MICRO: A Lightweight Middleware for Optimizing Cross-store Cross-model Graph-Relation Joins

Xiuwen Zheng
University of California, San Diego
La Jolla, USA
xiz675@ucsd.edu

Arun Kumar
University of California, San Diego
La Jolla, USA
akk018@@ucsd.edu

Amarnath Gupta
University of California, San Diego
La Jolla, USA
a1gupta@ucsd.edu

***Abstract*—Modern data applications increasingly involve heterogeneous data managed in different models and stored across disparate database engines, often deployed as separate installs. Limited research has addressed cross-model query processing in federated environments. This paper takes a step toward bridging this gap by: (1) formally defining a class of cross-model join queries between a graph store and a relational store by proposing a unified algebra; (2) introducing one real-world benchmark and four semi-synthetic benchmarks to evaluate such queries; and (3) proposing a lightweight middleware, MICRO, for efficient query execution. At the core of MICRO is CMLero, a learning-to-rank-based query optimizer that selects efficient execution plans without requiring exact cost estimation. By avoiding the need to materialize or convert all data into a single model—which is often infeasible due to third-party data control or cost—MICRO enables native querying across heterogeneous systems. Experimental results on the benchmark workloads demonstrate that MICRO outperforms the state-of-the-art federated relational system XDB by up to 2.1× in total runtime across the full test set. On the 93 test queries of real-world benchmark, 14 queries achieve over 10× speedup, including 4 queries with more than 100× speedup; however, 4 queries experienced slowdowns of over 5 seconds, highlighting opportunities for future improvement of MICRO. Further comparisons show that CMLero consistently outperforms rule-based and regression-based optimizers, highlighting the advantage of learning-to-rank in complex cross-model optimization.**

***Index Terms*—cross-model query optimization, heteregenous data management, learning-to-rank based query optimizer**

I. Introduction

In today's data-driven era, various applications in various domains, such as biomedical research, cybersecurity, healthcare etc. [1]–[6], are increasingly relying on heterogeneous datasets from various sources and managed across distributed sites. The outdated notion of "one size fits all" in data management no longer holds due to two major factors:

- Ownership: many datasets are controlled by third parties, making it infeasible to consolidate all into a single system;
- Performance: different data models excel at different query types (e.g., graph databases for path traversal).

As a result, users often need to jointly query data stored in different models and DBMSs. To reduce the burden, *polystore systems* [7]–[11] have emerged, supporting queries across heterogeneous backends.

While polystores support multiple models, query optimization remains underexplored, with no well-defined formulation across models. Key challenges include:

- the lack of a unified algebra and cost model;
- the autonomy of the underlying engines, which restricts access to their internal optimization logic.

For example, BigDAWG [7]–[9] only optimize queries within each "island"—a single-model environment—while data movement across islands must be manually specified by users. As an initial step toward bridging this gap, this paper studies a specific important type of cross-model query: join queries between relational and graph data. By narrowing the scope, we aim to isolate and address key optimization challenges in this multi-model context, laying the foundation for broader research in cross-store query optimization.

A. Cross-model Cross-store Workloads

Mixed queries over graph and relational data is increasingly common across applications in social sciences, cybersecurity, social media analytics, and recommendation systems. For instance, researchers in security analytics and data governance [12], [13] use citation graphs and co-authorship graphs combined with relational bibliographic and patent data to analyze anomalies in international collaborations and identifying thematic areas of undue information leakage and the parties involved. Some prior work has studied multi-model query execution [14]–[16] by extending an RDBMS to support graph algorithms and optimizing such queries over an RDBMS. In contrast, we take a cross-store approach, respecting data ownership and leveraging the strengths of each native engine.

Motivating Workload. We consider a real-world cross-store cross-model query scenario. As shown in Fig. 1, the graph is derived from a benchmark such as LDBC [17], where nodes represent entities (forums, posts, persons, universities), and edges, their relationships. Relations come from external datasets such as forum platforms, QS university rankings, and Wikipedia: `Post` (`P`) with post metadata, `University` (`Un`) with city and ranking info, `City` (`Ci`) mapping cities to countries, and `Country` (`Co`) mapping countries to continents. A representative query retrieves hashtags of posts authored by people in *top-ranked Computer Science* departments at *Asian* universities, along with the corresponding forums.

A naive plan is to run the graph query, then move results to the relational engine for joins: a path traversal with a filter on `University` node property department retrieves all `(Forum, Post, University)` tuples (forum name,

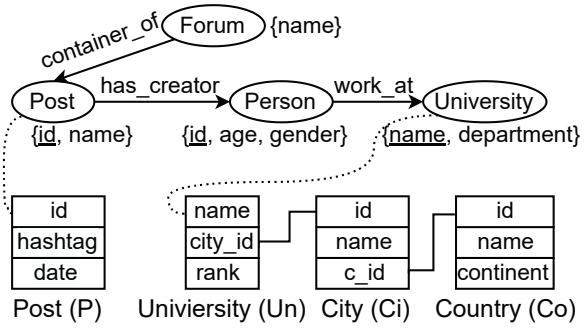

Fig. 1: Example cross-models workload

post id, university name). The results are then joined with tables using rank and continent filters. However, this plan can be inefficient: dense graphs yield large intermediate results, incurring high transfer costs.

Many alternative plans exist. For example, table `P` can be materialized as nodes (e.g., labeled `PR`) in the graph, enabling a node join with existing `Post` nodes. If `P` is small and `Post` nodes are central (high-degree), this plan can greatly reduce graph query result size and the subsequent data movement cost. However, if `P` is large, moving it to the graph is costly and the node join may introduce significant overhead. If the node joins are not selective, it may not reduce the graph query size much. Thus, determining which relations to move across systems is key to optimizing cross-model query execution.

B. Challenges

Optimizing this kind of cross-models join queries to find the best plan out of many candidates can be very challenging. We elaborate four main challenges as below:

C1: Development of a unified cost model. Each underlying DBMS has its own cost model, e.g., PostgreSQL's cost model yields relative cost units, while Neo4j's generates estimated cardinalities, making cross-engine cost comparisons infeasible. Additionally, these models depend on internal statistics (e.g., histograms, value distributions) to estimate cost. When sub-queries involve foreign data which is intermediate results produced by other systems and transferred at runtime, these statistics are missing, making cost estimation unreliable.

C2: Adaptiveness to dynamic environments. Federated databases are often deployed in distributed settings with varying hardware, locality, and network conditions. The cost of data movement across systems can differ significantly based on whether databases are co-located or geo-distributed. A practical middleware must be adaptive to these environmental factors to choose efficient plans.

C3: Lack of expressiveness in existing algebra. While prior work [18], [19] has extended relational algebra to support graph queries, existing frameworks still lack the expressiveness to represent cross-model queries. Similar to single-model optimizers, a unified algebraic foundation is needed to enable plan enumeration and transformation.

C4: Enlarged planning space. Unifying relational, graph, and cross-model operations into a shared algebra significantly increases the query plan search space. The optimizer must consider a combinatorial number of join orders and different data movement strategies. Even if an optimal plan is selected, the underlying DBMSs may not execute delegated sub-queries exactly as intended, introducing additional uncertainty in the actual runtime behavior.

C. Solution To The Challenges

To address these challenges, we build upon decades of research in RDBMSs, adapting their insights and techniques to this new context.

Recent work has explored using Machine Learning (ML) to help build query optimizers for both relational DBMSs and federated systems [20]–[23]. Many of these approaches propose a *learned cost model* or plan value function to estimate the quality (e.g., latency) of each candidate plan to replace the traditional cost models. However, training such models is often challenging: it requires a large number of training data considering the model complexity. In our cross-model, multi-store setting, the problem is even more complex since the total cost consists of multiple parts and involves two data models.

A recent system, Lero [24], states that predicting exact latency for each plan is an overkill since the goal of optimizer is to find the best execution plan. Instead of pointwise prediction (assigning a numeric cost to each plan), Lero proposes a learning-to-rank framework using a pairwise comparison model: given two plans, the model learns to predict which one is better. Their experiment proves that comparing to a pointwise regression model, training a binary classifier requires much fewer training samples.

S1: Predicting exact costs can be avoided. We adapt the learning-to-rank paradigm. C1 can be addressed by avoiding the need to predict exact cost.

S2: Learned model has better generalization. Unlike traditional rule-based or heuristics-based optimizers that rely on magic constants, learned models uses model parameters and can adapt to new system environments as long as sufficient training data is available.

S3: Introduction of cross-model join operators. Previous work [18], [19] has extended relational algebra to support graph queries by proposing graph relation and define operators on it. We define two new operators for joining between graph relations and relational tables. These operators enable algebraic expression for cross-model queries.

S4: Lightweight middleware architecture. We build a minimal middleware that focuses only on high-level decisions keeping the planning overhead low while enabling effective performance gain. It preserves the autonomy of individual engines, does not require modifications to their internal code which makes deployment easier, and delegates most optimization tasks to their native optimizers.

D. Contributions

We present **MICRO**, a lightweight MIddleware for CROss-model cross-engine queries. At its core is **CMLero** - a Cross-Model Learning-to-rank query optimizer that selects execution plans with low latency. Key contributions are:

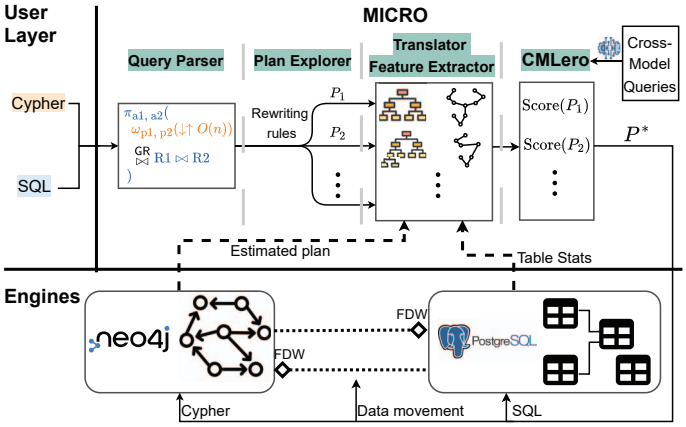

Fig. 2: System architecture for MICRO

- We formally define a class of cross-model join queries between graph and relational data, where a graph traversal is joined with relational tables via vertex properties and table columns. We provide a uniform algebra to express them.
- To the best of our knowledge, MICRO is the first to optimize such joins in a federated, cross-engine setting.
- We introduce a learning-to-rank framework for query optimization in the cross-model setting. It avoids the overhead of estimating exact costs of plans which is a complex task under cross-models cross-engines settings.
- We publish one real-world and four semi-synthetic benchmarks combining graph and relational data with hundreds of queries, benefiting future research in this area.
- MICRO is a lightweight middleware built on existing databases. It leverages mature native query optimizers while preserving autonomy, requiring no modifications to internal code and ensuring ease of adoption and extensibility.

The benchmark and MICRO code are publicly available to support reproducibility and future research.[1]

II. System Architecture

Fig. 2 provides an overview of MICRO architecture. MICRO is built on top of a graph database and a relational database, and takes as input a graph and a relational query. The system first parses both queries into a unified algebraic expression. A plan explorer then enumerates all candidate execution plans by rewriting the algebraic expression into equivalent forms. Each candidate plan is translated into corresponding graph and relational subqueries, along with the data movement operations for cross-model joins. Then they are encoded as feature vectors using statistics collected from the underlying databases. Then the learned comparator model, CMLero, will rank the plans. The top-ranked plan is selected and executed by delegating subqueries to the respective databases and transferring intermediate results across engines.

Extensibility of the system. The figure uses Cypher and SQL as example query languages for property graph and relations. MICRO is not tied to specific languages or engines. It can support other standard query languages—e.g., Gremlin for graphs—as long as appropriate parsers and translators between query syntax and the unified algebra are provided. Similarly, while PostgreSQL and Neo4j are used in our prototype, MICRO can integrate other engines, provided that the necessary features required by the learned comparator can be extracted.

Organization. The rest of the paper is organized as follows. Section III introduces background on relational graph algebra. Section IV presents our unified algebra for cross-model queries and formally defines the class of queries addressed in this work. Sections V formalizes the query optimization problem and describes the plan exploration framework. Section VI introduces the learned comparator model, CMLero. Section VII describes the benchmarks. Section VIII presents our experimental results. Finally, Sections IX and X discuss related work, conclude the paper, and outline future directions.

III. Preliminary

We briefly introduce the *Property Graph* data model, which is widely adopted by modern graph databases, and review prior work on extending relational algebra to support this model.

A. Data Model

We adapt the definition of property graph in [18], [19], [25].

Definition 1 (Property Graph): Let L_v, L_e be a set of vertices and edge labels respectively; P_v, P_e be a set of property names for vertices and edges respectively; D denote the domain of atomic values of all properties; $\text{SET}^+(X)$ be all subsets of set X. A property graph is defined as

$$G = \{V, E, \rho, \lambda_v, \lambda_e, \mu_v, \mu_e\},$$

where

- V is a set of vertices;
- E is a set of edges;
- $\rho : E \to V \times V$ is a function that associates each edge in E to a pair of source and target vertices.
- $\lambda_v : V \to \text{SET}^+(L_v)$ (or $\lambda_e : E \to \text{SET}^+(L_e)$) assigns vertex (or edge) labels to vertices in V (or edges in E).
- $\mu_v : V \times P_v \to D \cup \{\epsilon\}$ (or $\mu_e : E \times P_e \to D$) assigns each vertex (or edge) property to a value in D or a NULL value ϵ if the vertex (or edge) does not have that property.

To extend relational algebra to the property graph model, [18] introduced the notion of a *graph relation*, which allows graph elements to be represented relationally.

Definition 2 (Graph Relation): Given a property graph $G = \{V, E, \rho, \lambda_v, \lambda_e, \mu_v, \mu_e\}$, a *graph relation* r is a bag of tuples so that $\forall A \in sch(r) : dom(A) \subseteq V \cup E \cup D$, where $sch(r)$ is its schema and $dom(A)$ is the domain of attribute A.

B. Relational Graph Algebra

We first recap some *relational algebra operators* in Table I. Prior work [18], [19] has proposed *relational graph algebra*, which extends relational algebra with graph-specific operators for OpenCypher and Gremlin queries. In this work, we adapt and refine a subset of the operators proposed in [18], and present them in Table II. To avoid confusion, we use distinct

[1] https://github.com/xiz675/MICRO

notations to differentiate graph-specific operators from their relational counterparts. In schema definitions, we use $||$ to indicate column concatenation (i.e., appending attributes), and $\setminus$ to denote attribute removal.

Basic Operators. Some graph operators have direct analogues in relational algebra. Selection operator $\gamma_{\text{condition}}(r)$ filters tuples from a graph relation that satisfy a given condition. Projection operator $\omega_{x_1,x_2,\ldots}(r)$ extracts certain attributes from the graph relation. Join operator $\otimes$ is analogues to relational join $\bowtie$: it performs a Cartesian product between two graph relations and filters tuples based on common attribute values.

Graph-specific Extensions. Other operators are unique to the graph context. The get-vertices operator $O(v : L)$ retrieves all vertices with label L, returning a relation with a single attribute v. The path expansion operators $\uparrow_{v_1}^{v_2:L}[:T](p)$ and $\downarrow_{v_1}^{v_2:L}[e : T](p)$ traverse from a vertex v_1 to an adjacent vertex v_2 with label L in the outgoing or incoming direction via an edge with label T, and $\updownarrow_{v_1}^{v_2:L^+}[:T](p)$ traverse from v_1 to v_2 by one or more T edges. These operators extend the schema of the input graph relation with two additional attributes representing vertex v_2.

Example. With these operators, the graph pattern matching query from our motivated example can be expressed as:

$$p = \uparrow_{n_2}^{(n_3:\text{University})} \uparrow_{n_1}^{(n_2:\text{Person})} \uparrow_{n_4}^{(n_1:\text{Post})} O(n_4 : \text{Forum}),$$

where edge types and selection operators are omitted for brevity. To extract needed information, apply a project operator: $X = \omega_{n_1.\text{id}\to\text{pid},n_4.\text{name}\to\text{fname},n_3.\text{name}\to\text{uname}}(p)$

IV. Cross-Model Query Definition

To support unified cross-model query processing, we extend relational algebra with operators that enable joins between graph and relational data. This integrated algebra forms the foundation for formally defining cross-model join queries.

A. Operators for Cross-Model Joins

We introduce two operators to support joins between relational tables and graph vertices, summarized in Table III. Each operator transforms one data model into the other and applies a native join in the destination model.

TABLE I: Relational algebra operators

Operation	Notation	Description
Join	$r_1 \bowtie r_2$	Joins two relations r_1 and r_2 on their common attributes.
Projection	$\pi_{a_1,a_2,\ldots}(r)$	Extracts specified attributes (columns) from a relation r.
Selection	$\sigma_{\text{condition}}(r)$	Filters tuples from relation r that satisfy a given condition.

TABLE II: Relational graph operators

	Operation	Notation	Schema
Relation Analogous	Selection	$\gamma_{\text{condition}}(p)$	$\text{sch}(p)$
	Projection	$\omega_{x_1,x_2,\cdots}(r)$	$\langle x_1, x_2, \cdots \rangle$
	Join	$p_1 \otimes p_2$	$\text{sch}(p_1) \Vert (\text{sch}(p_2) \setminus \text{sch}(p_1))$
Graph Extension	Get-vertices	$O(v : L)$	$\langle v \rangle$
	Expansion	$\updownarrow_{v_1}^{v_2:L}[:T](p)$	$\text{sch}(p) \Vert \langle v_2 \rangle$
		$\updownarrow_{v_1}^{v_2:L^+}[:T](p)$	$\text{sch}(p) \Vert \langle v_2 \rangle$

TABLE III: Cross-model Join Operators

Operation	Notation	Output and Schema
Relation-Graph Join	$r \overset{\text{RG}}{\bowtie} p$	graph relation, $\text{sch(r)} \Vert (\text{sch(r)} \setminus \text{sch(p)})$
Graph-Relation Join	$p \overset{\text{GR}}{\bowtie} r$	relation, $\text{sch(p)} \Vert (\text{sch(p)} \setminus \text{sch(r)})$

Relation-Graph Join. The operator $r \overset{\text{RG}}{\bowtie} p$ first transforms the relational table r into a graph representation by mapping each tuple to a vertex and its attributes to vertex properties. A temporary label $l \notin L_v$ is assigned to these new vertices to avoid conflicts with existing graph labels. These vertices are then joined with the graph relation p using a graph-native join operator on vertex properties. Formally: $r \overset{\text{RG}}{\bowtie} p = \omega_{a_1,a_2,\ldots}(O(n : l)) \otimes p$ where a_i are properties used to match with vertices in p. The output is a graph relation.

Graph-Relation Join. The operator $p \overset{\text{GR}}{\bowtie} r$ first materializes the graph relation p as a relational table r_p, then joins it with the table r using relational join operator. Formally: $p \overset{\text{GR}}{\bowtie} r = r_p \bowtie r$. The output is a relational table.

With these integrated operators, the motivated workload in Fig. 1 can be expressed as:

$$\pi_{\text{fname, hashtag}}(\tag{1a}$$

$$X \overset{\text{GR}}{\bowtie}_{\text{pid=P.id}} \text{P} \bowtie_{\text{uname=Un.name}} \text{Un} \bowtie \text{Ci} \bowtie \text{Co}), \tag{1b}$$

Equation 1b applies graph-relation join to materialize the graph relation X as a relational table to join with the other tables by relational joins. 1a projects the output attributes using the relational projection operator.

B. Cross-model Join Query

Using the relational graph operators introduced, we formalize the notion of a graph query as follows:

Definition 3 (Graph query): A graph query can be expressed as an algebraic expression using the relational graph operators listed in Table II.

Building on this, we define a specific class of cross-model join queries that are the focus of this paper:

Definition 4 (Cross-model graph-relation join (CMGRJ) query): A CMGRJ query consists of a *single* graph query; and a subsequent *single* relational query over the graph query result materialized as a relation and relations, expressed using the relational operators in Table. I.

More generally, modern graph and relational databases support more operators (e.g., system-specific analytical ones). However, for the purposes of designing the benchmark queries and presenting a generalizable architecture, we focus on a core set of basic operators. There are different patterns of cross-model join queries, such as those where a complex relational query followed by a graph query that joins the materialized vertices of the relational query result with vertices of one or multiple graphs. The optimization of this type of queries is analogous to that of the CMGRJ queries. This paper mainly focuses on CMGRJ queries.

V. Design Decision, Optimization Problem and Plan Explorer

We revisit challenge **C4** from Section I-B through a concrete example. Given a CMGRJ query, the combination of relational, graph-specific, and cross-model operations introduces a vast plan space. For the motivated workload, Fig. 3 (A) shows a possible plan, where the blue-shaded regions denote operations performed in the relational database, while yellow-shaded regions denote operations in the graph database.

In this plan, the tables `P` and `Un` are moved to the graph database, transformed into vertices and joined with the `Post` and `University` vertices on properties, `id` and `name`. The partial path from `Post` → `Person` → `University` is then materialized as a relation and joined with `Ci` and `Co` in the relational database. The resulting table is moved back to the graph database and joined with another partial path `Forum` → `Post` on shared `Post` vertices.

Many alternative plans are possible by reordering joins, traversal steps, or choosing different points for data transfer across systems. The joint optimization of relational, graph, and cross-model operations results in an exponentially growing plan space. Even within the graph engine, traversal orders affect performance; similarly, the relational engine offers a large join-order space. Adding cross-model joins further increases complexity due to decisions on where each join should occur and their associated data movement costs.

A. Problem Definition

To address this complexity, we design a lightweight middleware that avoids exhaustively searching the entire plan space. Instead, it focuses on high-level strategic decisions about data movement and join execution locations for cross-model vertex-relation joins. It does not decompose the graph query; rather, the middleware operates as follows:

- Step 1: Execute filters and joins over relational data in relational engine and materialize the results as tables; apply filters on node properties and assign these nodes a new label.
- Step 2: Identify vertex types to move to relational engine, join with corresponding tables with results stored as tables.
- Step 3: Identify selected tables and the join results to transfer to the graph engine and materialize them as vertices;
- Step 4: Perform joins between new vertices and the original graph query in the graph engine;
- Step 5: Transfer the graph query result back to the relational engine and complete the remaining joins there.

In Step 3, vertex–table join results must be moved back to the graph engine to participate in the path traversal graph query. Under this design, we formalize the optimization problem:

Definition 5 (Cross-store Cross-model query optimization): Given a property graph in a graph database and a set of relations in a relational database, for a CMGRJ query, select vertices (by label) to materialize as tables in the relational database and relations to materialize as vertices in the graph database. The objective is to minimize end-to-end latency which includes: 1) data transfer cost across databases, and 2) native query execution time within each database.

B. Plan Explorer

Fig. 3 (A) illustrates the plan generation process for the motivating workload. A CMGRJ query, consisting of a Cypher query and a SQL query, is first parsed into an integrated algebraic expression. The relational filters and joins are pre-computed and materialized as relation `U`: $\sigma_{rank<500}(\text{Un}) \bowtie \text{Ci} \bowtie \sigma_{continent='Asia'}(\text{Co})$. The original algebraic expression derived from the query is simplified as: $\pi_{\text{fname, hashtag}}(X \overset{\text{GR}}{\bowtie}_{\text{pid=P.id}} \text{P} \bowtie_{\text{uname=U.name}} \text{U})$ which is the raw plan with no data movement. Candidate plans are generated by choosing different vertices and tables to materialize in the other engine. Each plan is translated to new Cypher and SQL queries with explicit data movement instructions. We omit parser and translator details and focus on plan generation.

Plan Space Complexity. For each vertex–relation join, there are three options: (1) move vertices to the relational engine, (2) move the relation to the graph engine, or (3) keep both in place and join the graph query result with the relation in the relational engine. We state a theorem on the plan space size.

Theorem 1: Let n be the number of relational tables with join attributes matching graph vertex properties, then the number of candidate plans is $O(3^n)$.

C. Pruning Rules

One possible plan for the motivating query is to move the Post vertices to the relational engine, joined with table `P`, and then transferred back to the graph engine to participate in the graph query. The plan is expressed by $\pi_{\text{fname, hashtag}}((O(:\text{Post}) \overset{\text{GR}}{\bowtie} \text{P}) \overset{\text{RG}}{\bowtie} X \overset{\text{GR}}{\bowtie} \text{U})$.

Alternatively, table `P` can be moved to the graph engine: $\pi_{\text{fname, hashtag}}(\text{P} \overset{\text{RG}}{\bowtie} X \overset{\text{GR}}{\bowtie} \text{U})$. The key difference is that the first plan materializes the post entity join in relational engine before executing the graph query, while the second lets the graph engine optimize the entity join with the original graph query natively by moving Post table to graph engine. The first plan also incurs two data transfers from graph to relational engine (two $\overset{\text{GR}}{\bowtie}$ operators) which can be costly.

In general, plans that move vertices to the relational engine add additional data transfers and force materialization for certain entity joins, limiting the benefit of the native graph optimizer. We therefore prune such plans. In the 5-step framework, we remove Step 2 and only consider table movements. This reduces the plan space from 3^n to 2^n. In the motivated example, the plans drop from 9 to 4. Fig. 3 (C) shows these plans where circles denote relations materialized as graph vertices, and rectangles denote intermediate results transferred to the relational engine. This pruning strategy significantly simplifies planning while preserving good performance.

D. Lightweight Middleware

We justify the design of our plan explorer and pruning rules along four dimensions.

Reducing Planning Complexity. By delegating intra-model planning to native engines, the middleware avoids enumerating the full cross-product of possible plans across models, reducing complexity and overhead.

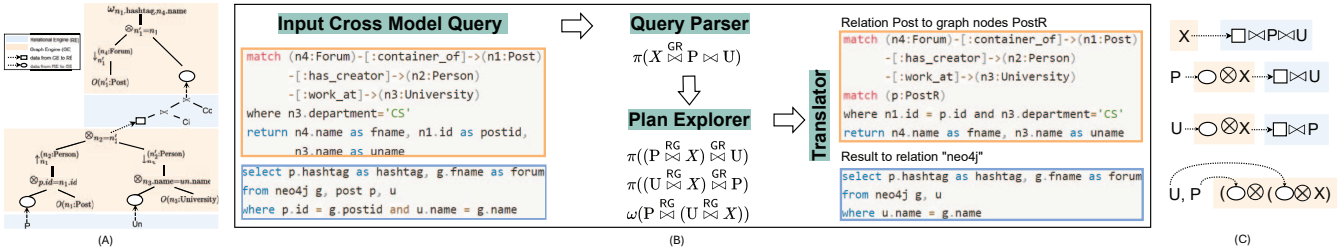

Fig. 3: (A) An example plan for motivated workload (B) Plan explorer framework; (C) Candidate plans for motivated workload.

Leveraging Native Optimizers. Graph systems excel at traversal and pattern matching, while relational systems excel at joins and filtering. The middleware reuses these native optimizers rather than re-implementing them.

Preserving Autonomy and Generalizability. Our design does not impose detailed plans on underlying engines. This preserves engine autonomy, maintains compatibility with future updates, and enables extension to other systems.

Enhancing Scalability and Reducing Data Movement. Data movement is often the primary bottleneck in cross-system queries [26]. Data transfers are limited to at most one in each direction (relational $\rightarrow$ graph and graph $\rightarrow$ relational), avoiding multiple round-trips, unlike the plan in Fig. 3 (A) (two relational $\rightarrow$ graph transfers) or pruned plans with two graph $\rightarrow$ relational transfers. Filters and joins are also pushed down to the relational engine to shrink the data moved initially.

VI. Learned Comparator Model

The plan comparator CMLero is a learned model designed to rank all the plans generated by the plan explorer for a given CMGRJ query. The structure of CMLero is shown in Fig. 4.

A. Feature Extractor

As shown in Fig. 3 (A), a candidate plan from the translator includes data movement, a modified Cypher and SQL query. We extract features from each plan.

1) Plan structure: As Fig. 4 feature extractor part shows, each plan consists of two parts:

Modified Neo4j Plan Tree. A tree-structured representation of the raw Cypher query plan, modified to account for data movement. If the Cypher query includes external tables moved from the relational DBMS, Neo4j cannot generate a complete plan. Thus, we first generate a query plan tree for the original Cypher query without external data, then modify it to include new nodes: `NodeFromRelation`: Represents tables moved to Neo4j and transformed into vertices; and `NodeHashJoin`: Represents joins between the new nodes and existing graph nodes on common node properties. These modifications integrate external data into the Neo4j plan tree structure. When a table `A` is moved and joined with label `L` nodes, the original plan tree will be traversed to find the plan node, say N, that firstly visits label `L` nodes. Then a `NodeHashJoin` plan node will be inserted so that its parents inherits the parents of N and N becomes its child node, and a `NodeFromRelation` plan node will be added to be the other child of `NodeHashJoin`. This modified plan may not be the same as the real plan executed by Neo4j after the tables are actually moved there, but it captures the cross-model joins information.

Join graph. A graph-structured representation of the SQL query, highlighting the star join pattern between the materialized graph relation and other relational tables since all the relational joins and filters on relations only are pre-computed and materialized. Operators include 1) `Neo4jResult`: Materializes the Cypher query result as a relation; `NodeByLabelScan` and `TableScan`: Project relevant columns from the graph relation or native tables; and `Join`: Represents relational joins.

2) Plan Node Encoding: Each plan node in the modified Neo4j plan tree or join graph is encoded into a feature vector. These vectors capture:

Node type. One-hot encoded for operator type.

Cardinality. In the Neo4j plan tree, Neo4j provides estimated output cardinality for native plan nodes. For the newly inserted nodes: we assign output cardinality of `NodeFromRelation` as the relation size, and that of `NodeHashJoin` as unknown. In the join graph, we instead use input cardinality for each operator node. For example, `Join` node has two children: `RelationScan` and `NodeByLabelScan`, thus its input cardinalities will be the relation size and the estimated size of the graph query result. Like Lero, a min-max normalization is applied to the logarithmic cardinality values to account for their wide range.

Touched tables and labels. Binary indicators for all tables and vertices labels that have been visited by the current node and all its descendants. For example, the `Neo4jResult` node in the join graph contains all the tables and labels that are touched by the modified Neo4j plan tree.

B. Model Structure

Similar to the Lero model [24], CMLero comprises of plan embedding layers and a comparison layer. It takes two encoded plans, maps each to a one-dimensional value through its embedding layers, and compares the two values to output a score $p \in [0, 1]$ that indicates which plan has lower latency.

1) Plan Embedding Layers: After encoding each plan node, the modified Neo4j plan tree is a tree structure of vectors and the join graph is a graph of vectors. Unlike Lero, which only embeds a tree structure, CMLero embeds both a tree and a graph structure. The embedding layers consist of:

Tree embedding layers. Tree embedding layers use Tree-Based Convolution Neural Network (TBCNN) [24], [27], [28].

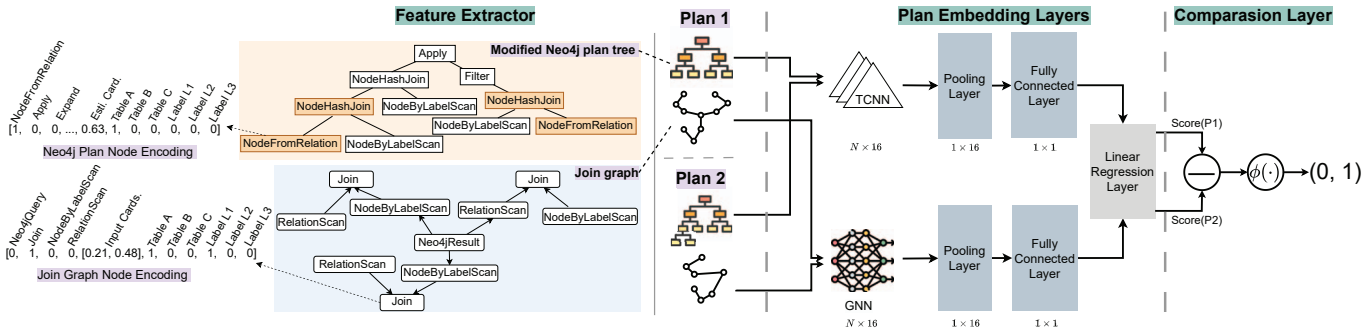

Fig. 4: CMLero: Learned comparator model structure.

Different from a general CNN which uses a small matrix of weights to slide across an input image, TBCNN uses a binary tree kernel (a parent node and two children node) to slide over an input tree. After applying three TBCNN layers, the original tree is transformed into a new tree with dimension $N \times 16$ (N is the number of plan nodes in the original tree) where each node aggregates the information of itself and its two children. Then a dynamic pooling layer aggregates the nodes features to generate a tree embedding with dimension 1×16. Lastly, a fully connected layers maps the aggregated tree features into a scalar latency-related score.

Graph embedding layers. Graph Embedding consists of two-layer Graph Convolution Network (GCN) that update each node's feature by aggregating its neighbors' features, a dynamic pooling layer that summarizes the entire graph into a fixed-length vector which is 16, and a fully connected layer to reduce this to a single latency-related scalar.

Linear regression layer computes the weighted sum of the two scalar output from tree and graph embeddings to produce a final scalar latency-related score of the plan.

2) Plan Comparator: Let $\mathcal{P}(q)$ be the candidate plans set for a query q, given two candidate plans $P_i, P_j \in \mathcal{P}(q)$, let $\text{Emb}(P_i), \text{Emb}(P_j)$ be their scalar embeddings. A sigmoid function is applied on their difference:

$$\hat{y_{ij}} = \sigma(\text{Emb}(P_j) - \text{Emb}(P_i)) = \frac{1}{1 + e^{-(\text{Emb}(P_j) - \text{Emb}(P_i))}}$$

$\hat{y_{ij}} \in [0, 1]$ indicates the probability that P_i is better than P_j in terms of latency. $\hat{y_{ij}} > t$ where t is a threshold value indicates P_i is preferable, a higher $\hat{y_{ij}}$ means P_i is more preferable; when $\hat{y_{ij}} \leq t$, P_j is preferable.

3) Loss Function: Let $y_{ij} \in \{0, 1\}$ be the true label. Let $y_{ij} = 1$ if P_i has lower latency and 0 if P_j is better with lower latency, then the likelihood of y_{ij} subjects to a Bernoulli distribution: $P(y_{ij}|\hat{y_{ij}}) = \hat{y_{ij}}^{y_{ij}} \cdot (1 - \hat{y_{ij}})^{(1-y_{ij})}$. The log-likelihood would be:

$$\log P(y_{ij}|\hat{y_{ij}}) = y_{ij} \log(\hat{y_{ij}}) + (1 - y_{ij}) \log(1 - \hat{y_{ij}})$$

For a set of workload queries $\mathcal{Q}$, the goal is to maximize likelihood over all candidate plan pairs of all queries, equivalent to minimizing the negative log-likelihood over all samples:

$$\min L = -\frac{1}{N} \sum_{q \in \mathcal{Q}} \sum_{1 \leq i < |\mathcal{P}(q)|} \sum_{i < j \leq |\mathcal{P}(q)|} \log P(y_{ij}|\hat{y_{ij}}) \quad (2)$$

where N is the total size of samples.

C. Model Training

The embedding and comparison layers are shared for two plans, so only one model needs to be trained. To train CMLero, i.e., updating parameters in the plan embedding layers, we use stochastic gradient descent (SGD) on a large set of training samples. Each sample consists of a pair of plans with a binary label indicating which plan has lower latency.

For each training query $q \in \mathcal{Q}$, the plan explorer generates all the candidates $\mathcal{P}(q)$. These plans are executed to collect the actual latencies. For each pair (P_i, P_j), a label is assigned based on which has lower latency (1 if P_i is better, 0 otherwise), yielding $N_q = \frac{|\mathcal{P}(q)|(|\mathcal{P}(q)|-1)}{2}$ training samples. The total number of training samples over $\mathcal{Q}$ is: $N = \sum_{q \in \mathcal{Q}} N_q$. Model parameters are trained to minimize Equation 2. SGD enables efficient optimization across this large dataset.

D. Inference

Once trained, CMLero ranks candidate plans for a test query q_t. The plan explorer generates $\mathcal{P}(q_t)$. Each plan is represented by a Neo4j plan tree and a join graph with each node encoded by feature vector. These are passed through the embedding layers to produce a single ranking score. Plans are then sorted by score, with lower scores indicating lower expected latency. The top-ranked plan is selected for execution.

VII. Benchmark

To the best of our knowledge, there are no publicly available benchmarks for evaluating cross-database, cross-model query systems. To fill the gap and assess the performance of MICRO, we develop two benchmarks: the **OpenAlex-USPTO Benchmark**, built from real-world heterogeneous data, and the **CM-LDBC Benchmark**, a semi-synthetic dataset extending the LDBC graph benchmark [17] with relational components. Both are publicly released to support future research in cross-model query optimization.

A. OpenAlex-USPTO (OA-USPTO) Benchmark

1) Dataset.: OpenAlex [29] is an open catalog of global scholarly research. We extract millions of recent publications and associated metadata—authors, institutions, and topics—filtered by: (1) physical sciences domain, (2) over 10

TABLE IV: Summary of OpenAlex node types and properties

Label	Node Properties	#Nodes
Author	id, name, works_count, cited_by_count	3,294,820
Work	id, name, year, type, cited_by_count	8,969,495
Institution	id, name, ror, works_count, cited_by_count	112,271
Field	id, name	10
Subfield	id, name	89
Topic	id, name, works_count, cited_by_count	1,571
Keyword	name	11,400

TABLE V: Summary of some OpenAlex relationship types

Relationship	Source Node	Target Node	#Relationships
WORK_AT	Author	Institution	30,573,987
CHILD_OF	Institution	Institution	22,060
CREATED_BY	Work	Author	16,244,003
CREATED_BY	Work	Institution	13,472,802
RELATED_TO	Work	Work	15,800,754
ABOUT	Work	Topic	22,171,963
HAS_SIBLING	Topic	Topic	67,510
BELONGS_TO	Keyword	Topic	15,705

citations, and (3) published after 2010. From this, we construct a graph dataset summarized in Tables IV and V.

We also collect recent patents (filed after 2022) from the USPTO Open Data Portal [30] that mention keywords like "large language model," "artificial intelligence," and "machine learning," yielding 124,110 patents. Metadata includes inventors, affiliated institutions, and cited publications are also collected as relations. The relational schema and statistics are shown in Table VI. Fields in **bold** denote derived or preprocessed attributes discussed in the next subsection.

2) Entity Matching: To enable cross-model joins, we resolve entities across the graph and relational datasets referring to the same real-world concepts (e.g., authors and inventors), with details in our technical report [31]. After entity matching, entities can be joined by matching node properties with relational columns as shown in Table VII.

3) Workloads: We generate 500 cross-model queries using the LLM-based query generation method proposed in [32]. Among these, 465 queries are validated to be both syntactically and semantically correct. To simulate real-world filtering conditions, we inject random predicates on graph node properties and relational table columns. We randomly split the validated query set into 372 training and 93 testing queries. The test set is used to evaluate and compare MICRO against baseline optimizers. The following example illustrates a benchmark query, where `neo4j` refers to the materialized result of a Cypher query:

TABLE VI: USPTO relational tables and schemas

Relation	Schema	#Rows
patent	id, year, country, issue_date, title	124,110
pub_cited	**publication_name**, patent_ids	699,104
inventors	name, patent_ids	210,573
institution	name, **ror**, patent_ids	5,460
maingroup	**astj_topic_id**, symbol, patent_ids, ⋯	418
subclass	**astj_subfield_id**, symbol, patent_ids, ⋯	82
class	**astj_field_id**, symbol, patent_ids, ⋯	16

TABLE VII: Entity matching for nodes and relational columns

Node Label. Property	Relation. Column
Author.name	inventors.name
Work.name	pub_cited.publication_name
Institution.ror	institution.ror
Field.id	class.astj_field_id
Subfield.id	subclass.astj_subfield_id
Topic.id	maingroup.astj_topic_id

```
MATCH (a1:Author)<-[:CREATED_BY]-(w1:Work)-[:RELATED_TO]
  ->(w2:Work)-[:CREATED_BY]->(a2:Author)
WHERE a1.works_count > 50 AND a2.works_count > 50
RETURN w2.name AS w2name, w1.name AS w1name, a1.name AS
     a1name, a2.name AS a2name;
SELECT g.a2name, g.a1name, g.w2name, g.w1name
FROM neo4j g,publication_cited w,inventors a0,inventors a1,
WHERE g.a1name = a0.name AND g.a2name = a1.name
 AND g.w1name = w.name AND cardinality(a1.patent_ids) > 2;
```

Listing 1: One benchmark query

B. CM-LDBC Benchmark

The LDBC Social Network Benchmark (SNB) is a widely-used synthetic graph benchmark for evaluating graph database performance. It models realistic social networks with entities such as people, posts, forums, tags and organizations. Following the method in [32], we adapt SNB to construct a semi-synthetic cross-model benchmark. Structural connections and node ids remain in the graph, while node attributes—including `id`—are stored in separate relational tables.

1) Dataset: The SNB datasets offer multiple scale factors; we use two: `SF-1` and `SF-10`. For each, we generate two sets of relational tables, $T1$ and $T2$, with varying sizes and join cardinalities between entities, resulting in 4 Cross-Model LDBC (CM-LDBC) datasets with stats in our tech report [31].

2) Workloads: We generate over 4,000 cross-model queries on CM-LDBC using the approach from [32]. As that work does not define a diversity metric, we perform manual filtering to ensure variation in Cypher paths structure and SQL join patterns. We select 505 distinct queries, split into 385 for training and 120 for evaluation to enable representative and fair comparisons across optimizers.

VIII. Experiment

We evaluate the performance of MICRO on the two proposed benchmarks and compare it against a baseline federated relational system, XDB [26]. We also conduct ablation studies to assess the effectiveness of our learned optimizer, CMLero.

A. Configuration

For the OpenAlex-USPTO benchmark, the datasets are curated and managed by UCSD researchers. The OpenAlex graph is stored in a Neo4j instance, and the USPTO tables reside in a PostgreSQL database, each hosted on separate physical servers. MICRO is deployed on a third server provisioned via CloudLab [33], a public research infrastructure. The machine features a 24-core AMD 7402P CPU, 128 GB RAM, 1.6 TB of disk space, and runs Ubuntu 18.04.

For the CM-LDBC benchmark, the entire setup is deployed on a single CloudLab machine. We use Docker to host three containers—PostgreSQL storing relations, Neo4j storing graph data, and Python running MICRO—all connected within the same Docker network.

B. Single-data model Setting

We compare MICRO with XDB [26], a single-model federated relational system that, like MICRO, supports in-situ data access without data movement overhead across mediator and databases. To support this comparison, we transform our cross-model benchmarks into a single-model format compatible with XDB. The relational part of the dataset remains unchanged. We made modifications to the graph data:

- Graph is transformed into relational tables: each node label or edge type becomes a separate table, with each row encode a node or edge and columns denote nodes/edges properties.
- These tables are loaded into a PostgreSQL server on the same physical node that previously hosted the Neo4j server.
- For index we built on previous graph node properties, we build similar index on the corresponding relational columns.

We implement a simplified yet effective version of XDB: since there are only two relational servers, we designate the one with more data as the primary and use PostgreSQL's SQL/MED extension for accessing remote tables on the other Postgres server. Optimization features such as predicate and join pushdown are enabled through SQL/MED.

C. Optimizers Baselines

We compare CMLero against three optimizer baselines: two rule-based baselines and one learned-model based:

- **Baseline-TS (Table Size):** A threshold-based heuristic using table row count. For each query, if a table that joins with the graph has a row count smaller than a predefined threshold t, it is pushed into Neo4j for join execution.
- **Baseline-FVN (First-Visited Nodes):** Uses Neo4j's estimated plan for the raw Cypher query to identify variables accessed by `NodeByLabelScan`. Tables joining with these variables are moved to Neo4j. Since such nodes are often traversal entry points, early joins may help filter them at the start of traversing, reducing traversal cost.
- **Baseline-RLM (Regression Learned Model):** Shares CMLero's architecture but omits the pairwise comparison layer. During training, each plan is passed through the embedding layers, and its latency is used as the label. At inference time, the plan with the lowest predicted latency is selected.

D. End-to-End Results

All results are reported on the test query set. Baseline-RLM and **CMLero** are trained on training queries.

Fig. 5 (A) shows the total runtime for all test queries across five datasets and five methods. A red mark indicate one query that exceeded the 5-minute timeout. Key observations include:

Baseline-TS exhibits varying performance across datasets: While it outperforms XDB on some datasets (e.g., SF1-T1 and SF1-T2), it performances worse on others. It also fails to complete one query within the timeout on OpenAlex-USPTO. Besides, It requires manual tuning to determine an effective table size threshold. For each dataset, we experimented with thresholds from 1K to 500K rows and selected the value that yielded the best performance. The optimal threshold is 5K for the OpenAlex-USPTO benchmark and 100K for CM-LDBC. Notably, when database servers are deployed on different physical machines—as in OpenAlex-USPTO—the cost of moving large tables is significant, requiring smaller thresholds to avoid transferring large tables.

Baseline-FVN consistently underperforms across all datasets. This is likely because the estimated plan for the raw Cypher query without alien tables does not match the actual execution plan after relational data is moved to Neo4j. Also, it only targets early filtering at path start nodes to reduce path traversal cost ignoring other factors such as table movement cost and join selectivity between moved tables and nodes.

Baseline-RLM performs on par with or better than XDB across all datasets. **CMLero** consistently outperforms all baselines, achieving speedups of up to 2.1x, 1.6x, 1.6x, 1.3x, and 1.3x over XDB across all datasets. The end-to-end runtime includes the total inference time, covering model loading, feature extraction, and evaluation for all test plans, which remains under 2 seconds even on CPU-only CloudLab nodes and is negligible compared to the total query execution time.

Fig. 5 (C) upper part shows per-query runtime differences between CMLero and XDB on 93 test queries in OpenAlex-USPTO. Queries are sorted by time difference, with 23 queries exceeding a 5-second gap highlighted. CMLero is slower on 4 (4.3%) and faster on 19 (20.4%) of these.

E. Advantage of Heterogeneous Database

Different databases have distinct strengths. Graph engines like Neo4j are designed for efficient path traversal, which can be a bottleneck in relational systems. To evaluate this advantage, we isolate variable-length queries from test workloads, 12 from OpenAlex-USPTO and 5 from CM-LDBC, and compare runtime performance across methods (Fig. 5 (B)).

The direct rewriting translates a variable length Cypher query to a `WITH RECURSIVE` clauses defining the result table of recursive join, followed by filters and joins with other tables. However, PostgreSQL handles such recursive queries inefficiently, particularly in the presence of dense relationships (e.g., `WORK_RELATED_TO_WORK`), leading to significantly degraded performance. To ensure a fair comparison, we manually optimize the XDB implementation by pushing filters and joins into the base relations within the `WITH RECURSIVE` clause to improve execution efficiency. In contrast, MICRO requires no manual rewriting. Despite these efforts for XDB, MICRO achieves 2×–6× speedups on these path-intensive queries, demonstrating the clear advantage of leveraging native graph engines within a heterogeneous query execution framework.

F. Engine-specific Implementation of Rewrites

We profiled the four queries where CMLero was slower than XDB and found that the bottleneck lay in executing the rewritten Cypher query. These queries involve node property joins between newly created nodes and original nodes combined with path traversal. For example, the second query joins topic and work entities across two databases. The plan chosen by CMLero (which was the best among candidates)

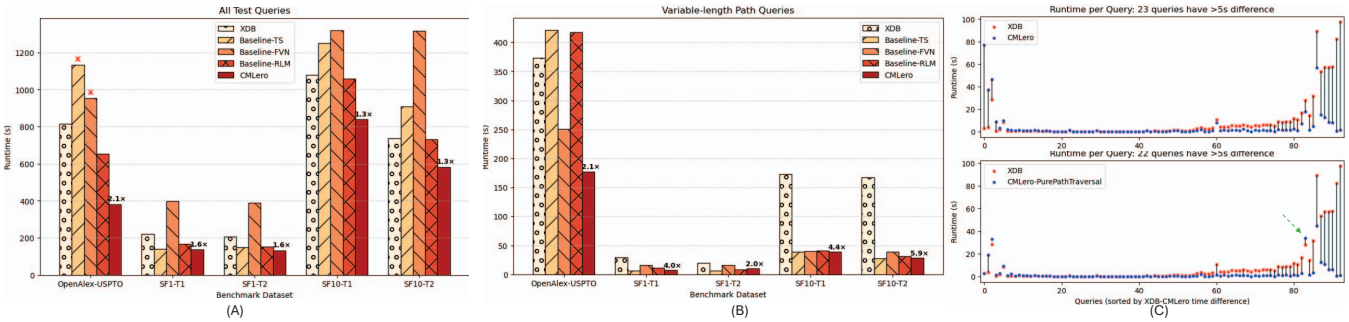

Fig. 5: (A) Total runtime of all test queries across two benchmarks and five methods; (B) Total runtime of variable-length queries across two benchmarks and five methods; (C) Per-query runtime for XDB and CMLero and highlighted queries with time difference larger than 5 secs.

produces the following Cypher query (we omit node property filter predicates and return clause for brevity):

```
MATCH (w1:Work)-[:ABOUT]->(t:Topic)<-[:ABOUT]-(:Work)-
    [:CREATED_BY]->(:Institution)
MATCH (l:TopicT) MATCH (a:WorkT) WHERE w1.name = a.name and
    t.id = l.id
```

Profiling this query in Neo4j shows that the engine performs the path traversal first, generating a large intermediate result, and only then applies the selective node property join. This yields around 4 million db hits in around 32 seconds, with the expensive value-hash join operator dominating cost. By contrast, XDB joins the small entity tables first, producing a much more efficient plan. Since Neo4j excels at path traversal, we replaced the node property join with explicit edges so that the query becomes a pure traversal. Concretely, when moving a table into Neo4j, we not only create nodes but also connect them to existing nodes if they share the same join property value. The node MATCH and WHERE clause then becomes:

```
MATCH (t)-[:EQ_T]->(:TopicT) MATCH (w1)-[:EQ_W]->(:WorkT)
```

This version yields around 1 million db hits in 3.6 seconds by pushing down the path traversal denoting small entity join. We integrated this translator into MICRO, retrained it, and evaluated it on the test queries. Results are shown in the lower half of Fig. 5 (C), aligned with the same query order. The four slow queries improved substantially. However, not all queries benefited: e.g., the query marked by the green arrow became slower because data movement time increased dramatically when creating both nodes and edges for the moved tables. If data movement cost outweighs traversal savings, this approach is worse. This highlights that deeper optimization requires engine- and workload-specific knowledge, but in this paper we focus on a general implementation, so we keep our original implementation of the translator.

G. Pruning Effectiveness and Training Cost

To test the effectiveness of the pruning rule, for the test queries of OpenAlex-USPTO benchmark, we run the pruned plans which move vertices from Neo4j to Postgres, and move the result tables back to Neo4j (without moving other tables), and compare their runtime with the corresponding plans in our planning space which move the tables of the same entities to Neo4j instead. We compare their runtime in Fig. 6, and out of the 634 plans, the retained plans perform better or similar in almost all cases; some pruned plans were faster but less than three seconds.

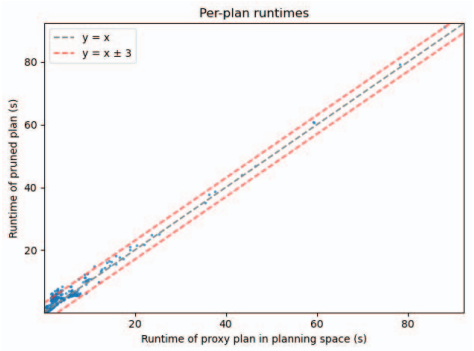

Fig. 6: Runtime comparison of pruned vs. retained plans.

TABLE VIII: Training data collection stats.

	OpenAlex-USPTO	SF10-T1
# plans	3192	4524
# pruned plans	13977	21411
collection time	7 hr 15 min	15 hr 20 min

Table VIII shows that pruning avoids nearly four to five times as many plans. Given the small (< 3s) advantage of the few better pruned plans, this loss is negligible relative to the training data collection time saved.

We further applied runtime-based pruning to stop long-running plans early. After materializing tables in Neo4j, we issue a `COUNT(<query>)` on the rewritten Cypher query. If the count takes longer than five minutes, we assign `EXTREME_LARGE_QUERY` as its runtime. If the count returns but exceeds a threshold where moving results back to Postgres would take > 5 minutes, we assign `LARGE_QUERY` time instead without running the queries. With these rules, execution time for plans of training queries on OpenAlex-USPTO were collected in about 7h15m, and on the largest benchmark, SF10-T1, in about 15h20m.

H. Effectiveness of CMLero Optimizer

We evaluate optimizer quality using two metrics:

TABLE IX: Optimizer comparison on CM-LDBC

Data	System	Top-3-HR	AvgQ	Q90	Q95
SF1-T1	Baseline-TS	0.66	7.67	3.54	34.93
	Baseline-FVN	0.20	18.96	10.07	24.04
	Baseline-RLM	0.67	5.13	2.74	3.54
	CMLero	**0.82**	**1.19**	**1.52**	**1.65**
SF1-T2	Baseline-TS	0.40	8.31	4.72	42.59
	Baseline-FVN	0.17	17.42	9.64	25.50
	Baseline-RLM	0.55	5.31	2.37	4.01
	CMLero	**0.86**	**2.40**	**1.82**	**2.35**
SF10-T1	Baseline-TS	0.68	7.14	3.12	6.57
	Baseline-FVN	0.32	5.27	8.12	15.31
	Baseline-RLM	0.51	5.65	3.34	4.90
	CMLero	**0.78**	**2.50**	**1.68**	**2.48**
SF10-T2	Baseline-TS	0.68	8.20	2.96	7.87
	Baseline-FVN	0.23	5.88	8.55	14.71
	Baseline-RLM	0.63	3.82	3.54	5.48
	CMLero	**0.83**	**1.99**	**1.53**	**2.55**

- **Q-Error**: The ratio of the latency of selected plan (ST) to the latency of the ground-truth best plan (GT): $Q = \frac{ST}{GT}$. We report the average Q-error across all test queries, along with the 90th and 95th percentile values.
- **Top-N Hit Rate (Top-N-HR)**: The proportion of test queries for which the selected plan falls within the top-N lowest latency plans: Top-N-HR $= \frac{|\text{Hit-Workloads}|}{|\text{Test-Workloads}|}$.

Table IX reports these metrics for all optimizers on the CM-LDBC benchmark. CMLero consistently achieves the best performance across all datasets, significantly outperforming the other heuristic-based baselines and regression based baseline in both Q-error and Top-3 Hit Rate. Notably, CMLero not only reduces total runtime but also ensures that the selected plans for all queries are generally robust, as indicated by the low 95th percentile Q-error.

Baseline-TS uses a fixed table size threshold across all CM-LDBC datasets, resulting in identical plans for SF1-T1 and SF10-T1 (and for SF1-T2 and SF10-T2) when relation sizes are the same. However, its performance varies, indicating that table size alone is an insufficient criterion. It overlooks key graph-specific and cross-model factors—such as how a table's role may shift with changes in graph statistics—leading to suboptimal plans. While it performs well in certain cases, it is consistently outperformed by CMLero. **Baseline-FVN**, in contrast, uses graph-side information but neglects the relational side and how the moved table affects the graph query. As a result, it performs poorly across all datasets.

Between the two learned models, **Baseline-RLM**, despite sharing the same model architecture as CMLero, performs worse on all metrics. This is largely due to its regression-based objective, which struggles with limited training data. As a result, it often fails to capture subtle performance differences between plans and is less reliable in selecting optimal or near-optimal plans. For example, for the SF10-T1 benchmark, the training queries produce a total of 4499 valid plans leading to 4499 training samples for the **Baseline-RLM** model, however, 31461 pairs of plans are generated which provides **CMLero** much more training data.

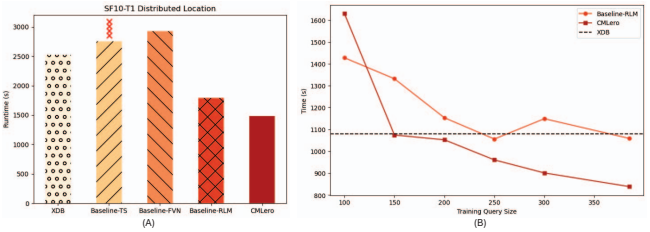

Fig. 7: (A) Runtime of all test queries on SF10-T1 benchmark in distributed environment; (B) Runtime of all test queries on SF10-T1 with different training sizes.

I. Adaptiveness to Distributed Environment

In our initial CM-LDBC experiments, Neo4j and PostgreSQL were deployed as Docker containers on a single CloudLab machine, connected via a shared Docker network. To assess the robustness of MICRO in more realistic deployment scenarios, we extended this setup to a distributed environment across three physical machines. Specifically, we deployed Neo4j, PostgreSQL, and MICRO on separate CloudLab nodes, connected through a shared internal network.

To emulate geographically distributed or cross-datacenter deployments, we imposed artificial network constraints between nodes: 50 Mbps bandwidth, 50 ms propagation latency, and a 64 KB burst allowance. We evaluate the SF10-T1 benchmark under this new setting with results shown in Fig. 7 (A). **Baseline-TS** has poor performance with 4 queries run out of time limit. **CMLero** achieves around 1.7× speedup over XDB, a more substantial improvement than 1.3x in the single-node setup. In distributed settings, data movement between databases has higher cost, and by choosing the right tables to move to Neo4j, CMLero avoids more cost. This demonstrates its adaptability to various distributed environments.

J. Optimizer Performance with Varying Training Query Sizes

To evaluate the impact of training size on the effectiveness of learned optimizers, for SF10-T1 benchmark, we vary the number of training queries from 100 to 385 and evaluate on the test query set. Fig. 7 (B) presents the total runtime of test queries under both learned optimizers, compared to the XDB baseline. **CMLero** reaches performance comparable to XDB with only 150 training queries and continues to improve as the training size increases. **Baseline-RLM**, however, requires around 250 queries to match XDB-level performance, and shows limited improvement beyond that point, indicating lower sample efficiency and less effective generalization.

K. Limitations and Discussion for Future Work

For ease of benchmark generation and model design, we currently restrict focus to a limited query pattern. More complex cases—such as nested joins or multi-round bidirectional joins—can arise in practice. An extension of CMLero is to decompose such queries into multiple Cypher+SQL segments, apply the model to each segment, and aggregate their scores into a final prediction. Future work may also explore attention mechanisms across segments to better capture interactions.

This extension does not require a redesign of the overall architecture, but rather incremental additions.

Besides, our experiments use offline training on a fixed benchmark, however, the framework can be extended when the data updates or fresh: the learned model can be incrementally retrained starting from existing parameters rather than from scratch, reducing training cost and preserving prior knowledge. CMLero uses estimated plan from Neo4j collected statically offline, adaptive re-optimization maybe needed when the actual Neo4j plan deviates a lot from the estimates. Since all candidate plans for a query share the Neo4j's plan for the original Cypher query, recollecting the real query plan can benefit the evaluation of all candidates. Exploring continuous or online fine-tuning is an interesting direction for future work.

IX. Related work

ML-Based Query Optimization for RDBMS. ML-based methods have been developed for query optimizers. A majority of them work on learned cardinality estimations (CE) and fall into two groups: data-driven approaches that model data distributions using generative models [34]–[37], and query-driven models trained on sampled queries [38]–[40]. Experiment [41] shows that these methods significantly outperform traditional techniques at CE, and integrating them into DBMSs has yielded real performance gains [42], [43]. Some ML approaches focus on learned cost model [20]–[22] which utilize neural networks to estimate the cost of a query; systems like Neo [27], Balsa [44] and Bao [23] formulate plan enumeration as a Reinforcement Learning (RL) problem and designs deep neural networks such as TCNN as the value network to predicate the latency of plans to choose the optimal one. As compared in Section I-C, these pointwise approaches have certain limitations and Lero [24] provides pairwise approach which overcomes these shortcomings.

Query Optimization in Federated Database System. Federated Database System (FDBS) integrates multiple databases into a unified system. Queries are decomposed and delegated to different databases and their results are integrated. Optimization must account for engine-specific costs. Early systems like Garlic and Disco [45]–[47] use site-specific wrappers to extract cost/cardinality estimates; [48] sends bids to the underlying system to acquire the cost of an operation; [49], [50] calibrate cost formulas for each system via execution of a large set of queries. More recently, XDB [26] adopts a mediator-free model leveraging SQL/MED for in-situ query execution. ML-based approaches also exist: [51] uses random forests to choose a single engine as the federated engine per query; Coral [52] optimizes join orders and engine selections using a RL model with deep Q-networks estimating the reward of each join action. However, these systems assume homogeneous relational data model, leaving cross-model challenges unaddressed.

Query Optimization in Polystore system. Polystores is a specialized FDBS that integrates engines with different data models. BigDAWG [7]–[9] groups engines into model-specific "islands", optimizes mainly within single islands by profiling operations on different engines within the island, and requires manual CASTs for cross-island queries, thus avoiding cross-model join optimization. ESTOCADA [10] focuses on query rewriting leveraging materialized views in different models, which focuses on a different optimization goal than ours. Wayang [11], [53]–[55] supports a variety of platforms including DBMSs and Spark through fine-grained API where fine-granular keyword/operator can be mapped to different platforms. Optimization selects platforms with the lowest cost per operation, but this approach does not preserve DBMS autonomy or utilize native query optimizers.

Cross-Model Join Optimization. A few works explore cross-model joins. [56] estimates selectivity for relational-tree joins. [14]–[16], [57] supports relational-graph hybrid queries by extending relational model for graph, enabling unified query planning. However, both assume a single RDBMS extended to support other data model—unlike our heterogeneous multi-engine setting.

X. Conclusions and Future Work

In this paper, we defined a class of cross graph-relational join queries as an important subset of cross-model cross-engine queries. We introduced a real-world and several semi-synthetic benchmarks to support research on query processing over heterogeneous data models. We presented MICRO, a lightweight middleware designed to efficiently execute such queries without incurring the overhead of an inflated plan space. Despite its simplicity, experimental results demonstrate that MICRO achieves strong performance on complex workloads. At the core of MICRO, the learning-to-rank optimizer CMLero consistently outperforms both rule-based and regression-based baselines, underscoring its effectiveness in selecting efficient execution plans in cross-model settings.

This work serves as an initial step toward addressing the broader challenges of federated query optimization across heterogeneous data models. Currently, the system supports one graph engine and one relational engine with multiple tables. Future work includes extending the architecture to support other cross-model operators, multiple graph and relational engines, and their interplay with indexes, all of which will require generalizing the design of CMLero. Beyond graph and relational models, real-world applications often involve additional data modalities, such as unstructured text served by specialized systems like Solr. Enabling efficient query processing across a wider spectrum of data models remains an important direction for future research.

REFERENCES

[1] V. Moustaka, A. Vakali, and L. G. Anthopoulos, "A systematic review for smart city data analytics," *ACM Computing Surveys (cSuR)*, vol. 51, no. 5, pp. 1–41, 2018.

[2] N. Mehta and A. Pandit, "Concurrence of big data analytics and healthcare: A systematic review," *International journal of medical informatics*, vol. 114, pp. 57–65, 2018.

[3] Y. Lu and L. Da Xu, "Internet of things (iot) cybersecurity research: A review of current research topics," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2103–2115, 2018.

[4] J. Mageto, "Big data analytics in sustainable supply chain management: A focus on manufacturing supply chains," *Sustainability*, vol. 13, no. 13, p. 7101, 2021.

[5] S. Rajendran, W. Pan, M. R. Sabuncu, Y. Chen, J. Zhou, and F. Wang, "Patchwork learning: A paradigm towards integrative analysis across diverse biomedical data sources," *arXiv preprint arXiv:2305.06217*, 2023.

[6] X. Shi, Z. Pan, and W. Miao, "Data integration in causal inference," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 15, no. 1, p. e1581, 2023.

[7] A. J. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska *et al.*, "A demonstration of the bigdawg polystore system," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, p. 1908, 2015.

[8] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The bigdawg polystore system," *ACM Sigmod Record*, vol. 44, no. 2, pp. 11–16, 2015.

[9] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker, "The bigdawg polystore system and architecture," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–6.

[10] R. Alotaibi, B. Cautis, A. Deutsch, M. Latrache, I. Manolescu, and Y. Yang, "Estocada: towards scalable polystore systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2949–2952, 2020.

[11] D. Agrawal, L. Ba, L. Berti-Equille, S. Chawla, A. Elmagarmid, H. Hammady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse *et al.*, "Rheem: Enabling multi-platform task execution," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 2069–2072.

[12] N. Houttekier, S. Van Hoeymissen, and C. Du Bois, "Sino-belgian research collaborations and chinese military power," *European Security*, vol. 34, no. 2, pp. 250–272, 2025.

[13] I. V. Nastasa, A.-R. Artamonov, S. S. Busnatu, D. G. Mincă, and O. Andronic, "From innovation to regulation: Insights from a bibliometric analysis of research patterns in medical data governance," in *Informatics*, vol. 12, no. 3. MDPI, 2025, p. 66.

[14] A. Jindal, S. Madden, M. Castellanos, and M. Hsu, "Graph analytics using vertica relational database," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 1191–1200.

[15] K. Zhao and J. X. Yu, "All-in-one: Graph processing in rdbmss revisited," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1165–1180.

[16] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, "Graphframes: an integrated api for mixing graph and relational queries," in *Proceedings of the fourth international workshop on graph data management experiences and systems*, 2016, pp. 1–8.

[17] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, "The ldbc social network benchmark: Interactive workload," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 619–630.

[18] J. Marton, G. Szárnyas, and D. Varró, "Formalising opencypher graph queries in relational algebra," in *Advances in Databases and Information Systems: 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings 21*. Springer, 2017, pp. 182–196.

[19] H. Thakkar, D. Punjani, S. Auer, and M.-E. Vidal, "Towards an integrated graph algebra for graph pattern matching with gremlin," in *Database and Expert Systems Applications: 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I 28*. Springer, 2017, pp. 81–91.

[20] J. Sun and G. Li, "An end-to-end learning-based cost estimator," *arXiv preprint arXiv:1906.02560*, 2019.

[21] J. K. Zhi Kang, Gaurav, S. Y. Tan, F. Cheng, S. Sun, and B. He, "Efficient deep learning pipelines for accurate cost estimations over large scale query workload," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1014–1022.

[22] B. Hilprecht and C. Binnig, "Zero-shot cost models for out-of-the-box learned cost prediction," *arXiv preprint arXiv:2201.00561*, 2022.

[23] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1275–1288.

[24] R. Zhu, W. Chen, B. Ding, X. Chen, A. Pfadler, Z. Wu, and J. Zhou, "Lero: A learning-to-rank query optimizer," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1466–1479, 2023.

[25] R. Angles, "The property graph database model." in *AMW*, 2018.

[26] H. Gavriilidis, K. Beedkar, J.-A. Quiané-Ruiz, and V. Markl, "In-situ cross-database query processing," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 2794–2807.

[27] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: a learned query optimizer," *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1705–1718, Jul. 2019. [Online]. Available: https://doi.org/10.14778/3342263.3342644

[28] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[29] J. Priem, H. Piwowar, and R. Orr, "Openalex: A fully-open index of scholarly works, authors, venues, institutions, and concepts," *arXiv preprint arXiv:2205.01833*, 2022.

[30] United States Patent and Trademark Office, "Uspto bulk data directory," 2023. [Online]. Available: https://data.uspto.gov/bulkdata/datasets

[31] A. G. Xiuwen Zheng, Arun Kumar, "Micro: A lightweight middleware for optimizing cross-store cross-model graph-relation joins [tech report]," https://github.com/xiz675/MICRO/blob/main/tr.pdf, 2025, accessed: Sep. 22, 2025.

[32] X. Zheng, A. Kumar, and A. Gupta, "Generating cross-model analytics workloads using llms," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 2024, pp. 4303–4307.

[33] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb *et al.*, "The design and operation of {CloudLab}," in *2019 USENIX annual technical conference (USENIX ATC 19)*, 2019, pp. 1–14.

[34] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica, "Neurocard: One cardinality estimator for all tables," *Proc. VLDB Endow.*, vol. 14, no. 1, pp. 61–73, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol14/p61-yang.pdf

[35] Z. Wu and A. Shaikhha, "Bayescard: A unified bayesian framework for cardinality estimation," *CoRR*, vol. abs/2012.14743, 2020. [Online]. Available: https://arxiv.org/abs/2012.14743

[36] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: Learn from data, not from queries!" *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 992–1005, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf

[37] R. Zhu, Z. Wu, Y. Han, K. Zeng, A. Pfadler, Z. Qian, J. Zhou, and B. Cui, "FLAT: fast, lightweight and accurate method for cardinality estimation," *Proc. VLDB Endow.*, vol. 14, no. 9, pp. 1489–1502, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf

[38] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. Narasayya, and S. Chaudhuri, "Selectivity estimation for range predicates using lightweight models," *Proceedings of the VLDB Endowment*, vol. 12, no. 9, pp. 1044–1057, 2019.

[39] Z. Wu, P. Yang, P. Yu, R. Zhu, Y. Han, Y. Li, D. Lian, K. Zeng, and J. Zhou, "A unified transferable model for ml-enhanced dbms," *Conference on Innovative Data Systems Research*, 2022.

[40] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," in *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[41] X. Wang, C. Qu, W. Wu, J. Wang, and Q. Zhou, "Are we ready for learned cardinality estimation?" *Proc. VLDB Endow.*, vol. 14, no. 9, p. 1640–1654, May 2021. [Online]. Available: https://doi.org/10.14778/3461535.3461552

[42] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui, "Cardinality estimation in dbms: a comprehensive benchmark evaluation," *Proc. VLDB Endow.*, vol. 15, no. 4, p. 752–765, Dec. 2021. [Online]. Available: https://doi.org/10.14778/3503585.3503586

[43] J. Zhang, C. Zhang, G. Li, and C. Chai, "Autoce: An accurate and efficient model advisor for learned cardinality estimation," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 2621–2633.

[44] Z. Yang, W.-L. Chiang, S. Luan, G. Mittal, M. Luo, and I. Stoica, "Balsa: Learning a query optimizer without expert demonstrations," ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 931–944. [Online]. Available: https://doi.org/10.1145/3514221.3517885

[45] M. T. Roth and P. M. Schwarz, "Don't scrap it, wrap it! a wrapper architecture for legacy data sources." in *VLDB*, vol. 97, 1997, pp. 25–29.

[46] L. Haas, D. Kossmann, E. Wimmers, and J. Yang, "Optimizing queries across diverse data sources," 1997.

[47] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling access to heterogeneous data sources with disco," *IEEE Transactions on knowledge and Data Engineering*, vol. 10, no. 5, pp. 808–823, 1998.

[48] A. Deshpande and J. M. Hellerstein, "Decoupled query optimization for federated database systems," in *Proceedings 18th International Conference on Data Engineering*. IEEE, 2002, pp. 716–727.

[49] W. Du, R. Krishnamurthy, and M.-C. Shan, "Query optimization in a heterogeneous dbms," in *VLDB*, vol. 92, 1992, pp. 277–291.

[50] G. Gardarin, F. Sha, and Z.-H. Tang, "Calibrating the query optimizer cost model of iro-db, an object-oriented federated database system." in *VLDB*, vol. 96. Citeseer, 1996, pp. 3–6.

[51] L. Xu, R. L. Cole, and D. Ting, "Learning to optimize federated queries," in *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2019, pp. 1–7.

[52] R. Gu, Y. Zhang, L. Yin, L. Song, W. Huang, C. Yuan, Z. Wang, G. Zhu, and Y. Huang, "Coral: federated query join order optimization based on deep reinforcement learning," *World Wide Web*, vol. 26, no. 5, pp. 3093–3118, 2023.

[53] K. Beedkar, B. Contreras-Rojas, H. Gavriilidis, Z. Kaoudi, V. Markl, R. Pardo-Meza, and J.-A. Quiané-Ruiz, "Apache wayang: A unified data analytics framework," *ACM SIGMOD Record*, vol. 52, no. 3, pp. 30–35, 2023.

[54] S. Kruse, Z. Kaoudi, B. Contreras-Rojas, S. Chawla, F. Naumann, and J.-A. Quiané-Ruiz, "Rheemix in the data jungle: a cost-based optimizer for cross-platform systems," *The VLDB Journal*, vol. 29, pp. 1287–1310, 2020.

[55] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani *et al.*, "Rheem: enabling cross-platform data processing: may the big data be with you!" *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1414–1427, 2018.

[56] C. Zhang and J. Lu, "Selectivity estimation for relation-tree joins," in *Proceedings of the 32nd International Conference on Scientific and Statistical Database Management*, 2020, pp. 1–12.

[57] W. Fu, "Joining entities across relation and graph with a unified model," *arXiv preprint arXiv:2401.18019*, 2024.