and robust query execution plans. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1777–1789.

[15] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 135–144.

[16] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.

[17] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[18] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th international conference on database theory*. 8–21.

[19] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu graph database management system. In *The Conference on Innovative Data Systems Research*, Vol. 7. 25–35.

[20] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*. 1433–1445.

[21] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[22] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[23] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[24] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 405–418.

[25] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. [n. d.]. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment* 13, 7 ([n. d.]).

[26] Himanshu and Sarika Jain. 2017. Impact of memory space optimization technique on fast network motif search algorithm. In *Advances in Computer and Computational Sciences: Proceedings of ICCCCS 2016, Volume 1*. Springer, 559–567.

[27] Tomaž Hočevar and Janez Demšar. 2017. Combinatorial algorithm for counting small induced graphs and orbits. *PloS one* 12, 2 (2017), e0171428.

[28] Neo4j Inc. 2024. *Neo4j*. https://neo4j.com/

[29] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast redundancy-reduced subgraph matching. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[30] Hasara Kalumin and Amol Deshpande. 2025. Optimizing Queries with Many-to-Many Joins. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 3668–3681.

[31] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*. 925–937.

[32] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming Subgraph Isomorphism for RDF Query Processing. *Proceedings of the VLDB Endowment* 8, 11 (2015).

[33] Kyoungmin Kim, Hyeonji Kim, George Fletcher, and Wook-Shin Han. 2021. Combining sampling and synopses with worst-case optimal runtime and quality guarantees for graph pattern cardinality estimation. In *Proceedings of the 2021 International Conference on Management of Data*. 964–976.

[34] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*.

[35] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJU4ayYgl

[36] Wilco v Leeuwen, Thomas Mulder, Bram Van De Wall, George Fletcher, and Nikolay Yakovets. 2022. Avantgraph query processing engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3698–3701.

[37] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[38] Andrei Leman and Boris Weisfeiler. 1968. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsiya* 2, 9 (1968), 12–16.

[39] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.

[40] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proceedings of the VLDB Endowment* 17, 2 (2023), 197–210.

[41] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-Based Rewrite System for Boosting Query Efficiency. *Proceedings of the VLDB Endowment* 18, 1 (2024), 53–65.

[42] Jie Liu and Barzan Mozafari. 2024. Query rewriting via large language models. *arXiv preprint arXiv:2403.09060* (2024).

[43] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. 2020. Neural subgraph isomorphism counting. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1959–1969.

[44] Ilya Loshchilov and Frank Hutter. [n. d.]. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[45] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.

[46] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.

[47] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1733–1746.

[48] Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. 2019. Provably powerful graph networks. *Advances in neural information processing systems* 32 (2019).

[49] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–11.

[50] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.

[51] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 984–994.

[52] Van-Quyet Nguyen, Quyet-Thang Huynh, and Kyungbaek Kim. 2022. Estimating searching cost of regular path queries on large graphs by exploiting unit-subqueries. *Journal of Heuristics* (2022), 1–21.

[53] Yue Pang, Lei Zou, Jeffrey Xu Yu, and Linglin Yang. 2024. Materialized View Selection & View-Based Query Planning for Regular Path Queries. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[54] Pedro Paredes and Pedro Ribeiro. 2013. Towards a faster network-centric subgraph census. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 264–271.

[55] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34, 3 (2009), 1–45.

[56] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[57] N Pržulj, Derek G Corneil, and Igor Jurisica. 2006. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics* 22, 8 (2006), 974–980.

[58] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proceedings of the VLDB Endowment* 17, 4 (2023), 740–752.

[59] Petar Ristoski and Heiko Paulheim. 2016. Rdf2vec: Rdf graph embeddings for data mining. In *International semantic web conference*. Springer, 498–514.

[60] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB journal* 29 (2020), 595–618.

[61] Tim Schwabe and Maribel Acosta. 2024. Cardinality estimation over knowledge graphs with embeddings and graph neural networks. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[62] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

[63] Wonseok Shin, Siwoo Song, Kunsoo Park, and Wook-Shin Han. 2024. Cardinality Estimation of Subgraph Matching: A Filtering-Sampling Approach. *Proceedings of the VLDB Endowment* 17, 7 (2024), 1697–1709.

[64] Tom AB Snijders, Philippa E Pattison, Garry L Robins, and Mark S Handcock. 2006. New specifications for exponential random graph models. *Sociological methodology* 36, 1 (2006), 99–153.

[65] Giorgio Stefanoni, Boris Motik, and Egor V Kostylev. 2018. Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In *Proceedings of the 2018 World Wide Web Conference*. 1043–1052.

[66] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment* 13, 3 (2019), 307–319.

[67] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[68] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapidmatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[69] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proceedings of the VLDB Endowment* 5, 9 (2012).

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[71] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[72] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P Chakkappen. 2015. Join size estimation subject to filter conditions. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1530–1541.

[73] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. 2022. Neural subgraph counting with Wasserstein estimator. In *Proceedings of the 2022 International Conference on Management of Data*. 160–175.

[74] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2022. Reinforcement learning based query vertex ordering model for subgraph matching. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 245–258.

[75] Tobias Weller and Maribel Acosta. 2021. Predicting instance type assertions in knowledge graphs using stochastic neural networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2111–2118.

[76] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryGs6iA5Km

[77] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query planning for evaluating SPARQL property paths. In *Proceedings of the 2016 International Conference on Management of Data*. 1875–1889.

[78] Linglin Yang, Lei Yang, Yue Pang, and Lei Zou. 2022. gCBO: A Cost-based Optimizer for Graph Databases. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 5054–5058.

[79] Linglin Yang, Lei Zou, and Zhao Chunshan. 2025. NeuSO: Neural Optimizer for Subgraph Queries. *arXiv preprint arXiv:2509.23775* (2025).

[80] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a query optimizer without expert demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*. 931–944.

[81] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment* 14, 1 (2020), 61–73.

[82] Zhiming Yao, Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2025. A query optimization method utilizing large language models. *arXiv preprint arXiv:2503.06902* (2025).

[83] Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Efficient Exact Subgraph Matching via GNN-Based Path Dominance Embedding. *Proceedings of the VLDB Endowment* 17, 7 (2024), 1628–1641.

[84] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1297–1308.

[85] Bohang Zhang, Guhao Feng, Yiheng Du, Di He, and Liwei Wang. 2023. A complete expressiveness hierarchy for subgraph gnns via subgraph weisfeiler-lehman tests. In *International Conference on Machine Learning*. PMLR, 41019–41077.

[86] Jie Zhang, Yuxiao Dong, Yan Wang, Jie Tang, and Ming Ding. 2019. Prone: Fast and scalable network representation learning. In *IJCAI*, Vol. 19. 4278–4284.

[87] Yu Zhang and Qiang Yang. 2021. A survey on multi-task learning. *IEEE transactions on knowledge and data engineering* 34, 12 (2021), 5586–5609.

[88] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–29.

[89] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A learned sketch for subgraph counting. In *Proceedings of the 2021 International Conference on Management of Data*. 2142–2155.

[90] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*. 1525–1539.

[91] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using monte carlo tree search. *Proceedings of the VLDB Endowment* 15, 1 (2021), 46–58.

[92] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. 2011. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment* 4, 8 (2011), 482–493.

Received April 2025; revised July 2025; accepted August 2025