R2O: A Dual-Layer Framework for Joint Rewriting and Ordering in Distributed Property Graph Query Optimization

MIN SHI, Hunan University, China
PENG PENG*, Hunan University, China
XIN XIAO, Hunan University, China
LEI ZOU, Peking University, China
KENLI LI, Hunan University, China
XU ZHOU, Hunan University, China

In distributed property graph systems, complex pattern queries are typically decomposed into subqueries that can be independently executed within individual partitions. However, the integration of their results requires inter-partition joins, which incur significant communication overhead. Moreover, the order of inter-partition joins plays a pivotal role in determining the size of intermediate results, which subsequently affects the overall efficiency of distributed query processing. To address these challenges, we propose **R2O (Rewriting to Ordering)**, a dual-layer framework that jointly optimizes both inter-partition joins and join order for distributed pattern queries. We first introduce a partition-aware query rewriting strategy, which restructures and merges subqueries at partition boundaries to reduce intermediate results during inter-partition joins. Building on this strategy, R2O employs graph neural networks and reinforcement learning to construct a local rewriting model and a global ordering model. This unified end-to-end model effectively reduces intermediate results during inter-partition joins and identifies effective join orders, enabling efficient distributed query plan optimization. Experimental results on billion-scale property graphs indicate that R2O is compatible with different partitioning methods and yields 1–2 orders of magnitude speedup (up to 3 orders in some queries) over state-of-the-art query optimization techniques.

CCS Concepts: • **Information systems → Distributed retrieval**; • **Mathematics of computing → Graph algorithms**.

Additional Key Words and Phrases: Distributed query processing, graph pattern query

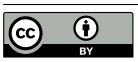
ACM Reference Format:
Min Shi, Peng Peng, Xin Xiao, Lei Zou, Kenli Li, and Xu Zhou. 2026. R2O: A Dual-Layer Framework for Joint Rewriting and Ordering in Distributed Property Graph Query Optimization. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 71 (February 2026), 27 pages. https://doi.org/10.1145/3786685

1 Introduction

Property graphs are widely used in real-world applications such as social networks, recommendation systems, bioinformatics, and transportation [34]. In a property graph, each node and edge is associated with labels and properties, supporting both structural and semantic relationships within complex networks. With the continuous growth of data scale, efficient management in distributed

*Peng Peng is the corresponding author.

Authors' Contact Information: Min Shi, shimin22@hnu.edu.cn, Hunan University, Changsha, Hunan, China; Peng Peng, hnu16pp@hnu.edu.cn, Hunan University, Changsha, Hunan, China; Xin Xiao, xiaoxinxx@hnu.edu.cn, Hunan University, Changsha, Hunan, China; Lei Zou, zoulei@pku.edu.cn, Peking University, Beijing, China; Kenli Li, lkl@hnu.edu.cn, Hunan University, Changsha, Hunan, China; Xu Zhou, zhxu@hnu.edu.cn, Hunan University, Changsha, Hunan, China.



ACM 2836-6573/2026/2-ART71
https://doi.org/10.1145/3786685

environments has become increasingly critical, leading to the division of large property graphs into multiple partitions distributed across different sites [11].

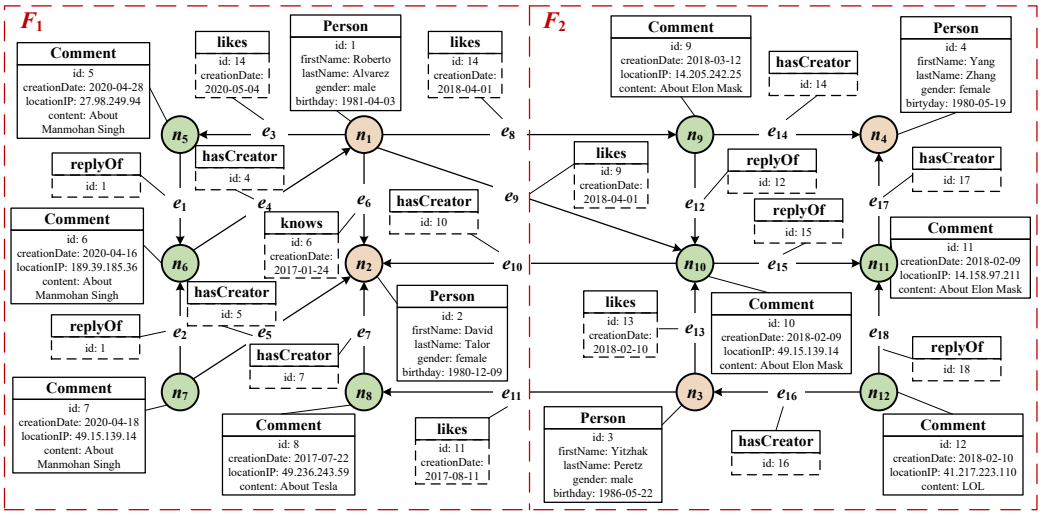


Fig. 1. An Example of Distributed Property Graph.

Example 1. ***(Distributed Property Graph)*** *Figure 1 illustrates an example of a distributed property graph G representing a social network. In this graph, nodes $\{n_1, \ldots, n_4\}$ are labeled as "Person" and are interconnected by directed edges labeled as "knows", which indicates that one person knows another. The nodes $\{n_5, \ldots, n_{12}\}$ are labeled as "Comment" and are connected by directed edges labeled as "replyOf", representing that one comment replies to another. Additionally, a "Comment" node is linked to a "Person" node via the "hasCreator" edge, signifying that the comment's creator is that person. Similarly, a "Person" node is connected to a "Comment" node through the "likes" edge, indicating that the person likes the comment.*

This social network graph is divided into two partitions F_1 and F_2, where edges $\{e_8, e_9, e_{10}, e_{11}\}$ are referred to as inter-partition edges. For illustration we depict a node-disjoint partitioning.

Graph pattern queries are essential for extracting information from property graphs [7, 13] and form the foundation of most query languages, including Cypher [10], GQL [28], and Gremlin [33]. Such queries identify structural patterns and apply property constraints, enabling complex relationship analysis.

In distributed property graph systems, large graphs are typically partitioned into multiple partitions. To ensure completeness, boundary elements are replicated in all relevant partitions—i.e., inter-partition edges (together with the labels of their remote endpoints) under node-disjoint partitioning, and inter-partition nodes (labels only) under edge-disjoint partitioning [32]. Consequently, pattern queries are decomposed into subqueries that can be executed independently on each partition [20, 45]. The decomposition is defined by the chosen partitioning scheme and handled by the underlying system. An optimizer then generates an execution plan [23] that specifies how these subqueries are allocated and joined across partitions. Each site executes its allocated subqueries, and the intermediate results are then joined or aggregated to form the final results. Throughout

the paper, examples are used for exposition and do not assume any edge directions or partitioning strategy[1].

Example 2. ***(Graph Pattern Query Q_1)*** *Figure 2 illustrates a graph pattern query Q_1, which matches a Person" node p_1 and a Comment" node c_1 connected by a "hasCreator" edge. The query specifies that p_1 must have gender = female and c_1 must satisfy creationDate < 2019-01-01.*

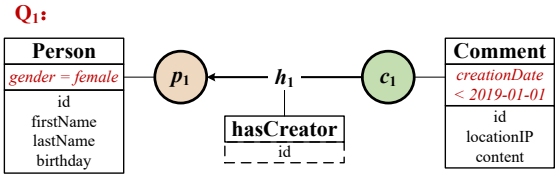


Fig. 2. An Example of Graph Pattern Query Q_1.

When Q_1 is executed on the distributed property graph G, it cannot be processed entirely within a single partition and is typically decomposed into two subqueries (according to the partitioning method), each of which can be executed independently on each partition (in this example, the minimal unit is a star with a local center retaining properties). Concretely, q_1: (p_1:Person)<-[h_1:hasCreator]-(:Comment) and q_2: (:Person)<-[h_1:hasCreator]-(c_1:Comment). Accordingly, q_1 produces five matches in total across F_1 and F_2: (n_2)<-[e_5]-(n_7), (n_2)<-[e_7]-(n_8), (n_2)<-[e_{10}]-(n_{10}), (n_4)<-[e_{14}]-(n_9), and (n_4)<-[e_{17}]-(n_{11}). Similarly, q_2 yields five matches: (n_2)<-[e_7]-(n_8), (n_2)<-[e_{10}]-(n_{10}), (n_4)<-[e_{14}]-(n_9), (n_4)<-[e_{16}]-(n_{12}), and (n_4)<-[e_{17}]-(n_{11}). After aggregation and joining of these intermediate results, only four final results are generated: (n_2)<-[e_7]-(n_8), (n_2)<-[e_{10}]-(n_{10}), (n_4)<-[e_{14}]-(n_9), and (n_4)<-[e_{17}]-(n_{11}).

1.1 Motivation

However, distributed query process often generates a large number of intermediate results, of which only a small fraction contribute to the final results [40]. Furthermore, aggregation and join operations introduce substantial communication overhead across partitions. Therefore, effective query planning is crucial to minimize redundant intermediate results and reduce communication costs, thereby improving the performance of distributed graph pattern queries [9, 43].

This inefficiency stemming from excessive intermediate results primarily arises from two factors: (1) some intermediate results do not contribute to the final result and can be eliminated within their local partitions; (2) some final results are entirely contained within a single partition and do not require inter-partition joins, making such communication unnecessary.

To illustrate this in more detail, consider the query process presented in Example 2. First, two intermediate results (n_2)<-[e_5]-(n_7) and (n_4)<-[e_{16}]-(n_{12}) do not contribute to the final output and could be joined and eliminated within their respective partitions. Due to the limitations of the current query decomposition strategy, these results are nevertheless communicated across partitions, leading to unnecessary communication overhead. Second, some final results, such as (n_2)<-[e_7]-(n_8), (n_4)<-[e_{14}]-(n_9), and (n_4)<-[e_{17}]-(n_{11}), do not involve inter-partition joins. Despite this, they must still be communicated across partitions for aggregation and joining, resulting in redundant communication costs.

On the other hand, to show the importance of effective query planning in reducing redundant intermediate results and reducing communication costs, consider another example query Q_2.

[1]Notation (Cypher-style [10]): nodes (x : Label) (named) and (: Label) (anonymous); edges as -[e : Label]-> or <-[e : Label]- with the arrow indicating direction.

Example 3. ***(Graph Pattern Query Q_2)*** *Figure 3 extends Q_1 to Q_2 by adding another "Person" node (p_2), which is connected to c_1 via an edge (l_1) labeled as "likes". This indicates that p_2 likes the comment created by p_1.*

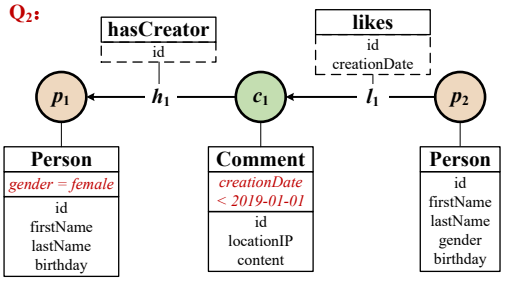


Fig. 3. An Example of Graph Pattern Query Q_2.

When Q_2 is executed on graph G, its answer can be obtained by joining and aggregating the outputs of the logical subqueries q_3: (:Comment)<-[l_1:likes]-(p_2:Person) and q_4: (p_1:Person)<-[h_1:hasCreator]-(c_1:Comment)<-[l_1:likes]-(:Person). Concretely, evaluating q_3 on G yields five matches: (n_5)<-[e_3]-(n_1), (n_9)<-[e_8]-(n_1), (n_{10})<-[e_9]-(n_1), (n_8)<-[e_{11}]-(n_3), and (n_{10})<-[e_{13}]-(n_3). Evaluating q_4 on G yields four matches: (n_2)<-[e_7]-(n_8)<-[e_{11}]-(n_3), (n_2)<-[e_{10}]-(n_{10})<-[e_9]-(n_1), (n_4)<-[e_{14}]-(n_9)<-[e_8]-(n_1), and (n_2)<-[e_{10}]-(n_{10})<-[e_{13}]-(n_3). At runtime, q_3 and q_4 may be further decomposed into subqueries that can be executed on each partition (see section 2.2), and only their outputs participate in the global joins/aggregations. After aggregation and joining, the final results of Q_2 are: (n_2)<-[e_7]-(n_8)<-[e_{11}]-(n_3), (n_2)<-[e_{10}]-(n_{10})<-[e_9]-(n_1), (n_4)<-[e_{14}]-(n_9)<-[e_8]-(n_1), and (n_2)<-[e_{10}]-(n_{10})<-[e_{13}]-(n_3).

The order in which subqueries are executed has a significant impact on the generation of unnecessary intermediate results. For example, in existing join strategies such as the semi-join, executing q_3 before q_4 produces an unnecessary intermediate result (n_5)<-[e_3]-(n_1), while executing q_4 first avoids this extra computation. Therefore, the choice of execution order directly affects both the amount of intermediate data generated and the overall communication overhead in distributed query processing.

1.2 Our solution

To address the above challenges, we propose **R2O (Rewriting to Ordering)**, a novel dual-layer learning framework for distributed property graph pattern queries. R2O consists of two key components: a local rewriting model and a global ordering model.

First, R2O introduces a partition-aware query rewriting strategy. By leveraging inter-partition edge information, this approach restructures each pair of neighboring, independently executable subqueries into two components: Boundary Subqueries, which focus on inter-partition edges and extended nodes, and Local-Join Subqueries, which involve only internal nodes and edges within a single partition. For instance, in Example 2, intermediate results that do not contribute to the final output can be joined and eliminated locally within their respective partitions, without being transmitted across the network. The local rewriting model, based on graph neural networks (GNNs) and reinforcement learning, iteratively selects and rewrites high-quality pairs of candidate subqueries. A negative reward discourages unnecessary rewriting, encourages termination when further rewriting is unlikely to yield benefits.

Second, to further optimize distributed query execution, R2O employs a global ordering model that determines the optimal join sequence of the rewritten subqueries, as motivated by the challenges in Example 3. This model leverages GNN-based feature representations and reinforcement learning to select the best execution order, minimizing unnecessary computation and further reducing intermediate results. The local rewriting and global ordering models are trained jointly in an end-to-end fashion: in each iteration, candidate subqueries are first rewritten by the local model, and then the rewritten subqueries are globally ordered. The overall query execution time is used as the global reward to guide learning.

Therefore, the contributions of this paper are as follows:

- This paper proposes a novel partition-boundary–aware query rewriting strategy that leverages inter-partition edge information and internal/extended (I/E) annotations to explicitly identify locally executable regions and factor a pattern query into boundary subqueries and local-join subqueries, significantly reducing intermediate results and communication overhead.
- This paper designs a graph neural network (GNN) model that incorporates property graph features, including node/edge labels and property information, augmented with partition-aware statistics (e.g., label frequencies, degree/selectivity, property constancy, replica densities, and inter-partition edge ratios) for accurate representation learning in distributed environments.
- This paper integrates reinforcement learning to jointly train the rewriting and ordering models, forming the R2O dual-layer learning framework that supports end-to-end optimization of distributed query plans. The MDP action space includes both rewriting and global ordering, and the reward directly targets intermediate sizes and inter-partition communication.
- Extensive experiments on large-scale property graphs demonstrate that R2O significantly reduces query execution time and communication overhead, achieving strong scalability and efficiency across different partitioning strategies. Our code has been released on GitHub[2].

2 Background

In this section, we introduce the preliminary concepts, notations, and definitions used throughout the paper.

2.1 Preliminaries

In this paper, we adopt a widely accepted definition of property graphs, which is commonly used in property graph systems such as JanusGraph [14]. According to this definition, each node or edge can have an arbitrary number of property/value pairs. In addition to properties, each node or edge is assigned a label, and the set of all label types in a property graph is represented by $\mathcal{L}$.

Definition 1. ***(Property Graph)*** *A property graph is a directed labeled multigraph denoted as a tuple $G = \langle N_G, E_G, \tau, \pi, \lambda \rangle$ where:*

- *N_G is a finite set, referred to the nodes of G;*
- *E_G is a finite set, referred to the edges of G;*
- *$\tau: E_G \to N_G \times N_G$ is a function that maps each edge $e \in E_G$ to an ordered pair of nodes (n_s, n_t), where n_s is the source node and n_t is the target node of r;*
- *$\pi: N_G \cup E_G \times \mathcal{P} \to \mathcal{V}$ is a function that maps each property of nodes and edges to a value. $\mathcal{P}$ and $\mathcal{V}$ are countable sets where $p \in \mathcal{P}$ represents a property and $v \in \mathcal{V}$ represents the content of a property;*
- *$\lambda: N_G \cup E_G \to \mathcal{L}$ is a function that maps each node or edge to a label.* □

In a pattern query, the user defines a query graph that specifies the required labels for the nodes and edges, along with their structure and property/value pairs.

[2]https://github.com/sh1min22/R2O.

Definition 2. ***(Graph Pattern Query)*** *A graph pattern query over a property graph $G = \langle N_G, E_G, \tau, \pi, \lambda \rangle$ is a labeled multigraph denoted as $Q = \langle N_Q, E_Q, \tau_Q, \pi_Q, \lambda_Q \rangle$. Which aims to identify subgraphs in the property graph G that match the query graph Q, where:*

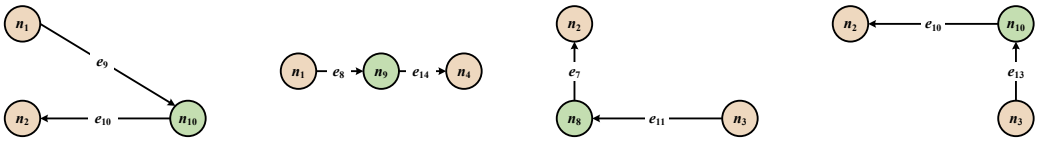
- *N_Q is a finite set of nodes, as defined for property graph G;*
- *E_Q is a finite set of edges, as defined for property graph G;*
- *τ_Q: $E_Q \rightarrow N_Q \times N_Q$ is a function that maps each edge $e \in E_Q$ to an ordered pair of nodes (n_s, n_t), where n_s is the source node and n_t is the target node of e in the query.*
- *π_Q: $N_Q \cup E_Q \times \mathcal{P} \rightarrow \mathcal{V} \cup \mathcal{V}_{var}$ is a mapping from properties to their corresponding values. For each $v \in \mathcal{V}_{var}$, a set of constraints (e.g., $<, \leq, \neq, \geq, >$) can be applied. Multiple constraints on the same property can be combined using logical operators such as "AND" and "OR" to define complex filtering conditions (e.g., "$x > 10$ AND $x < 20$" for regular expression matching).*
- *λ_Q: $N_Q \cup E_Q \rightarrow \mathcal{L}$ is a label mapping where each node or edge in the query graph is assigned a label from the label set $\mathcal{L}$.* □

A graph pattern query identifies all subgraphs in the data graph that match structure, labels, and property constraints. Each result maps the query's nodes and edges to those in the graph, with variable property values assigned according to the query's constraints.

Definition 3. ***(Pattern Match)*** *Given a graph pattern query $Q = \langle N_Q, E_Q, \tau_Q, \pi_Q, \lambda_Q \rangle$ over $G = \langle N_G, E_G, \tau, \pi, \lambda \rangle$, a pattern match is a mapping μ: $N_Q \cup E_Q \rightarrow N_G \cup E_G$, where for $\forall a \in N_Q \cup E_Q$:*

- *If $a \in E_Q$ with $\tau_Q(a) = (n_s, n_t)$, then $\tau(\mu(a)) = (\mu(n_s), \mu(n_t))$;*
- *If $\lambda_Q(a) \in \mathcal{L}$, then $\lambda(\mu(a)) = \lambda_Q(a)$;*
- *If $\mathcal{P}_a$ denotes the set of properties associated with a, for $\forall p \in \mathcal{P}_a$:*
  - *If $\pi_Q(p) \in \mathcal{V}$, $\pi(p) = \pi_Q(p)$;*
  - *If $\pi_Q(p) \in \mathcal{V}_{var}$, then $\pi(p) = v$, where v is a value of p that satisfies all constraint condition defined in $\pi_Q(p)$ (if there is no constraint, it represents an arbitrary value);*

The set of all such pattern matches μ for pattern query Q on property graph G is collectively denoted as $[[Q]]_G$, where each μ includes mappings and any variable values satisfying π_Q. We adopt subgraph homomorphism: node/edge mappings need not be injective as long as label, direction, and property constraints are satisfied. □



(a) μ_1 (b) μ_2 (c) μ_3 (d) μ_4

Fig. 4. Pattern Matches of graph pattern query Q_2 in Example 3 on property graph G in Example 1.

Example 4. ***(Pattern Match)*** *Figure 4 shows four pattern matches (μ_1, μ_2, μ_3 and μ_4) for Q_2 (Example 3) on distributed property graph G (Example 1). In Q_2, node p_1 ("Person") with gender = female matches n_2 and n_4 in G. c_1 ("Comment") with creationDate < 2019-01-01 matches n_8, n_9, and n_{10}. Node p_2 and both edges have variable properties without additional constraints, so any node or edge satisfying the label requirements can be matched. Each match satisfies the pattern's structure, labels, and property constraints.*

2.2 Distributed Query Processing Framework

In distributed environments, property graphs are partitioned and stored across multiple sites. In this paper, we focus on node-disjoint partitioning, where nodes are assigned to partitions, and each edge is stored with its endpoints. For completeness, inter-partition edges and replicated endpoints (label only) are included in all relevant partitions. Our optimization can be extended to edge-disjoint partitioning, but the discussion is based on node-disjoint partitioning.

Definition 4. ***(Graph Partitioning)*** *Given a property graph $G = \langle N_G, E_G, \tau, \pi, \lambda \rangle$ and the number n of partitions, a partitioning $\mathcal{F}$ of G is a set of subgraphs $\mathcal{F} = \{F_1, ..., F_n\}$, each subgraph $F_i = \langle N_{F_i}, E_{F_i}, \tau, \pi, \lambda \rangle$ is a vertex-induced subgraph of G, where:*

- *$N_G = \bigcup_{i=1}^{n} N_{F_i}$ and $N_{F_i} \cap N_{F_j} = \emptyset$ for $i \neq j$, where N_{F_i} is the set of nodes in F_i;*
- *$E_G = \bigcup_{i=1}^{n} E_{F_i}$, where E_{F_i} is the set of edges in F_i;*
- *For each $e = (n_s, n_t) \in E_G$:*
  - *If $n_s \in N_{F_i}$ or $n_t \in N_{F_i}$, then $e \in E_{F_i}$;*
  - *If $e \in E_{F_i}$, but $n_x \notin N_{F_i}$ ($n_x \in \{n_s, n_t\}$), a replicated node n'_x is added to N_{F_i} and $\lambda(n'_x) = \lambda(n_x)$, $\pi(n'_x) = \emptyset$.* □

For each inter-partition edge, a copy of the edge and a replica of its remote endpoint (storing only the label) are included in both partitions. This ensures query completeness in distributed settings.

Definition 5. ***(Inter-Partition Edge)*** *Given a property graph $G = \langle N_G, E_G, \tau, \pi, \lambda \rangle$ and a partitioning $\mathcal{F} = \{F_1, ..., F_n\}$ of G, an inter-partition edge is an edge $e \in E_G$, such that there exist distinct partitions $F_i, F_j \in \mathcal{F}$, with $e \in F_i$ and $e \in F_j$, where $i \neq j$.* □

Since the property graph is partitioned and distributed across different sites, complex pattern queries typically cannot be fully matched within a single partition. As a result, queries are usually decomposed into several subqueries based on the partitioning method, enabling independent execution within each partition.

Definition 6. ***(Query Decomposition)*** *Given a graph pattern query $Q = \langle N_Q, E_Q, \tau_Q, \pi_Q, \lambda_Q \rangle$ over a property graph G that is partitioned into $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ as defined in Definition 4, the query decomposition produces a set of subqueries $\boldsymbol{Q} = \{q_1, q_2, \ldots, q_m\}$, where for each subquery $q_i = \langle N_{q_i}, E_{q_i}, \tau_Q, \pi_Q, \lambda_Q \rangle$:*

- *$N_Q = \bigcup_{i=1}^{m} N_{q_i}$ and $N_{q_i} \cap N_{q_j} = \emptyset$ for $i \neq j$, where N_{q_i} is the set of nodes in q_i;*
- *$E_Q = \bigcup_{i=1}^{m} E_{q_i}$, where E_{q_i} is the set of edges in q_i;*
- *For each edge $e = (n_s, n_t) \in E_{q_i}$:*
  - *If $n_s \in N_{q_i}$ or $n_t \in N_{q_i}$, then $e \in E_{q_i}$;*
  - *If $e \in E_{q_i}$, but $n_x \notin N_{q_i}$ ($n_x \in \{n_s, n_t\}$), a replicated node n'_x is added to N_{q_i} and $\lambda(n'_x) = \lambda(n_x)$, $\pi(n'_x) = \emptyset$.*
- *Each subquery q_i can be independently executed on each partition $F_j \in \mathcal{F}$ without inter-partition join operations, i.e., $[\![q_i]\!]_G = \bigcup_{j=1}^{n} [\![q_i]\!]_{F_j}$.* □

Example 5. ***(Query Decomposition of Q_2)*** *Figure 5 illustrates the decomposition of Q_2 into three subqueries: q_1 ((p1)<-[h1]-(:Comment)), q_2 ((:Person)<-[h1]-(c1)<-[l1]-(:Person)), and q_3 ((:Comment)<-[l1]-(p2)). Each subquery includes its core nodes and edges from Q_2, along with replicas of remote endpoints as needed (e.g., q_1 and q_3 each include a replica of c_1, and q_2 includes replicas of p_1 and p_2). These subqueries can be executed independently within partitions, without inter-partition joins during local execution.*

In a typical distributed query execution process, after query decomposition, the subqueries are executed independently and in parallel within each partition. Their partition-aggregated results

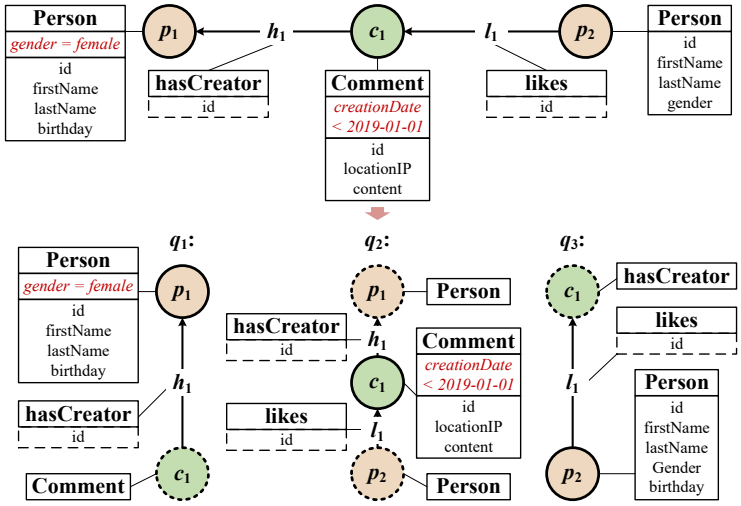


Fig. 5. An Example of Query Decomposition of Q_2.

are then joined (or semi-joined) in a specified global order to produce the final query result. The formal definition is as follows:

Definition 7. ***(Distributed Query Execution)*** *Given a property graph G that is partitioned into $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ as defined in Definition 4, and a set of subqueries $\mathcal{Q} = \{q_1, q_2, \ldots, q_m\}$ produced by query decomposition (as defined in Definition 6), the distributed query execution process follows these steps:*

- *Each subquery Q_i is executed independently and in parallel on every partition $F_j \in \mathcal{F}$, producing partial results $[\![Q_i]\!]_{F_j}$ for each partition F_j. And the final result of Q_i is obtained by combining the partial results across all partitions: $[\![Q_i]\!]_G = \bigcup_{j=1}^{n} [\![Q_i]\!]_{F_j}$.*
- *The final result of Q is obtained by sequentially joining the results of all subqueries: $[\![Q]\!]_G = [\![Q_1]\!]_G \bowtie [\![Q_2]\!]_G \bowtie \ldots \bowtie [\![Q_m]\!]_G$, where $\bowtie$ represents the join operation.* □

3 Partition-Aware Query Rewriting for Inter-Partition Joins

This section presents a partition-aware query rewriting strategy for inter-partition joins, designed to minimize intermediate results and communication cost by restructuring subqueries after query decomposition. The method leverages graph partitioning and subquery relationships to optimize distributed execution.

3.1 Node Labeling

To capture partitioning information, we assign each node a label during graph partitioning, distinguishing between **internal** and **extended** nodes:

- **Internal Nodes** refer to nodes that are entirely contained within a single partition. They possess complete property information, and all their connections, including neighboring edges and nodes, are within the same partition. The property and structural data of internal nodes do not rely on any other partition, meaning they can be processed independently within their partition without the need for inter-partition communication.
- **Extended Nodes** correspond to internal nodes from other partitions, replicated in the current partition via inter-partition edges. Unlike internal nodes, extended nodes do not have complete

property information; they only store the label and minimal information necessary for inter-partition connectivity. Full properies remain in the partition of the original internal node.

Based on the characteristics of these two types of nodes, if the matched results of the neighboring nodes in two neighboring subqueries are both internal nodes, this implies that the matching results for this portion of these queries are contained within the same partition. We materialize this status as a data-side property $label(u) \in \{I, E\}$ attached to each node for the partitions where it is visible, which records whether the node is internal or an extended replica and serves as partition metadata.

3.2 Partition-Aware Query Rewriting Strategy

After partitioning and node labeling, we further optimize query execution by rewriting subqueries to reduce inter-partition communication. This strategy distinguishes query components that require inter-partition joins from those executable within a single partition, and applies to any pair of neighboring subqueries (i.e., two subqueries that share at least one identical edge). In the rewriting, the expressions $label = I$ and $label = E$ are used as query-side predicates that test the data-side tags to determine locality.

The rewriting process produces two types of subqueries:

- **Boundary Subqueries** add node-label constraints to distinguish nodes involved in inter-partition joins. For any pair of neighboring subqueries sharing an edge, the node with full properties is marked as internal, and its counterpart in the other subquery is marked as extended. This clarifies data dependencies across partitions while preserving query correctness.
- **Local-Join Subqueries** are formed by merging two neighboring subqueries when all shared nodes are internal. The merged subquery can be executed entirely within a single partition, reducing intermediate results and redundant inter-partition joins, and improving execution efficiency.

Algorithm 1: Query Rewriting Algorithm

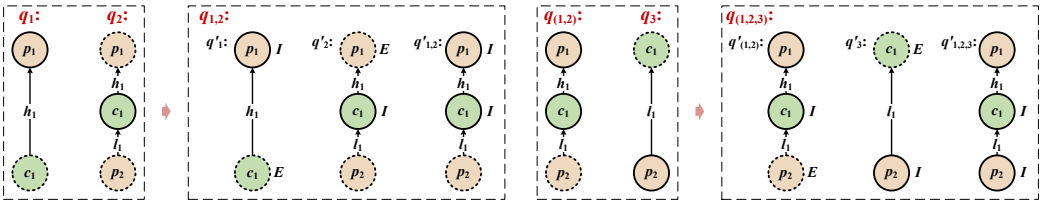
Input: Subquery q_1 and its neighbor q_2 ($E_{q_1} \cap E_{q_2} \neq \emptyset$)
Output: Eexecution of q_1, q_2 after rewriting (Boundary subqueries q'_1, q'_2 and local-join subquery $q'_{1,2}$)

1 **foreach** $e(n_i, n_j) \in E_{q_1} \cap E_{q_2}$ **do**
2 **if** $\exists n_k \in \{n_i, n_j\}$ *such that* n_k *has associated properties* **then**
3 Add condition $label = I$ to n_k;
4 $q'_{1,2} \leftarrow q_1 \bowtie q_2$;
5 **foreach** $e(n_i, n_j) \in E_{q_1} \cap E_{q_2}$ **do**
6 **if** $\exists n_k \in \{n_i, n_j\}$ *such that* $label(n_k) = I$ **then**
7 Let $n_l \in \{n_i, n_j\} \setminus \{n_k\}$;
8 Add condition $label = E$ to n_l;
9 $q'_1 \leftarrow q_1$;
10 $q'_2 \leftarrow q_2$;
11 **return** $([[q'_1]] \bowtie [[q'_2]]) \cup [[q'_{1,2}]]$

To formalize the rewriting process, we propose Algorithm 1, which takes two neighboring subqueries as input and transforms them into an optimized execution plan. The algorithm first identifies shared edges between the two subqueries and assigns internal labels (I) to nodes with associated properties (Lines 1–3). It then constructs the local-join subquery by merging the two

subqueries (Line 4). Next, for each shared edge, the algorithm assigns extended labels (E) to the opposite node of an internal-labeled node to ensure boundary consistency (Lines 5–8). Finally, it generates the boundary subqueries and returns the updated execution order, which reduces intermediate results and minimizes inter-partition communication (Lines 9–11). The execution plan $([[q'_1]] \bowtie [[q'_2]]) \cup [[q'_{1,2}]]$ produces the same final result as the original execution plan $[[q_1]] \bowtie [[q_2]]$, as both approaches preserve all valid matches from the original query. However, the rewritten plan restructures the computation to reduce intermediate results and optimize execution efficiency.

The overall computational complexity of Algorithm 1 is $O(|N_{q_1}| + |N_{q_2}| + |E_{q_1}| + |E_{q_2}|)$. Since queries are typically small, the computational overhead of the rewriting process remains small.



(a) Rewriting of q_1 and q_2 in Example 5 (b) Rewriting of $q_{(1,2)}$ and q_3 in Example 5

Fig. 6. Rewriting Process.

Example 6. ***(Query Rewriting for q_1 and q_2)*** *Figure 6(a) illustrates the rewriting process of subqueries q_1 and q_2 from Example 5. Through the rewriting strategy, the original execution plan $[[q_1]] \bowtie [[q_2]]$ is transformed into a new execution plan consisting of three subqueries: the boundary subqueries q'_1 and q'_2, and the local-join subquery $q'_{1,2}$. In the rewritten query, the boundary subquery q'_1 produces a single match in G: (n_{10})-$[e_{10}]$->(n_2). Similarly, q'_2 yields two matches in G: (n_1)-$[e_9]$->(n_{10})-$[e_{10}]$->(n_2) and (n_3)-$[e_{13}]$->(n_{10})-$[e_{10}]$->(n_2). Meanwhile, the local-join subquery $q'_{1,2}$ produces two matches in G: (n_3)-$[e_{11}]$->(n_8)-$[e_7]$->(n_2) and (n_1)-$[e_8]$->(n_9)-$[e_{14}]$->(n_4). By executing the rewritten query according to the new execution plan $([[q'_1]] \bowtie [[q'_2]]) \cup [[q'_{1,2}]]$, the final matches obtained are: (n_3)-$[e_{11}]$->(n_8)-$[e_7]$->(n_2), (n_1)-$[e_9]$->(n_{10})-$[e_{10}]$->(n_2), (n_1)-$[e_8]$->(n_9)-$[e_{14}]$->(n_4) and (n_3)-$[e_{13}]$->(n_{10})-$[e_{10}]$->(n_2). Compared to the original query plan, the rewritten execution significantly reduces the number of intermediate results, requiring only two inter-partition joins.*

Example 7. ***(Query Rewriting for $q_{(1,2)}$ and q_3)*** *Figure 6(b) illustrates the rewriting process where the merged subquery $q_{(1,2)}$ from the previous step is further rewritten with q_3. Following the same rewriting strategy, the original execution plan $[[q_{(1,2)}]] \bowtie [[q_3]]$ is transformed into the optimized execution plan $[[q'_{(1,2)}]] \bowtie [[q'3]] \cup [[q'1, 2, 3]]$. In the rewritten query, the boundary subquery $q'_{(1,2)}$ produces three matches in G: (n_1)-$[e_9]$->(n_{10})-$[e_{10}]$->(n_2), (n_1)-$[e_8]$->(n_9)-$[e_{14}]$->(n_4) and (n_3)-$[e_{11}]$->(n_8)-$[e_7]$->(n_2). Similarly, q'_3 produces three matches in G: (n_1)-$[e_8]$->(n_9), (n_1)-$[e_9]$->(n_{10}) and (n_3)-$[e_{11}]$->(n_8). Meanwhile, the local-join subquery $q'_{1,2,3}$ generates a single match in G: (n_3)-$[e_{13}]$->(n_{10})-$[e_{10}]$->(n_2). Thus, the final matching results obtained from the rewritten execution plan are: (n_3)-$[e_{11}]$->(n_8)-$[e_7]$->(n_2), (n_1)-$[e_9]$->(n_{10})-$[e_{10}]$->(n_2), (n_1)-$[e_8]$->(n_9)-$[e_{14}]$->(n_4) and (n_3)-$[e_{13}]$->(n_{10})-$[e_{10}]$->(n_2). Compared to the previous query execution plan, the rewriting process similarly reduces the number of intermediate results.*

Notably, the rewritten execution plan from Example 6 serves as the foundation for the subsequent rewriting process in Example 7. These two examples illustrate that query rewriting effectively reduces the number of intermediate result joins, with Example 6 achieving a more significant

reduction compared to Example 7. However, since rewriting introduces additional query steps, it also incurs extra communication overhead. Consequently, the benefits of rewriting are not always strictly positive. This underscores the necessity of a strategic approach to determine when rewriting should be applied, which will be explored in the following section.

4 R2O Model

This section introduces the proposed R2O (Rewriting to Ordering) framework, a dual-layer learning model designed for query plan generation in distributed property graph systems. R2O operates between query decomposition and execution, and jointly optimizes subquery rewriting and ordering to generate efficient query execution plans for distributed property graph pattern queries.

4.1 Framework

As illustrated in Figure 7, after query decomposition, candidate subqueries are enriched with data graph information to construct feature representations, which are then processed by a graph neural network to obtain informative embeddings.

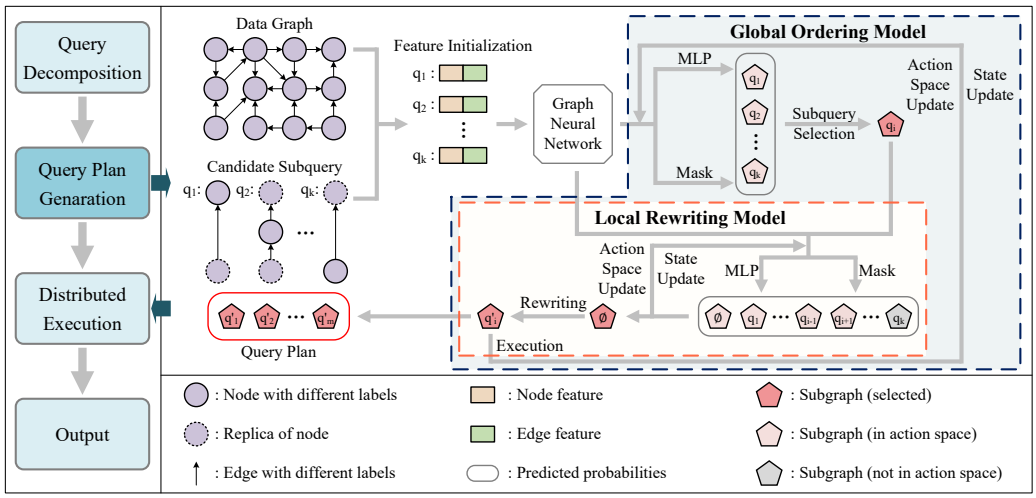


Fig. 7. Framework of the R2O Model.

The R2O model consists of two coordinated modules, each modeled as a separate Markov Decision Process (MDP). The Global Ordering Module determines the execution order of subqueries by sampling from a learned probability distribution over candidate subqueries, based on their feature embeddings. Once a subquery is selected, the Local Rewriting Module evaluates whether and how to rewrite it with its neighboring subqueries by predicting a probability distribution over possible rewriting actions and a termination option. If a neighboring subquery is selected, it is rewrited with the current one to minimize inter-partition joins. Otherwise, the rewriting process ends, and the (possibly rewritten) subquery is incorporated into the final query plan.

This process continues, with the Global Ordering Module dynamically updating its candidate subqueries and action space at each step, ensuring that rewriting decisions are made strategically. The cycle repeats until all subqueries have been processed, resulting in an optimized execution plan.

These two modules work in tandem within the R2O framework, generating optimized query plans that drive efficient distributed execution of property graph pattern queries. Further details are provided in the subsequent subsections.

4.2 Feature Representations

In the R2O framework, each subquery is characterized by a set of node and edge features that encode essential properties of the query graph and its correspondence to the data graph. These features serve as critical inputs for decision-making in the dual-layer query plan generation process, enabling the learning framework to assess the selectivity and execution priority of subqueries.

4.2.1 Node Features. Let n_i denote a node in the query graph. Node features capture structural attributes, selectivity, and property constraints, all of which influence the execution order of subqueries.

Label. Nodes are classified by label type, which determines their role in the query graph. Rare labels in the data graph increase selectivity, making subqueries containing such nodes more likely to be prioritized. This feature is encoded and normalized as follows:

$$\mathbb{L}_{n_i} = \frac{encode(\lambda_Q(n_i))}{|\mathcal{L}_N|} \tag{1}$$

where $encode(\lambda_Q(n_i))$ represents the encoded label of node n_i, and $|\mathcal{L}_N|$ is the total number of unique node labels in the data graph.

Degree. The degree of a node reflects its connectivity. A higher-degree node generally has fewer matches, increasing its selectivity:

$$\mathbb{D}_{n_i} = \frac{degree_{in}(n_i) + degree_{out}(n_i)}{\alpha_{degree}} \tag{2}$$

where $degree_{in}(n_i)$ and $degree_{out}(n_i)$ denote the in-degree and out-degree of n_i, respectively, and α_{degree} is a scaling factor.

Global Node Count. This feature quantifies how frequently nodes with the same label as n_i appear in the data graph. Fewer occurrences indicate higher selectivity:

$$\mathbb{C}_{n_i} = \frac{|\{n \in N_G : \lambda(n) = \lambda_Q(n_i)\}|}{|N_G|} \tag{3}$$

where N_G represents the set of all nodes in the data graph.

Property Vector. Each node in the query graph possesses a set of properties that influence its selectivity. To ensure comparability, numerical properties are normalized based on their global range, while categorical properties are one-hot encoded:

$$\mathbb{P}_{n_i} = \left[\frac{encode(\pi(p_1))}{\alpha_1}, \ldots, \frac{encode(\pi(p_k))}{\alpha_k} \right] \tag{4}$$

where $encode(\pi(p_j))$ represents the encoding of the j-th property, and α_j is a scaling factor.

Constant Value. This feature counts the number of properties of a node that have fixed values, indicating higher selectivity:

$$\mathbb{V}_{n_i} = \sum_{p \in \mathcal{P}_{n_i}} \mathbf{1}_{\{\pi(p) \in \mathcal{V}\}} \tag{5}$$

where $\mathcal{P}_{n_i}$ denotes the property set of n_i.

4.2.2 Edge Features. Let e_i denote an edge in the query graph. Edge features capture structural significance and constraints of edges, contributing to the evaluation of a subquery's execution priority.

Label. Edges represent relationships between nodes, with rare labels increasing selectivity. This feature is encoded and normalized similarly to node labels:

$$\mathbb{L}_{e_i} = \frac{encode(\lambda_Q(e_i))}{|\mathcal{L}_E|} \tag{6}$$

where $|\mathcal{L}_E|$ is the number of unique edge labels in the data graph.

Edge Degree. The degree of an edge is derived from the degrees of its endpoint nodes. Higher degrees indicate lower selectivity:

$$\mathbb{D}_{e_i} = \frac{\mathbb{D}_{n_s} + \mathbb{D}_{n_t}}{2} \tag{7}$$

where n_s and n_t are the source and target nodes of e_i.

Global Edge Count. This feature measures the frequency of edges with the same label as e_i in the data graph. Lower frequency implies higher selectivity:

$$\mathbb{C}_{e_i} = \frac{|\{e \in E_G : \lambda(e) = \lambda_Q(e_i)\}|}{|E_G|} \tag{8}$$

where E_G represents all edges in the data graph.

Property Vector. Edges can also contain properties that affect query selectivity. These properties are encoded using the same strategy as node properties:

$$\mathbb{P}_{e_i} = \left[\frac{encode(\pi(p_1))}{\alpha'_1}, \ldots, \frac{encode(\pi(p_k))}{\alpha'_k} \right] \tag{9}$$

where α'_j is a normalization factor.

Constant Value. Edges with fixed property values are more constrained and have higher selectivity:

$$\mathbb{V}_{e_i} = \sum_{p \in \mathcal{P}_{e_i}} \mathbf{1}_{\{\pi(p) \in \mathcal{V}\}} \tag{10}$$

where $\mathcal{P}_{e_i}$ represents the property set of edge e_i.

By integrating these features, R2O effectively captures structural and semantic variations between query nodes and edges while incorporating data graph distribution information. This enriched representation facilitates reinforcement learning and graph neural networks in automatically deriving optimized subquery ordering, leading to more efficient distributed query execution.

4.3 Global Ordering Model

In R2O, the Global Ordering Model is a core component of this dual-layer learning framework. It determines the execution order of subqueries to optimize query efficiency by leveraging carefully designed features and a graph neural network. This module formulates subquery ordering as a Markov Decision Process (MDP): at each time step t, the agent selects one subquery from the action space $\mathcal{A}_t^{(o)}$, based on the current state $\mathcal{S}_t^{(o)}$, and adds it to the query plan. Once all subqueries are executed, a global reward based on the total execution time is used to optimize the overall strategy.

4.3.1 State. The state $\mathcal{S}_t^{(o)}$ at time step t consists of the current set of candidate subqueries (those neighboring to the previously selected subqueries) along with their corresponding feature representations.

$$\mathcal{S}_t^{(o)} = (\mathcal{Q}_t^{(o)}, \mathcal{H}_t^{(o)}) \tag{11}$$

where:

- $\mathcal{Q}_t^{(o)} = \{q_i \mid q_i \text{ is neighboring to selected subqueries at time } t\}$ represents the set of candidate subqueries available for selection at time t;
- $\mathcal{H}_t^{(o)} = \{\mathbf{H}_{q_i} \mid q_i \in \mathcal{Q}_t^{(o)}\}$ contains the feature representations of the candidate subqueries, encoding their selectivity, connectivity, and structural properties.

Each subquery's pattern graph is encoded by a GNN, with node and edge features initialized from Section 4.2.

4.3.2 Action. The action space $\mathcal{A}_t^{(o)}$ at time step t consists of selecting one subquery from the candidate set $\mathcal{Q}_t^{(o)}$ that has not yet been included in the query plan. Formally,

$$\mathcal{A}_t^{(o)} = \{a_i \mid q_i \in \mathcal{Q}_t^{(o)}, q_i \text{ not selected at time } t\} \tag{12}$$

where a_i corresponds to the action of adding subquery q_i to the execution plan.

At each step, the policy network outputs a probability distribution over the available actions in $\mathcal{A}_t^{(o)}$. Actions are sampled according to this distribution, balancing exploration and exploitation. This sampling mechanism prevents the policy from prematurely converging to suboptimal choices and enhances the global optimality of the final query plan.

In particular, for the selection of the initial subquery in the plan, the system gives priority to subqueries containing constant predicates. Such subqueries can more effectively constrain the search space in the early stage of execution, thus providing a more discriminative and informative starting point for the overall plan.

4.3.3 Reward Design. In the Global Ordering Model, the core objective of reward design is to minimize the total query execution time. To this end, we define the immediate reward at each step based on the total execution time $T(Q)$ of the generated query plan:

$$\mathcal{R}_t^{(o)} = -T(Q) \tag{13}$$

Due to variations in query complexity and data scale across different query graphs, the raw execution time may differ by several orders of magnitude. To mitigate the risk of gradient vanishing or unstable convergence during training, we apply z-score standardization to the rewards across different queries. This normalization ensures stability and comparability in the learning process under diverse query workloads.

Since the actual execution time $T(Q)$ can only be measured after the entire query plan is generated, the final reward is shared across all decision steps. Specifically, the same final reward is assigned to every time step t, i.e., for $\forall t$, $\mathcal{R}_t^{(o)} = \mathcal{R}^{(o)}$, during the query plan generation process. This reward-sharing design enables the model to account for the long-term impact of each decision on the overall query execution outcome, thereby encouraging global optimization rather than short-sighted, greedy choices at individual steps.

Since the model does not receive immediate feedback during query plan generation, the final reward is distributed across all decision steps using a discount factor γ. The cumulative return for the entire sequence is thus defined as:

$$\mathcal{R}_Q = \sum_{t=1}^{|Q|} \gamma^t \mathcal{R}_t^{(o)} \tag{14}$$

where $|Q|$ is the number of subqueries, $\mathcal{R}_t^{(o)}$ is the (shared) reward at step t, and $\gamma \in (0, 1]$ is a discount factor that progressively reduces the influence of later steps. This design encourages the model to focus more on earlier decisions, which have a greater impact on the final query plan.

4.4 Local Rewriting Model

While the Global Ordering Model determines the subquery execution order, relying on it is insufficient to compress intermediate results or address local structural redundancy. Therefore, after a subquery is selected by the Global Ordering Model, the Local Rewriting Model examines its neighborhood and selects subqueries for structural rewriting, further reducing intermediate results before execution. This rewriting process is modeled as a Markov Decision Process (MDP), where the model focuses on the current subquery and selects actions among its neighboring subqueries. At each step, the model either rewrites the current subgraph with a neighboring subquery or terminates the expansion and returns control to the global ordering model to select the next starting subquery.

4.4.1 State. The state $\mathcal{S}_t^{(r)}$ at time step t during the local rewriting process consists of the currently constructed subquery (i.e., the merged subgraph), the set of candidate neighboring subqueries, and the feature representations of these candidates. Formally:

$$\mathcal{S}_t^{(r)} = (C_t, \mathcal{Q}_t^{(r)}, \mathcal{H}_t^{(r)}) \tag{15}$$

where:

- C_t denotes the set of subqueries that have been merged into the current subgraph up to step t, starting from the initial subquery selected by the global ordering model;
- $\mathcal{Q}_t^{(r)}$ denotes the set of neighboring subqueries that are candidates for merging at step t;
- $\mathcal{H}_t^{(r)} = \{\mathbf{H}_{q_i} \mid q_i \in \mathcal{Q}_t^{(r)}\}$ denotes the feature representations of the candidate subqueries at step t, where each candidate subquery's feature representation $\mathbf{H}_{q_i}$ is derived from GNN-based message passing over the query structure and shared with the Global Ordering Model.

4.4.2 Action. The action space $\mathcal{A}_t^{(r)}$ at time step t is the set of candidate neighboring subqueries that can be merged with the current rewritten subgraph. In addition, a special action a_{stop} is included to indicate the termination of the rewriting process. Formally,

$$\mathcal{A}_t^{(r)} = \{a_i \mid q_i \in \mathcal{N}_{C_t}\} \cup \{a_{\text{stop}}\} \tag{16}$$

where $\mathcal{N}_{C_t}$ denotes the set of subqueries neighboring to the current subgraph C_t. At each step, the model either selects a neighboring subquery for merging or chooses a_{stop} to end the local rewriting phase and return control to the global ordering model.

The model samples actions from the probability distribution predicted by the policy network, thereby dynamically deciding whether to continue expanding the rewritten subgraph or to terminate rewriting at each step.

4.4.3 Reward Design. The reward function in the Local Rewriting Model is designed to measure the effectiveness of each rewriting operation in reducing query execution cost. At each step, let $T(Q_t)$ and $T(Q_t')$ denote the execution times of the subquery before and after rewriting, respectively. The immediate reward is defined as the relative reduction in execution time:

$$\mathcal{R}_t^{(r)} = \frac{T(Q_t) - T(Q_t')}{T(Q_t)} \tag{17}$$

A positive reward is given when rewriting leads to improved performance, while a negative reward penalizes less effective actions.

To prevent the model from repeatedly selecting rewriting steps that do not lead to meaningful performance improvements, a penalty mechanism is introduced. If the relative improvement falls

below a threshold ϵ (set to 0.01 in this work), that is,

$$\frac{T(Q_t) - T(Q_t')}{T(Q_t)} < \epsilon, \tag{18}$$

an additional fixed negative reward is assigned to discourage redundant or ineffective rewrites.

Overall, this reward design encourages the model to focus on substantial performance gains, avoids local loops of trivial optimizations, and ensures stable convergence to high-quality rewriting strategies even in complex graph structures.

4.5 Joint Training and Optimization

This section presents the architecture of the policy and value networks for both the global ordering and local rewriting models in R2O, and introduces the joint training strategy based on a two-layer actor-critic framework.

4.5.1 Policy and Value Networks. Both the policy and value networks in R2O adopt a multi-layer perceptron (MLP) architecture. The policy network integrates a masking mechanism to ensure that only valid actions are considered at each step. Given the current state $\mathcal{S}_t$, the policy outputs a probability distribution over all available actions:

$$P_{a_i}^t = \pi(a_i \mid \mathcal{S}_t) = \text{Softmax}(\text{Mask}(q_i) \cdot (\mathbf{W}_2 \cdot \sigma(\mathbf{W}_1 \mathbf{H}_{q_i}))) \tag{19}$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are learnable parameters of the Actor network, and $\mathbf{H}_{q_i}$ is the feature embedding of subquery q_i. The output dimension is one scalar per candidate, representing the logit score for each action. The binary mask function $\text{Mask}(q_i)$ is defined as:

$$\text{Mask}(q_i) = \begin{cases} 1, & \text{if } q_i \text{ has not been executed} \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

which guarantees that executed subqueries are excluded from selection.

The value network (critic) receives the state embedding as input and outputs a scalar estimate $V(\mathcal{S}_t)$:

$$V(\mathcal{S}_t) = \mathbf{W}_4 \cdot \text{ReLU}(\mathbf{W}_3 \mathcal{S}_t) \tag{21}$$

where $\mathbf{W}_3$ and $\mathbf{W}_4$ are learnable parameters of the Critic network.

The value prediction is used to compute the advantage function, guiding the policy update in the actor-critic optimization.

4.5.2 Policy Training. The policy and value networks are jointly trained in R2O through an actor-critic paradigm that combines both offline pretraining and online interactive learning. In each training episode, actions are sampled from the current policy, and state-action-reward trajectories are collected to compute returns.

Offline pretraining leverages historical query execution data to initialize network parameters, exposing the model to a wide range of query patterns. During online training, the model interacts with the database engine in real time: at each step, the policy network selects a subquery or rewriting action, receives the observed execution time, and records the corresponding immediate or cumulative reward. For the global ordering model, the normalized execution time of the full query plan is used as the reward for all steps; for the local rewriting model, each step's reward reflects the relative improvement in execution cost.

After the query plan is executed, advantage estimates are computed for each step. The policy (actor) networks are updated by minimizing the policy loss, encouraging actions with higher-than-expected returns. The value (critic) networks are optimized by minimizing the difference between predicted and empirical returns.

The total loss function integrates the policy loss, value loss, and an entropy regularization term to balance exploration and exploitation:

$$L_{\text{total}} = \beta_1 L_{\text{critic}} + L_{\text{actor}} - \beta_2 L_{\text{entropy}} \tag{22}$$

where L_{critic} and L_{actor} denote the value and policy losses, L_{entropy} is the policy entropy, and β_1, β_2 are balancing coefficients.

All parameters are updated end-to-end via stochastic gradient-based optimization. While the global ordering and local rewriting models share feature representations, they maintain separate policy and value networks. This joint training scheme enables the R2O framework to adaptively optimize query plans under dynamic query workloads and data distributions, achieving both efficiency and robustness.

5 Experiment and Evaluation

5.1 Experiment Setup

In this section, we evaluate the performance of our dual-layer model on different datasets and partitioning algorithms. All experiments are conducted on a cluster consisting of seven physical servers. The main site server is equipped with a 16-core CPU (2.1 GHz), 320 GB of memory, and runs Ubuntu 22.04.5 LTS without a GPU. Each sub-site server has a 4-core CPU (3.1 GHz), 32 GB of memory, and runs Ubuntu 20.04.2 LTS. Neo4j 3.5.28 is deployed on each sub-site as the distributed storage engine. In this software and hardware environment, the proposed dual-layer model and its related algorithms are fully implemented and deployed on the main site. The six sub-sites collaboratively perform distributed query execution and data management tasks. All servers communicate using MPICH-3.2.1. In the experiments, all algorithms are implemented and evaluated under the same environment. Our code is available on GitHub[1] and is implemented in Python 3.8.

5.1.1 Datasets. We evaluated our R2O model on four property graph datasets of varying sizes. The specific parameters of each dataset are listed in Table 1.

Table 1. Statistics of Property Graph Datasets.

Datasets	LDBC-SNB			DBpedia
	SF-3	SF-10	SF-30	v2014
# Nodes	9,281,922	29,987,835	88,789,833	139,483,229
# Edges	52,695,735	176,623,445	540,915,404	790,050,505
# Properties	215,093,554	417,792,113	1,265,960,004	2,176,994,801

LDBC-SNB (Linked Data Benchmark Council - Social Network Benchmark) [8] simulates the data structure and interactions of social network platforms. We generate datasets with three different scale factors, SF-3, SF-10, and SF-30, corresponding to 3GB, 10GB, and 30GB of data, respectively.

DBpedia [22] is a real-world knowledge graph dataset extracted from structured information in Wikipedia. As it is only available in RDF format, we converted the 2014 version of DBpedia into a property graph [4, 39] for our experiments.

Based on these datasets, we generate multiple sets of graph pattern queries according to the frequency of label occurrences. Each query graph contains 4 to 8 nodes and the edges between them, and node or edge properties are randomly selected as additional query conditions.

5.1.2 Partitioning Methods. To comprehensively evaluate the applicability and robustness of R2O under different partitioning strategies, we adopt two representative graph partitioning methods. Each dataset is divided into six partitions, and queries are decomposed based on the partitioning results for performance testing.

METIS [15] is a classic multilevel partitioning algorithm that reduces inter-partition communication by minimizing the number of edge cuts and is widely used in distributed graph processing systems. METIS tends to preserve query locality, resulting in smaller independently executable subqueries.

MPC (Minimum Property-Cut) [32] aims to minimize the number of property labels on cross-partition edges, which allows more queries to be decomposed into larger independently executable subqueries.

For each dataset and partitioning method, we train a dedicated R2O model. We use 15 randomly generated queries for training and 30 disjoint randomly generated queries for testing, and the trained policy is frozen during testing.

Table 2. Training Efficiency and Convergence Across Datasets and Partitions (20 Epochs per Model).

Dataset	Partitioner	Tot(h)[1]	Ep(m)[2]	ConvEp[3]	L5R[4]
LDBC-SNB (SF-3)	METIS	1.7	5.1	5	0.964
	MPC	1.6	4.8	4	0.969
LDBC-SNB (SF-10)	METIS	4.1	12.3	7	0.953
	MPC	4.0	12.0	6	0.958
LDBC-SNB (SF-30)	METIS	12.0	36.0	10	0.938
	MPC	12.8	38.4	12	0.942
DBpedia (v2014)	METIS	5.1	15.3	6	0.982
	MPC	4.7	14.1	5	0.985

[1] Tot means total hours; [2] Ep means minutes per epoch; [3] ConvEp means epoch at validation convergence; [4] L5R means last 5 validation reward.

5.1.3 Training Protocol and Convergence. We train one R2O model per dataset and partitioner (METIS/MPC) for 20 epochs. As shown in Table 2, training time grows with data scale: SF-30 is longest (36.0–38.4 min/epoch, 12.0–12.8 h total), while SF-3 and DBpedia finish within 1.6–5.1 h. For the same dataset, METIS and MPC are similar; on SF-30, MPC is slightly longer (12.8 h vs. 12.0 h). We define convergence as the first epoch where the validation moving average over the last five trajectories is $\leq$0.95 with 95% CI width $\leq$0.02, Critic RMSE $\leq$0.05, and policy entropy in [0.15, 0.30]. Under this criterion, models converge in 5–12 epochs. The last-5-trajectory validation rewards stay high (0.938–0.985), indicating that a few extra epochs help stabilize Actor–Critic estimates and cover rarer, high-impact query patterns.

5.2 Effectiveness of Partition-Aware Query Rewriting Strategy

To evaluate the effectiveness of the proposed partition-aware query rewriting strategy, we conducted experiments on the LDBC-SNB datasets with two partitioning methods. We randomly selected 30

pairs of adjacent subqueries from the test set and executed them before and after rewriting. Two key metrics were used for evaluation: query execution time and the scale of intermediate results.

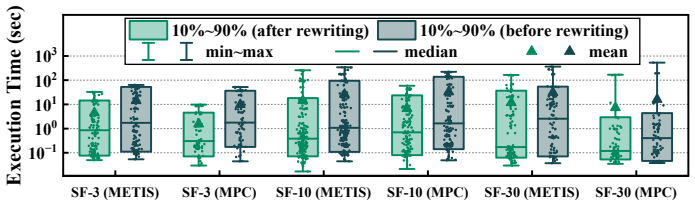


Fig. 8. Performance of Query Rewriting under Two Partitioning Methods on LDBC-SNB.

5.2.1 Query Execution Time. To reduce the impact of system load and network fluctuations, each set of queries was executed three times. As shown in Figure 8, the rewriting approach consistently improved query time by nearly an order of magnitude across different data scales and partitioning methods, with a maximum speedup of up to 663 times (CYPHER 16 under SF-30 (METIS): from 53.17s to 0.08s). However, some rewritten queries showed only slight improvements or even negative effects (CYPHER 30 under SF-3 (METIS): from 0.20s to 3.36s). This highlights the importance of the local rewriting model in R2O for decision making.

Table 3. Query Rewriting Performance on LDBC-SNB under Different Intermediate Result Reductions.

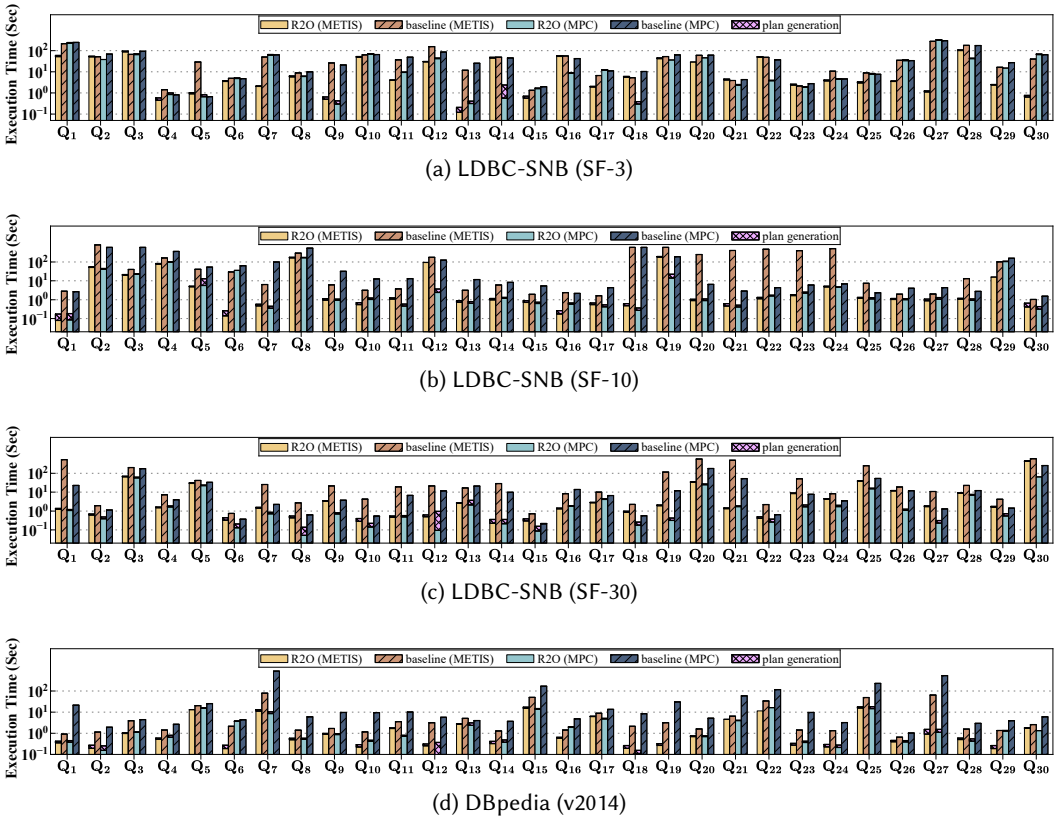
Intermediate Result Reduction (%)	[0, 30)	[30, 70)	[70, 100]
Sample Proportion (%)	36.85	14.20	48.95
Max Speedup (%)	271.03	4544.38	66316.71
Min Speedup (%)	7.80	46.48	75.68
Average Speedup (%)	91.09	227.61	1662.01

5.2.2 Scale of Intermediate Result. The scale of intermediate results refers to the number of partial matches generated during query execution. For each query, we define the intermediate-result size as the total number of partition-local partial matches actually consumed by subsequent operators. Under the proposed rewriting strategy, the scale of intermediate results usually decreases, or at most remains unchanged. Table 3 shows the query rewriting performance across different reduction intervals. The results indicate a strong positive correlation between the reduction in intermediate results and query speedup. Greater reduction in intermediate results leads to better acceleration. When the reduction is low ([0, 30)%), the average speedup remains below 1, suggesting that rewriting can sometimes introduce extra overhead. There is also a large gap between the maximum and minimum speedup within each interval, which reflects the impact of query characteristics and local rewriting decisions. Notably, nearly half of the queries (48.95%) achieved more than 70% reduction in intermediate results, highlighting the necessity and effectiveness of the rewriting strategy.

5.3 Performance of R2O

5.3.1 Baseline Algorithm. In distributed property graph systems, unoptimized query plans often result in extremely poor performance, frequent timeouts, or even system failures. To enable fair and meaningful comparison, we adopt a global degree-based subquery ordering strategy as our baseline. This heuristic is a simple and pragmatic baseline commonly used in graph pattern matching and join planning [36].

Specifically, the baseline algorithm prioritizes subqueries containing property predicates with constants, as these typically yield higher selectivity and smaller intermediate results. At each step, the algorithm selects the next subquery from candidates connected to the current plan, always choosing the one with the smallest global degree (i.e., the total frequency of its edge labels in the data graph). For disconnected components, the procedure repeats by selecting the remaining subquery with the minimum global degree as a new starting point, until all subqueries are ordered. Compared to randomly generated query plans, baseline already achieves an average 21.42× speedup for complex queries that produce results.



(a) LDBC-SNB (SF-3)

(b) LDBC-SNB (SF-10)

(c) LDBC-SNB (SF-30)

(d) DBpedia (v2014)

Fig. 9. Query Execution Time of R2O and Baseline under METIS and MPC Partitioning Methods.

5.3.2 Query Performance. We evaluated the query performance of R2O and the baseline algorithm under two partitioning strategies, METIS and MPC. For each dataset and partitioner, we randomly generated 45 complex queries. We report results on 30 test queries and use the remaining 15 queries for training. Each test query was run three times, and the average execution time was reported. As shown in Figure 9, R2O achieves average speedups of 11.61×, 81.29×, 20.99×, and 22.38× on LDBC-SNB (SF-3, SF-10, SF-30) and the DBpedia (v2014), respectively, with the maximum speedup reaching 1637.99×. It is worth noting that the reported query time includes the time required for query plan generation (R2O averages 1.1 seconds, while the baseline averages 0.02 seconds), providing a comprehensive evaluation of our method's effectiveness in distributed query scenarios.

Table 4. Stage-Wise Average Query Time (sec).

Dataset	Partitioner	R2O				Baseline			
		CT[1]	LET[2]	JT[3]	Total[4]	CT	LET	JT	Total
LDBC-SNB (SF-10)	METIS	4.26	15.47	1.45	21.18	23.83	136.07	5.66	165.56
	MPC	3.52	12.98	0.82	17.32	16.72	97.21	3.66	117.59
DBpedia (v2014)	METIS	1.41	0.81	0.87	3.09	2.46	6.74	2.65	11.85
	MPC	1.64	1.15	0.43	3.22	6.62	52.73	13.76	73.11

[1]CT means the communication time; [2]LET means the local evaluation time;
[3]JT means the join time; [4]Total excludes planning time.

5.3.3 Evaluation of Each Stage. We decompose the end-to-end query time, excluding planning, into communication time (CT), local evaluation time (LET), and join time (JT). Table 4 reports averages over 30 test queries for LDBC-SNB SF-10 and DBpedia under METIS and MPC. On SF-10, LET dominates baseline. With partition-aware global ordering and local rewriting, R2O reduces LET and lowers JT and CT. Under METIS on SF-10, baseline LET and JT are 136.07s and 5.66s, whereas R2O reports 15.47s and 1.45s, and CT decreases from 23.83s to 4.26s. Under MPC, the same pattern holds, with baseline LET and JT of 97.21s and 3.66s compared to 12.98s and 0.82s under R2O, and CT decreasing from 16.72s to 3.52s. On DBpedia, absolute times are much smaller. R2O reduces LET and JT under both partitioners, while the reduction in CT is modest because the system imposes fixed communication overheads due to serialization and synchronization in the message-passing layer. Overall, the largest gains arise from lowering per-partition LET, with smaller but consistent decreases in JT and CT.

5.3.4 Comparison with Existing Query Optimizer. To further evaluate R2O, and given that there is no prior distributed optimizer for property-graph pattern queries, we compared it with GOpt [23], a state-of-the-art graph-native query optimizer that combines rule-based and cost-based strategies. We implemented a lightweight GOpt variant (with Neo4j as the underlying engine) and applied it to LDBC-SNB (SF-10) queries. Execution plans and planning times were collected, and the resulting query plans were decomposed according to data partitioning and executed in a distributed environment, enabling a direct comparison with our R2O framework.

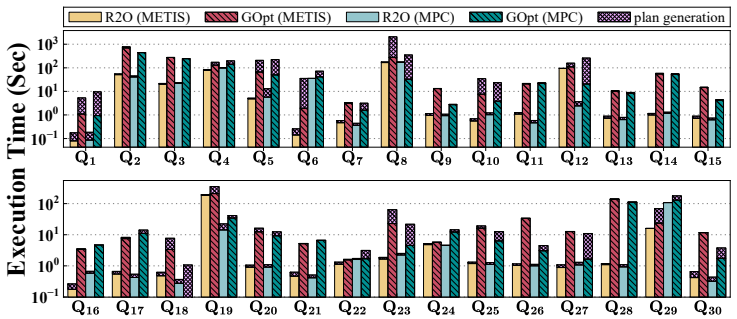


Fig. 10. Performance of R2O and GOpt on LDBC-SNB (SF-10) in Distributed Query Execution.

As shown in Figure 10, R2O achieves an average speedup of 20.23× over GOpt in query execution time, with the maximum speedup reaching 137.18×. Notably, the plan generation time is also

significantly reduced, with R2O averaging 62× faster than GOpt. This significant difference in planning time is primarily due to GOpt's extensive rule and cost estimation. Furthermore, as GOpt is not designed for distributed settings, its generated plans often yield slower distributed query execution. In some cases, GOpt's performance is even inferior to simple heuristic baselines. Overall, R2O generates more efficient plans and consistently outperforms GOpt in distributed query processing.

5.3.5 Comparison with Existing Distributed Graph System. In the absence of a distributed optimizer for property-graph pattern queries, we benchmark R2O on DBpedia (v2014) [22] against the distributed SPARQL engine gStoreD [30, 31], using it as a external baseline. R2O operates on the property-graph conversion with METIS/MPC partitions, while gStoreD runs on native RDF; both use 6-node cluster. We translate our Cypher patterns into semantically equivalent SPARQL BGPs (preserving labels, predicates, and property filters), and report the average of three runs.

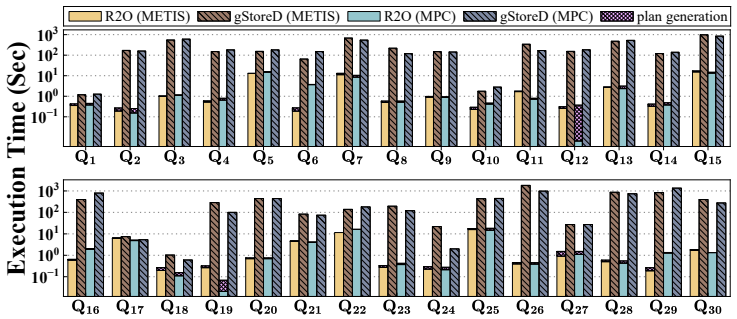


Fig. 11. Performance of R2O (Property Graph/Cypher) and gStoreD (RDF/SPARQL) on DBpedia (v2014).

As shown in Figure 11, R2O attains lower end-to-end runtimes on most DBpedia queries under both partitioners, with the largest gains on star or hybrid patterns with selective properties because partition-aware ordering and local rewriting shrink intermediates and inter-partition traffic. We observe a maximum speedup of about 3975× (Q_{26} under METIS, from 1809.90s to 1.16s). For a small number of path-heavy queries with high cross-partition traversals, the gap narrows because result assembly and synchronization dominate (Q_{17} under MPC, from 5.34s to 5.07s).

Table 5. Average Query Time (sec) on LDBC-SNB (SF-3) Before and After Data Updates.

Partitioner	Before Update			After Update		
	R2O	Base	Speedup	R2O	Base	Speedup
METIS	21.40	52.46	2.45×	22.10	53.40	2.41×
MPC	37.34	53.99	1.45×	38.20	56.00	1.46×

5.3.6 Model Reusability after Data Updates. We evaluate whether the learned R2O policy can be reused after data changes without retraining. We delete 5% of the data and add 5% new data, then rerun the same query workload on the updated database using the original model. Table 5 reports average query time on LDBC-SNB SF-3 under METIS and MPC, before and after update. Latency stays in the same range, indicating a 10% content change does not invalidate the learned policy. The speedup of R2O over the baseline is preserved. Under METIS, the baseline is 2.45× slower

than R2O before the update and 2.41× slower after the update. Under MPC, the baseline is 1.45× slower before the update and 1.46× slower after the update. This suggests R2O can be reused on an updated database without retraining.

5.4 Ablation Study

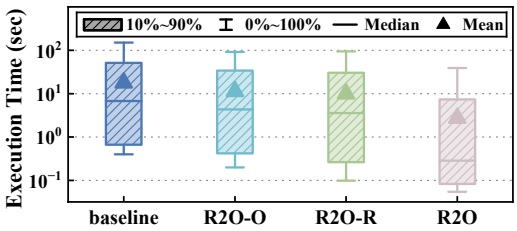


Fig. 12. Ablation Study of Global Ordering and Local Rewriting on LDBC-SNB (SF-30).

Figure 12 shows query execution time distributions for four variants on LDBC-SNB (SF-30): baseline, global ordering only (R2O-O), local rewriting only (R2O-R), and the full R2O framework. Compared to the baseline, R2O-O and R2O-R reduce the median execution time by 35.62% (from 6.76s to 4.35s) and 47.29% (from 6.76s to 3.56s), and the mean by 36.47% (from 17.84s to 11.33s) and 43.96% (from 17.84s to 10.00s). Both single-module variants still show long upper whiskers, indicating limited improvement for long-tail queries. The full R2O framework achieves the best performance, reducing the median and mean by 95.77% (from 6.76s to 0.29s) and 84.66% (from 17.84s to 2.74s). The distribution is more concentrated and stable. The upper whisker of the full R2O drops by 74% (from 149.91s to 39.09s), showing substantial benefits for even the most complex queries. These results show that global ordering and local rewriting provide complementary strengths: the former avoids costly execution paths, while the latter optimizes intermediate results.

6 Related Work

6.1 Traditional Graph Query Optimization

In graph query processing, pattern queries are typically formalized as subgraph isomorphism or homomorphism with attribute constraints. Early work focused on backtracking and join-based matching [21]. For example, QuikSI [6] orders matches by weighting query edges by label frequency, while RI [37] repeatedly selects the highest-degree vertex. These methods improve efficiency through candidate filtering, order optimization, and pruning. Many integrate constraint propagation and neighborhood-consistency checks to discard infeasible candidates, stabilizing performance under skewed labels and high-degree vertices. For distributed graph processing, systems such as TriAD [12] use graph summaries for join-ahead pruning to reduce intermediates and communication. Wukong [38] applies full-history pruning to avoid unnecessary inter-partition joins. Lothbrok [1] leverages index-based cardinality estimation for data-aware subquery scheduling. S2RDF [35] uses semi-join preprocessing to shrink input, and Galaxybase [40] optimizes adjacency storage for locality and network efficiency. On distributed backends, semi-join reductions are paired with partition summaries and locality-conscious layouts to curb cross-partition traffic. Recent work includes large-scale query frameworks using Datalog-style interfaces and global operator optimizations [46], relational DBMS extensions for graph computation [47], and scalable graph analytics via storage- and workload-aware partitioning with pipelined parallelism [19]. Theoretical algorithms such as Delta-BigJoin [3, 5] use worst-case-optimal multi-way joins to bound computation and communication. Other advances include hybrid cardinality estimation via sampling and

synopses [18], benchmarks for subgraph matching cardinality [29], and pruning/equivalence-based matching to reduce search space [16]. However, heuristic and theoretical optimizations often fail to reach globally optimal plans for large or complex queries, leading to local minima and limited adaptivity.

6.2 Learning-Based Query Optimization

Learning-based query optimization is increasingly common. In relational systems, ReJOIN [26] and Neo [25] apply deep reinforcement learning to join order selection, using neural models to capture plan–reward signals. Tree-LSTM and graph neural networks further improve join tree representations [44], and Bao [24] uses learned hints to cut training data and system overhead. These ideas extend to graph queries. Mhedhbi et al. [27] propose a hybrid optimizer for multi-way joins, and Kim et al. [17] introduce TurboHOM++ to reduce RDF query intermediates via optimized enumeration and pruning. RL-OVO [41] and ReJOOSp [42] apply reinforcement learning to join order optimization for graph and SPARQL queries. Learned policies exploit signals (e.g., label frequencies, degrees) and data skew to guide enumeration, yielding adaptive plans and fewer intermediates under skewed workloads. In SPARQL, Aimonier-Davat et al. [2] combine dynamic programming with cost sampling for property-path join ordering, improving path query efficiency on datasets such as Wikidata. Despite this progress, prior work does not address both computation and communication cost in distributed property graph settings. We propose a dual-layer learning framework that integrates query rewriting and global ordering for adaptive optimization of complex distributed queries.

7 Conclusion

This paper presents R2O, a dual-layer reinforcement learning framework for optimizing distributed property graph queries. R2O applies partition-aware query rewriting to reduce intermediate results and communication by leveraging cross-partition edge information, and uses node and edge features that capture structural and semantic properties. By combining a GNN-based local rewriting model with an RL-based global ordering model, trained jointly end to end, R2O produces efficient distributed query plans. Experiments on large-scale benchmarks show that R2O outperforms baselines, reducing query time and communication across partitioning methods and data scales. R2O offers a scalable solution for distributed graph query optimization, and in future work we plan to extend it to multi-modal property graphs and more complex patterns to support heterogeneous, knowledge-intensive workloads.

Acknowledgments

Xu Zhou is also a corresponding author. The research was supported by the National Natural Science Foundation of China (Grant Nos. 62472156, U23A20317, 62576051, 62572182, 62532001, 62402481), the Creative Research Groups Program of the National Natural Science Foundation of China (Grant No. 62321003), and the Natural Science Foundation of Hunan Province (Grant No. 2025JJ50352, 2023JJ10016).

References

[1] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2023. Optimizing SPARQL queries over decentralized knowledge graphs. *Semantic Web* 14, 6 (2023), 1121–1165. doi:10.3233/SW-233438

[2] Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli, Minh-Hoang Dang, and Brice Nédelec. 2023. Join Ordering of SPARQL Property Path Queries. In *The Semantic Web*, Catia Pesquita, Ernesto Jimenez-Ruiz, Jamie McCusker, Daniel Faria, Mauro Dragoni, Anastasia Dimou, Raphael Troncy, and Sven Hertling (Eds.). Springer Nature Switzerland, Cham, 38–54.

[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 691–704.

[4] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. 2020. Mapping RDF Databases to Property Graph Databases. *IEEE Access* 8 (2020), 86091–86110.

[5] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. 2021. Worst-Case Optimal Graph Joins in Almost No Space. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 102–114. doi:10.1145/3448016.3457256

[6] 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.

[7] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2246–2258.

[8] 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 *SIGMOD*. Association for Computing Machinery, New York, NY, USA, 619–630.

[9] Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Sungpack Hong, and Hassan Chafi. 2023. Better Distributed Graph Query Planning With Scouting Queries. In *Proceedings of the 6th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Seattle, WA, USA) *(GRADES-NDA '23)*. Association for Computing Machinery, New York, NY, USA, Article 3, 9 pages. doi:10.1145/3594778.3594884

[10] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. doi:10.1145/3183713.3190657

[11] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. 2018. A study of partitioning policies for graph analytics on large-scale distributed platforms. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 321–334. doi:10.14778/3297753.3297754

[12] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 289–300. doi:10.1145/2588555.2610511

[13] Kongzhang Hao, Zhengyi Yang, Longbin Lai, Zhengmin Lai, Xin Jin, and Xuemin Lin. 2019. PatMat: A Distributed Pattern Matching Engine with Cypher. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM '19)*. Association for Computing Machinery, New York, NY, USA, 2921–2924. doi:10.1145/3357384.3357840

[14] JanusGraph. 2023. *Distributed, open source, massively scalable graph database.* http://janusgraph.org/

[15] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. arXiv:https://doi.org/10.1137/S1064827595287997 doi:10.1137/S1064827595287997

[16] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 925–937. doi:10.1145/3448016.3457265

[17] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *Proc. VLDB Endow.* 8, 11 (July 2015), 1238–1249.

[18] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 964–976. doi:10.1145/3448016.3457246

[19] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 395–410. doi:10.1145/3183713.3196915

[20] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.* 12, 10 (June 2019), 1099–1112. doi:10.14778/3339490.3339494

[21] Zhengmin Lai, Zhengyi Yang, and Longbin Lai. 2019. Improving Distribued Subgraph Matching Algorithm on Timely Dataflow. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 266–273. doi:10.1109/ICDEW.2019.000-2

[22] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.

[23] Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, Wenyuan Yu, Ying Zhang, and Jingren Zhou. 2025. A Modular Graph-Native Query Optimization Framework. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 566–579. doi:10.1145/3722212.3724425

[24] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. doi:10.1145/3448016.3452838

[25] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. doi:10.14778/3342263.3342644

[26] 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* (Houston, TX, USA) *(aiDM'18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. doi:10.1145/3211954.3211957

[27] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. doi:10.14778/3342263.3342643

[28] Anthony Papantonakis and Peter J. H. King. 1994. Gql, a declarative graphical query language based on the functional data model. In *Proceedings of the Workshop on Advanced Visual Interfaces* (Bari, Italy) *(AVI '94)*. Association for Computing Machinery, New York, NY, USA, 113–122. doi:10.1145/192309.192336

[29] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1099–1114. doi:10.1145/3318464.3389702

[30] Peng Peng, Lei Zou, and Runyu Guan. 2019. Accelerating Partial Evaluation in Distributed SPARQL Query Evaluation. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 112–123.

[31] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. 2016. Processing SPARQL queries over distributed RDF graphs. *The VLDB Journal* 25, 2 (April 2016), 243–268. doi:10.1007/s00778-015-0415-0

[32] Peng Peng, M. Tamer Özsu, Lei Zou, Cen Yan, and Chengjun Liu. 2022. MPC: Minimum Property-Cut RDF Graph Partitioning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 192–204. doi:10.1109/ICDE53745.2022.00019

[33] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (Pittsburgh, PA, USA) *(DBPL 2015)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/2815072.2815073

[34] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. 2021. The Future is Big Graphs: A Community View on Graph Processing Systems. *Commun. ACM* 64, 9 (2021), 62–71.

[35] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF querying with SPARQL on spark. *Proc. VLDB Endow.* 9, 10 (June 2016), 804–815. doi:10.14778/2977797.2977806

[36] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 364–375. doi:10.14778/1453856.1453899

[37] 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.

[38] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 317–332.

[39] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *SIGMOD*. Association for Computing Machinery, New York, NY, USA, 1887–1901.

[40] Bing Tong, Yan Zhou, Chen Zhang, Jianheng Tang, Jing Tang, Leihong Yang, Qiye Li, Manwu Lin, Zhongxin Bao, Jia Li, and Lei Chen. 2024. Galaxybase: A High Performance Native Distributed Graph Database for HTAP. *Proc. VLDB*

Endow. 17, 12 (Aug. 2024), 3893–3905. doi:10.14778/3685800.3685814

[41] 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)*. 245–258. doi:10.1109/ICDE53745.2022.00023

[42] Benjamin Warnke, Kevin Martens, Tobias Winker, Sven Groppe, Jinghua Groppe, Prasad Adhiyaman, Sruthi Srinivasan, and Shridevi Krishnakumar. 2024. ReJOOSp: Reinforcement Learning for Join Order Optimization in SPARQL. *Big Data and Cognitive Computing* 8, 7 (2024). doi:10.3390/bdcc8070071

[43] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2049–2062. doi:10.1145/3448016.3457237

[44] 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)*. 1297–1308. doi:10.1109/ICDE48307.2020.00116

[45] Ling Yuan, Jiali Bin, and Peng Pan. 2020. Optimized Distributed Subgraph Matching Algorithm Based on Partition Replication. *Electronics* 9, 1 (2020). doi:10.3390/electronics9010184

[46] Qizhen Zhang, Akash Acharya, Hongzhi Chen, Simran Arora, Ang Chen, Vincent Liu, and Boon Thau Loo. 2019. Optimizing Declarative Graph Queries at Large Scale. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1411–1428. doi:10.1145/3299869.3300064

[47] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-One: Graph Processing in RDBMSs Revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1165–1180. doi:10.1145/3035918.3035943

Received July 2025; revised October 2025; accepted November 2025