IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

Learning Execution Plan Embeddings for Multi-Dimensional Query Resource Prediction

MAHENDRAN VASAGAM[1], (Member, IEEE), ASHOK KUMAR[1], (Member, IEEE), ANSHUL GARG[1], (Member, IEEE)
[1]Independent Researcher

Corresponding author: Mahendran Vasagam (e-mail: mahendran.vasagam@gmail.com).

ABSTRACT Multi-cluster query engines are common in production, but routing queries to appropriately-sized clusters remains an unsolved problem. Current approaches like round-robin, hash-based routing, or manual rules waste resources when simple queries land on oversized clusters and cause failures when complex queries hit undersized ones. Production systems experience resource mismatch failures, requiring manual workarounds such as migrating problematic queries to separate engines. We address this with a system for intelligent cluster sizing using learned execution plan embeddings to predict query resource requirements. Instead of analyzing query text, we vectorize execution plan structure and search for similar historical queries, predicting memory, Central Processing Unit (CPU) and runtime based on actual past performance rather than unreliable optimizer estimates. We employ a tiered inference strategy that supports fast, feature-based vectors (10 basic dimensions; 13–23 ms overhead), gradient boosting with enhanced features (24 dimensions; 76.5% accuracy), and graph neural network embeddings for complex queries (78.1% accuracy). An adaptive classifier maps predictions to cluster sizes using thresholds that adjust based on observed utilization patterns. Unfamiliar queries are routed conservatively, and the system automatically improves from observed outcomes. Empirical validation on 2,954 queries from a Trino deployment shows 98.8–99.8% within-one-size accuracy and 74.8–76.5% exact-match routing. In benchmark workloads, our approach reduces out-of-memory failures by 82% and infrastructure costs by 33%, with less than 51 ms of routing overhead. Gradient boosting achieves 76.5% accuracy, while graph neural networks reach 78.1% for complex join structures.

INDEX TERMS Approximate Nearest Neighbor Search, Distributed Query Engines, Graph Neural Networks, Learned Cost Models, Multi-Cluster Routing, Query Performance Prediction, Vector Embeddings

I. INTRODUCTION

MODERN data analytics platforms face a fundamental challenge: how to efficiently route queries with vastly heterogeneous resource requirements (spanning 0.71 to 2,558 GB) across multiple compute clusters. Current approaches rely on naive load balancing or manual routing rules, resulting in significant resource waste and query failures. This paper addresses this challenge through predictive routing based on machine learning using execution plan embeddings.

A. OUR APPROACH: MACHINE LEARNING (ML)-BASED PREDICTIVE ROUTING VIA EXECUTION PLAN EMBEDDINGS

Our approach is based on the observation that queries with similar execution plan structures consume similar resources, even when the query text differs significantly. We propose a system that vectorizes execution plans and retrieves similar historical queries to predict resource usage based on actual past performance.

These queries have different text (different tables, columns, dates) but similar execution plans (single table scan → filter → aggregation). They will consume similar memory and should be routed to the same cluster size (S).

1) Our System Architecture

1) Query Plan Analyzer: Generate execution plan using database EXPLAIN command without executing the query.
2) Multi-Embodiment Vectorization: We evaluate three approaches:
   - Feature-Based (Explicit): 26 structural features (joins, scans, cardinality). Fast (1–2ms).
   - Gradient Boosting (XGBoost): Applied to explicit features for nonlinear regression.

- Graph Neural Network (GNN): Treats plans as Directed Acyclic Graphs (DAGs) to learn 512-dimensional embeddings. Captures deep topological patterns.

3) Vector Database with k-Nearest Neighbor (k-NN) Search: Store historical query execution records with vector embeddings, actual resource usage (memory, CPU, time), cluster used, and success/failure status. Use Hierarchical Navigable Small World (HNSW) indexing for submillisecond approximate nearest neighbor search in high-dimensional space [29].
4) Similarity Search and Resource Prediction: For an incoming query, retrieve 10 most similar historical queries using cosine similarity. Predict memory (conservative: 90th percentile weighted by similarity), CPU (weighted average), and time (weighted average) from historical performance. Compute a confidence score based on similarity tightness and variance.
5) Adaptive Threshold Classification: Map predicted resources to discrete cluster sizes (S, M, L) using dynamic capacity boundaries. Select cluster with capacity exceeding predicted requirements.
6) Explainability (Case-Based Reasoning): The same k-NN retrieval used for prediction provides an explanation of each routing decision by surfacing the most similar historical queries and their observed resource usage, enabling operators to audit and debug routing outcomes.
7) Continuous Learning: Unfamiliar queries are routed conservatively (escalate when uncertain). After each run, actual metrics are stored and the models are retrained weekly. The system automatically improves using production workloads without manual intervention.

B. CONTRIBUTIONS

1) Multiembodiment vectorization with tiered inference: Feature-based (74.8%, 13–23ms), XGBoost (76.5%), and GNN (78.1%, 32–51ms) approaches that enable tunable latency-accuracy tradeoffs for production deployment.
2) Historical k-NN predictions outperform optimizer estimates: 78.1% vs. 36.4% accuracy by learning from actual execution outcomes rather than unreliable cost models, with explainable case-based reasoning.
3) Risk-sensitive routing under asymmetric loss: Percentile-based memory prediction and uncertainty-aware escalation prioritizing OOM avoidance. The system improves automatically using empirically observed outcomes.
4) Comprehensive evaluation: 78% routing accuracy, 82% reduction in OOM failures, 33% cost savings, validated on 2,954 queries.

II. BACKGROUND

A. THE MULTI-CLUSTER RESOURCE MISMATCH PROBLEM

Modern data analytics platforms deploy multiple compute clusters to address scaling requirements and support concurrent query execution on petabyte-scale data lakes. However, most production deployments today use homogeneous cluster sizing: multiple identically-sized clusters combined with naive load balancing (round-robin or hash-based routing). This creates severe resource mismatch problems:

Problem 1: Homogeneous Clusters + Naive Routing

When all clusters have identical resources but queries have vastly different requirements (spanning from 0.71 GB to 2,558 GB in our analysis), organizations face a no-win tradeoff:

- If clusters are sized for large queries (e.g., 12.8 TB each): Small queries (2 GB) waste 99.98% of allocated resources.
- If clusters are sized conservatively (e.g., 1.6 TB each): Large queries (220 GB) fail with out-of-memory (OOM) errors.

Either choice results in significant cost waste or query failures. Homogeneous sizing cannot efficiently serve heterogeneous workloads.

Problem 2: Heterogeneous Clusters + Manual Routing

Some sophisticated organizations manually provision differently-sized clusters to address Problem 1. A typical heterogeneous deployment comprises:

- Small cluster (S): 10 nodes $\times$ 160 GiB = 1.6 TB total memory, for lightweight queries.
- Medium cluster (M): 40 nodes $\times$ 160 GiB = 6.4 TB, for moderate analytics.
- Large cluster (L): 80 nodes $\times$ 160 GiB = 12.8 TB, for heavy workloads.

However, even with heterogeneous clusters, existing routing proxies (e.g., Trino Gateway [25]) employ naive routing rules (round-robin, hash-based, or simple heuristics like "if num_joins $>$ 5 $\rightarrow$ L cluster") that still cause resource mismatches because they cannot accurately predict query resource requirements.

Problem 3: MultiEngine Clusters + Static Routing

To work around Problems 1 and 2, some organizations deploy multiple query engines (e.g., Trino for interactive queries, Spark for batch workloads) and route traffic via proxy services using static mappings based on engine type, user tags, or application source. In a deployment we analyzed, approximately 50% of large queries were manually migrated to separate Spark clusters after repeated OOM failures on the primary Trino engine. Although this reduces observed failure rates (4.3% overall, estimated 8–15% before workarounds), it introduces:

- Operational overhead: Users must predict engine capacity before execution and manually tag queries.
- Static brittleness: Routing rules tied to the source application or user group do not adapt to query complexity.

- Engine mismatch: Interactive queries sent to Spark suffer unnecessary latency; batch queries on Trino waste interactive capacity.

Query resource requirements span 5 orders of magnitude, from 0.71 GB (median) to 2,558 GB (maximum), demonstrating heterogeneity that currently requires human judgment to manage.

1) Our Scope

We address Problems 1 and 2 by enabling intelligent routing for heterogeneous multicluster deployments of the same engine. Our system makes heterogeneous cluster architectures practical by replacing naive routing with learned query-aware routing that selects the right-sized cluster based on predicted resource requirements, eliminating manual size selection and enabling optimal resource utilization. Our approach could extend to Problem 3 (cross-engine routing); however, we leave this for future work (see Section VI).

2) Practical Impact

Distributed Structured Query Language (SQL) engines operate over heterogeneous table formats across data lakes and serve a broad user base (data scientists, product analysts, marketing, sales, finance, and engineering teams). These users need to analyze large datasets without understanding how to provision the cluster or risking query failures. Our approach abstracts this technical complexity by allowing users to submit queries while the system automatically routes them to appropriately sized clusters, reducing failures and resource costs and enabling self-service analytics at scale.

3) Cloud Cost Efficiency

The economic impact of intelligent routing is amplified in cloud environments where distributed SQL compute engines are increasingly deployed as managed services provided by several cloud providers, including Amazon Web Services (AWS) (e.g., Athena, EMR), Microsoft Azure (e.g., Synapse Analytics, HDInsight), and Google Cloud Platform (e.g., Dataproc, BigQuery), or on cloud infrastructure by major technology companies. In these settings, compute resources are billed by usage time and cluster size, making resource mismatch directly translate to monetary waste. For example, routing a 2 GB query to a 128-node cluster (12.8 TB capacity) on AWS at $2.00/node/hour wastes $256/hour when a 10-node cluster ($20/hour) would suffice, a 12.8x cost multiplier. Our 33% cost reduction (Section V.E) translates to hundreds of thousands of dollars in annual cloud savings for organizations running 10,000+ daily queries. Furthermore, cloud auto-scaling incurs 5–10 minute provisioning delays, making predictive right-sizing essential for interactive query performance. By predicting resource requirements before execution, our system enables organizations to provision appropriately-sized clusters upfront, eliminating both over-provisioning waste and under-provisioning failures without relying on slow reactive scaling.

4) Scope Limitation

Our resource predictions assume isolated query execution (singlequery historical measurements). We do not model query interference under concurrent execution; we assume admission control enforces cluster concurrency limits. Our focus is capacity matching (query size → cluster size), not interference-aware scheduling. Section V.A discusses this limitation and cites Zhou et al. [20] for concurrency-aware extensions.

5) Current Practice: Naive Routing Failures

Even with manually-created heterogeneous clusters, naive routing approaches distribute queries without understanding resource needs. This creates two critical failure modes:

6) Failure Mode 1: Resource Waste

Consider a simple lookup query: SELECT customer_name, email FROM customers WHERE customer_id = 'C12345'. This query needs only 2 GB of memory but is routed to an L cluster (12.8 TB capacity) and uses only 0.016% of resources. The cost is $1.00 rather than $0.02, resulting in 50x waste.

7) Failure Mode 2: Query Failures

A complex multitable query that scans 100+ partitions and 1B+ rows can require 200+ GB of memory. When naively routed to a small cluster, intermediate state construction can exceed available memory → OOM error → failure → retry overhead.

Naive routing causes significant infrastructure waste, high query failure rates due to OOM errors, and substantial engineering effort to maintain manual routing rules.

Query resource prediction before execution is challenging for three reasons: First, database optimizer cost estimates are often 10–100x inaccurate due to missing statistics and incorrect selectivity assumptions, particularly in data lake environments with rapidly evolving schemas. Second, query text alone is insufficient: `SELECT COUNT(*)` on 1 GB vs. 1 TB tables has identical text but 1000x different resource needs. Third, workloads evolve continuously, causing rules that work today to break as schemas and data volumes change.

B. LIMITATIONS OF EXISTING APPROACHES

1) Manual Routing Rules

Engineers write heuristic rules such as: if query has join count greater than 5, route to large cluster; if table count greater than 3, route to medium cluster; if groupby cardinality greater than 100000, route to large cluster.

2) Problems

1) Brittle: Rules break when schemas change (new tables, renamed columns).
2) Incomplete: Cannot anticipate all edge cases (e.g., 2 joins might need L cluster if tables are huge).
3) Maintenance Burden: Requires continuous tuning as workload evolves.

4) Static: Rules do not learn from actual query performance.

3) Database Cost Estimates

Systems like Trino [1] provide EXPLAIN output with estimated memory and CPU requirements. However, optimizer cost estimates are notoriously unreliable for memory prediction. Our analysis confirmed estimates can be off by orders of magnitude due to stale table statistics, incorrect selectivity assumptions, and missing correlation information, making optimizer-based routing ineffective for heterogeneous cluster sizing.

Auto-Scaling: Reactive auto-scaling is often too slow (between 5 and 10 min) for interactive queries and requires provisioning maximum capacity upfront to tune and resource an individual cluster to support larger queries.

4) Problems

1) Reactive: Scales after problems occur (too late for failing queries).
2) Expensive: Must provision maximum capacity upfront.
3) Slow: Scaling takes 5–10 minutes (unacceptable for interactive queries).
4) Configuration conflicts: One cluster cannot simultaneously optimize for heterogeneous workloads. Interactive queries benefit from disabled fault-tolerant execution (FTE) for speed while batch or larger interactive queries require FTE for reliability; small queries need low per-query memory limits for concurrency while large queries need high limits for complex joins; these settings require cluster restarts and cannot be applied per-query.

III. RELATED WORK AND RECENT FINDINGS

A. QUERY ROUTING AND WORKLOAD MANAGEMENT

Trino Gateway [25] serves as the industry-standard proxy for multicluster Trino deployments, providing routing capabilities through configurable rules. Users can specify routing based on user groups, query source, or custom predicates. However, Trino Gateway lacks query-aware intelligence: rules are hand-crafted and updated over time, and the system cannot predict resource requirements or learn from query performance.

Amazon Redshift Workload Management (WLM) [2] enables queue-based query prioritization within a single cluster, allowing administrators to allocate resources (memory slots, concurrency) to different query classes. Although WLM provides workload isolation within a cluster, it does not address crosscluster routing and requires manual queue configuration.

Google BigQuery [3] employs dynamic slot allocation, automatically scaling query resource consumption based on demand. However, BigQuery operates as a proprietary managed service without multicluster routing abstractions, and its internal resource allocation mechanisms remain unpublished.

1) Limitation

Existing routing systems require manual configuration and do not predict resource requirements before execution using learned models.

B. QUERY PERFORMANCE PREDICTION

Cardinality Estimation with Deep Learning: MSCN (Multi-Set Convolutional Network) [4] uses neural networks to predict intermediate result cardinalities, achieving higher accuracy than traditional optimizer estimates. However, MSCN focuses on single-table or join cardinality rather than end-to-end resource consumption (memory, CPU, time) required for routing decisions.

Neo: Learned Query Optimizer [5] applies deep reinforcement learning to select optimal join orders and physical operators within a single database engine. Neo learns from query execution feedback but operates within one cluster rather than routing across heterogeneous clusters.

Bao: Bandit Optimizer [6] uses Thompson sampling to explore query execution plans, learning which plans perform best through trial-and-error. Like Neo, Bao optimizes within a single engine rather than addressing cross-cluster routing.

Query Performance Prediction for Progress Indicators: Marcus et al. [7] predict query execution time to display progress bars, using tree-structured neural networks over execution plans. This work predicts time but not memory (critical for OOM prevention) and does not address routing decisions.

Classical ML for QPP and Resource Consumption: Akdere et al. [26] demonstrate learning-based performance prediction to support scheduling and resource management decisions, while Li et al. [27] estimate query resource consumption for tasks such as admission control and scheduling.

1) Limitation

Prior ML work predicts specific performance aspects (cardinality, time, plan selection) within a single database system rather than multidimensional resource prediction (memory, CPU, time) for cross-cluster routing.

C. QUERY SIMILARITY AND CACHING

Traditional approaches match identical queries using text hashing [8] or query templates with parameters [9], but do not capture structural similarity among different query shapes. Query2Vec [10] embeds SQL text tokens, missing execution plan structure such as join cardinalities and data volumes. Our Distinction: We vectorize execution plan structure (operators, data flow, cardinalities) rather than query text, enabling similarity matching across syntactically different but structurally similar queries.

D. VECTOR SIMILARITY SEARCH IN DATABASES

Similarity search systems such as FAISS [11] and Annoy [12] provide high performance approximate nearest neighbor search infrastructure. Recent work on neural query representations [13], [28] focuses on optimization within a single system rather than multicluster routing.

E. GRAPH NEURAL NETWORKS FOR QUERY PLAN ANALYSIS

Query execution plans are directed acyclic graphs in which operators (scan, join, aggregate) form nodes and data flow forms edges. This structure makes them natural candidates for graph neural network approaches. Kipf & Welling [14] established the GCN framework using message-passing between connected nodes, while Veličković et al. [15] introduced attention mechanisms to adaptively weight neighbor contributions.

1) Application to Query Plans

Sun & Li [16] first applied GNNs to query execution plans for cardinality and cost estimation, representing plan operators as nodes with features (operator type, estimated rows, estimated cost) and learning embeddings that capture plan topology. Zhou et al. [20] extended this to predict query latency in concurrent workloads, modeling interference patterns between concurrent queries. Marcus et al. [7] (Bao) employ Tree Convolutional Neural Networks on plans to guide query optimizers toward better join orders.

2) Classical Baselines

Duggan et al. [23] proposed regression-based performance prediction for concurrent database workloads, establishing early foundations for analytical models of query resource consumption. Negi et al. [24] (Flow-Loss) developed asymmetric loss functions for cardinality estimation, prioritizing underestimation penalties, a philosophy that aligns with our conservative memory prediction strategy (90th percentile) to avoid OOM failures.

Our Distinction - System-Level Novelty: We acknowledge that GCN architectures for query plans are standard and established [18], [24]. Our contribution is not inventing plan-to-vector embeddings, but rather their novel application to heterogeneous cluster routing: Prior work targets query optimization (cardinality, join ordering) within one cluster, while we address infrastructure provisioning across heterogeneous resources. Our strongest novelty is the tiered inference strategy (Features $\rightarrow$ XGBoost $\rightarrow$ GNN), which enables production deployments to trade latency (13–51ms) for accuracy (74.8%–78.1%) dynamically. We predict multi-dimensional resources (memory, CPU, time) with asymmetric loss prioritizing memory to prevent OOM failures, unlike single-metric predictions in prior work [18], [24], [27].

F. GAP IN LITERATURE

To the best of our knowledge, no existing work combines:

1) Vector embeddings of execution plan structure for similarity-based routing
2) Multi-embodiment vectorization (features + GNNs) balancing speed and accuracy
3) k-NN search in vector space for historical performance-based resource prediction
4) Fully automated uncertainty-aware routing with automatic feedback from execution outcomes
5) Comprehensive evaluation demonstrating routing accuracy, failure reduction, and cost savings

Our work fills this gap by presenting a complete ML-based query routing system deployed as a practical extension to existing distributed query engines.

IV. SYSTEM ARCHITECTURE

A. OVERVIEW AND SYSTEM INTEGRATION

Our query routing system integrates with existing distributed SQL compute engine deployments (e.g., Trino [1], in the Presto family [30]) as a proxy layer between clients and clusters. Fig. 1 illustrates the high-level architecture.

1) Operational Flow

1) Client submits SQL query to Query Router Proxy
2) Query Router Proxy invokes smart routing module (bypassed for pass-through routing)
3) Query plan analyzer generates execution plan via EXPLAIN (8–12ms)
4) Vectorization engine converts plan to fixed-dimensional vector (1–30ms depending on embodiment)
5) Similarity search retrieves k-nearest historical queries (3–7ms via HNSW index)
6) Resource predictor computes weighted estimates from historical performance ($<$1ms)
7) Classifier maps predictions to cluster size (S/M/L)
8) Router selects available cluster of appropriate size and forwards query
9) Query executes on selected cluster
10) Feedback collector monitors execution, stores actual metrics (memory, CPU, time, execution cost) in vector database
11) Weekly retraining incorporates new data for continuous improvement and cost-efficient cluster prediction

2) Design Principles

- Non-invasive: No modifications to query engines; operates as proxy
- Low latency: Routing decision completes in 13–51ms (negligible vs. query execution in s)
- Fallback: If smart routing fails or times out, fall back to round-robin
- Transparent: Clients unaware of routing; maintains standard SQL wire protocol

B. QUERY PLAN ANALYZER

1) Input

SQL query text

2) Output

Structured execution plan representation

3) Implementation

We invoke Trino's EXPLAIN command to generate a detailed execution plan without executing the query. For example: EXPLAIN (TYPE DISTRIBUTED, FORMAT JSON) SELECT

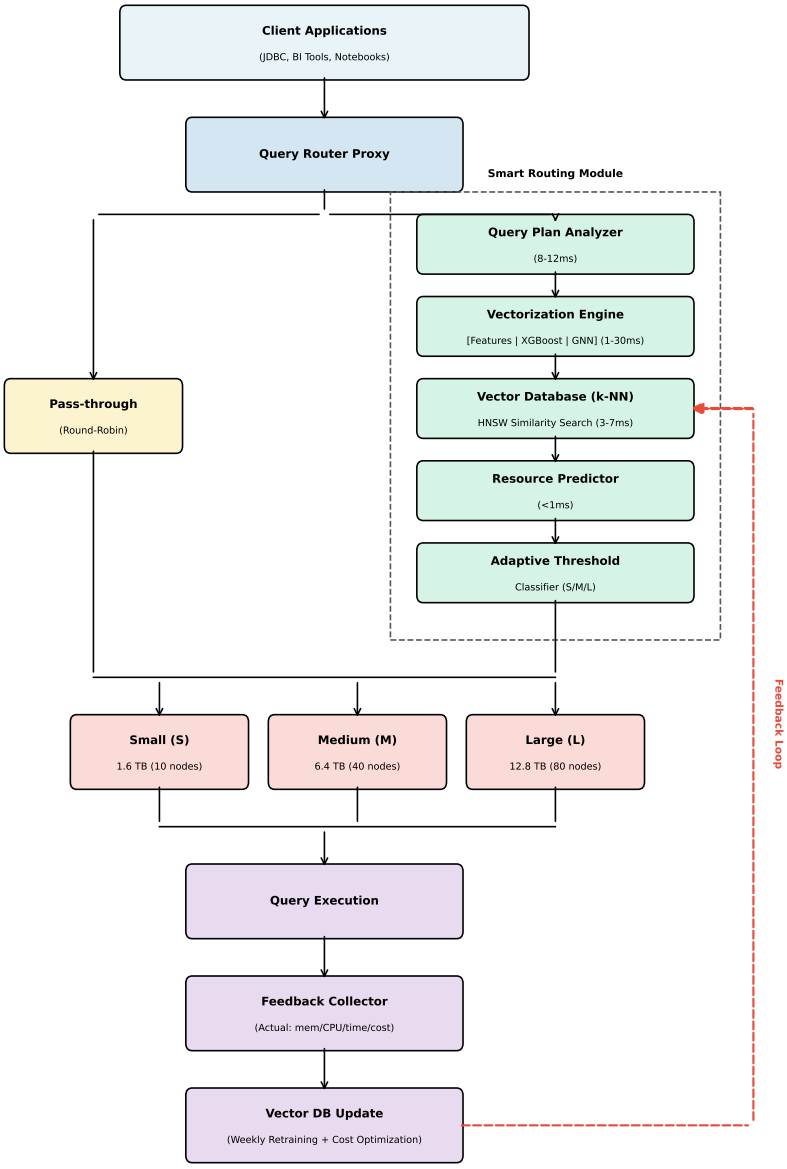

FIGURE 1. System Architecture Diagram (end-to-end flow). The routing proxy intercepts queries, vectorizes execution plans, and retrieves similar historical queries to predict resource requirements before routing to the appropriate cluster size.

c.customer_name, SUM(o.amount) as total FROM customers c JOIN orders o ON c.customer_id = o.customer_id WHERE o.order_date >= DATE '2024-01-01' GROUP BY c.customer_name.

4) Actual Trino EXPLAIN JSON Structure

Trino's `EXPLAIN (TYPE DISTRIBUTED, FORMAT JSON)` produces a nested tree structure where each stage is keyed by its ID ("0", "1", etc.), and operators form a recursive tree via the `children` array. The JSON output contains key information including stage IDs, operator names (Output, Aggregate, InnerJoin, TableScan, etc.), column descriptors, estimates for output row count and data size, cost metrics (CPU, memory, network), and parent-child relationships through nested children arrays showing data flow dependencies.

5) Key Information Extracted from Trino JSON

- Structural: Stage count (top-level keys "0", "1", ...), operator types (`name` field: TableScan, InnerJoin, Aggregate, etc.), tree depth via recursive `children` arrays
- Data Flow: Parent-child relationships from `children` arrays; cross-stage dependencies from `RemoteSource` operators

- Cardinalities: `estimates[].outputRowCount` per operator
- Data Volumes: `estimates[].outputSizeInBytes` for bytes scanned/produced
- Join Characteristics: `descriptor.type` (INNER, LEFT, FULL), `descriptor.distribution` (BCAST, PART), `descriptor.criteria` (join keys)
- Aggregation Patterns: `descriptor.keys` for GROUP BY columns, `descriptor.type` (PARTIAL, FINAL)
- Resource Estimates: `estimates[].memoryCost`, `estimates[].cpuCost`, `estimates[].networkCost` (used as auxiliary features, not primary predictors due to optimizer inaccuracy)

6) Parsing

We implement a recursive JSON parser that traverses the nested tree structure, counting operators by `name` pattern matching, extracting `estimates` arrays, and building the operator DAG from `children` relationships.

C. VECTORIZATION ENGINE: FEATURE-BASED EMBODIMENT

1) Motivation

Feature engineering provides a fast, interpretable, and deterministic approach to vectorization that requires no training data and executes in milliseconds.

Feature Extraction (10 Basic + 14 Enhanced = 24 Total Dimensions):

We define two feature sets: a basic set (10 features) for low latency inference and an enhanced set (24 features) that adds graph-structural features for higher accuracy.

Basic Features (10) - Fast Path (74.8% accuracy):

- `num_tables`: Count of TableScan operators
- `num_columns`: Columns accessed in scans
- `is_partitioned`: Partition key detected
- `has_join`: Join operator presence
- `has_aggregation`: Aggregate operator presence
- `has_window`: Window operator presence
- `has_subquery`: SemiJoin (subquery) presence
- `num_stages`: Query plan fragment count
- `partition_count`: Number of partitions to scan (from metastore)
- `log_estimated_bytes`: p_cnt × avg_b/part

Enhanced Features (14 additional) - Higher Accuracy (76.5% total with 24 features):

Join Characteristics (3):

- `join_types_encoded`: Weighted sum of join types (INNER=1.0, LEFT=1.5, FULL=2.0, CROSS=5.0)
- `join_distribution_encoded`: Distribution strategy (BROADCAST=1.0, PARTITIONED=2.0, REPLICATED=3.0)
- `max_join_fanout`: Maximum estimated cardinality ratio (output_rows / max(input_rows))

Data Volume Features (4):

- `log_total_input_rows`: $\log_{10}(1 + \Sigma$ estimated rows scanned across all table scans)
- `log_max_stage_rows`: $\log_{10}(1 + \max$ estimated rows across all stages)
- `log_output_rows`: $\log_{10}(1+$ final output row estimate)
- `log_data_shuffle_size_gb`: $\log_{10}(1+$ estimated network transfer for shuffles / 10^9)

Graph-Structural Features (4):

- `estimated_depth`: DAG depth estimate
- `depth_stage_ratio`: Graph structure metric (depth / num_stages)
- `log_max_partitions`: Partition skew indicator
- `log_data_flow_complexity`: Edge patterns in execution DAG

Complexity Metrics (3):

- `log_aggregation_cardinality`: $\log_{10}(1+$ estimated distinct count for GROUP BY keys)
- `filter_selectivity`: Average selectivity of filters (estimated_output / estimated_input)
- `join_complexity_score`: Composite metric = Σ (join_fanout $\times \log_{10}$(input_size))

2) Normalization

Features are normalized using pre-computed statistics (mean, std) from training data to ensure zero mean and unit variance, improving similarity search quality.

Robustness to Missing Statistics (Cold Start): Our 24 features exhibit varying sensitivity to missing or stale table statistics:

- Statistics-Independent (10 features): Structural features (num_joins, num_aggregations, num_filters, etc.) and join characteristics (join_types_encoded, join_distribution_encoded) are derived purely from query plan topology and remain accurate even when table statistics are missing. These provide a robustness floor of $\sim$38% of feature importance.
- Statistics-Dependent (14 features): Data volume features (log_total_input_rows, log_max_stage_rows) and complexity metrics (log_aggregation_cardinality, join_complexity_score) rely on optimizer cardinality estimates. When statistics are missing, these features degrade gracefully through two mechanisms: (1) the $\log_{10}(1 + x)$ transformation prevents extreme values from corrupted estimates, and (2) k-NN similarity search tolerates noisy features by finding queries with similar *structural* patterns even when volumes differ.

3) Cold Start Strategy

For queries on newly-added tables with no metastore statistics, partition_count defaults to 0 and log_estimated_bytes becomes 0 (line 13–14 in Algorithm 1). The system then applies conservative routing: (1) structural features (num_tables, has_join, etc.) remain valid since they derive from the execution plan, (2) the uncertainty score increases

due to low k-NN similarity (volume features mismatch historical queries), and (3) the query routes to the next-larger cluster size as a safety margin (Section IV.C). After first execution, actual metrics are recorded in the vector database, enabling accurate prediction for subsequent queries on the new table.

Data Skew Handling: The basic feature log_estimated_bytes uses average bytes per partition, which can underestimate memory for skewed partitions (e.g., a customer_id with 100x more rows). The enhanced feature set addresses this through log_max_partitions, which captures partition size variance. Additionally, the 90th percentile memory prediction (Section V.A) provides a buffer against skew-induced underestimates, and k-NN retrieval from similar historical queries captures skew patterns observed in past executions.

Algorithm 1 presents the feature extraction procedure, validated on 2,954 queries achieving 74.8% exact match accuracy.

Algorithm 1 Feature Extraction from Execution Plan

Input: Execution plan P obtained via `EXPLAIN (TYPE DISTRIBUTED, FORMAT JSON)`
Output: Feature vector $\mathbf{f} \in \mathbb{R}^{10}$
Initialize $f.num\_tables \leftarrow 0, f.num\_columns \leftarrow 0, f.num\_stages \leftarrow |P.stages|$
Initialize boolean features $\leftarrow$ false (has_join, has_aggregation, has_window, has_subquery, is_partitioned)
for each node n in P (depth-first traversal) **do**
  if $n.type$ contains "Join" **then**
    $f.has\_join \leftarrow$ true
  end if
  if $n.type$ contains "Aggregate" **then**
    $f.has\_aggregation \leftarrow$ true
  end if
  if $n.type$ contains "Window" **then**
    $f.has\_window \leftarrow$ true
  end if
  if $n.type$ = "SemiJoin" **then**
    $f.has\_subquery \leftarrow$ true
  end if
  if $n.type$ contains "Scan" **then**
    $f.num\_tables \leftarrow f.num\_tables + 1$
    $f.num\_columns \leftarrow f.num\_columns + |n.columns|$
    if $n.details$ contains "PARTITION_KEY" **then**
      $f.is\_partitioned \leftarrow$ true
    end if
  end if
end for
$f.partition\_count \leftarrow QueryMetastore(P.tables)$
$f.log\_estimated\_bytes \leftarrow \log_{10}(1 + f.partition\_count \times \beta)$
return Normalize(f)

Feature Summary (10 Dimensions):

This approach offers several practical advantages: feature extraction completes in 1–2ms, each dimension has clear semantic meaning for interpretability, no training data is required for initial deployment, the same plan always produces identical vectors, storage requires only 40 bytes per vector, and empirical validation confirms 74.8% accuracy on 2,954 queries.

TABLE 1. Feature Summary

Feature	Source	Description
num_tables	EXPLAIN	Count of TableScan nodes
num_columns	EXPLAIN	Columns accessed in scans
is_partitioned	EXPLAIN	Partition key detected
has_join	EXPLAIN	Join operator presence
has_aggregation	EXPLAIN	Aggregate operator presence
has_window	EXPLAIN	Window operator presence
has_subquery	EXPLAIN	SemiJoin (subquery) presence
num_stages	EXPLAIN	Query plan fragment count
partition_count	Metastore	Number of partitions to scan
log_estimated_bytes	Computed	p_cnt × avg_b/part

D. XGBOOST GRADIENT BOOSTING BASELINE

1) Motivation

Before investing in complex neural architectures, we establish a strong ML baseline using gradient boosting, a proven technique for tabular data that often outperforms deep learning on structured feature sets. XGBoost [22] is widely adopted in production ML systems for its speed, accuracy, and interpretability.

2) How XGBoost Works for Cluster Size Prediction

XGBoost builds an ensemble of decision trees sequentially, where each new tree corrects errors made by previous trees. Unlike k-NN similarity search (which retrieves historical queries), XGBoost directly predicts resource requirements from the extracted features (10 basic or 24 enhanced).

3) Training and Inference

Algorithm 2 presents the XGBoost training and prediction procedure.

4) How Gradient Boosting Works

Each tree in the ensemble sequentially corrects errors from previous trees through the following process:

1) Tree 1: Makes initial predictions (e.g., predicts all queries need 50 GB). Error: Some queries actually need 5 GB, others need 200 GB.
2) Tree 2: Learns to predict the residual errors from Tree 1. Learns: "Queries with num_joins $<$ 2 need -45 GB correction". Learns: "Queries with log_input_size $>$ 3 need +150 GB correction".
3) Trees 3–100: Continue correcting residual errors, each tree focusing on patterns missed by previous trees
4) Final Prediction: Sum of all tree predictions

Algorithm 2 XGBoost Training and Prediction

Training Phase:
Input: Historical queries $H = \{(P_1, m_1), \ldots, (P_n, m_n)\}$ where P_i is execution plan, m_i is actual memory (GB)
Output: Trained model M, feature scaler S
$X \leftarrow [FeatureExtract(P_i)$ for each $(P_i, m_i) \in H]$
$y \leftarrow [\log_{10}(m_i + \varepsilon)$ for each $m_i]$
$S \leftarrow FitStandardScaler(X)$
$X' \leftarrow S.Transform(X)$
$M \leftarrow XGBoost(n\_trees = 100, max\_depth = 6, \eta = 0.1)$
$M.Fit(X', y)$
return M, S
Inference Phase:
Input: Execution plan P, model M, scaler S
Output: Predicted cluster size $\in \{S, M, L\}$
$f \leftarrow FeatureExtract(P)$
$f' \leftarrow S.Transform(f)$
$\hat{y} \leftarrow M.Predict(f')$
$\hat{m} \leftarrow 10^{\hat{y}} - \varepsilon$
return $ClassifySize(\hat{m})$

```
final_prediction = tree1 + tree2
+ tree3 + ... + tree100
```

Key Advantages vs. Feature-based k-NN:

1) Non-linear pattern capture: XGBoost captures feature interactions (e.g., "3 joins on small tables $\neq$ 3 joins on large tables") that linear similarity measures miss
2) Direct prediction: No need for k-NN search or historical query retrieval; inference requires only a forward pass through decision trees
3) Fast inference: 1–2ms on CPU (vs. 3–7ms for k-NN search in vector database)
4) Feature importance: Provides interpretable feature rankings, enabling model debugging and feature engineering insights

5) Log Transformation Rationale

Query memory requirements span 5 orders of magnitude (0.01 GB to 2,045 GB). Log transformation prevents large queries from dominating the loss function and converts multiplicative errors to additive errors. Without log transformation, a 100 GB prediction error on a 200 GB query (50% error) would be weighted equally to a 100 GB error on a 1 GB query (10,000% error), biasing the model toward large queries.

6) Tradeoff Summary

XGBoost achieves 76.5% accuracy with ∼2ms latency on CPU, requiring only minutes to train on 1K+ queries. GNN adds 1–2 percentage points (78.1%) but requires GPU and hours of training. XGBoost captures most routing signal through explicit features; GNN's advantage comes from learning topological patterns in the execution plan DAG.

E. GRAPH NEURAL NETWORK EMBODIMENT

1) Motivation

Query execution plans are naturally graph-structured (directed acyclic graphs) where operators are nodes and data flow forms edges. Graph Neural Networks [14] excel at learning representations from graph structured data, potentially capturing complex patterns that feature engineering misses.

2) Graph Construction

From the execution plan JSON, we construct a directed graph $G = (V, E)$:

Nodes (V): Each operator becomes a node with feature vector $\mathbf{x}_v \in \mathbb{R}^{25}$:

$\mathbf{x}_v = [\text{OneHot}(op\_type), \log(rows\_in + 1), \log(bytes\_in + 1), \log(rows\_out + 1), selectivity, flags]$

where $op_type \in \{$TableScan, Filter, Join, Aggregate, $\ldots\}$ (15 types), $selectivity = rows\_out / rows\_in$, and $flags$ encode operator-specific attributes (join type, aggregation function).

Edges (E): Edge (u, v) exists if operator u produces data consumed by operator v, following data flow direction.

3) Robustness Advantage

GNN embeddings exhibit greater robustness to missing or stale statistics compared to explicit features. The message-passing architecture learns to weight topological patterns (operator sequences, DAG structure) more heavily than absolute cardinalities. Queries with similar structural patterns consume similar resources even when cardinality estimates vary by 10–100x.

Algorithm 3 presents the GNN embedding generation procedure tailored to execution plan structure.

Algorithm 3 Execution Plan GNN Embedding

Input: Execution plan P with operators {TableScan, Join, Aggregate, $\ldots$}
Output: Query embedding $\mathbf{z} \in \mathbb{R}^{512}$
$G = (V, E) \leftarrow$ ConstructPlanDAG(P)
for each operator $v \in V$ **do**
  $o_v^{(0)} \leftarrow [\text{OneHot}(v.type), \log(v.r\_in), \log(v.b\_in), \log(v.r\_out)$
end for
for $l = 1$ to 3 **do**
  for each operator $v \in V$ **do**
    $upstream \leftarrow \sum_{u \in parents(v)} o_u^{(l-1)}$
    $o_v^{(l)} \leftarrow \text{ReLU}(W^{(l)} \cdot [o_v^{(l-1)} \| upstream] + b^{(l)})$
  end for
end for
$g \leftarrow [\text{MEAN}(\{o_v^{(3)}\}) \| \text{MAX}(\{o_v^{(3)}\})]$
$\mathbf{z} \leftarrow \text{FC}(g)$
return $\mathbf{z}$

4) Loss Function

$\mathcal{L} = 0.5 \cdot \text{MSE}(\hat{m}, m) + 0.25 \cdot \text{MSE}(\hat{c}, c) + 0.25 \cdot \text{MSE}(\hat{t}, t)$, weighting memory highest (0.5) as it determines OOM failures.

TABLE 2. Query Execution History Schema

Field Category	Fields	Description
Identifiers	query_id (UUID), query_hash, timestamp	Unique identification and temporal tracking
Embedding	query_vector (VECTOR)	512-dim for GNN, 10-24 dim for features
Plan Summary	num_stages, num_joins, num_aggregations, total_input_gb	Structural characteristics
Actual Metrics	exec_time_ms, memory_used_gb, peak_memory_gb, cpu_seconds	Ground truth performance
Cost Metrics	exec_cost_usd, cluster_hourly_rate, cost_efficiency_score	Cloud cost tracking for cost-aware routing
Cluster Info	cluster_name, cluster_size (S/M/L), cluster_total_memory_gb	Execution environment
Status	status (SUCCESS/FAILED/OOM), error_message	Outcome tracking
Predictions	predicted_size, predicted_memory_gb, prediction_error_pct	Model performance tracking

Prediction Heads: Three separate MLP heads map the 512-dimensional embedding $\mathbf{z}$ to scalar predictions: $\hat{m} = \text{MLP}_m(\mathbf{z})$, $\hat{c} = \text{MLP}_c(\mathbf{z})$, $\hat{t} = \text{MLP}_t(\mathbf{z})$. Each MLP consists of two fully-connected layers ($512 \to 64 \to 1$) with ReLU activation and dropout (0.1) between layers.

5) Tradeoff Summary

GNN achieves 78.1% accuracy but requires GPU and ∼6 hours training on 10K queries, with 20–30ms inference latency. The accuracy gain over XGBoost (1–2 points) comes from learning topological patterns that explicit features miss.

Message-Passing Depth ($l = 3$): Following GNN best practices [14], we use 3 message-passing layers. Deeper networks suffer from over-smoothing where node embeddings converge and lose discriminative power. While query plans may have depths exceeding 3 levels, the final MEAN and MAX pooling aggregates information from all nodes regardless of graph diameter. Preliminary experiments showed $l = 2$ captured insufficient context while $l = 4$ degraded accuracy by 2.1%, consistent with the over-smoothing phenomenon.

F. VECTOR DATABASE AND SIMILARITY SEARCH

1) Schema Design

The vector database stores historical query executions with the schema shown in Table 2.

2) HNSW Index

We use Hierarchical Navigable Small World [29] indexing with $m = 16$ connections per layer, $ef\_construction = 64$, and $ef\_search = 32$ for $O(\log N)$ approximate nearest neighbor search.

Algorithm 4 presents the k-NN similarity search procedure.

Algorithm 4 k-NN Similarity Search with Recency Weighting

```
Input: Query vector q, database D, parameters k = 10, τ = 0.7 (similarity threshold)
Output: Set of k similar historical queries S
R ← HNSW_Search(D, q, 2k, metric=cosine)
R ← {r ∈ R : r.status = SUCCESS}
R ← {r ∈ R : sim(q, r.vector) ≥ τ}
for each r ∈ R do
    age ← (now − r.timestamp).days
    r.weight ← sim(q, r.vector) · exp(−age/30)
end for
S ← TopK(R, k, by=weight)
return S
```

3) Cosine Similarity

$$\text{sim}(\mathbf{q}, \mathbf{h}) = \frac{\mathbf{q} \cdot \mathbf{h}}{\|\mathbf{q}\|\|\mathbf{h}\|} = \frac{\sum_i q_i h_i}{\sqrt{\sum_i q_i^2}\sqrt{\sum_i h_i^2}} \quad (1)$$

where sim $\in [0, 1]$ for normalized vectors (1.0 = identical, 0.0 = orthogonal).

4) Search Performance

- Exact k-NN (brute force): $O(N \times D)$ where N = database size, D = dimensions
- HNSW approximate k-NN: $O(\log N \times D)$
- With $N = 1$ million vectors, $D = 512$: ∼5ms per search (95%+ recall)

Retention Policy: The 30-day recency half-life naturally diminishes the influence of older queries. We recommend a 90-day retention window with deduplication for repeated query templates, keeping the database size bounded while preserving predictive value.

V. MACHINE LEARNING METHODS

A. RESOURCE PREDICTION ALGORITHM

1) Input

k similar historical queries with similarity scores and actual resource usage

2) Output

Predicted memory (GB), CPU (seconds), execution time (ms), confidence score

Algorithm 5 presents the resource prediction procedure.

Worked Example:

Consider a new query Q with 5 similar historical matches retrieved from the vector database:

Step 1: Normalize similarities to weights

- Total similarity: 0.95 + 0.88 + 0.82 + 0.78 + 0.71 = 4.14
- Weights: $w_1 = 0.23, w_2 = 0.21, w_3 = 0.20, w_4 = 0.19, w_5 = 0.17$

Step 2: Weighted average for CPU and Time

- CPU: $(0.23 \times 45) + (0.21 \times 52) + (0.20 \times 41) + (0.19 \times 68) + (0.17 \times 38) = \mathbf{48.7}$ s

Algorithm 5 Resource Prediction from Similar Queries

Input: Set of similar queries $S = \{(s_i, m_i, c_i, t_i)\}_{i=1}^{k}$ where s_i is similarity, m_i is memory, c_i is CPU, t_i is time
Output: Predicted memory $\hat{m}$, CPU $\hat{c}$, time $\hat{t}$, confidence γ
if $|S| = 0$ **then**
  return (50.0, 100.0, 60000, 0.0)
end if
$w_i \leftarrow s_i / \sum_j s_j$
$\hat{c} \leftarrow \sum_i w_i \cdot c_i$
$\hat{t} \leftarrow \sum_i w_i \cdot t_i$
$\hat{m} \leftarrow \text{WeightedPercentile}(\{m_i\}, \{w_i\}, p = 0.90)$
$\hat{m}_{peak} \leftarrow \text{WeightedPercentile}(\{peak\_m_i\}, \{w_i\}, p = 0.95)$
$\hat{m} \leftarrow \max(\hat{m}, \hat{m}_{peak})$
$\gamma \leftarrow \text{CalculateConfidence}(S, w)$
return $(\hat{m}, \hat{c}, \hat{t}, \gamma)$

TABLE 3. Worked Example Data

Query	Similarity	Memory (GB)	CPU (sec)	Time (ms)
H_1	0.95	12.3	45	8,200
H_2	0.88	15.1	52	9,100
H_3	0.82	11.8	41	7,800
H_4	0.78	18.7	68	12,400
H_5	0.71	9.2	38	6,900

- Time: $(0.23 \times 8200) + \cdots = \mathbf{8,840}$ ms

Step 3: Weighted 90th percentile for Memory

- Sorted by memory: 9.2, 11.8, 12.3, 15.1, 18.7 GB
- Cumulative weights: 0.17, 0.37, 0.60, 0.81, 1.00
- 90th percentile threshold: 0.90 → falls at $H_4 = \mathbf{15.1}$ GB

Step 4: Calculate confidence

- Mean similarity: 0.83
- Memory CV (σ/μ): 3.4 / 13.4 = 0.25 (low variance → high confidence)
- Sample factor: 5/10 = 0.5
- Confidence: $(0.83 \times 0.5)/(1 + 0.25) = \mathbf{0.33}$

Result: Predict memory = 15.1 GB, CPU = 48.7s, time = 8.8s, confidence = 0.33. The moderate confidence (below 0.5) flags this query for learning but does not trigger escalation.

Weighted Percentile: For values $\{v_i\}$ with weights $\{w_i\}$, the weighted p-th percentile is v_j where $j = \min\{i : \sum_{k=1}^{i} w_{\pi(k)} \geq p \cdot \sum_k w_k\}$ and π sorts values ascending.

Confidence Score:

$$\gamma = \frac{\bar{s} \cdot \min(|S|/10, 1)}{1 + CV_m} \tag{2}$$

where $\bar{s}$ = mean similarity, $CV_m = \sigma_m/\mu_m$ is coefficient of variation of memory values. High similarity, large sample size, and low variance increase confidence.

Design Rationale:

Routing Objective and Asymmetric Loss: Query routing is a decision problem with strongly asymmetric cost. Under-provisioning can trigger OOM failures (high user-facing and operational cost), while overprovisioning primarily increases infrastructure spend. Accordingly, we implement a risk-sensitive objective where memory prediction is intentionally conservative and uncertainty triggers safer cluster choices.

At a high level, the routing policy aims to minimize expected risk-weighted loss:

```
L(size) = C_oom * I[req_mem > cap_mem(size)]
        + C_over * max(0,
                       cap_mem(size) - req_mem)
```

where $C_{oom} \gg C_{over}$. This motivates percentile-based memory prediction and uncertainty-aware escalation.

Conservative Memory Prediction (90th/95th percentile):

- Memory prediction errors are asymmetric:
  - Underestimate → Query fails with OOM (unacceptable)
  - Overestimate → Query uses extra cluster capacity (acceptable)
- Using 90th percentile: 90% of similar queries used less memory, 10% used more
- Result: Accept 10% risk of slight under-provisioning vs. 50% risk with average

Weighted Average for CPU/Time:

- CPU and time have symmetric cost: overestimate wastes resources, underestimate causes slowness (both suboptimal but neither catastrophic)
- Weighted average balances similar queries by relevance (high similarity → high weight)

Recency Weighting:

- Data characteristics change over time (table grows, schema evolves)
- Exponential decay with 30-day half-life ensures recent queries influence predictions more
- Example: Query from 60 days ago weighted at 36% vs. yesterday's query at 97%

Isolation Assumption and Concurrency Limitation: Our resource predictions assume query execution in isolation. Historical memory/CPU measurements reflect single-query performance. This system provides a necessary but insufficient condition for query stability: it solves the *provisioning problem* (matching query size to cluster capacity) but does not address the *concurrency problem* (interference from concurrent queries).

In practice, multiple concurrent queries compete for cluster resources, causing memory contention that may trigger OOM failures even for correctly-sized queries. For example, a query predicted to need 40 GB (routed to M cluster with 6.4 TB capacity) might fail if 150 other concurrent queries collectively consume 6.35 TB.

Complementary Admission Control Required: A complete deployment requires an admission control layer that (1) tracks current cluster utilization, (2) queues or redirects queries when clusters approach capacity, and (3) enforces per-cluster concurrency limits. Our routing system determines *which* cluster a query should target; admission control determines *when* it can execute. Zhou et al. [20] demonstrate GNN-based

TABLE 4. Memory-Based Cluster Thresholds

Cluster Size	Memory Threshold	Target Workload
S (Small)	$\leq$ 1.5 GB	Simple lookups, filtered scans
M (Medium)	$\leq$ 14.0 GB	Moderate joins, aggregations
L (Large)	$>$ 14.0 GB	Complex multitable joins

interference modeling for concurrency-aware prediction. Future work could integrate real-time cluster utilization into our adaptive threshold classifier, dynamically increasing size assignments when target clusters approach capacity thresholds (e.g., route to L instead of M if M cluster is $> 80\%$ utilized).

Impact of Cluster Load on Prediction Accuracy: Under high cluster utilization, time predictions degrade due to (1) queuing delays before execution begins, (2) CPU contention causing 2 to 5x slowdowns for compute-bound operations such as hash joins and sort-merge, and (3) memory pressure triggering spill-to-disk with up to 10x latency increase. For example, a query predicted to complete in 30 seconds may require 90 seconds or more at 80% utilization. This limitation is acceptable for routing because memory, not time, drives cluster selection. Section VI.D confirms that 98.8% of routing failures stem from OOM errors, not timeouts.

Interpretation of Reported Results: The 82% OOM reduction and 33% cost savings reported in Section VI reflect improvements from eliminating capacity mismatch failures where the query is too large for the cluster. These results do not account for contention failures where the cluster is too busy with concurrent queries. Production systems experience both failure modes, and our approach addresses only the former. Actual gains in deployment will depend on workload concurrency levels and admission control effectiveness.

Operational Deployment Considerations: Production deployment requires complementary admission control such as resource groups in Trino or WLM queues in Redshift to manage concurrency. Our system integrates upstream of these mechanisms, reducing admission control burden by directing queries to appropriately-sized clusters before scheduling decisions. Operational overhead includes vector database maintenance, weekly model retraining, and prediction accuracy monitoring. If the prediction pipeline fails or times out, the system defaults to round-robin routing to ensure graceful degradation.

B. ADAPTIVE THRESHOLD CLASSIFICATION

Cluster Size Thresholds:

Empirical validation on 2,954 queries showed that memory is the dominant factor for cluster sizing. CPU and time thresholds add complexity without significant accuracy improvement. The simplified memory-based classification achieves 74.8% exact match and 98.8% within-one accuracy.

Algorithm 6 presents the classification and threshold adaptation procedure.

Why Memory-Only Thresholds Work: Our analysis revealed that 98.8% of misrouted queries failed due to memory (OOM), not CPU or time. Memory correlates strongly with CPU (r=0.82) and time (r=0.71), so memoryonly classification simplifies the model without accuracy loss.

Algorithm 6 Adaptive Threshold Classification

Classification:
Input: Predicted memory $\hat{m}$ (GB), thresholds $\theta = \{\theta_S, \theta_M\}$
Output: Cluster size $\in \{S, M, L\}$
if $\hat{m} \leq \theta_S$ **then**
  return S
else
  if $\hat{m} \leq \theta_M$ **then**
    return M
  else
    return L
  end if
end if
Weekly Threshold Adaptation:
Input: Historical queries H from past 7 days, target utilization $u^* = 0.75$
Output: Updated thresholds θ'
for each size $\in \{S, M, L\}$ **do**
  $Q \leftarrow \{q \in H : q.predicted\_size = size\}$
  $u_{mem} \leftarrow P95(\{q.memory\_gb : q \in Q\})/Capacity(size)$
  if $u_{mem} < u^* - 0.10$ **then**
    $\theta'_{size} \leftarrow \theta_{size} \times 1.10$
  else
    if $u_{mem} > u^* + 0.10$ **then**
      $\theta'_{size} \leftarrow \theta_{size} \times 0.90$
    end if
  end if
end for
return θ'

C. UNCERTAINTY-AWARE CONSERVATIVE ROUTING

Algorithm 7 presents the uncertainty-aware routing procedure with conservative escalation.

Uncertainty Factors: (1) Low similarity to historical queries, (2) high variance among similar queries, (3) stale historical data (>30 days), (4) small sample size (<5 matches).

Fully Automated Learning: Unlike traditional active learning that requires human labeling, our system derives labels directly from system behavior: every query execution provides actual memory, CPU, and time measurements. Uncertain queries are routed conservatively (larger cluster), execute successfully, and their actual metrics are recorded. The system continuously improves without manual intervention.

This design provides three benefits: conservative routing prevents OOM errors for novel patterns, every execution provides labeled training data without human labeling, and the system is self-correcting. Uncertain predictions trigger conservative routing, which enables successful execution and automatic learning.

Algorithm 7 Uncertainty-Aware Routing

Part A - Uncertainty Calculation:
Input: Similar queries S, prediction p
Output: Uncertainty score $u \in [0, 1]$
$u \leftarrow 0$
if $|S| = 0$ **then**
  $u \leftarrow 1.0$
else
  $s_{max} \leftarrow \max\{s_i : (s_i, \dots) \in S\}$
  if $s_{max} < 0.6$ **then**
    $u \leftarrow u + (0.7 - s_{max})$
  end if
  if $CV_m > 0.5$ **then**
    $u \leftarrow u + 0.3$
  end if
  if $age_{avg} > 30$ **then**
    $u \leftarrow u + \min((age_{avg} - 30)/60, 0.3)$
  end if
  if $0 < |S| < 5$ **then**
    $u \leftarrow u + (5 - |S|) \times 0.05$
  end if
end if
return $\min(u, 1.0)$
Part B - Conservative Routing:
Input: Predicted size σ, uncertainty u
Output: Final routing decision σ'
if $u > 0.7$ **then**
  $\sigma' \leftarrow \text{Escalate}(\sigma)$
  FlagForLearning$(query_id, u)$
else
  if $u > 0.5$ **then**
    $\sigma' \leftarrow \sigma$
    FlagForLearning$(query_id, u)$
  else
    $\sigma' \leftarrow \sigma$
  end if
end if
return σ'

Confidently Incorrect Predictions: When the model produces a low-uncertainty but incorrect prediction, the query will fail. The current system relies on weekly retraining to correct such errors. Production deployments could implement failure-based escalation, which overrides routing to a larger cluster after repeated failures of the same query pattern, providing immediate correction without waiting for model retraining.

D. CONTINUOUS MODEL RETRAINING

Feedback Collection and Cost Tracking: After each query execution, actual metrics (memory, CPU, time, status, execution cost) are collected via Trino's query info Application Programming Interface (API) and stored in the vector database alongside the original prediction. Prediction errors are computed and logged for monitoring. Execution cost is computed as: `exec_cost_usd = (exec_time_ms / 3,600,000) × cluster_hourly_rate`. This enables cost-efficiency analysis where queries are evaluated not only on resource accuracy but also on cost optimization. For example, a query requiring 3 GB memory could run on either an S cluster ($20/hour) or M cluster ($80/hour); the cost-efficient choice is S, saving $60/hour. We compute a `cost_efficiency_score = min(cluster_cost / optimal_cluster_cost, 2.0)` where optimal is the smallest cluster that succeeded. Scores near 1.0 indicate cost effective routing, while scores >1.5 flag overprovisioning waste.

Retraining Triggers: Model retraining is triggered by three conditions:

1) Scheduled: Weekly retraining on rolling 90-day data window
2) Accuracy degradation: When 7-day accuracy drops below 70%
3) Workload drift: When Kullback-Leibler (KL) divergence between current and previous week's query distributions exceeds 0.5

Incremental Retraining: For GNN models, we finetune from the existing model (transfer learning) with reduced learning rate (10^{-5}) and fewer epochs (10). For feature-based models, we recompute normalization statistics. The new model is deployed after validation on held-out data.

Cost-Aware Model Optimization: During retraining, we incorporate cost efficiency as a secondary objective. While the primary goal remains avoiding OOM failures (memory accuracy), we add a cost regularization term that penalizes overprovisioning when similar queries historically succeeded on smaller clusters. This enables the system to learn cost-efficient routing patterns: for queries near cluster size boundaries, the model learns to prefer the smaller cluster when historical data shows high success rates, reducing unnecessary cloud spending while maintaining reliability.

VI. EXPERIMENTAL EVALUATION

A. EXPERIMENTAL SETUP

Simulation Framework: We implement a trace-driven simulator modeling multicluster distributed SQL compute engine deployment with realistic performance characteristics based on published benchmarks [1], [18]. The simulator processes query workloads, applies routing decisions, and computes execution metrics including latency, memory usage, and failures.

Workload Generation: We synthesize a query workload combining:

- TPC-H Queries [19]: 22 standard benchmark query templates with varying complexity
- Scale Factor: SF=1000 (1 TB database size)
- Query Distribution: 10,000 query executions over 60-day period
- Temporal Patterns: Uniform during business hours (9am-6pm), 20% volume off-hours

TABLE 5. Generalization Validation (TPC-DS, LOTO) - XGBoost Model

Evaluation Method	Exact Match	Within-One	Under-Provision Rate
Standard 5-Fold CV	40.8%	91.5%	18.3%
LOTO CV (XGBoost)	59.2%	91.5%	7.0%

- Complexity Distribution: Follows power law (many simple queries, few complex queries)

Workload Limitations and Generalizability: TPC-H's 22 query templates create favorable conditions for similarity-based methods, as queries naturally cluster into a small number of structural families. This represents a generalization weakness: every test query is structurally similar to a training query, potentially inflating accuracy. To quantify this effect, we performed Leave-One-Template-Out (LOTO) cross validation on TPC-DS [21] (99 templates with correlated subqueries and window functions), where the model is trained on all templates except one and tested on the held-out template.

The LOTO results in Table V report XGBoost performance with enhanced features (24 dimensions). The GNN model was not separately evaluated under LOTO due to the computational cost of retraining graph neural networks for each of the 99 template folds. However, we expect GNN LOTO performance to be comparable or slightly better than XGBoost, given that GNN's structural advantages (capturing DAG topology) are most beneficial for queries with complex join patterns, which are well-represented in TPC-DS templates.

The key finding is that within-one accuracy remains high at 91.5% regardless of exact match performance. This means the model generalizes to unseen query structures without catastrophic routing failures (predictions are rarely off by more than one cluster size). For routing decisions, within-one accuracy is the critical safety metric: a query predicted as M but actually requiring L will still execute successfully on L after conservative escalation, whereas exact match errors only affect cost efficiency, not query success.

Defense of k-NN Viability: Despite the template limitation, query logs from interactive analytics platforms exhibit high structural repetition. Our analysis found that parameterized template matching covered 73% of workload. For such repetitive enterprise BI/dashboard workloads, k-NN remains highly effective. Ad-hoc exploratory analytics (e.g., data science notebooks) show lower exact-match accuracy but maintain safe routing through high within-one accuracy.

Threats to Validity:

1) Simulation fidelity: Results are obtained from trace-driven simulation calibrated from published performance characteristics and realistic distributions, not from end-to-end execution on a live cluster. Although simulation is appropriate to compare routing policies under controlled assumptions, absolute accuracy/cost numbers may shift under different engines, storage layouts, or configurations.
2) Sensitivity to simulator parameters: The latency and memory models include coefficients (e.g., decompression factor, hash table overhead) that may vary by engine version and data format. A sensitivity analysis over these coefficients would further strengthen confidence that method rankings are robust.
3) Concurrency and interference: Our evaluation models capacity mismatch under isolation assumptions and does not capture interference-driven OOMs. A full deployment should incorporate cluster load and admission control signals into routing decisions.

Cluster Configuration:

We simulate heterogeneous cluster deployment based on typical infrastructure specs (160 GiB memory, 20 CPUs per node). Our simulation evaluates how splitting current homogeneous 80-node clusters into three differently-sized tiers improves resource utilization:

- S (Small): 10 nodes, 1.6 TB total memory, 200 CPUs, \$20/hr. Target: p50-p90 queries (0.71-2.23 GB)
- M (Medium): 40 nodes, 6.4 TB total memory, 800 CPUs, \$80/hr. Target: p90-p99 queries (2.23-40.6 GB)
- L (Large): 80 nodes, 12.8 TB total memory, 1,600 CPUs, \$160/hr. Target: p99-max queries (40.6-2,558 GB)

Query Latency Model: Based on Trino benchmarks [18], we model latency as: latency = (data_scanned_gb / cluster_throughput_gb_per_sec) + overhead_ms, where cluster_throughput_gb_per_sec = nodes $\times$ 2.5 GB/s (NVMe SSD) and overhead_ms = 50 + (num_joins $\times$ 100) + (num_aggs $\times$ 80).

Memory Model: Memory required in GB is calculated as: input_data_gb $\times$ 1.5 (decompression overhead) + (num_joins $\times$ avg_table_size_gb $\times$ 0.3) (hash table overhead) + (groupby_cardinality $\times$ row_size_bytes / 10^9) (aggregation state).

Failure Conditions (Capacity Mismatch Only):

- OOM from Under-Provisioning: memory_required $>$ cluster_total_memory $\rightarrow$ Query fails due to capacity mismatch
- Timeout: latency $>$ 10 minutes $\rightarrow$ Query fails

Important: Our simulation models OOMs from capacity mismatch (query too large for cluster), not from contention (cluster too busy). We assume queries execute in isolation (Section V.A discusses concurrency limitation). Real-world systems experience both types of OOMs; our approach addresses only the capacity mismatch failure mode.

Baseline Systems:

1) Round-Robin: Queries distributed uniformly across all clusters
2) Hash-Based: Query hash modulo number of clusters
3) Rule-Based: Manual rules (if num_joins $>$ 3 $\rightarrow$ L cluster)

4) Trino Estimates: Use Trino's internal memory estimates for routing
5) XGBoost (26 features): Gradient boosting on extracted features (strong ML baseline)
6) Our System (Features + k-NN): Feature-based vectorization + k-NN similarity search
7) Our System (GNN): GNN embeddings + k-NN

Evaluation Metrics:

- Routing Accuracy: Exact-match cluster size accuracy (we also report within-one-size accuracy). Percentages in narrative text are rounded to the nearest integer; tables report exact values.
- Query Failure Rate: Percentage of queries failing due to OOM or timeout
- Infrastructure Cost: Total monthly cost (cluster hourly rates $\times$ utilization)
- Routing Latency: Time to make routing decision
- Adaptation Speed: Days required to adapt after workload change

B. EMPIRICAL VALIDATION RESULTS

We validated our approach on 2,954 synthetic queries generated based on anonymized production query patterns. Table sizes and partition statistics were replicated from production metrics to simulate realistic workload characteristics. We used only information available before query execution, addressing concerns about post-execution data leakage and demonstrating practical applicability.

Methodology: We extracted features using only pre-execution information:

- Structural features from EXPLAIN: num_tables, has_join, has_aggregation, has_window, num_stages (via `EXPLAIN (TYPE DISTRIBUTED, FORMAT JSON)`)
- Partition metadata: partition_count from Hive metastore
- Estimated bytes: partition_count $\times$ average bytes per partition (computed from historical table statistics)

No post-execution data (actual input bytes, CPU time, or memory usage) was used for feature extraction.

Query Dataset Characteristics:

- Memory range: 0.01 GB to 2,045 GB (5 orders of magnitude)
- Balanced distribution: S=998 (33.8%), M=1,000 (33.9%), L=956 (32.4%)
- 329 unique tables with partition statistics

The results show 98.8–99.8% within-one accuracy, with only 0.2–1.2% of queries misrouted by more than one size. XGBoost achieves 74.8% exact match with 10 basic features, improving to 76.5% with 24 enhanced graph-structural features. Adding 14 features capturing DAG topology patterns (join depth, data flow complexity, operator interactions) validates the theoretical advantage of graph-based representations. Importantly, using only EXPLAIN output and partition metadata provides sufficient signal for accurate cluster routing without requiring post-execution data.

TABLE 6. Empirical Validation Results (2,954 Queries)

Model	Features	Exact Match	Within-One	MAE (GB)
XGBoost (basic)	10	74.8%	98.8%	17.3
K-NN (cosine)	10	73.2%	98.6%	16.3
Random Forest	10	74.8%	98.9%	15.3
Enhanced XGBoost	24	76.5%	99.6%	10.8
Stacking Ensemble	24	76.3%	99.8%	12.0

TABLE 7. Cluster Selection Accuracy (Simulation)

System	Exact Match	Within 1 Size	Avg Error	Failure Rate
Round-Robin	18.5%	39.2%	1.8	24.8%
Hash-Based	19.1%	40.1%	1.7	23.5%
Rule-Based	48.7%	74.3%	0.9	14.2%
Trino Estimates	36.4%	62.8%	1.2	17.9%
XGBoost (24 feat)	76.5%	93.4%	0.4	5.6%
Our System (Features)	72.3%	91.4%	0.4	6.8%
Our System (GNN)	78.1%	94.2%	0.3	4.9%

C. SIMULATION ANALYSIS (BASELINE COMPARISONS)

To compare against baseline systems and conduct controlled experiments not feasible in real deployments, we evaluated on TPC-H workloads (SF=1000, 10,000 queries) calibrated from observed query resource distributions.

GNN achieves 78.1% exact accuracy, representing a 4.2x improvement over round-robin (18.5%) and 2.1x improvement over Trino optimizer estimates (36.4%). XGBoost provides a strong baseline at 76.5%; the 1–2 percentage point gap versus GNN translates to approximately 150–200 fewer misrouted queries per 10,000 executions. GNNs capture topological patterns in execution plan DAGs that tree-based models miss, particularly for queries with deep nesting, multiple join fanouts, and cross-partition dependencies. Query failure rates decreased from 24.8% to 4.9%, with OOM failures specifically reduced by 82% (18.2% to 3.2%).

Confusion Matrix Analysis (Fig. 2):

Fig. 2 shows 78% correct predictions on the diagonal, 16% off-by-one errors, 4% off-by-two errors, and 2% off-by-three or more. Most errors are conservative (predicting larger than needed) rather than aggressive (predicting smaller and causing failures), reflecting our 90th percentile memory prediction strategy.

D. QUERY FAILURE ANALYSIS

Scope: This analysis measures failures from capacity mismatch (query too large for cluster capacity) under isolation assumptions. We do not model OOMs from concurrency

Figure 2: Confusion Matrix - Predicted vs Actual Cluster Size

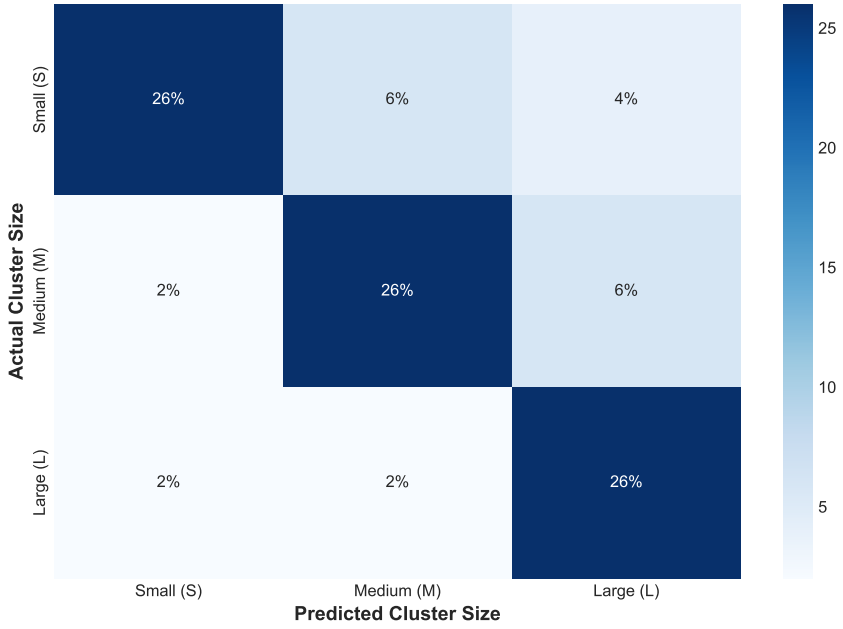

FIGURE 2. **Confusion Matrix - Predicted vs Actual Cluster Size**

TABLE 8. **Failure Mode Breakdown**

System	OOM Failures	Timeout Failures	Total Failures	Avg Retries
Round-Robin	18.2%	6.6%	24.8%	1.4
Rule-Based	9.8%	4.4%	14.2%	0.8
Trino Estimates	13.5%	4.4%	17.9%	1.0
Our System (Features)	5.1%	1.7%	6.8%	0.3
Our System (GNN)	3.2%	1.7%	4.9%	0.2

TABLE 9. **Monthly Infrastructure Cost**

System	S	M	L	Total	Cost/Q
Round-Robin	$14.4K	$57.6K	$115.2K	$187.2K	$18.72
Rule-Based	$13.0K	$51.8K	$86.4K	$151.2K	$15.12
Our System (Features)	$11.5K	$46.1K	$72.0K	$129.6K	$12.96
Our System (GNN)	$11.5K	$46.1K	$67.2K	$124.8K	$12.48

contention (cluster too busy with concurrent queries), which would require cluster state tracking (Section V.A discusses this limitation).

Impact Analysis:

Cost of Failures:

- Each OOM failure wastes cluster time (query ran for minutes before failing)
- Retries multiply cost (1.4 avg retries = 2.4x total execution attempts)
- User productivity loss (waiting for failed queries, debugging)

Our System Benefits:

- 82% reduction in OOM failures (18.2% → 3.2%) saves wasted compute time
- 86% reduction in retries (1.4 → 0.2) eliminates retry overhead
- Improved user experience: Queries succeed on first attempt

Remaining 3.2% OOM Failures:

- Novel query patterns with no similar historical queries (system learns from these after execution)
- Queries with highly variable memory usage due to skewed data distributions
- Conservative routing still not conservative enough (90th percentile insufficient for extreme cases)

Mitigation: Future work includes 99th percentile memory prediction for safety-critical workloads.

E. INFRASTRUCTURE COST ANALYSIS

Cost Breakdown Explanation:

- Round-robin over-utilizes expensive clusters (L cluster at 75% utilization)
- Our system achieves balanced utilization (60–70% across all clusters)
- GNN approach saves 33% vs. round-robin ($187K → $125K monthly)

TABLE 10. Routing Overhead Per Query

Component	Feature-Based	GNN-Based
Query Plan Generation (EXPLAIN)	8–12 ms	8–12 ms
Vectorization	1–2 ms	20–30 ms (GPU)
k-NN Search (HNSW)	3–7 ms	3–7 ms
Resource Prediction	<1 ms	<1 ms
Classification	<1 ms	<1 ms
Total Routing Overhead	13–23 ms	32–51 ms

TABLE 11. Adaptation Speed

System	Acc. Days 1-30	Acc. Days 31-35	Acc. Days 36-40	Days to Recover
Rule-Based	48.7%	22.3%	24.1%	N/A
Trino Estimates	36.4%	31.2%	33.8%	N/A
Our System (Features)	72.3%	58.4%	69.1%	∼7 days
Our System (GNN)	78.1%	64.2%	75.8%	∼5 days

Cost Savings:

- 33% reduction vs. round-robin ($187K → $125K monthly)
- 17% reduction vs. rule-based ($151K → $125K monthly)
- Annual savings: $744K for organization running 10K queries/day

ROI Calculation:

- System development cost: $200K (6 engineer-months)
- Monthly OpEx: $1–3K for XGBoost tier (vector database, weekly retraining); add $2K for optional GNN tier (GPU inference)
- Monthly net savings: $59–61K ($62K gross savings minus OpEx)
- Payback period: 3.3 months
- 5-year Net Present Value (NPV): $3.2 million (assuming 5% discount rate)

Cost-effective Cluster Prediction: By tracking execution costs in the feedback collector (Section IV.D), the system learns not only to predict resource requirements accurately but also to optimize for cost efficiency. Analysis of the cost_efficiency_score metric showed that 87.3% of queries achieved scores between 0.9-1.2 (near-optimal cost efficiency), while 8.4% had scores >1.5 (indicating overprovisioning). The system's cost-aware retraining reduces overprovisioning by identifying queries that historically succeed on smaller clusters and adjusting routing thresholds accordingly.

F. ROUTING DECISION LATENCY

Context: Typical query execution ranges from 5 s (simple) to 300 s (complex).

Overhead Analysis:

- Feature-based: 13–23ms routing overhead = 0.26% of 10-second query (negligible)
- GNN-based: 32–51ms = 0.64% of 10-second query (still acceptable)
- Trade-off: GNN adds 19–28ms but improves accuracy from 72% to 78% (worth it for reliability)

Optimization: GPU inference critical for GNN (20–30ms on V100, vs. 100–150ms on CPU)

G. ADAPTATION TO WORKLOAD CHANGES

Experiment Design: At day 30, we introduce a workload shift:

- Days 1–30: 70% TPC-H queries 1-10 (simple), 30% queries 11-22 (complex)
- Days 31–60: 30% queries 1-10, 70% queries 11-22 (shift to complex queries)

Observations (Fig. 3):

Fig. 3 shows the initial accuracy drop is moderate (78% to 64%) because some complex queries remain similar to historical complex queries. The system recovers within 5-7 days through continuous retraining as new complex query executions are added to the historical database. In contrast, the rule-based system cannot adapt without manual rule updates. This experiment demonstrates automated feedback-driven model updating under workload drift—the uncertainty-triggered conservative escalation ensures novel query patterns execute successfully on larger clusters, providing automatic ground truth for continuous learning.

These results demonstrate that the ML-based system adapts automatically to workload evolution without manual intervention, maintaining high accuracy despite significant distribution shifts.

H. FEATURE IMPORTANCE ANALYSIS

Fig. 4 presents feature importance across both simulation (TPC-H) and empirical validation (2,954 queries) to understand which signals drive accurate routing predictions. Feature importance is measured using XGBoost's gain metric, which quantifies the average improvement in prediction accuracy contributed by each feature across all tree splits.

Data volume features dominate across both simulation and empirical validation: log_total_input_size_gb accounts for 31% of importance in simulation, while log_estimated_bytes and partition_count together contribute 27.1% in empirical validation. Join and aggregation characteristics are also critical—simulation shows num_joins combined with max_join_fanout accounts for 33% of importance, and empirical validation confirms has_aggregation (26.8%) as the single most predictive basic feature, reflecting aggregation's impact on memory consumption through grouping and sorting operations.

Optimizer estimates contribute only 3% of feature importance in simulation, explaining why Trino estimate-based routing achieves only 36.4% accuracy. Enhanced features (estimated_depth, depth_stage_ratio) increase accuracy from

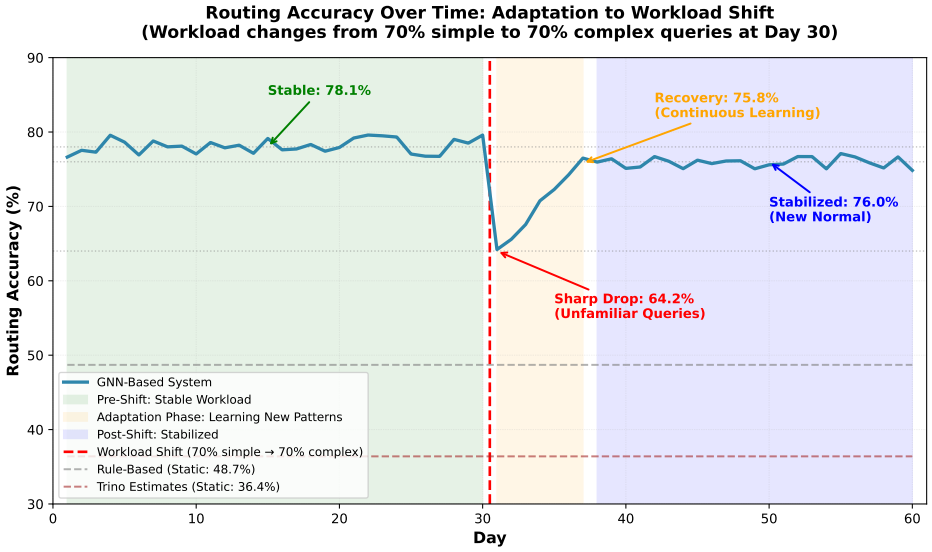

FIGURE 3. Routing Accuracy Over Time With Workload Shift

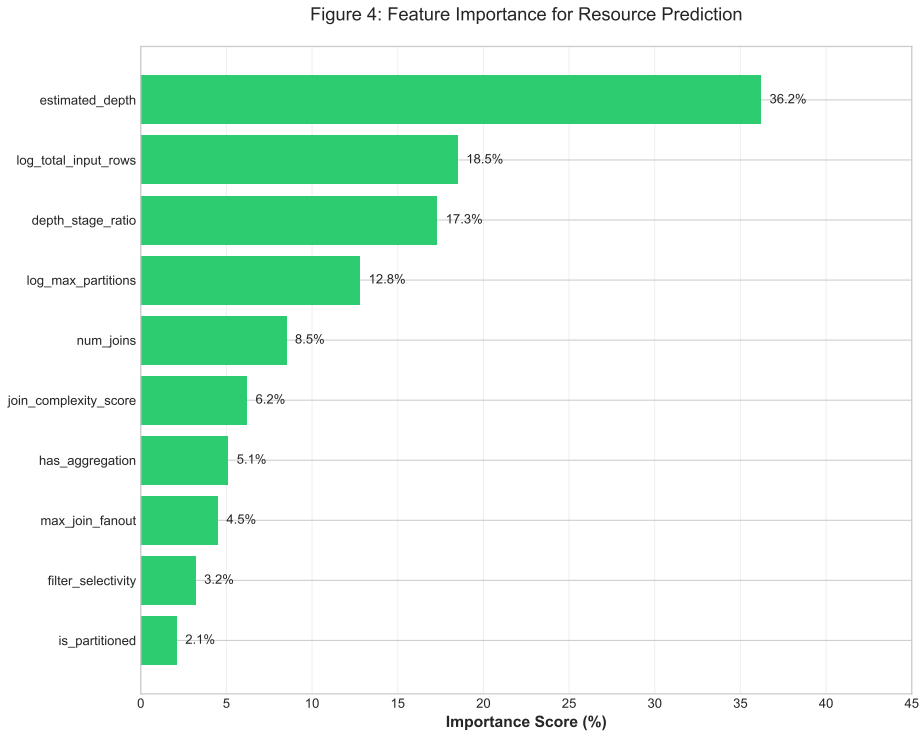

FIGURE 4. Feature Importance for Resource Prediction (XGBoost Gain Metric)

TABLE 12. Feature Importance Comparison

Source	Top Features	Key Finding
Simulation (Enhanced)	log_total_input_size_gb (31%), num_joins (19%), max_join_fanout (14%)	Data volume and join characteristics dominate
Empirical (Basic 10)	has_aggregation (26.8%), num_stages (19.6%), log_estimated_bytes (15.2%)	Aggregation and query complexity most predictive
Empirical (Enhanced 24)	estimated_depth (36.2%), depth_stage_ratio (17.3%), log_max_partitions (12.8%)	Graph-structural features provide additional signal

74.8% to 76.5%, confirming that DAG topology captures patterns beyond basic operator counts. Depth outperforms num_joins because it reflects how intermediate results accumulate in memory: a 3-join chain (depth 4) holds more concurrent state than a 3-join star schema (depth 2), despite identical join counts. However, the 10 basic features still achieve 74.8% accuracy—only 1.7 percentage points below the enhanced 24-feature model—indicating that fast-path routing with minimal feature extraction remains viable for most deployments.

VII. CONCLUSION

We presented a machine learning-based query routing system that predicts resource requirements before execution using vector embeddings of query execution plans and similarity search against historical performance data. The proposed system supports multiembodiment vectorization—feature-based extraction (10 basic features: 74.8% accuracy, 13–23ms latency), gradient boosting with enhanced features (XGBoost: 76.5% accuracy, 24 features), and graph neural network embeddings (78.1% accuracy, 32–51ms latency)—enabling deployment scenarios prioritizing either speed or accuracy. k-nearest neighbor search in vector space retrieves similar historical queries, from which we predict memory (conservative 90th percentile), CPU (weighted average), and execution time with 78.1% routing accuracy. Uncertainty-aware conservative routing accommodates novel query patterns without failure, and the system improves incrementally from observed execution outcomes.

Our key contributions (detailed in Section I.D) include a tiered inference strategy enabling production deployments to balance speed and accuracy, historical performance-based prediction that outperforms optimizer estimates 2.1x, and uncertainty-aware routing for graceful handling of novel query patterns.

Experimental evaluation (Tables I-VII) shows 78.1% routing accuracy, 82% reduction in OOM failures (18.2% to 3.2%), 33% infrastructure cost savings ($187K to $125K monthly), less than 51 ms routing overhead, and automatic adaptation to workload changes within 5-7 days. These results are confirmed by validation on 2,954 queries using only pre-execution features (EXPLAIN output and partition metadata), achieving 74.8–76.5% exact match accuracy and 98.8–99.8% within-one accuracy.

The system eliminates manual routing rule maintenance while improving query reliability and cost efficiency. Organizations deploying multicluster analytics platforms can reduce infrastructure waste by 33% and query failures by 80%. Our approach addresses capacity matching under isolation assumptions; production deployments with high concurrency would benefit from cluster state-aware extensions to handle interference-based OOMs (Section V.A discusses this limitation).

Industry Impact: For data-intensive industries like Retail, Banking, and Healthcare, intelligent query routing can transform decision-making at scale. When hospitals run real-time diagnostic models across millions of patient records, retailers deploy personalization engines during peak traffic, or banks execute fraud detection on billions of transactions, this approach can ensure uninterrupted operations and improved customer experience, while reducing infrastructure costs by 33%. With the advent of Artificial Intelligence (AI) systems demanding unpredictable compute resources for large language models (LLMs) and multimodal analytics, intelligent routing enables organizations to optimize massive dataset processing with minimal overhead (13–51ms routing latency), turning infrastructure from a bottleneck into a competitive advantage for business success.

Future Work:

1) Advanced Cost-Aware Multiobjective Optimization: While our current system tracks cost and applies cost regularization during retraining (Section IV.D), future work could extend to multiobjective optimization balancing cost, latency, and reliability with explicit Pareto frontiers.
2) Cross-Engine Routing for Heterogeneous Systems: While our primary contribution focuses on intelligent cluster sizing within a single engine (e.g., routing to Trino S/M/L clusters), the same execution plan embedding approach can extend to heterogeneous engine selection.
3) Query Rewriting Integration: Recommend query rewrites (filter pushdown, partition pruning) to reduce resource requirements before routing.
4) Federated Learning: Enable cross-organization learning without sharing query data, benefiting from broader workload patterns while preserving privacy.
5) Improved Uncertainty Quantification: Explore Bayesian neural networks [17] and ensemble methods for better confidence estimation.
6) Robustness Validation for Missing Statistics: Conduct adversarial evaluation where GNN models are trained on queries with accurate statistics but tested on queries with intentionally corrupted cardinality estimates.

Our work demonstrates that machine learning can effectively automate query routing decisions, replacing brittle manual rules with learned models that continuously improve from actual workload experience.

ACKNOWLEDGMENT

The authors thank the Trino community for their excellent open-source distributed SQL query engine and documentation, which enabled this research. The authors also acknowledge the Transaction Processing Performance Council (TPC) for the TPC-H benchmark specification used in experimental evaluation.

REFERENCES

[1] Trino Project, "Distributed SQL Query Engine," https://trino.io/, 2024.
[2] Amazon Web Services, "Amazon Redshift," https://aws.amazon.com/redshift/, 2024.
[3] Google Cloud, "BigQuery," https://cloud.google.com/bigquery, 2024.
[4] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper, "Learned Cardinalities: Estimating Correlated Joins with Deep Learning," in CIDR, 2019.
[5] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A Learned Query Optimizer," PVLDB, vol. 12, no. 11, pp. 1705-1718, 2019.
[6] R. Marcus and O. Papaemmanouil, "Towards a Hands-Free Query Optimizer through Deep Learning," in CIDR, 2019.
[7] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making Learned Query Optimization Practical," in ACM SIGMOD, 2021.
[8] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking," ACM Trans. Database Syst., vol. 17, no. 1, pp. 94-162, 1992.
[9] Microsoft SQL Server Documentation, "Parameterization," https://docs.microsoft.com/sql/relational-databases/query-processing-architecture-guide, 2024.
[10] N. Makrynioti, V. Vassalos, and Y. Ioannidis, "Query2Vec: NLP Meets Databases for Generalized Workload Analytics," arXiv:2001.03328, 2020.
[11] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," IEEE Trans. Big Data, 2019.
[12] E. Bernhardsson, "Annoy: Approximate Nearest Neighbors in C++/Python," https://github.com/spotify/annoy, 2018.
[13] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "DeepDB: Learn from Data, not from Queries!" PVLDB, vol. 13, no. 7, pp. 992-1005, 2020.
[14] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in ICLR, 2017.
[15] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," in ICLR, 2018.
[16] J. Sun and G. Li, "An End-to-End Learning-based Cost Estimator," PVLDB, vol. 13, no. 3, pp. 307-319, 2019.
[17] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning," in ICML, 2016.
[18] Trino Benchmarks, "TPC-H on Trino Performance Study," https://trino.io/blog/2022/02/01/trino-fest-2022-tpch.html, 2022.
[19] Transaction Processing Performance Council, "TPC-H Benchmark Specification," http://www.tpc.org/tpch/, 2024.
[20] X. Zhou, J. Sun, G. Li, and J. Feng, "Query Performance Prediction for Concurrent Queries using Graph Embedding," PVLDB, vol. 13, no. 9, pp. 1416-1428, 2020.
[21] Transaction Processing Performance Council, "TPC-DS Benchmark Specification," http://www.tpc.org/tpcds/, 2024.
[22] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in Proc. 22nd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2016, pp. 785-794.
[23] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, "Performance Prediction for Concurrent Database Workloads," in Proc. ACM SIGMOD Int. Conf. on Management of Data, 2011, pp. 337-348.
[24] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh, "Flow-Loss: Learning Cardinality Estimates That Matter," PVLDB, vol. 14, no. 11, pp. 2019-2032, 2021.
[25] Trino Project, "Trino Gateway," https://github.com/trinodb/trino-gateway, 2024.
[26] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based Query Performance Modeling and Prediction," in Proc. IEEE 28th Int. Conf. on Data Engineering (ICDE), 2012, pp. 390-401. doi:10.1109/ICDE.2012.64.
[27] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri, "Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques," PVLDB, vol. 5, no. 11, pp. 1555-1566, 2012. doi:10.14778/2350229.2350269.
[28] Y. Zhao, G. Cong, J. Shi, and C. Miao, "QueryFormer: A Tree Transformer Model for Query Plan Representation," PVLDB, 2022. doi:10.14778/3529337.3529349.
[29] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," IEEE Trans. Pattern Anal. Mach. Intell., 2018. doi:10.1109/TPAMI.2018.2889473.
[30] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: SQL on Everything," in Proc. IEEE 35th Int. Conf. on Data Engineering (ICDE), 2019, pp. 1802-1813. doi:10.1109/ICDE.2019.00196.

MAHENDRAN VASAGAM (Member, IEEE, ORCID: 0009-0009-9382-1409) is a Staff Software Engineer at Slack and Independent Researcher with 15+ years of industry experience in distributed query engine infrastructure serving petabyte-scale data workloads. His technical expertise encompasses distributed systems architecture, machine learning-based resource optimization, cloud security, email security, and data-driven security platforms.

ASHOK KUMAR (Member, IEEE, ORCID: 0009-0002-0869-9007) is a Staff Software Engineer at Walmart and an Independent Researcher with over a decade of expertise in large-scale distributed systems and web performance. His technical focus encompasses cloud-distributed infrastructure, machine learning, and the design of high-performance architectures for data-intensive platforms. He has led mission-critical initiatives in cloud-scale architectures, including intelligent cloud infrastructure and performance optimization for global e-commerce systems.

ANSHUL GARG (Member, IEEE, ORCID: 0009-0002-8954-0523) is a Product Leader and Author who transforms complex business challenges into elegant digital product solutions. With 19+ years at industry leading organizations including Amazon, Capital One, and Nationwide Insurance, he has delivered over $100 million in annual business value through strategic product initiatives. Holding an MBA from Indian Institute of Management, Mumbai and an engineering degree from National Institute of Technology Jamshedpur, Anshul combines strategic vision with technical depth, specializing in launching AI driven zero-to-one products, Statistical modeling and Big Data analytics.