Graph Transformers for Query Plan Representation: Potentials and Challenges

Chenghao Lyu, Guillaume Lachaud, Gabriel Lozano, Yanlei Diao

Graph Transformers for Query Plan Representation: Potentials and Challenges

Chenghao Lyu*
University of Massachusetts, Amherst
chenghao@cs.umass.edu

Guillaume Lachaud*
Gabriel Lozano
Ecole Polytechnique
guillaume.lachaud@polytechnique.edu
lozanopinzon@lix.polytechnique.fr

Yanlei Diao
Ecole Polytechnique
University of Massachusetts, Amherst
yanlei.diao@polytechnique.edu

ABSTRACT

Query Plan Representation (QPR) is central to workload modeling, with various deep-learning based architectures proposed in the literature. Our work is motivated by two key observations: (i) the research community still lacks clarity on which model, if any, best suits the QPR problem; and (ii) while transformers have revolutionized many fields, their potential for QPR remains largely underexplored. This study examines the strengths and challenges of Graph Transformers for QPR. We introduce a new taxonomy that unifies deep-learning based QPR techniques along key design axes. Our benchmark analysis of common QPR architectures reveals that Graph Transformer Networks (GTNs) consistently outperform alternatives, but can degrade under limited training data. To address this, we propose novel data augmentation techniques to enhance training diversity and refine GTN architectures by replacing ineffective language-model-inspired components with techniques better suited for query plans. Evaluation on JOB, TPC-H, and TPC-DS benchmarks shows that with sufficient training data, enhanced GTNs outperform existing models for capturing complex queries (JOB Full and TPC-DS) and enable the query embedder trained on TPC-DS to generalize to TPC-H queries out of the box.

PVLDB Reference Format:
Chenghao Lyu, Guillaume Lachaud, Gabriel Lozano, and Yanlei Diao. Graph Transformers for Query Plan Representation: Potentials and Challenges. PVLDB, 18(13): 5716 - 5730, 2025.
doi:10.14778/3773731.3773745

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/udao-moo/udao-spark-modeling.

1 INTRODUCTION

Machine learning for data analytics systems has become a central topic in both the research community and industry. In this evolving landscape, a key challenge is modeling the workload in a data analytics system. While early solutions often relied on a *black-box approach*, treating the query workload as an opaque function [40, 78, 89, 98, 100, 101], more recent methods adopt a *white-box approach*, leveraging explicit query plan representations [9, 25, 47, 48, 52, 53, 55, 87, 91, 102]. This paper focuses on the white-box approach as it provides richer insights into query properties, enabling broader applications such as cardinality estimation [38, 61, 67, 86, 102]; query performance prediction [25, 55, 74, 87, 102]; knob tuning and cost-performance optimization [47, 48, 91]; and query plan improvement [3, 9, 10, 52, 92, 105].

At its core, the query plan representation (QPR) problem can be divided into two questions: how to encode each node of a query plan (*node encoding*); and how to encode the structure of the plan (*structure encoding*) to capture the interactions between nodes and aggregate all the plan information into a final plan representation (a single numerical vector for the query plan). Finally, the plan representation is passed to a downstream task. For example, the task of cardinality estimation [38, 61, 67, 86, 102] or query latency estimation [47, 48, 87, 91] can be modeled by a regressor using the plan representation and other relevant information (e.g., data properties and system knobs) to make the final prediction.

While many techniques exist for query plan representation, our work in this paper was motivated by two key observations:

Lacking Understanding of Model Differences. Existing research has explored a range of plan representation techniques, from Tree Recurrent Networks [25, 55] to Tree LSTMs [74, 95, 96] to Tree Convolutional Networks [3, 10, 11, 52, 54, 92, 105] to Graph Convolutional Networks [87, 91]. However, the community lacks a clear understanding of which model, if any, is the most suitable for query plan representation. The latest benchmark paper [103] offers two findings: (*i*) Node encoding plays a crucial role in model performance, and including a rich set of features—such as predicate encoding and cardinality estimates—is a strong default choice. (*ii*) Structural encoding has a relatively minor impact on performance. However, an important question remains unanswered: why does structural encoding contribute so little to performance, especially for complex query plans? Is it really true that the different methods of capturing node interactions and aggregating them into the final representation lead to similar performance outcomes? This gap in understanding forms a central motivation for our work.

Unproven Potential of Transformers. Concurrently, transformer architectures have revolutionized multiple fields including natural language processing (NLP) [7, 12, 80], computer vision [14, 44], and molecular modeling [31, 68]. In all of these domains, transformers have demonstrated remarkable success and superior performance over traditional architectures such as Convolutional Neural Networks (CNNs) and Graph Neural Networks (GNNs). Given transformers' ability to capture long-range dependencies and contextual interactions, recent studies have begun exploring their application

*The authors contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 18, No. 13 ISSN 2150-8097.
doi:10.14778/3773731.3773745

to query plan representation, including Tree Transformers [102] and Graph Transformers [47, 48]. However, a fundamental question remains unaddressed: Are transformers—originally designed for language modeling—truly well-suited for query plan representation? While their self-attention mechanism could offer advantages in capturing hierarchical structures, there is still no in-depth analysis of whether they outperform existing models in the QPR domain. Understanding transformers' strengths, limitations, and practical applicability to QPR remains an open question.

Our study in this paper unfolds the potentials and challenges of graph transformers for query plan representation. We begin with a new taxonomy of deep-learning based QPR techniques, which unites all of them via several key design axes. Then we perform a benchmark analysis of these techniques using the JOB, TPC-H, and TPC-DS workloads. Our analysis extensively covers the major architectures used in the literature, including Tree Recurrent Networks (TRN) [25, 55], Tree LSTM (TSLTM) [74, 95, 96], Tree CNN (TCNN) [3, 10, 11, 52, 54, 92, 105], Tree Transformer Network (TTN) [102], Graph Convolutional Network (GCN) [87, 91], and Graph Transformer Network (GTN) [47, 48]. Our analysis reveals that for structure encoding, model differences do exist and *Graph Transformer Networks (GTNs) provide stable, superior performance over other alternatives*. However, when the test queries change from the same distribution as the training data (in-distribution) to new plan structures never seen during training (out-of-distribution), all models exhibit a severe loss of performance. The performance gap between in-distribution (ID) and out-of-distribution (OOD) queries poses a major challenge to all models and blocks the GTNs from reaching its full potential. To reduce the gap between ID and OOD queries, we introduce two novel techniques to augment GTNs:

Training Data Augmentation. Our work is inspired by recent machine learning practices for domain-specific models [4, 75], where limited training data can be enhanced with carefully crafted synthetic data to boost model performance. However, the challenge in our setting is generating synthetic queries that fill the "gaps" in the embedding space of GTNs—areas where the model struggles to generalize due to limited training data, (a.k.a., *epistemic uncertainty* of the model). This is a non-trivial problem because no inverse mapping from the vast, sparse regions of the embedding space to valid SQL queries exists. In our work, we explore two approaches: (i) using LLMs to generate synthetic queries that are *maximally distinct* from the existing training set; (ii) using database technology to generate synthetic, random queries with varied join structures and predicates, via sampling the schema and column statistics.

Enhanced GTN Architectures for Query Plans. We then dive deeper into understanding whether the Transformer architecture [80], initially designed for the language technology, is the best fit for SQL query plans. Our analysis reveals that (i) the transformer's *positional encodings*, crucial for understanding languages, bring noise to query plan representation; (ii) the transformer's *self-attention* mechanism, the dominant form of capturing interactions among different units (e.g., words in the language) is not an ideal fit for capturing interactions between operators (nodes) in query plans. Therefore, we propose novel GTN architectures to best encode query plans, i.e., removing position encodings and using learnable attentions among operators.

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

- We introduce a new taxonomy that unites all major deep learning architectures for query plan representation (QPR) and positions each technique precisely in the design space, enabling a clear understanding of how they relate to each other (Section 3).
- We perform a benchmark analysis of common QPR architectures, showing that GTNs provide stable, superior performance over other alternatives (Section 4).
- We propose new data augmentation methods to diversify training data and reduce the gap with unseen test queries, hence unlocking the potential of advanced models (Section 5).
- We then propose enhanced GTN architectures that replace ineffective language-model-inspired components with new techniques specifically tailored to query plans. We further reveal that enhanced GTNs better suit complex, long-running queries than GCN, TTN, etc., due to **context-aware attention** and **effective information flow** within query plans (Section 6).

We evaluated modeling techniques for out-of-the-box performance (i.e., test queries are seen the first time), using Spark SQL on JOB, TPC-H, and TPC-DS. (1) As existing training sets for these workloads are insufficient for OOD, our data augmentation techniques generate diverse training data and significantly improve OOD performance. (2) Under sufficient training data (JOB Light), our enhanced GTN can *significantly reduce the gap between ID and OOD queries*. (3) For complex queries (JOB Full and TPC-DS), our enhanced GTNs outperform GCN, TTN, etc. due to *context-aware attention and effective information flow*. (4) For TPC-H, treated as a hold-out set for transfer learning, our results show that the GTN query embedder trained on TPC-DS *captures TPC-H queries out-of-the-box* (*zero-shot learning*). Moreover, with these representations, only a small fraction of TPC-H observations is needed to train an effective regressor for query latency prediction (*few-shot learning*).

A final contribution of our paper is high-quality training sets for JOB and TPC-DS: (i) Our Spark SQL query sets exhibit lower performance variance than prior PostgreSQL datasets [103] due to better resource isolation, offering a more reliable testbed for model evaluation. (ii) We provide 47K synthetic, random training queries (1039 join signatures) for JOB, and 50K parameterized TPC-DS queries augmented with 55K synthetic queries (5518 join structures).

2 RELATED WORK

Workload modeling is a central topic in ML for analytics systems.

Black-box approaches. These methods treat queries or workloads as opaque functions that are refined via iterative exploration, e.g., using Gaussian Process (GP) or Bayesian Optimization (BO). OtterTune [78], iTuned [15], and OpAdviser [101] build workload-specific GP models, while PACE [98] builds separate GP models to optimize query performance over multiple rounds. GPTuner [33] harnesses DBMS manuals with a Large Language Model to guide knob tuning. ResTune [99] accelerates GP training through meta-learning, and Tuneful [19] narrows the search to influential knobs. LOCAT [89] adapts GP tuning to data size changes in Spark, whereas Li et al. [40] apply BO with a meta-learner for periodic Spark jobs. Finally, OnlineTuner [100] uses contextual BO with "safe" exploration policies. Although these black-box approaches are effective in many settings, they do not leverage explicit query plan representations that yield richer query insights and broader applications.

Cardinality estimation. Accurate cardinality estimation depends heavily on how query plans are represented, particularly in capturing operator information like table joins, filter conditions, and their interactions. Traditional methods [23, 37, 39, 63, 64] that rely on histograms or sample-based estimation often struggle with complex data patterns. Data-driven methods such as CloudCard [84], NeuroCard [93, 94], BayesCard [88], DeepDB [26], FLAT [106], and UAE [85] learn data distributions operator by operator through neural or Bayesian networks, yet they tend to be restricted to certain query patterns and seldom generalize to broader ML for data analytics (ML4DA) tasks. The multi-set convolution network (MSCN) approaches [32, 60, 61, 86] represent query plans as flat vectors by concatenating table, join, and predicate features, then averaging over these elements. FACE [83], Fauce [42], JGMP [67] and ALECE [38] apply similar flat-vector encodings with more elaborate feature engineering for predicates and joins. While simple, these flat representations oversimplified structural information and can yield suboptimal performance on complex queries. Addressing this, QueryFormer [102] introduces a Tree Transformer Network to embed query plans more effectively, and can be extended to other ML4DA tasks such as index selection and query optimization.

Performance prediction and cost-performance optimization. Effective performance tuning and cost-performance optimization critically rely on accurate cost-performance models, which in turn depend on a query plan's representation. Heuristic cost models [29, 36, 66, 71, 82] approximate performance through simplified closed-form or operator-level rules. Recent learning-based approaches closely encode the query plan structure to predict query latency, exploring a wide range of architectures: Tree Recurrent Networks [25, 55], Tree LSTMs [74], Tree Transformer Networks [41, 102] and Graph Convolutional Networks [87]. Similar ideas have been adapted for knob tuning [72, 91] and cost-performance optimization [47, 48], incorporating Graph Transformer Networks [16] to capture more complex structural dependencies. Despite the variety of models used, the differences among them are not well understood. In particular, the latest benchmark paper [103] concluded that the model differences are not significant, and our paper will revisit this issue with an in-depth analysis.

Query optimization. Query optimization tasks in traditional databases—including plan improvement, index selection, and view selection—also benefit from robust query plan representations. BAO [52], NEO [54], Lero [105], LEON [10], Balsa [92], BASE [11], AutoSteer [3], Balsa [92], RTOS [96], HybridQO [95], ROGER [9] and PerfGuard [28] embed query plans as fixed-size vectors through tree-based or graph-based encoders to guide plan refinement. Meanwhile, AIMeetsAI [13] and AVGDL [97] flatten query plans for index and view selection, respectively. These prototypes are primarily developed in PostgreSQL to validate their feasibility and performance. In big data systems like Spark, however, there is less emphasis on indexes and query plans are mainly improved during execution with the support of adaptive query execution [18, 47, 90].

3 PROBLEM AND SYSTEM OVERVIEW

In this section, we formally define the problem of query plan representation, further divide it into several technical questions, and finally present a comprehensive taxonomy of existing solutions.

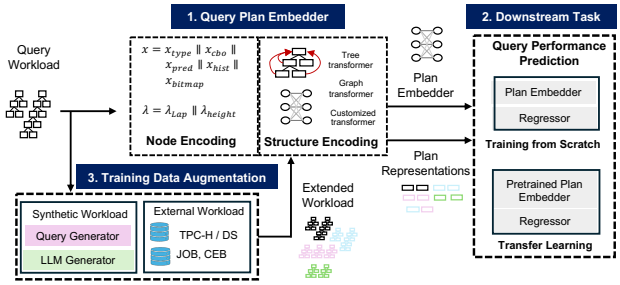


Figure 1: System overview. Node encodings x are defined in Section 3.2, positional encodings λ in Section 6.1.

3.1 Problem Statement

Let $G = (V, E)$ be a query plan, where V is the set of relational operators and E the set of edges between them. In our work, we consider a query plan as a graph because in big data systems like Spark, an operator's output can be reused by multiple downstream operators (e.g., a scan operator serving as input for a self-join), resulting in a directed acyclic graph instead of a strict tree structure.

Our task is divided in two parts: (i) learn a representation of the query plan $z = \phi(G)$, where $\phi : G \mapsto \mathbb{R}^d$, called a **plan embedder**; and (ii) use the representation to assist in a **downstream task**. In a database or big data system, typical downstream tasks include cardinality estimation, query performance prediction, cost-performance optimization, and query plan refinement. Among them, the latest benchmark paper [103] shows that the most challenging task, from a query plan representation perspective, is **query performance prediction (QPP)**. It is because this task needs to capture all the operator behaviors, interactions among themselves, as well as their interactions with the system environment (e.g., resources available and other system knobs) to predict query latency. For this reason, our paper studies the query plan representation problem in the setting of QPP as the downstream task.

Figure 1 gives an overview of our system, including (1) a query plan embedder, (2) prediction for the downstream task, QPP, and (3) training data augmentation, which is optional but can boost model performance. We review key techniques for these steps below.

3.2 Node Encoding

The goal of the plan embedder ϕ is to generate a vector $z \in \mathbb{R}^d$ that is representative of the query plan G. To begin with, we first derive a numerical representation x of each node $v \in V$. This representation, called a **node (operator) embedding**, can be obtained by concatenating several features, e.g.,

$$x = [x_{type} \| x_{cbo} \| x_{pred} \| x_{hist} \| x_{bitmap}]$$

where $\|$ is the concatenation operator. We now describe the different features, which align with state-of-the-art approaches [102, 103].

(1) *type.* We employ one-hot encoding to represent the operator type. Each operator type is mapped to a fixed-length dummy vector.

(2) *cbo.* This feature captures the cardinality estimates from the Cost-Based Optimizer (CBO), including the estimated number of input records and their total size (in bytes). Both quantities are treated as numeric values, and augmented by their logarithmic transformations to better capture their distributions.

Table 1: Taxonomy of query plan embedders

Method	Positional encoding	Neighborhood	Update mechanism	Information flow	Final representation
TRN [55]	-	direct children	neural network	recurrent	root node representation
TLSTM [74]	-	direct children	LSTM gates	recurrent	root node representation
TCNN [54]	-	direct children	tree convolution	layer-wise	maximum pooling
TTN [102]	height encoding	all descendants	self-attention	layer-wise	super node
GCN [87]	-	direct children	graph convolution	layer-wise	root node representation
GTN [47]	Laplacian eigenvectors	direct children	self-attention	layer-wise	average pooling
LSTM-TTN [41]	-	all descendants (TTN)	LSTM & self-attention	recurrent & layer-wise	MLP

(3) *pred*. We embed predicates using word2vec [58] trained on training set predicates. We tokenize, remove stop words and special characters, replace math symbols with reserved words, then average the embeddings of remaining tokens.

(4) *hist*. We use histogram encoding from [102] for cardinality estimation. Database systems (e.g., Spark SQL) maintain equi-depth histograms on columns. We re-bin all column histograms into 50 buckets and encode each predicate as a 50-dimensional vector, where each value (0-1) indicates how well the bucket satisfies the predicate. Multiple predicates on the same column are combined (via union or intersection) based on their logical relationship.

(5) *bitmap*. We add cardinality estimation via bitmap encoding [32, 74, 102]. We randomly sample 1000 tuples from each table and construct a 1000-dimensional bit vector for each operator's column predicates, where each bit indicates whether the corresponding tuple satisfies the predicates (1) or not (0).

Remarks. Some of the above features may be unavailable in production (e.g., missing column statistics or unavailable samples due to implementation constraints). Nonetheless, our work seeks to characterize the "upper bound" of model performance by incorporating all techniques proposed in the literature, aligning with ongoing industry efforts to enrich statistics for optimization.

3.3 Structure Encoding

The next step in the plan embedder ϕ is to generate a vector $z \in \mathbb{R}^d$, called a **plan embedding**, representative of the entire query plan G. To this end, most plan embedders capture the interactions between relevant nodes and then aggregate all the information into a final plan representation z. We outline below the popular deep learning plan embedder architectures for doing so.

Tree Recurrent Network (TRN) [25, 55] builds separate neural units for different operators and connects them along the structure of the query plan. Information flows from child operators to their parent, enabling the model to capture operator-level correlations and produce a final plan embedding at the root operator.

Tree LSTM (TLSTM) [74, 95, 96] represents the entire query using the hidden state of its root node, which is recursively updated along the tree structure via a tree variant of the LSTM network [76].

Tree CNN (TCNN) [3, 10, 11, 52, 54, 92, 105] learns parent and children relations in the query plan and generate the plan representation by (1) binarizing the plan tree, (2) sliding a set of shared triangular filters (parent, left child, right child) over the tree, and (3) aggregating operator features via dynamic pooling.

Tree Transformer Network (TTN), QueryFormer [102], generalizes transformers to tree-structured query plans by encoding operator height and computing attention between nodes and their children. A "super" node links all operators to provide a global plan embedding. A variant (**LSTM-TTN**) [41] removes height encoding and initializes operator embeddings using LSTM based on topological order, then applies similar attention. The final representation averages node features through a fully connected network.

Graph Convolutional Network (GCN) [87, 91] employs convolutional operations on graph-structured data to aggregate neighborhood features and capture both local and multi-hop dependencies in query plans. By stacking multiple layers, it integrates information across the plan, yielding a final embedding for downstream tasks.

Graph Transformer Network (GTN) [9, 47, 48] extends the transformer to graph data by combining Laplacian positional encodings of nodes, self-attention among nodes, and multi-head attention. These methods enable the model to capture long-range operator dependencies, resulting in a richer query plan representation.

Taxonomy of Plan Embedder Architectures. We next propose a comprehensive taxonomy to enable a better understanding of the relationships between the embedder models. In essence, deep plan embedders use multiple layers to compute hidden representations for each node in the graph, then apply a final layer to extract a vector out of the graph. Each layer in a query embedder can be interpreted along the following five axes, as summarized in Table 1.

Positional encoding. In the original transformer architecture [80], positional encodings enrich token features, ensuring that the same token has different features depending on its position in a sentence. Transformers-based architectures (TTN, GTN) follow the same practice. TTN encodes the position of the node as the longest path of the node to one of the leaves [102], while GTN encodes the position using the Laplacian eigenvectors of the graph [16].

Neighborhood. Since operators only impact downstream operators in query plans, embedders fall into two groups: those transferring information to direct children (TRN, TCNN, TLSTM) versus all descendants (TTN, LSTM-TTN). While graph-based approaches like GTN can theoretically attend to all nodes, they use only direct children to reduce computational complexity and leverage graph structure [16]. Figure 2(a) shows the neighborhood (green nodes) used by GTN to update the blue node. The neighborhood per layer, combined with the number of layers, determines the information flow (which we will explain shortly).

Update mechanism. Update mechanisms are functions that take as inputs a node and its neighbors (based on the neighborhood definition above), and output a new representation. They can be divided into two main classes: recurrent (TRN, TLSTM, LSTM-TTN) and non-recurrent (TCNN, TTN, GCN, GTN) approaches.

In recurrent approaches, the inputs are processed *sequentially*. When the last node of the plan is processed, the model weights are supposed to have captured knowledge of all the nodes in the plan.

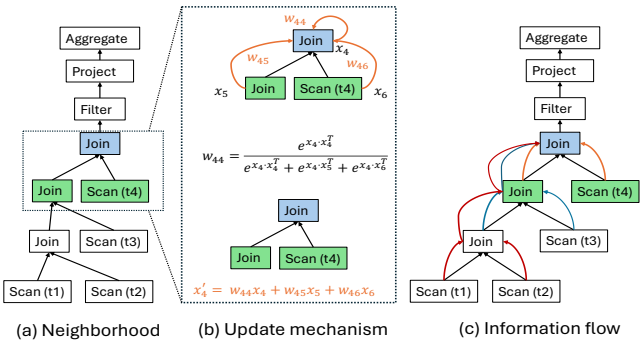


Figure 2: Example of information flow in a GTN.

For example, LSTM-based approaches (TLSTM, first part of LSTM-TTN) use several gates (matrix multiplications) to produce rich representations for each node [27]. While recurrent-based models can theoretically capture long-range dependencies [27], in practice, they suffer from the vanishing and exploding gradient problem, which blocks the flow of information [62].

In contrast, non-recurrent approaches process all the nodes *in parallel*, and use the model weights to detect the same patterns on different nodes. They can be further divided into two families.

Convolution (TCNN, GCN) approaches are **context-agnostic**: convolutional filters apply matrix multiplication and non-linear activation [34]. The same filter weights are reapplied across all nodes to detect similar patterns, sharing learned features throughout the plan irrespective of node location.

Attention (TTN, GTN) approaches are **context-sensitive** and subsume convolutional filters, using context (e.g., neighbors, node position) to compute representations. The dominant form is *self-attention*, popularized by transformers [80], which computes similarity scores between nodes to weight how much each neighbor contributes to a node's updated representation. Figure 2(b) shows an example of self-attention in a query graph. Self-attention will be discussed in detail in Section 6.

Information flow across layers. For non-recurrent models (TTN, TCNN, GCN, GTN), neighborhood and layer count determine information flow distance. For example, a GTN with four layers attending only to children can transmit information at most four hops away. Figure 2(c) shows that after 3 layers, the selected node receives information from all ancestors (red arrows). However, adding more layers to capture longer range dependencies causes oversmoothing, where nodes from different classes become indistinguishable [8].

Final representation. After passing through the model layers, node representations are aggregated to form the plan representation. (1) Recurrent approaches (TRN, TLSTM) use the last processed node, which contains information from the entire plan [55, 74]. (2) Non-recurrent approaches use pooling (TCNN, GTN)—averaging node representations or selecting salient features—or advanced methods (TTN) using super nodes that attend to all other nodes [21], or MLPs for final representation (LSTM-TTN).

3.4 Effective Training Strategies

In this section, we consider a range of training strategies to effectively train the plan embedder with its downstream task.

Training from Scratch. The plan embedder outputs a vector representation of the query plan, $z = \phi(G)$. Take QPP as a downstream task. The plan representation z is passed to a regressor, implemented as a multilayer perceptron (MLP), that takes the plan z, a system configuration θ consisting of all tuneable parameters that control resources and runtime behaviors, and a data encoding α capturing the input data characteristics. The task is to learn the prediction $y = \Psi(z, \theta, \alpha)$. Given a training set of query plans $\{G_i\}$ with their latency observations $\{l_i\}$, we can train the plan embedder $\phi(G)$ and regressor $\Psi(z, \theta, \alpha)$ jointly through backpropagation. This follows the training strategy used in foundation models [5].

Data Augmentation. There are cases when an application does not have sufficient data for training the plan embedder with the regressor. Consider two settings in evaluating the trained models: (i) **In-distribution** (ID), where the test queries are drawn from the same distribution as the training set, e.g., with modest changes of the predicates; (ii) **Out-of-distribution** (OOD), where the test queries are drastically different from the training set, e.g., a new query structure that was never seen in training. A trained model may incur a significant increase in error from ID to OOD evaluation, as the training data does not allow it to generalize to unseen query structures. To tackle this issue, our work provides several data augmentation methods (Section 5) to enrich the training set with synthetic queries that diversify the predicates and query structures, offering the potential to reduce the OOD error.

Transfer Learning. In the third training strategy, which is the ideal but is the hardest to achieve, we hope that for the applications without sufficient training data, a pre-trained model from other domains will work "out-of-the-box", meaning that it understands the query plans in this new domain despite being trained in other domains (*zero-shot learning*). Further, the derived plan representations can be used to support the downstream task with limited observational data (*few-shot learning*). Our approach differs from existing zero-shot learning work [25] in that: (i) we use much richer node encodings; (ii) we use a GTN approach for structure encoding, whereas the prior work used a TRN approach. In later evaluation, we reserve TPC-H as this new domain and show the effectiveness of transferring the model trained on TPC-DS to TPC-H.

4 INITIAL BENCHMARK ANALYSIS

In this section, we present an extensive benchmark analysis of model structures and operator encoding methods using the QPP task, highlighting new findings that differ from prior work [103].

4.1 Experimental Setup

We use the Join Order Benchmark (JOB) [35], TPC-H [2], and TPC-DS [1] benchmarks in our analysis. In addition, we use Spark SQL as the execution engine—as we shall show shortly, resource isolation in Spark allows us to collect datasets with less variance in query runtime than in the previous model benchmark study using PostreSQL [103], hence enabling more stable model performance results. More specifically, the training set and test sets, for both **in-distribution** (ID) and **out-of-distribution** (OOD) evaluation are collected for each benchmark as follows.

JOB [35] is derived from an IMDB dataset of 2.5 million movie titles and includes 113 multi-join queries (up 16 joins). To enable training, Kipf et al.[32] generated three subsets: `Train` with 6 tables, 100,000 queries with up to 2 joins; `Synthetic` with 5,000 queries of up to 2 joins; and `Light` with a subset of 70 JOB queries of up to

4 joins, using star joins on the *title* table and simplified predicates. We divide `Train` 90%/10% into training/validation sets. We then test on `Synthetic` for ID performance; on `Light` for OOD performance when train and test queries have somewhat comparable complexity (except from 2 to 4 joins), denoted as OOD1; and on the full 113 queries, `JOB Full`, for OOD performance when some of the test queries are significantly more complex than `Train` (up to 16 joins and using complex predicates), denoted as OOD2.

TPC-H and TPC-DS. TPC-H contains 22 queries with relatively simpler joins and aggregations, whereas TPC-DS has 102 queries involving more complex multi-joins, window operations, subqueries, and aggregations. Following previous studies [47, 55, 102, 103], we use each TPC-H or TPC-DS query as a template and generate 50,000 parameterized queries per benchmark. The ID experiments use stratified sampling so each query template appears in both training/validation (80%/10%) and test (10%) splits. OOD experiments partition the 22 TPC-H and 102 TPC-DS templates into 10 folds each, holding one fold out as the test set and using remaining folds for training and validation, with results averaged over all 10 folds.

Modeling Target. Across three benchmarks, to run each query, we randomly sample a Spark configuration using Latin Hypercube Sampling [57] across 19 parameters [47] (denoted as θ in the previous section), which control resource allocation, runtime context, and query optimization. Our goal is to predict the end-to-end latency of these queries based on the logical query plan, the chosen Spark configuration, and data statistics. We compare different plan embedder approaches via a multilayer perceptron (MLP) regressor that predicts query latency based on the plan representation and other parameters stated above (see Appendix A.1 for details).

Metrics. Prediction accuracy is measured by (i) weighted mean absolute percentage error (**WMAPE**), which weights the APE of each query by the ratio of its latency over the entire latency distribution and is our actual *training loss*; (ii) the 50th and 90th percentiles of weighted absolute percentage error (**WAPE**), emphasizing long-running queries; and (iii) **Q-Error** [59] defined as the maximum ratio error between the true y and predicted values $\hat{y}$, $\max(\frac{y}{\hat{y}}, \frac{\hat{y}}{y})$, with reported values of its mean and the percentiles (P50, P90, P99, and max) for capturing the tail performance. The Q-Error enables a direct comparison with the previous Q-Error based study [103].

Hardware. All experiments were conducted on two 6-node Spark 3.5.0 clusters, where each node runs CentOS and has two 16-core Intel Xeon Gold 6130 processors, 768 GB RAM, and 100 Gbps Ethernet. 20 nodes of the above setup were used to train models.

4.2 Model Comparison

We first benchmark structure encoding methods on JOB, TPC-H, and TPC-DS, while fixing the node encoding method using all the features in Section 3.2. We compare 7 recent query embedding methods (Section 3.3): TRN [55], TLSTM [74], TCNN [54], TTN(QF) [102], LSTM-TTN [41] (which is our best-effort implementation as the original code is not publicly available), GCN [87], and GTN [47].

In-Distribution Performance. We first examine in-distribution (ID) performance in Table 2 , where the test queries are drawn from the same distribution as the training set.

Model Comparison. We first consider the WAPE metrics for model comparison as they directly relate to the training loss. First, TRN [55] and LSTM-TTN [41] exhibit consistently worse performance than the others. For example, on TPC-DS, TRN's WMAPE is 0.467 and LSTM-TTN's WMAPE is 0.419, while other methods are in the range of 0.163-1.180. Similar significant gaps can be observed for the P90 WAPE metric. Second, TLSTM's performance is not stable: it exhibits poor performance for JOB, but better performance for TPC-H and TPC-DS. Third, the remaining four methods, TCNN, TTN, GCN, and GTN are closer in performance, with GTN consistently outperforming its peers: GTN dominates TCNN for 20 out of 21 metrics reported while being very close for the one metric without dominant performance. The same observation could be made for GTN's dominance over GCN. GTN and TTN are even closer in performance, but in most cases, GTN provides the best performance and exhibits close approximation in a few other cases when TTN is better. As such, we see that **in the ID setting, GTN provides the overall best performance across the LSTM, convolutional, and transformer families** (**R1**, denoting major result 1).

Workload Complexity. We observe that workload complexity varies considerably. TPC-DS yields the highest WMAPE (0.163-0.467), hence the hardest, followed by TPC-H (WMAPE 0.111-0.360) and JOB (WMAPE 0.057-0.251).

Latency distributions are heavily skewed, with GTN's WMAPE double its P50 WAPE due to a small number of long-running queries.

Out-of-Distribution (OOD) Performance. We report OOD performance in Table 3, where test queries are drawn from the templates not seen in training. Here, we focus on the 4 leading models based on the ID performance: TCNN, TTN, GCN, and GTN.

Large Gaps between ID and OOD Performance. First, for TPC-H, all models exhibit severe performance degradation. The root cause is that TPC-H's OOD results are statistically unreliable due to having only 22 query templates. This entails highly unrepresentative test sets, i.e., the two hold-out test templates are poorly represented by the other 20 templates in training, hence large errors (e.g., WMAPE $\geq$ 0.699). For this reason, we omit TPC-H here and reserve it exclusively for a later study using transfer learning.

For other workloads with more templates in training, we observe consistently that **all models degrade OOD performance significantly from the respective ID performance** (**R2**). For JOB Light, the best ID WMAPE (0.057) and P90 WAPE (0.114) degrade by 2.5–3.2× to 0.140 and 0.363 in OOD performance. For JOB Full, the best ID WMAPE and P90 WAPE degrade by a factor of 6.4–6.7× in OOD performance. This larger degration is due to the fact that JOB Full includes much more complex patterns, including up to 16 joins and complex predicates, that do not exist in training data. Finally, for TPC-DS, the best ID WMAPE (0.163) and P90 WAPE (0.336) nearly triple, indicating that 90% of 102 TPC-DS tempates are not enough for generalizing to the 10% unseen templates.

Model Comparison. **GTN has the best OOD performance, in WMAPE, on the two complex workloads, i.e., JOB Full and TPC-DS** (**R3**). Other models suffer from worse performance for at least one of the two workloads: TCNN and TTN lose performance on TPC-DS, while GCN loses on JOB Full.

Performance Stability. We further compare our Q-Error metrics (obtained from Spark SQL queries) with those reported on PostgreSQL in a recent study [103]. For the JOB benchmark, TTN [102] (from the authors of [103]) achieves a P99 Q-Error of 1.417 (ID) and

Table 2: In-distribution (ID) performance of query embedding methods on JOB Synthetic, TPC-H, and TPC-DS datasets.

	Weighted Absolute Percentage Error (WAPE)									Q-Error												
	JOB Synthetic (ID)			TPC-H (ID)			TPC-DS (ID)			JOB Synthetic (ID)				TPC-H (ID)				TPC-DS (ID)			
	Mean	P50	P90	Mean	P50	P90	Mean	P50	P90	P50	P90	P99	Max	P50	P90	P99	Max	P50	P90	P99	Max
TRN	0.251	0.163	0.596	0.360	0.132	0.814	0.467	0.180	1.004	1.209	1.677	2.816	4.837	1.259	1.978	3.577	6.866	1.342	2.442	7.771	29.96
LSTM-TTN	0.185	0.077	0.489	0.235	0.088	0.584	0.419	0.157	0.930	1.090	1.473	2.329	3.779	1.151	1.860	3.076	9.848	1.323	2.530	7.792	41.51
TLSTM	0.195	0.085	0.522	**0.111**	**0.045**	0.246	0.164	0.074	0.337	1.099	1.508	2.337	4.655	1.078	**1.244**	1.745	4.868	1.130	**1.405**	2.085	15.60
TCNN	0.064	0.033	0.129	0.112	0.046	0.241	0.171	0.076	0.348	1.038	1.118	1.459	3.251	**1.076**	1.266	1.757	5.282	1.132	1.437	2.124	15.37
TTN(QF)	**0.057**	**0.031**	0.115	0.127	0.049	0.270	0.168	**0.072**	0.342	**1.035**	**1.104**	1.417	2.934	1.085	1.307	1.872	5.886	**1.126**	1.426	2.105	16.29
GCN	0.061	0.032	0.133	0.113	**0.045**	0.240	0.180	0.075	0.340	1.037	1.117	1.432	2.962	1.076	1.257	1.758	5.576	1.130	1.417	2.163	16.40
GTN	**0.057**	0.032	**0.114**	**0.111**	**0.045**	**0.235**	**0.163**	0.073	**0.336**	1.036	1.106	**1.413**	**2.844**	1.077	1.250	**1.725**	**4.648**	**1.126**	1.416	**2.074**	**10.01**

Table 3: Out-of-distribution (OOD) performance of query embedding methods on JOB Light, JOB Full, and TPC-DS datasets.

	Weighted Absolute Percentage Error (WAPE)									Q-Error												
	JOB Light (OOD)			JOB Full (OOD)			TPC-DS (OOD)			JOB Light (OOD)				JOB Full (OOD)				TPC-DS (OOD)			
	Mean	P50	P90	Mean	P50	P90	Mean	P50	P90	P50	P90	P99	Max	P50	P90	P99	Max	P50	P90	P99	Max
TCNN	0.141	0.069	**0.363**	0.418	0.300	**0.729**	0.542	0.237	1.296	1.087	1.380	1.898	**2.021**	1.501	2.381	6.309	7.973	1.385	2.816	6.005	22.41
TTN(QF)	**0.140**	0.072	0.367	0.391	**0.208**	0.782	0.520	0.249	1.227	1.089	**1.362**	1.842	2.156	**1.334**	2.283	**3.401**	**6.680**	1.432	3.298	7.638	26.27
GCN	0.148	**0.066**	0.395	0.436	0.280	0.933	0.479	0.201	1.159	**1.084**	1.364	1.929	2.191	1.519	2.470	5.117	10.89	1.349	2.551	**5.114**	**15.97**
GTN	0.151	0.090	0.387	**0.384**	0.228	0.824	**0.456**	**0.199**	**1.056**	1.098	1.377	**1.834**	2.138	1.354	**2.252**	4.072	9.722	**1.320**	**2.458**	5.203	19.70

1.842 (OOD) on our Spark SQL dataset, compared to 13.05 (ID) and 176.6 (OOD) on PostgreSQL—9.5× and 95.9× higher, respectively. The authors of [103] attributed their high errors to the high variance in query performance caused by resource contention. **Our stable performance demonstrates the value of our Spark SQL datasets (R4)**, with better resource isolation and less variance of query performance, hence more suitable for model performance analysis than the prior PostgreSQL datasets [103].

Node Encoding Comparison. We finally take GTN as the default model structure and compare different feature combinations for node encoding. We confirm that using all operator features is the most robust choice in both ID and OOD settings, consistent with the results reported in [103]. See details in Appendix A.2.

In summary, GTN stands out as the most robust model structure across different datasets and ID/OOD scenarios. Therefore, we focus on GTN in the following sections to devise new techniques.

5 DATA AUGMENTATION

As our analysis has shown, a salient challenge in query plan representation is that the model learned from the training set may fail to generalize to the test set, a phenomenon known as the *out-of-distribution* (OOD) behavior. In data analytics environments, the system may frequently encounter "unfamiliar" queries with remarkably different plan structures or operators from historical traces, and the model is unable to generalize. The literature on uncertainty [30, 50] refers to this type of model behavior as *epistemic* (knowledge or systematic) uncertainty, resulting from inadequate knowledge about the true underlying patterns (e.g., due to sparse regions in query modeling). Figure 3 illustrates a scenario where OOD queries exhibit high epistemic uncertainty, reflected by large WMAPEs. Queries with higher errors deviate significantly from the training data points (blue points), whereas queries with smaller errors show greater overlap with the training points. This example underscores the need for data-driven solutions to reduce these knowledge-related deficits.

In this section, we introduce approaches to augmenting existing training data with *synthetic queries* to reduce the discrepancy between in-distribution (ID) and OOD scenarios. We explore approaches that have the potential to fill the void of the model embedding space, enabling the model to better interpolate in the embedding space. This aligns with the latest ML practices, where under limited training data, models are boosted using carefully crafted synthetic data [4], but in the new setting of query embedding space.

5.1 Diverse Queries via LLMs

Large Language Models (LLMs) [65] have shown promise in natural language (NL) to SQL query generation and database tuning using manuals [20, 33, 43, 77]. Based on the hypothesis that LLMs understand the SQL syntax, we push them to the more challenging task of "*generating the most diverse queries from the existing queries*", e.g., from TPD-DS queries. This task is harder than NLtoSQL translation for several reasons: (i) NL2SQL typically involves pattern matching against the training data of LLMs. In contrast, generating diverse queries means not just reproducing *similar* queries but *intentionally varying* the structure and operators from the given TPC-DS query set. (ii) The LLM model must maintain syntactic correctness while generating queries that are novel and diverse. Therefore, it cannot be based on modifying existing queries slightly. (iii) Generating diverse yet meaningful queries also requires deep scheme comprehension beyond syntax. (iv) LLMs are prone to *mode collapse* [22, 70], meaning that a generative model fails to produce diverse outputs and instead generates repetitive, similar samples. Achieving true diversity in query structure—different joins, filters, aggregates, and subqueries—is a difficult optimization problem.

In our approach, we carefully instrument *prompt engineering* to guide ChatGPT-4.0 in generating diverse query templates to mitigate epistemic uncertainty. Given the complexity of simultaneously generating 1000's of distinct SQL templates, we adopt an iterative prompting procedure, as illustrated in Figure 4. Specifically, we first prompt ChatGPT-4.0 to produce an initial batch of 10 distinct query templates in Template Pattern Language (TPL) format, parameterized explicitly to ensure diversity and validity. Subsequently, we iteratively prompt for additional batches, each consisting of 10 distinct templates, until 1,000 templates are collected. During each

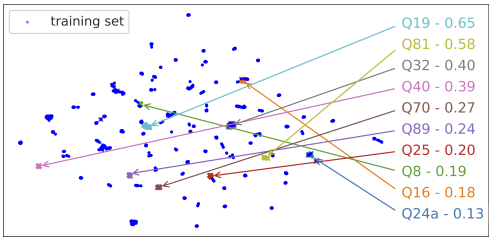


Figure 3: Illustration of epistemic uncertainty using OOD queries (colored by TPC-DS query templates). The legend shows the WMAPE per query template. Queries with large errors deviate significantly from training points, indicating insufficient training data coverage.

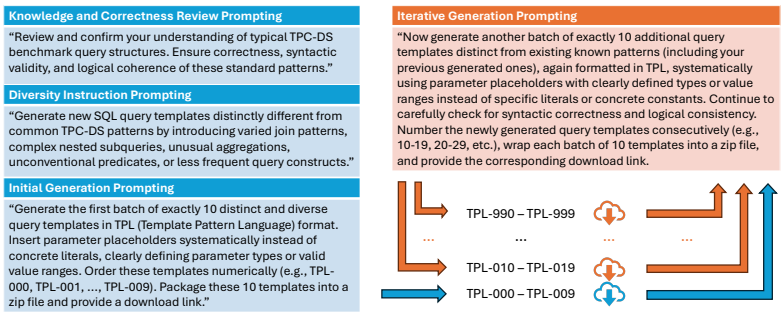


Figure 4: The iterative prompt engineering procedure to generate diverse TPC-DS query templates.

iteration, the newly generated templates are checked for syntactic correctness and logical consistency, with any invalid templates discarded or corrected. The post-filtering step removes templates that fail to execute, followed by systematic parameterization to yield a total of 6,000 executable queries for TPC-DS.

5.2 Synthetic, Random Query Generation

Given all of the abovementioned challenges to LLMs, we explore an alternative approach that generates random queries from a given schema. The idea is that if we randomly sample the query structures permitted by the schema, we have a better chance of covering the embedding space than any manually derived query set, like TPC-DS. Given the large query space to sample, our synthetic query generator focuses on the `select-from-where` block of SQL.

Join structure generation. Given a database schema, we translate it to a complete join graph where each table is a node and each (foreign key, primary key) pair is an edge. The generator performs a random walk on this graph, starting from a fact table and guided by two parameters, the maximum number of hops away from the initial table and the probability of keeping an edge. It outputs a new, unseen subgraph that serves as a unique **join signature**.

Predicate Generation. Given a join signature, we further generate diverse column predicates (range, equality and IN-list) that can be satisfied by the input data. For each predicate, we start by randomly selecting a column. If we are to generate a range predicate, we impose a minimum selectivity threshold and select a range of values based on the histogram to guarantee the selectivity. For an equality predicate, we impose a minimum relative frequency, and use most common values (MCVs) to find a candidate (if present). A similar process is applied to IN predicates, with additional elements drawn from the histogram boundary values to achieve the desired list length. Finally, we filter the query if it returns an empty set.

Encouraging Diversity. We encourage queries that cover a diverse range of join output sizes. To do so, we initiate the random walk under different parameter settings and bin the synthetic queries based on the join output size. As we sample enough parameter values, we apply stratified sampling over the bins to produce queries with a (relatively) even spread across the join output size.

5.3 Filtering Synthetic Queries in GTN Space

Another potential solution is to directly generate queries that lie in the low-density regions of the embedding space. However, graph generation from the embedding space is a known hard problem in the ML literature, and the latest ML techniques, including variational flow methods, only manage to achieve it for two real world applications, molecule generation [17, 104] and point-cloud denoising [51]. In contrast, query plan graph generation requires complex node features, which current methods cannot yet achieve.

Alternatively, we proposes an optimization that filters a large set of synthetic queries via the GTN embedding space and selects those the most distant from the training set, e.g., landing in the sparse region. Its benefits include better balancing the training set between sparse and dense regions, and improving training efficiency as adding more queries to the dense region will not add much value. For more details, see Appendix G.

6 ENHANCED GTN FOR QUERY PLANS

In this section, we identify GTN limitations and propose enhanced architectures that replace language-model-inspired components with techniques tailored for query plans.

6.1 Background on Graph Transformers

Following the taxonomy of Section 3.3, we review the graph transformer architecture for query performance prediction (QPP) as illustrated in Figure 5. We explain node i's update as a running example, using single-head self-attention for simplicity.

A. Node and Positional encoding. Node encoding x_i is enriched by positional encoding λ_i via a linear transformation W_λ: $h_i = x_i + W_\lambda \lambda_i$. Denote H as all enriched node encodings of a plan.

B. Update mechanism. Using the Transformer self-attention computation [80], the updated representation for node i is given by

$$\hat{h}_i = \left[\text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d}}\right)\right]_i V$$

where d is a scaling factor, Q, K, V are linear projections from H.

We break down the self-attention computation into several steps. First, we define s_{ij} as the dot product between vectors h_i and h_j: $s_{ij} = h_i \cdot h_j$. Then, we extract attention scores w_{ij} by normalizing s_{ij} with a softmax operator $w_{ij} = \exp(s_{ij})/\sum_{k \in \mathcal{N}(i)} \exp(s_{ik})$, where $\mathcal{N}(i)$ denotes the neighborhood of node i (in graph transformers, only a subset of nodes). We then have $\hat{h}_i = \sum_{j \in \mathcal{N}(i)} w_{ij} W h_j$.

Next, h_i and $\hat{h}_i$ are summed and normalized, i.e., we have $\bar{h}_i = \text{Norm}(h_i + \hat{h}_i)$. The normalization ensures that $\bar{h}_i$ has features in the same range as h_i and $\hat{h}_i$. Finally, we obtain the output representation $h'_i = \text{Norm}(MLP(\bar{h}_i) + \bar{h}_i)$ where the MLP combines multiple heads.

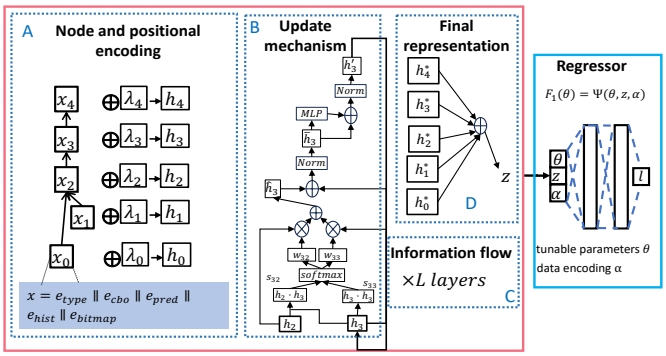


Figure 5: Latency prediction with graph transformers

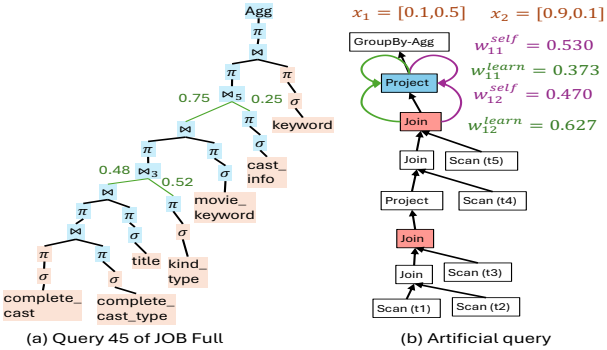


Figure 6: Running query examples

The skip connection (+) ensures that the output representation is at least as informative as the intermediate representation.

C. Information flow. The update procedure described above is performed for each layer of the graph transformer (except the last one which will be described shortly). As the input h_i pass through the layers, its updated representation gradually includes information about nodes farther away: after the first layer, the node has received information from its direct neigbhors. After the third layer, it has received information from nodes 3-hop away.

D. Final representation. The final layer of the graph transformer aggregates the final node representations $h_1^*, \ldots, h_N^*$, by average pooling: the query plan representation is defined as $z = (1/N) \sum_{i=1}^{N} h_i^*$, which is then passed to a downstream regressor.

6.2 Enhanced GTN

We identify two limitations of current GTN models, inspired by the language technology, for query embedding. We then propose new GTN variants that address these limitations. Our discussion uses running examples in Figure 6, where part (a) shows Query 45 of the JOB Full dataset with real model weights, and part (b) shows a toy example with 2-dim features to permit simple computation.

Limitation of Positional Encodings. The same operators occurring in different parts of the query graph should produce the same result, given the same inputs. Consider the two red join operators in Figure 6(b). The behaviour of a join operator depends only on its inputs, not its position in the query plan. After the first GTN layer processes the query plan, join operators with similar inputs should have similar updated features. However, positional encodings can disrupt this by altering node representations based on their locations, introducing artificial differences between similar operators in different parts of the plan. To address this, we propose to **remove positional encoding from the GTN architecture** and refer to this revised architecture as **GTN$_{\neg \text{PE}}$**.

Limitation of Self-Attention. We discover that self-attention may restrict the information flow between nodes with different features. Consider the join operator (red) followed by the project operator (blue) in Figure 6(b), with their respective node encodings h_2 and h_1. For simplicity, we assume two-feature encodings and omit positional encoding. Suppose $x_1 = [0.1, 0.5]$ and $x_2 = [0.9, 0.1]$. The self-attention coefficient w_{11}^{self} is computed as $\exp(x_1 \cdot x_1)/[\exp(x_1 \cdot x_1) + \exp(x_1 \cdot x_2)]$, resulting in $w_{11}^{self} = 0.530$. With a similar computation, we get $w_{12}^{self} = 0.470$, which is too low for the information to flow from the join operator to the project operator. In fact, the maximum attainable self-attention weight, w_{12}^{self}, is capped at 0.5 (achieved when $x_1 = x_2$), which is too low for information to flow. Consequently, self-attention sets a strict upper bound on how much information can be exchanged between distinct operators. Since many consecutive relational operators have dissimilar feature embeddings, this mechanism creates a bottleneck for information flow.

Learnable Attention to Maximize Information Flow. The failure case above happens because self-attention calculates scores purely based on the similarity of feature vectors. However, in query embedding, we want the *attention mechanism to produce scores that enable information flow even between nodes with highly contrasting features.* To achieve that goal, we can replace self-attention with learnable attention [6, 81]. This mechanism uses a parameterized vector $\mathbf{a}$ to compute the unnormalized attention score between two feature vectors, denoted as $e(h_i, h_j) = \mathbf{a}^T \text{ LeakyReLU}(W_e \cdot [h_i \| h_j])$, where W_e is a linear transformation, and $\|$ is the concatenation operator. Compared to s_{ij}, $\mathbf{a}$ allows control over how information flows between dissimilar operators. For instance, omitting the LeakyReLU and linear transformation for simplicity, if $\mathbf{a} = [0.1, 0.2, 0.7, 0.1]$, we have $e(h_1, h_1) = [0.1, 0.2, 0.7, 0.1] \cdot [0.1, 0.5, 0.1, 0.5] = 0.23$ and $e(h_1, h_2) = [0.1, 0.2, 0.7, 0.1] \cdot [0.1, 0.5, 0.9, 0.1] = 0.75$. Using the softmax operator to get the normalized scores w_{11}^{learn}, and w_{12}^{learn}, we get $w_{11}^{learn} = 0.373$ and $w_{12}^{learn} = 0.627$. Information is now able to flow from the join operator to the project operator. Therefore, we propose to **replace the self-attention with learnable attention in the GTN**, and we denote by **GAT** the new architecture.

6.3 Comparison to Competitive Models

We now compare our approach with competitive state of the art query embedders, namely GCN and TTN.

Importance of Context Awareness. Information flow must prioritize the relevant operators. Consider the joins 3 and 5 of Query 45 in Figure 6(a) from JOB. A convolution based model such as GCN will attribute fixed weights (here, 0.5 for each node) to the incoming operators, regardless of the features of the nodes, and will not be able to favour any of the two paths. In contrast, a GAT can attribute different weights to different operators: 0.75 and 0.25 for the inputs of join 5, 0.52 and 0.48 for the inputs of join 3. The 0.75 coefficient means that the left branch contains more relevant information to be passed to the next operator, and leads to an improved prediction (query latency: 12.45 s; GCN prediction: 8.23 s; GAT prediction: 12.246 s). Thus, attention allows information contextualization.

Importance of Sparse Attention. Efficient computation of information flow is critical. Consider the JOB query in Figure 6(a). The Tree Transformer Network (TTN) incorporates two mechanisms to promote long-range information propagation: (1) attention directed towards all downstream children of a node, and (2) the inclusion of a super node that receives attention from all nodes in the graph. The super node is used as the final graph representation and replaces the average pooling operation typically used in the final layer of graph transformers.

These mechanisms fail to emphasize critical operators. First, super nodes tend to homogenize node representations across the graph [73]. Second, the combination of super node and attention to all downstream children entails high weights of leaf nodes in the final graph representation. This happens because the super node is attended by a single leaf several times: directly by the leaf itself, and indirectly by all the leaf's children (two-hop neighbourhoods). Figure 6(a) illustrates this effect on JOB Query 45. Nodes highlighted in blue contribute less to the final graph representation than they would under average pooling, whereas pink-highlighted nodes contribute more. Here, TTN disproportionately focuses on table scan operators while underemphasizing join operators, leading to an underestimated query latency (predicted latency: 9.85 s).

7 PERFORMANCE EVALUATION

In this section, we evaluate our data augmentation methods and enhanced GTN architectures for improving the effectiveness of query plan representation. We also compare our techniques against the most competitive models identified in Section 4.

We implemented our (extended) GTN models in PyTorch, relying on its default parameter initialization [24]. For the downstream QPP task, we train the GTN and an MLP regressor (augmented with skip connections for improved robustness) jointly via backpropagation, using the AdamW optimizer [46]. To ensure stable convergence, we combine a cosine annealing schedule [45] with a linear warmup phase [49] for adaptive learning rate control. For each model, we conducted hyper-parameter tuning, up to one week of running time, over the number of graph layers (L), hidden dimension (d_{hidden}), the size of the final plan representation (d_{final}), and whether to use feature masks or not [61], as well as the structure of the downstream MLP. Additional details are provided in Appendix E.

7.1 Improving OOD with Data Augmentation

We begin by evaluating the effectiveness of data augmentation for improving OOD performance.

7.1.1 Impact on JOB. Recall that we have two settings for evaluating JOB OOD performance. JOB Light includes 70 test queries that are unseen but somewhat comparable to JOB Train in complexity (comparable predicates and modest increase of the number of joins., e.g, from 2 to 4 from training to test). JOB Full includes the full set of 113 JOB queries that are much more complex than JOB Train (complex predicates and up to 16 joins).

JOB Light. We augment training data for JOB Light to simulate the scenario that the training data has sufficient coverage of the (unseen) test queries – this scenario will enable us to demonstrate true model differences. To this end, we generated 27k synthetic queries with up to 4 joins (EXT_v1), using the generator in Section 5.2. Table 4 shows the results of data augmentation in the first 3 rows. First, as we increase the training data from 10k to 100k of JOB Train, OOD errors initially reduce, but beyond 30K queries, become worse, highlighting the limited diversity of additional queries in JOB Train. In contrast, adding our 27k synthetic queries reduces OOD errors (WMAPE from 13.6% to 10.2%), without negatively affecting ID accuracy. Figure 7 visualizes the effectiveness in broadening coverage, where a t-SNE projection is applied to query embeddings trained over Train_30K queries plus EXT_v1. The *left* plot shows that the points in Train (grey dots) cannot fully cover the test queries (red dots). In contrast, the *right* plot shows that the added synthetic queries are diverse and occupy many of these previously uncovered areas, thereby improving the OOD accuracy.

Table 5: OOD results of TPC-DS with different training data options

	WAPE			Q-error			
	Mean	P50	P90	P50	P90	P99	MAX
DS_50K	0.456	0.199	1.056	1.320	2.458	5.203	19.70
DS_50K+LLM_6K	0.458	0.213	1.039	1.315	2.547	5.682	20.39
DS_50K+EXT_10K	0.407	0.162	0.957	1.269	2.369	5.347	19.88
DS_50K+EXT_30K	0.394	0.159	0.982	1.267	2.369	4.765	22.73
DS_50K+EXT_50K	0.438	0.163	1.063	1.268	2.401	5.322	20.18
DS_50K+EXT*_30K (filtered)	0.387	0.156	0.937	1.256	2.301	4.864	19.48
DS_50K+EXT*_30K+JOB*	0.350	0.169	0.798	1.264	2.455	4.878	18.72

JOB Full. Table 4 shows, in the bottom row group, that neither JOB Train or Train_30K + EXT_v1 is sufficient for the more complex JOB Full, although the latter performs better than the former. (1) Given that JOB Full includes up to 16 joins, we ran our query generator with parameters to enable more random walks on the join graph and more predicates. *Without* peeking into the test set, our query generator produces 859 join signatures (1-11 joins) and 20K new queries (EXT_v2). (2) Alternatively, we added to training 13.6K CEB queries developed on IMDB, with 5-11 joins, based on 15 fixed templates [60]. Table 4 shows that Train_30K + EXT_v2 is more effective than Train_30K + CEB. This is confirmed by the t-SNE plots in Figure 8 where CEB covers a few concentrated regions of the embedding space, due to the fixed 15 templates, while EXT_v2 has broad, diverse coverage of the embedding space. Given that CEB has large joins and LIKE predicates (which are not available in EXT_v2), we finally combine both sets in training, achieving the best OOD WMAPE of 25.1%, compared to the initial 38.4%.

Summary. Controlling the training size of JOB Train to 30 K avoids overfitting ID patterns and improves the generalization ability of OOD queries. For data augmentation, the key finding is that **expanding the query variety, rather than merely increasing the total number of queries, is key to improving OOD performance** (**R1**, denoting major result 1).

7.1.2 Impact on TPC-DS. For TPC-DS, our original training data included 50 K parameterized queries from the 102 query templates (DS_50K). We report OOD performance of different data augmentation methods in Table 5.

LLM Queries. TPC-DS includes complex operators like UNION, CROSS-JOIN, SUBQUERY, WINDOW, and EXPAND. To support such diverse operators, we first explore LLMs to generate diverse queries that differ maximally from existing TPC-DS queries. The LLM method (ChatGPT 4.0) managed to generate 1000 distinct query

Table 4: JOB OOD Results (WAPE) of GTN under training data options

Training for JOB OOD	Mean	P90
Train_100K (**JOB Light**)	0.151	0.387
Train_30K	0.136	0.307
Train_30K+EXT_v1	0.102	0.200
Train_100K (**JOB Full**)	0.384	0.824
Train_30K	0.382	0.873
Train_30K+EXT_v1	0.334	0.748
Train_30K+EXT_v2	0.284	0.581
Train_30K+CEB	0.319	0.669
Train_30K+EXT_v2+CEB	0.251	0.496

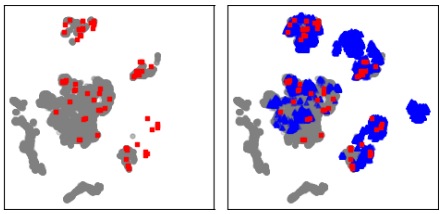


Figure 7: T-SNE visualization of JOB Light: Train_30K (grey dots), JOB EXT_v1 (blue dots), JOB Light queries (red dots).

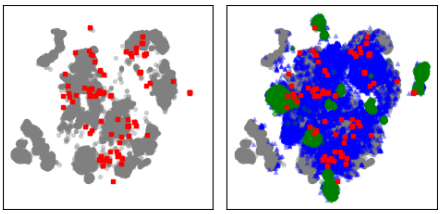


Figure 8: T-SNE visualization of JOB Full: Train_30K (grey dots), JOB EXT_v2 (blue dots), CEB (green dots), JOB Full queries (red dots).

templates and we parameterized them to 6000 queries (DS-LLM). The row, DS_50K+LLM_6K, in Table 5 shows that adding the LLM_6K query set does not improve OOD performance. Although 1,000 templates were requested from LLM, only 174 of them are runnable ones, covering just four distinct query-plan structures, due to the model collapse issue of LLMs [22]. Hence, DS_50K+LLM_6K yields a WMAPE of 46%, worse than the 45% baseline. This indicates that LLM-based query generation remains mostly at a syntax level, lacking a deeper understanding of the diverse query plans on the complex schema of TPC-DS, especially given only high-level prompts.

Random, Synthetic Join Queries. We also used our query generator to create 7000 join signatures (up to 9 joins) and 55,180 parameterized queries (DS-EXT). Table 5 shows that random query synthesis works better than LLM. From the row group, DS_50K+EXT_xK, in Table 5 we see that adding 10K-30K star join queries improves OOD errors, with the best WMAPE (39.4%). Beyond 30 K, performance regresses slightly, suggesting overfitting to newly introduced join patterns. With more internal profiling, we see that limiting up to four joins, denoted as EXT*_30K (filtered), achieves the best WMAPE (38.7%), suggesting that moderate join complexity offers better balance for generalization. Finally, we further add JOB* (the best JOB training data), achieving additional OOD gains (WMAPE from 38.7% to 35.0%). While JOB queries do not overlap with original TPC-DS queries in the embedding space, their diversity helps mitigate overfitting to current DS join queries.

We further examined the queries to understand the remaining errors with all the data augmentation. We observe that there exist many diverse operators, including UNION, WINDOW, EXPAND, CROSS-JOIN, and SUBQUERY. Queries involving these operators, which are insufficiently covered in training data, often experience high errors. For more details, see Appendix C.2.

Summary. We summarize the main observations from the TPC-DS data augmentation study: (i) **LLM Limitations (R2)**. LLM-based generation is insufficient for handling the structural and semantic complexity of TPC-DS: without deeper knowledge of query planning and optimization, it produces mainly low-quality templates (most of them are not runnable and include a very small number of distinct query structures). (ii) **Partial Gains from Join Queries (R3)**. Although additional join queries and JOB queries reduce the ID–OOD gap, the overall improvement remains modest. A main factor is that TPC-DS includes a very diverse set of query operators, and there is not enough coverage of these operators in the current training set. (iii) **Challenge of a Vast SQL Space (R4)**. Real-world query spaces are both huge and sparse, making it difficult for the model to cover every corner of it. Simply adding more synthetic queries, unless carefully aligned with the OOD patterns, may yield limited benefits as this may steer the model toward a limited region. A significantly larger, diverse training set is needed to have a good coverage of the complex TPC-DS workload.

7.2 Improving OOD with Enhanced GTN

We now evaluate the performance of our enhanced GTN on the augmented JOB and TPC-DS datasets. We focus on the four variants proposed in Section 6.2: (1) the original GTN; (2) $\text{GTN}_{\neg\,\text{PE}}$, with positional encodings removed; (3) GAT, with learnable attentions; (4) $\text{GAT}_{\neg\,\text{PE}}$, with positional encodings removed. We also compare to the most competitve models identified in Section 4: (5) GCN [87], (6) TTN(QF) [102], and (7) TCNN [54].

7.2.1 Evaluation on JOB. Table 6 compares the baseline GTN (on the original JOB Train), GTN variants (1-4) trained on augmented data, and competitve models (5-7) trained on augmented data.

JOB Light. After data augmentation improves OOD, the model differences start to be more visible. Most notably, the GTN family shows significant benefits over the competive models: while GCN and TCNN perform better than TTN, they lose to the best GTN model in mean and more significantly, in P90 result. Among the GTN variants, $\text{GAT}_{\neg\,\text{PE}}$ gives the overall best WMAPE and P90. It has reduced the gap between ID and OOD sufficiently enough that they appear to be drawn from a similar distribution. This means that **under sufficient training data, $\text{GAT}_{\neg\,\text{PE}}$ can reduce the gap between ID and OOD (R5)** due to custom architectural choices for query plans, namely, removing noisy positional encodings while adapting the attention between different operator types.

JOB Full is more challenging than JOB Light due to the additional large join queries and complex predicates. While all the models are affected by this challenge, again the GTN family of models outperform the competitive models significantly in the P90 range. This means that **advanced GTN variants can better capture complex, long-running queries with context-aware weights, hence outperforming competitive models (R6)**. An example of JOB Q45 was given in Section 6.

7.2.2 Evaluation on TPC-DS. Table 7 shows the TPC-DS OOD results of the 7 models. This workload is harder due to the diverse SQL operators used, which are not sufficiently covered in training data. Therefore, all the models exhibit high errors. Consistently, we see that the GTN models outperform the most competitve model, GCN, in the mean (with a modest 4% difference) and P90 (with more significant 11% difference) in WAPE.

Table 6: Enhanced GTNs vs competitve models for JOB Light and JOB Full on augmented data

Model	JOB-ID		JOB Light-OOD		JOB Full-OOD		
	Mean	P90	Mean	P90	Mean	P90	P99
GTN (BL)	0.057	0.114	0.151	0.387	0.384	0.824	2.80
GTN	0.072	0.156	0.102	0.200	0.266	0.556	2.48
GTN$_{\neg \text{ PE}}$	0.073	0.145	0.093	**0.189**	**0.254**	**0.441**	2.37
GAT	**0.071**	**0.143**	**0.089**	0.200	0.258	0.487	**1.69**
GAT$_{\neg \text{ PE}}$	0.073	0.165	**0.089**	0.190	0.269	0.666	2.26
GCN	0.076	0.175	0.109	0.277	0.280	0.615	**2.21**
TTN (QF)	0.121	0.313	0.231	0.525	0.276	0.569	2.40
TCNN	0.080	0.182	0.120	0.244	0.265	0.634	2.58

Table 7: Enhanced GTNs vs competitive models for TPC-DS on augmented data

Model	TPC DS-ID		TPC DS-OOD	
	Mean	P90	Mean	P90
GTN (BL)	0.163	0.336	0.456	1.056
GTN	0.158	0.326	0.350	**0.798**
GTN$_{\neg \text{ PE}}$	0.161	0.333	0.385	0.884
GAT	**0.156**	**0.312**	0.373	0.873
GAT$_{\neg \text{ PE}}$	0.159	0.323	**0.347**	0.818
GCN	0.164	0.324	0.387	0.908
TTN (QF)	0.160	0.319	0.47	1.156
TCNN	0.164	0.321	0.425	0.997

Table 8: Transferring the TPC-DS embedder to TPC-H for QPP

Fine-tuning Regressor w. Queries per Template (%)	WAPE		
	Mean	P50	P90
0 (Zero-shot/OOD)	0.605	0.372	1.340
5 (Few-shot)	0.396	0.212	0.939
10 (Few-shot)	0.211	0.098	0.484
20	0.162	0.077	0.345
50	0.141	0.064	0.298
90	0.118	0.047	0.253
Train-from-scratch (OOD)	0.786	0.556	1.643
Train-from-scratch (ID)	0.111	0.045	0.235

7.3 Use Case 1: Transfer Learning

We next evaluate how our learned plan embedder works for unseen workloads, a common use case in cloud analytics when new user workloads arrive and the model needs to work out-of-the-box. Here, we use TPC-H as the unseen workload, and transfer the plan embedder of TPC-DS, trained using the best GTN architecture (GTN*) on the augmented training sets (TPC-DS and JOB combined).

We assess the TPC-H query performance predictor (QPP) built on the transferred plan embedder from TPC-DS. This OOD scenario aligns with a *zero-shot learning* setup, where the embedder trained on TPC-DS is directly used without any retraining on TPC-H queries. In comparison, the regressor for the downstream task, QPP, may be fine-tuned using a small set of TPC-H data, aka, *few-shot learning*. First, as reference, we show the performance of the embedder and regressor trained natively on TPC-H in the bottom two rows of Table 13. As can be seen, the ID performance is good (WMAPE = 11.1%) but OOD performance is poor (WMAPE = 78.6%). Then in the top part of the table, we show that (1) by using the TPC-DS embedder out-of-the-box, without fine-tuning, transfer learning can already reduce the OOD WMAPE to 60.5%; (2) as we perform fine tuning of the regressor, only 10% parameterized queries per TPC-H template reduce the OOD WMAPE to 21.1%, highlighting substantial gains even with limited domain-specific data. WMAPE decreases steadily as fine-tuning employs more data (20-90%), until reaching the ID performance, all using the TPC-DS embedder.

7.4 Use Case 2: Job Scheduling

Finally, we use a job scheduling task to evaluate end-to-end performance in a model-centric setting—this task allows us to directly observe the net effect of a model, without tight coupling with an optimizer (which may come with its own constraints). The task requires classifying queries as "long" or "short" based on predicted latency, where short queries are sent to a single queue for scheduling and long queries are directed to two burst clusters [79]. Our evaluation proceeds in two stages. First, we measure classification accuracy (Table 9) on several OOD test sets, using the 80th latency percentile as the long/short cutoff. While all models identify short queries with high accuracy (95-100%), correctly identifying long queries is more challenging. Here, our best GTN model excels in long query accuracy, improving the most competitive model, GCN, with 21% on JOB Full and 8% on TPC-DS, while also being more accurate for short query classification. Second, to translate this accuracy to system-level gains, we simulate E2E workload latency by allowing five concurrent short queries in one cluster and two burst clusters for long queries. Misclassifying a query will send it to the wrong cluster, causing delays to this query and subsequent ones (which is sensitive to the order of long-short query mixes). The model's inference time of ~5 ms per query is negligible (<0.1%) compared to the shortest query execution time (4.87-5.22s) in Spark SQL. Our GTN model's superior accuracy translates to E2E time, consistently outperforming the competitive models by 4-17%, 7-10% and 6-11% on JOB Light, JOB Full and TPC-DS, respectively.

Table 9: Long-Short query classification accuracy and E2E Latency.

Model	JOB Light				JOB Full				TPC-DS			
	Short	Long	All	E2E(s)	Short	Long	All	E2E(s)	Short	Long	All	E2E(s)
GTN	1.00	**0.79**	**0.96**	**98**	0.99	0.12	0.81	219	0.95	**0.71**	**0.90**	**169**
GTN-PE	0.98	0.71	0.93	100	1.00	0.21	0.83	219	0.93	**0.71**	0.89	179
GAT	0.98	0.57	0.90	105	0.99	**0.33**	**0.85**	**202**	0.94	0.68	0.89	179
GAT-PE	0.98	0.71	0.93	100	1.00	0.21	0.83	213	0.95	0.69	0.89	179
GCN	0.95	**0.79**	0.91	118	0.98	0.12	0.80	224	0.94	0.63	0.87	190
TTN	0.89	0.43	0.80	105	0.97	0.12	0.79	218	0.93	0.36	0.82	182
TCNN	0.98	0.64	0.91	102	0.98	0.12	0.80	221	0.93	0.56	0.85	180

8 CONCLUSIONS

In this paper, we examined the strengths and challenges of Graph Transformer Networks (GTNs) for query plan representation (QPR). We proposed data augmentation techniques to enhance training diversity and refined GTN architectures by replacing ineffective language-model-inspired components with techniques better suited for query plans. Evaluation results show that with sufficient training data, enhanced GTNs reduce the gap between in-distribution (ID) and out-of-distribution (OOD) queries (JOB Light); outperform existing models for capturing complex queries (JOB Full and TPC-DS); and enable the query embedder trained on TPC-DS to generalize to TPC-H queries out of the box.

Our evaluation also reveals future research directions to realize the full potential of GTNs for QPR. We need larger high-quality training sets, covering diverse query operators and different ways of combining them, for the model to generalize for complex workloads like TPC-DS. In addition, we will explore alternative, self-supervised learning methods to reduce the number of query observations needed for training the GTN.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) Horizon 2020 research and innovation programme (grant n725561), and the HORIZON ERC Proof of Concept Grant (grant 101213783).

REFERENCES

[1] [n.d.]. TPC-DS Benchmark. https://www.tpc.org/tpcds/.

[2] [n.d.]. TPC-H Benchmark. https://www.tpc.org/tpch/.

[3] Christoph Anneser, Nesime Tatbul, David E. Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (2023), 3515–3527. https://doi.org/10.14778/3611540.3611544

[4] Abdul Fatir Ansari, Lorenzo Stella, Caner Turkmen, Xiyuan Zhang, Pedro Mercado, Huibin Shen, Oleksandr Shchur, Syama Syndar Rangapuram, Sebastian Pineda Arango, Shubham Kapoor, Jasper Zschiegner, Danielle C. Maddix, Hao Wang, Michael W. Mahoney, Kari Torkkola, Andrew Gordon Wilson, Michael Bohlke-Schneider, and Yuyang Wang. 2024. Chronos: Learning the Language of Time Series. *arXiv preprint arXiv:2403.07815* (2024).

[5] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2021).

[6] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks?. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=F72ximsx7C1

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[8] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. [n.d.]. Measuring and Relieving the Over-Smoothing Problem for Graph Neural Networks from the Topological View. 34, 04 ([n. d.]), 3438–3445. Issue 04. https://doi.org/10.1609/aaai.v34i04.5747

[9] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans. *Proc. VLDB Endow.* 16, 7 (2023), 1777–1789. https://doi.org/10.14778/3587136.3587150

[10] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. LEON: A New Framework for ML-Aided Query Optimization. *Proc. VLDB Endow.* 16, 9 (2023), 2261–2273. https://doi.org/10.14778/3598581.3598597

[11] Xu Chen, Zhen Wang, Shuncheng Liu, Yaliang Li, Kai Zeng, Bolin Ding, Jingren Zhou, Han Su, and Kai Zheng. 2023. BASE: Bridging the Gap between Cost and Latency for Query Optimization. *Proc. VLDB Endow.* 16, 8 (2023), 1958–1966. https://doi.org/10.14778/3594512.3594525

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, June 2-7, 2019, Volume 1*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[13] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1241–1258. https://doi.org/10.1145/3299869.3324957

[14] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=YicbFdNTTy

[15] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *PVLDB* 2, 1 (2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[16] Vijay Prakash Dwivedi and Xavier Bresson. 2021. A Generalization of Transformer Networks to Graphs. *AAAI Workshop on Deep Learning on Graphs: Methods and Applications* (2021).

[17] Floor Eijkelboom, Grigory Bartosh, Christian Andersson Naesseth, Max Welling, and Jan-Willem van de Meent. 2024. Variational flow matching for graph generation. *Advances in Neural Information Processing Systems* 37 (2024), 11735–11764.

[18] Wenchen Fan, Herman van Hovell, and MaryAnn Xue. 2020. Adaptive Query Execution: Speeding Up Spark SQL at Runtime. https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html.

[19] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. To Tune or Not to Tune? In Search of Optimal Configurations for Data Analytics. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 2494âĂŞ2504. https://doi.org/10.1145/3394486.3403299

[20] Victor Giannakouris and Immanuel Trummer. 2024. Demonstrating λ-Tune: Exploiting Large Language Models for Workload-Adaptive Database System Tuning. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 508–511. https://doi.org/10.1145/3626246.3654751

[21] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 1263–1272. http://proceedings.mlr.press/v70/gilmer17a.html

[22] Sil Hamilton. 2024. Detecting Mode Collapse in Language Models via Narration. arXiv:2402.04477 [cs.CL] https://arxiv.org/abs/2402.04477

[23] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1035–1050. https://doi.org/10.1145/3318464.3389741

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. IEEE Computer Society, 1026–1034. https://doi.org/10.1109/ICCV.2015.123

[25] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374. https://doi.org/10.14778/3551793.3551799

[26] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[27] Sepp Hochreiter and JÃijrgen Schmidhuber. [n.d.]. Long Short-Term Memory. 9, 8 ([n. d.]), 1735–1780.

[28] H. M. Sajjad Hossain, Marc T. Friedman, Hiren Patel, Shi Qiao, Soundar Srinivasan, Markus Weimer, Remmelt Ammerlaan, Lucas Rosenblatt, Gilbert Antonius, Peter Orenberg, Vijay Ramani, Abhishek Roy, Irene Shaffer, and Alekh Jindal. 2021. PerfGuard: Deploying ML-for-Systems without Performance Regressions , Almost! *Proc. VLDB Endow.* 14, 13 (2021), 3362–3375. https://doi.org/10.14778/3484224.3484233

[29] Zhiyao Hu, Dongsheng Li, Dongxiang Zhang, Yiming Zhang, and Baoyun Peng. 2021. Optimizing Resource Allocation for Data-Parallel Jobs Via GCN-Based Prediction. *IEEE Trans. Parallel Distributed Syst.* 32, 9 (2021), 2188–2201. https://doi.org/10.1109/TPDS.2021.3055019

[30] Eyke Hüllermeier and Willem Waegeman. 2021. Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods. *Mach. Learn.* 110, 3 (2021), 457–506. https://doi.org/10.1007/S10994-021-05946-3

[31] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin ťídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Magdalena Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly Accurate Protein Structure Prediction with AlphaFold. *Nature* 596, 7873 (2021), 583–589. https://www.nature.com/articles/s41586-021-03819-2

[32] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. 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. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[33] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (2024), 1939–1952. https://www.vldb.org/pvldb/vol17/p1939-tang.pdf

[34] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. [n.d.]. Deep Learning. 521, 7553 ([n. d.]), 436–444.

[35] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[36] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (2021), 1606–1612. http://www.vldb.org/pvldb/vol14/p1606-leis.pdf

[37] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf

[38] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (2023), 197–210. https://www.vldb.org/pvldb/vol17/p197-li.pdf

[39] Rundong Li, Ningfang Mi, Mirek Riedewald, Yizhou Sun, and Yi Yao. 2019. Abstract cost models for distributed data-intensive computations. *Distributed Parallel Databases* 37, 3 (2019), 411–439. https://doi.org/10.1007/S10619-018-7244-2

[40] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, and Bin Cui. 2023. Towards General and Efficient Online Tuning for Spark. *Proc. VLDB Endow.* 16, 12 (2023), 3570–3583. https://doi.org/10.14778/3611540.3611548

[41] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, and Zhiyong Peng. 2022. A Resource-Aware Deep Cost Model for Big Data Query Processing. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 885–897. https://doi.org/10.1109/ICDE53745.2022.00071

[42] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proc. VLDB Endow.* 14, 11 (2021), 1950–1963. http://www.vldb.org/pvldb/vol14/p1950-liu.pdf

[43] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? *CoRR* abs/2408.05109 (2024). https://doi.org/10.48550/ARXIV.2408.05109 arXiv:2408.05109

[44] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 10012–10022.

[45] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Skq89Scxx

[46] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

[47] Chenghao Lyu, Qi Fan, Philippe Guyard, and Yanlei Diao. 2024. A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning. *Proc. VLDB Endow.* 17, 11 (2024), 3562–3579. https://doi.org/10.14778/3681954.3682021

[48] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, and Jingren Zhou. 2022. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. *Proc. VLDB Endow.* 15, 11 (2022), 3098–3111. https://www.vldb.org/pvldb/vol15/p3098-lyu.pdf

[49] Jerry Ma and Denis Yarats. 2021. On the Adequacy of Untuned Warmup for Adaptive Optimization. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 8828–8836. https://doi.org/10.1609/AAAI.V35I10.17069

[50] Andrey Malinin, Liudmila Prokhorenkova, and Aleksei Ustimenko. 2021. Uncertainty in Gradient Boosting via Ensembles. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=1Jv6b0Zq3qi

[51] Aihua Mao, Biao Yan, Zijing Ma, and Ying He. 2024. Denoising point clouds in latent space via graph convolution and invertible neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5768–5777.

[52] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1275–1288. https://doi.org/10.1145/3448016.3452838

[53] Ryan Marcus and Olga Papaemmanouil. 2019. Flexible operator embeddings via deep learning. *arXiv preprint arXiv:1901.09090* (2019).

[54] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[55] Ryan C. Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[56] Leland McInnes, John Healy, and James Melville. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426* (2018).

[57] Michael D. McKay, Richard J. Beckman, and William J. Conover. 2000. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output From a Computer Code. *Technometrics* 42, 1 (2000), 55–61. https://doi.org/10.1080/00401706.2000.10485979

[58] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 3111–3119. https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html

[59] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (2009), 982–993. https://doi.org/10.14778/1687627.1687738

[60] Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. http://www.vldb.org/pvldb/vol14/p2019-negi.pdf

[61] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[62] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. [n.d.]. On the Difficulty of Training Recurrent Neural Networks. In *International Conference on Machine Learning* (2013). Pmlr, 1310–1318.

[63] PostgreSQL Global Development Group. 2020. PostgreSQL Documentation 12. Chapter 70.1: Row Estimation Examples. https://www.postgresql.org/docs/current/row-estimation-examples.html. Accessed: February 2025.

[64] Yuan Qiu, Yilei Wang, Ke Yi, Feifei Li, Bin Wu, and Chaoqun Zhan. 2021. Weighted Distinct Sampling: Cardinality Estimation for SPJ Queries. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1465–1477. https://doi.org/10.1145/3448016.3452821

[65] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[66] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: eloquent performance models for Resource Optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. 415–427. https://doi.org/10.1145/2987550.2987566

[67] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proc. VLDB Endow.* 17, 4 (2023), 740–752. https://www.vldb.org/pvldb/vol17/p740-reiner.pdf

[68] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying WEI, Wenbing Huang, and Junzhou Huang. 2020. Self-Supervised Graph Transformer on Large-Scale Molecular Data. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12559–12571. https://proceedings.neurips.cc/paper_files/paper/2020/file/94aef38441efa3380a3bed3faf1f9d5d-Paper.pdf

[69] Amazon EMR Serverless. 2022. The pricing for Amazon EMR Serverless. https://aws.amazon.com/emr/pricing/#Amazon_EMR_Serverless

[70] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross J. Anderson, and Yarin Gal. 2024. AI models collapse when trained on recursively generated data. *Nat.* 631, 8022 (2024), 755–759. https://doi.org/10.1038/S41586-024-07566-Y

[71] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 99–113. https://doi.org/10.1145/3318464.3380584

[72] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant J. Shenoy. 2021. Spark-based Cloud Data Analytics using Multi-Objective Optimization. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 396–407. https://doi.org/10.1109/ICDE51399.2021.00041

[73] Joshua Southern, Francesco Di Giovanni, Michael Bronstein, and Johannes F Lutzeyer. 2025. Understanding Virtual Nodes: Oversquashing and Node Heterogeneity. In *International Conference on Learning Representations (ICLR)*.

[74] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296

[75] Ruoxi Sun, Sercan ï£¡. Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and Tomas Pfister. 2024. SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL (extended). arXiv:2306.00739 [cs.CL] https://arxiv.org/abs/2306.00739

[76] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. The Association for Computer Linguistics, 1556–1566. https://doi.org/10.3115/v1/p15-1150

[77] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 190–203. https://doi.org/10.1145/3514221.3517843

[78] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[79] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[80] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[81] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro LiÃš, and Yoshua Bengio. [n.d.]. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings* (2018). OpenReview.net.

[82] Lalitha Viswanathan, Alekh Jindal, and Konstantinos Karanasos. 2018. Query and Resource Optimization: Bridging the Gap. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1384–1387. https://doi.org/10.1109/ICDE.2018.00156

[83] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84. http://www.vldb.org/pvldb/vol15/p72-li.pdf

[84] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *Proc. VLDB Endow.* 12, 3 (2018), 210–222. https://doi.org/10.14778/3291264.3291267

[85] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2009–2022. https://doi.org/10.1145/3448016.3452830

[86] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proc. ACM Manag. Data* 2, 1 (2024), 38:1–38:27. https://doi.org/10.1145/3639293

[87] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 280–294. https://doi.org/10.1145/3626246.3653391

[88] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2020. BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation. https://doi.org/10.48550/ARXIV.2012.14743

[89] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 674–684. https://doi.org/10.1145/3514221.3526157

[90] Maryann Xue, Yingyi Bu, Abhishek Somani, Wenchen Fan, Ziqi Liu, Steven Chen, Herman Van Hovell, Bart Samwel, Mostafa Mokhtar, Rk Korlapati, Andy Lam, Yunxiao Ma, Vuk Ercegovac, Jiexing Li, Alexander Behm, Yuanjian Li, Xiao Li, Sriram Krishnamurthy, Amit Shukla, Michalis Petropoulos, Sameer Paranjpye, Reynold Xin, and Matei Zaharia. 2024. Adaptive and Robust Query Execution for Lakehouses At Scale. *Proc. VLDB Endow.* 17, 12 (2024), 3947–3959. https://doi.org/10.14778/3685800.3685818

[91] Yu Yan, Junfang Huang, Hongzhi Wang, Jian Geng, Kaixin Zhang, and Tao Yu. 2024. KnobCF: Uncertainty-aware Knob Tuning. arXiv:2407.02803 [cs.DB] https://arxiv.org/abs/2407.02803

[92] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 931–944. https://doi.org/10.1145/3514221.3517885

[93] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73. https://doi.org/10.14778/3421424.3421432

[94] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292. https://doi.org/10.14778/3368289.3368294

[95] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (2022), 3924–3936. https://doi.org/10.14778/3565838.3565846

[96] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116

[97] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1501–1512. https://doi.org/10.1109/ICDE48307.2020.00133

[98] Jintao Zhang, Chao Zhang, Guoliang Li, and Chengliang Chai. 2024. PACE: Poisoning Attacks on Learned Cardinality Estimation. *Proc. ACM Manag. Data* 2, 1 (2024), 37:1–37:27. https://doi.org/10.1145/3639292

[99] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2102–2114. https://doi.org/10.1145/3448016.3457291

[100] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 631–645. https://doi.org/10.1145/3514221.3526176

[101] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. 2023. An Efficient Transfer Learning Based Configuration Adviser for Database Tuning. *Proc. VLDB Endow.* 17, 3 (2023), 539–552. https://www.vldb.org/pvldb/vol17/p539-zhang.pdf

[102] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670. https://www.vldb.org/pvldb/vol15/p1658-zhao.pdf

[103] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2023. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *Proc. VLDB Endow.* 17, 4 (2023), 823–835. https://www.vldb.org/pvldb/vol17/p823-zhao.pdf

[104] Cai Zhou, Xiyuan Wang, and Muhan Zhang. 2024. Unifying generation and prediction on graphs with latent graph diffusion. *Advances in Neural Information Processing Systems* 37 (2024), 61963–61999.

[105] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (2023), 1466–1479. https://doi.org/10.14778/3583140.3583160

[106] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (2021), 1489–1502. http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf

Table 11: TPC-DS template splits into 10 folds (F1-F10) for OOD evaluation.

F1	F2	F3	F4	F5
1, 11, 28, 38, 39b, 42, 44, 59, 65, 88	5, 13, 18, 24b, 29, 31, 74, 77, 83, 95	10, 15, 22, 37, 41, 50, 66, 71, 96, 98	6, 12, 26, 33, 52, 62, 63, 76, 82, 99	8, 16, 19, 24a, 25, 32, 40, 70, 81, 89
F6	**F7**	**F8**	**F9**	**F10**
4, 7, 9, 14a, 17, 23b, 36, 46, 75, 78	34, 43, 47, 51, 53, 64, 67, 68, 87, 93	30, 39a, 45, 54, 55, 58, 73, 79, 86, 97	2, 3, 21, 27, 35, 49, 56, 60, 92, 94	14b, 20, 23a, 48, 57, 69, 72, 80, 84, 85, 90, 91

Table 12: Node encoding ablation study (in WAPE)

	ID				OOD			
	JOB		TPC-DS		JOB		TPC-DS	
	Mean	P90	Mean	P90	Mean	P90	Mean	P90
type	0.160	0.390	0.166	0.334	0.207	0.435	0.530	1.318
cbo	0.061	0.127	0.160	0.324	0.178	0.380	0.519	1.274
pred	0.134	0.317	0.165	0.335	0.211	0.510	0.517	1.223
hist	0.161	0.386	0.180	0.374	0.294	0.619	0.637	1.572
bitmap	0.077	0.171	0.160	0.317	0.257	0.699	0.572	1.390
¬type	0.059	0.120	0.161	0.314	0.177	0.420	0.543	1.261
¬cbo	0.061	0.122	0.166	0.340	0.172	0.446	0.593	1.418
¬pred	**0.057**	**0.113**	0.164	0.331	0.162	0.422	0.502	1.133
¬hist	0.058	**0.111**	0.169	0.341	**0.142**	0.366	0.495	1.158
¬bitmap	0.061	0.128	**0.158**	**0.312**	0.156	**0.346**	0.474	1.141
all	**0.057**	0.114	0.163	0.336	0.151	0.387	**0.456**	**1.056**

Table 10: Characteristics of the JOB benchmark datasets.

JOB Dataset	#Tables	#Queries	#Joins
Train	6	100,000	0–2
Synthetic	6	5,000	0–2
Light	6	70	1–4

A MORE DETAILS OF INITIAL BENCHMARK ANALYSIS

A.1 More Details of Experimental Setup

Our initial analysis runs on two identical Spark 3.5.0 clusters, each with six CentOS-based nodes. Every node is equipped with two 16-core Intel Xeon Gold 6130 processors, 768 GB of RAM, and RAID disks, all connected via 100 Gbps Ethernet. For each query, we collect the end-to-end latency under a randomly sampled Spark configuration, generated using Latin Hypercube Sampling [57] across 19 parameters [47] that affect resource allocation, runtime context, and query optimization. The objective is to predict this query latency based on the logical query plan, the chosen configuration, and relevant statistics at both the operator and query levels. We adopt a white-box modeling strategy, investigating various query-embedding and operator-encoding approaches, with a multilayer perceptron (MLP) serving as the downstream regressor. Evaluations consider both data in-distribution (ID) and out-of-distribution (OOD) scenarios.

JOB Benchmark. The Join Order Benchmark (JOB) [35] is built from an IMDb dataset containing over 2.5 million movie titles spanning 133 years, involving 234,997 companies and 4 million actors (3.88 GB total). Though originally 113 queries, JOB has become a standard for evaluating cardinality estimation [32, 61, 88], query optimization [9, 10], and query-performance prediction [74, 102, 103] through various extensions. Kipf et al.[32] generated 100,000 queries (`Train`) and 5,000 additional queries (`Synthetic`) by varying the number of tables (1–3) and join predicates (0–2), e.g.

```
SELECT COUNT(*)
FROM [1-3 tables]
WHERE [0-2 join predicates]
      [1-3 column predicates]
```

Seventy of the original JOB queries (`Light`) are further picked to serve as a test set with potentially higher complexity, involving up to four tables or three join predicates. Table 10 summarizes these datasets: `Train` (training/validation), `Synthetic` (ID test), and `Light` (OOD test). Any join follows a star schema with *title* as the fact table. Typically, `Train` is split 90%/10% for training/validation, and then tested on both `Synthetic` (ID) and `Light` (OOD).

TPC-H and TPC-DS Workloads. TPC-H [2] includes 22 decision-making analytical queries, while TPC-DS [1] provides 102. TPC-H queries tend to be simpler, whereas TPC-DS involves more complex multi-joins, window operations, subqueries, and aggregations. Following previous work [47, 55, 102, 103], we treat each benchmark query as a *template* and generate parameterized queries by substituting constants. We create 50,000 distinct queries each for TPC-H and TPC-DS, with each template yielding about 2272 queries in TPC-H and about 500 in TPC-DS. Every query is executed under a distinct Spark configuration, again sampled via Latin Hypercube Sampling. For ID evaluations, we partition each benchmark with stratified sampling so that every template appears in the training/-validation sets (80%/10%) and test set (10%). For OOD evaluations, we split the 22 TPC-H and 102 TPC-DS templates into 10 folds, as illustrated in Table 11 for TPC-DS. In each run, one fold is used exclusively for testing (OOD), while the remaining folds provide training and validation; the final OOD performance is averaged over all 10 folds.

Metrics. We measure performance using two sets of metrics. First, we compute the *weighted mean absolute percentage error* (WMAPE) and the 50th/90th percentiles (P50/P90) of the *weighted absolute percentage error* (WAPE). Second, we report *Q-Error* [59], defined as $\max(\frac{y}{\hat{y}}, \frac{\hat{y}}{y})$, including its 50th (P50), 90th (P90), 99th (P99), and maximum (Max) percentiles.

A.2 Node Encoding Comparison

We next present an ablation study on node (operator) encodings for JOB and TPC-DS, using GTN as the structure encoding approach. Table 12 shows three main strategies for encoding features in each node: (i) using each encoding option alone (`type`, `cbo`, `pred`, `hist`, `bitmap`); (ii) leaving out exactly one option (`¬type`, `¬cbo`, `¬pred`, `¬hist`, `¬bitmap`); and (iii) employing all options together (`all`).

In the ID setting, some single-feature approaches are already competitive with the best. For example, `cbo` or `bitmap` alone yields a WMAPE of 0.160 in TPC-DS, which is close to 0.158 (the best ID result). In JOB, `cbo` likewise achieves near-top performance. Moreover, omitting a single feature rarely harms accuracy compared to using all features: WMAPE differs by only 0.004–0.011,

Algorithm 1: JOB Query Generation Process

Input: Number of queries n;
Fact table T_{title};
Dimension tables $\mathcal{T}_d$;
Random sampler $\mathcal{S}$ (without replacement, default size 1);
Set of arithmetic operators OP;
Precomputed constants for each column, stored in a hashmap $\mathcal{M}$
Output: List of queries $\mathcal{Q}$

Initialize $\mathcal{Q} \leftarrow \{\}$;
foreach $i \in [1, n]$ **do**
  $n_j \leftarrow \mathcal{S}(\{3, 4\})$;
  $\mathcal{P} \leftarrow \mathcal{S}(\mathcal{T}_d,\ \text{size} = n_j)$;
  $\mathcal{F} \leftarrow \{T_{\text{title}}\} \cup \mathcal{P}$;
  // Prepare WHERE clause
  Initialize $\mathcal{W} \leftarrow \{\}$;
  // Generate join predicates
  foreach $T_d \in \mathcal{P}$ **do**
    Add T_{title}.id = T_d.movie_id to $\mathcal{W}$;
  // Generate column predicates
  $n_c \leftarrow \mathcal{S}(\{2, 3\})$;
  $\mathcal{P}_c \leftarrow \mathcal{S}(\mathcal{P},\ \text{size} = n_c)$;
  foreach $T \in \mathcal{P}_c$ **do**
    $c \leftarrow \mathcal{S}(T.\text{nonindex\_columns})$;
    $op \leftarrow \mathcal{S}(OP)$;
    $v \leftarrow \mathcal{S}(\mathcal{M}[c])$;
    Generate a predicate $p \leftarrow T.c\ op\ v$;
    $\mathcal{W}$.add(p);
  Form WHERE clause $w \leftarrow$ " AND ".join($\mathcal{W}$);
  $q \leftarrow$ "SELECT COUNT(*) FROM " $\mathcal{F}$ " WHERE " w;
  $\mathcal{Q}$.add(q);
return $\mathcal{Q}$;

and P90 WAPE by 0.017–0.029 across both datasets. This robustness arises because multiple features (`cbo`, `hist`, `bitmap`) overlap in the cardinality information they capture, and the model is easier to generalize with limited information in the ID setting.

In the OOD setting, overall performance degrades, and relying on just one encoding option is no longer sufficient since unfamiliar query structures demand richer information to generalize. For TPC-DS, the most complex workload, excluding one feature still lags behind the all-features option. In comparison, for JOB, dropping `hist` or `bitmap` can still match the accuracy of all-features, suggesting that these two features convey similar information for JOB but contribute more distinct signals in TPC-DS. Overall, **using `all` operator features is the most robust choice in both ID and OOD settings** (**R6**), consistent with the results reported in [103].

B DATA GENERATION METHODS

B.1 JOB Query Generation Process

Algorithm 1 provides the details of our star join algorithm for JOB, which follows these main steps:

B.2 TPC-DS Star-Join Query Generation

Based on domain knowledge that TPC-DS queries often join fact tables with dimension tables, we adapt the star-join approach previously used in the JOB environment to TPC-DS's seven fact tables, each with 3-18 foreign keys. For a given fact table with n foreign keys, there are up to 2^n possible star-join "signatures" (i.e., combinations of dimension tables). We select up to 1,000 signatures per fact table as star-join templates, ranging from the fewest to the large number of dimension tables. Each template is further parameterized using various column filters, producing 10 runnable queries per template and yielding 55,180 synthetic queries, referred to as `DS-EXT`.

C ADDITIONAL EXPERIMENTS

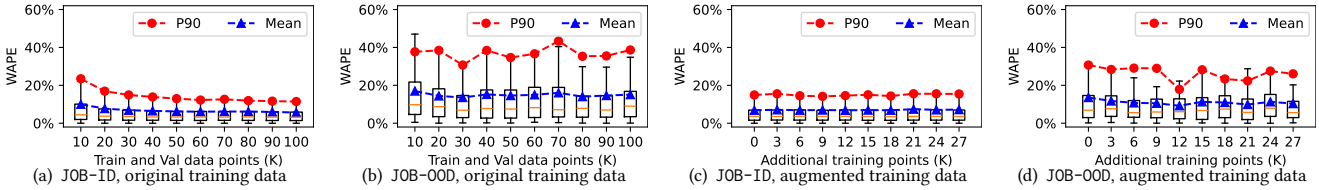
C.1 Impacts of Training Data for JOB

Impact of Training Set Size for JOB Figures 9(a)-9(b) present WAPE for in-distribution (`JOB-ID`) and out-of-distribution (`JOB-OOD`) test queries, respectively, as the training set increases from 10 K to 100 K. Each boxplot shows the distribution of errors, with blue triangles for mean and red circles for the 90th percentile (P90). First, expanding the dataset from 10 K to 30 K yields significant gains in both ID and OOD. For instance, `JOB-ID` WMAPE drops from 10% (P90 24%) to 6.9% (P90 15%), while `JOB-OOD` falls from 17% (P90 38%) to 14% (P90 31%). Second, beyond 30 K, `JOB-ID` continues to improve (reaching 5.7% WMAPE at 100 K), but `JOB-OOD` largely plateaus or worsens (up to 15% WMAPE at 100 K). This indicates that additional queries of limited diversity can overfit ID patterns without broadening OOD coverage. Third, P90 errors in `JOB-OOD` vary noticeably (31%–39% WAPE), indicating that the small test set (70 queries) is sensitive to outliers. Overall, 30 K emerges as an optimal balance: it sufficiently reduces OOD errors while avoiding diminishing returns or additional training overhead in ID. We made similar observations using the Q-error metric; the plots are omitted in the interest of space.

Effect of Adding Synthetic Queries. Figures 9(c)-9(d) show WAPE results for `JOB-ID` and `JOB-OOD` after incrementally adding up to 27 K synthetic `JOB-EXT` queries to the existing 30 K train set. First, `JOB-ID` errors remain nearly unchanged: WMAPE hovers around 7%, indicating the original ID distribution is well-captured and not adversely affected by the extra queries. Second, `JOB-OOD` WMAPE steadily improving from 14% (P90 31%) to about 9.2% (P90 18%) at 12 K new queries, becoming closer to `JOB-ID` (7% WMAPE, P90 14%). Finally, with 27 K synthetic queries, `JOB-OOD` achieves its lowest boxplot distribution in WAPE, without harming the ID performance.

C.2 Per-fold OOD Analysis for TPC-DS

As shown in Figure 10(a), augmenting the baseline with additional datasets does not affect ID performance; the model already captures ID queries sufficiently. However, certain folds, such as F8 and F9, exhibit substantial error reductions when synthetic queries (star-join or JOB) are introduced, whereas other folds, e.g., F4 and F7, ranges between 50%-70% WMAPE. Further analysis shows that these high-error folds predominantly involve unfamiliar operators that are not well covered by either the original training set or the extended query set. These operators including `UNION`, `WINDOW`, and `EXPAND`: for example, `UNION` in Q76 yields WMAPE 323%, `WINDOW` in Q67 yields 86%, `EXPAND` in Q36 yields 86%. To validate, we re-evaluate the OOD performance on queries excluding these operators (`Drop`), reducing

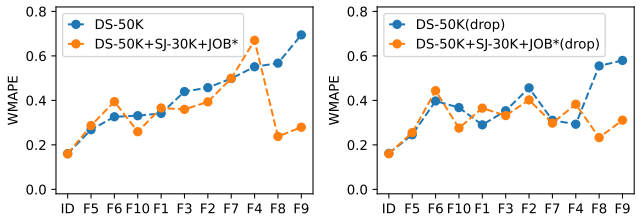


(a) JOB-ID, original training data (b) JOB-OOD, original training data (c) JOB-ID, augmented training data (d) JOB-OOD, augmented training data

Figure 9: JOB errors in ID and OOD against the size of training set

Table 13: WAPE of the plan embedder for different modeling targets

	JOB				TPC-DS			
	ID		OOD		ID		OOD	
	Mean	P90	Mean	P90	Mean	P90	Mean	P90
Latency (original)	0.057	0.115	0.151	0.387	0.163	0.336	0.456	1.056
Latency (GTN*)	0.068	0.148	0.089	0.190	0.158	0.326	0.350	0.798
Shuffle Size (original)	0.061	0.086	0.079	0.089	0.144	0.065	1.430	4.848
Shuffle Size (GTN*)	0.064	0.109	0.040	0.050	0.045	0.051	0.836	2.176
Cloud Cost (original)	0.051	0.098	0.113	0.299	0.138	0.087	1.168	3.940
Cloud Cost (GTN*)	0.057	0.120	0.059	0.133	0.051	0.067	0.724	1.894



(a) All TPC-DS queries (b) ¬ WINDOW, UNION, EXPAND

Figure 10: TPC-DS OOD Per-fold WMAPE breakdown, sorted by the baseline (DS-50K) WMAPEs.

the high error rates in F4 and F7 to below 40% (Figure 10(b)) and the OOD WMAPE to 32.2% (Table 5). When we further exclude queries containing additional operators `CROSS-JOIN` and `SUBQUERY`, which are not covered by the extended training set (`Drop+`), we achieve a WMAPE of 28.0% in the OOD setting (Table 5).

C.3 Generality across different modeling targets.

Besides *query latency*, we evaluate our best model for two additional modeling targets: *shuffle size*, which belongs to the task of cardinality estimation in big data systems, and *cloud cost* [69], a metric needed for cost-performance optimization. For the JOB benchmark, our augmented data and customized structure encoding generalizes well to shuffle size and cloud cost, improving OOD accuracy and reducing the gap with the ID performance. Specifically, shuffle size predictions achieve lower WMAPE and P90 errors in the OOD case because the augmented JOB queries fully cover the OOD query space, including more complex joins with longer running times and thus benefiting metrics that emphasize long-running queries. For TPC-DS, augmenting data significantly improves predictions by: (1) greatly reducing WMAPE for shuffle size (from 14.4% to 4.5%) and cloud cost (from 13.8% to 5.1%) in the ID scenario, and (2) notably improving performance in the OOD scenario, where WMAPE drops by 59% for shuffle size and by 205% for cloud cost. We observe that predicting shuffle size in the OOD scenario is more challenging because, without prior observations to adjust predictions for specific query plan patterns, predictions become more sensitive to the accuracy of cardinality estimations at the compile time, which are often unreliable in Spark SQL.

D END-TO-END EVALUATIONS

Finally, we evaluate the end-to-end model performance impact to two real-world tasks: job scheduling and auto-scaling.

D.1 Job Scheduling.

Query performance prediction is crucial for job scheduling [79], where the system must classify incoming queries to different performance categories to optimize resource allocation. We classify the queries into short and long categories based on their P80 latency, and simulate the job scheduling process to compare the effects of different model options. The simulation assumes a fixed cluster for short queries and two burst clusters for long queries. The fixed cluster is able to run up-to five queries in parallel, while each burst cluster is designed to run one long queries at a time, with three seconds overhead of initialization and shutdown. Underestimating a query from long to short leads to more delay for the short queue, and overestimating a query from short to long leads to more overhead of using the burst clusters force more long queries to wait for the cluster to be available.

E ADDITIONAL JOB EXPERIMENTS

E.1 Hyperparameter tuning

Initial search of best learning parameters yielded $H = 4$ (number of attention heads), $\eta = 0.003$ (learning rate).

We allocate one cluster node per model for seven days. We specifically vary the number of layers (2, 3, 4), the hidden dimensions of the model (64, 128, 256), and the final representation (256, 300). After a week, we cut off the queued training configurations not yet explored. We use TPC-DS ID as our training set for the hyperparameter tuning.

E.2 Detailed evaluation on JOB*

We evaluate each data improvement on JOB Full independently. Aggregated results appear in Table 14. Figure 11 shows a comparison between several models on the longer running queries of JOB Full.

F SNOWFLAKE QUERY GENERATION

Our query generation is divided into four main components:

(1) Column-wise statistics collection
(2) Join structure generation

Table 14: Data augmentation evaluation on JOB Full

	JOB (30K)		+EXTV2		+CEB		+EXTV2+CEB	
	Mean	P90	Mean	P90	Mean	P90	Mean	P90
GTN	0.382	0.873	0.284	0.581	0.319	0.669	0.251	0.496
GTN$_{\neg\ PE}$	0.344	0.764	0.275	0.463	0.281	0.603	0.268	0.650
GAT	0.373	0.770	0.264	0.557	0.318	0.752	0.265	0.521
GAT$_{\neg\ PE}$	0.372	0.839	0.274	0.652	0.280	0.635	0.264	0.613
GCN	0.399	0.847	0.277	0.582	0.366	0.868	0.290	0.632
TTN	0.475	0.953	0.291	0.619	0.326	0.747	0.283	0.674
TCNN	0.343	0.668	0.292	0.694	0.332	0.695	0.284	0.548
Masked GTN	0.390	0.875	0.312	0.653	0.370	0.770	0.266	0.556
Masked GTN$_{\neg\ PE}$	0.357	0.862	0.265	0.522	0.296	0.712	0.254	**0.441**
Masked GAT	0.379	0.769	0.297	0.649	0.347	0.734	0.258	0.487
Masked GAT$_{\neg\ PE}$	0.383	0.884	0.284	0.624	0.299	0.649	0.269	0.666
Masked GCN	0.414	0.881	0.302	0.614	0.326	0.814	0.280	0.615
Masked TTN	0.475	0.953	0.269	0.600	0.371	0.782	0.276	0.569
Masked TCNN	0.451	0.767	0.290	0.619	0.324	0.732	0.265	0.634

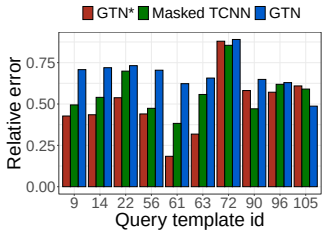


Figure 11: Latency prediction improvement with augmented data

(3) Predicate Generation
(4) LLM augmentation

Our query generation produces two datasets for training: the Snowflake dataset and the LLM-augmented dataset. The Snowflake dataset consists of queries structured as a simple join with a predicate; we test these queries to ensure they return a non-empty answer set. The LLM-augmented dataset is created by using an LLM to modify Snowflake queries, yielding queries with more complex and diverse structures.

F.1 Column-wise statistics

To guarantee not empty answer sets we collect histograms and most common values (MCV) from the datasets to guide the predicate generation. We extract histograms to generate range predicates and most common values for equality and IN-list predicates.

F.2 Join structure generation

We generate a unique join structure based on a subgraph of the database, where each table is a node and each foreign key is an edge. We take as input the foreign-key graph, the set of previously seen subgraphs, the maximum number of hops allowed, and the probability of keeping an edge; we then output a new, unseen subgraph that serves as a unique **join signature**. The algorithm uses a breadth-first search: starting from a fact table, for each neighbor we use the *keep edge* parameter to randomly decide whether to include that foreign key. If we include a foreign key, we add the connected table to the BFS queue and continue until we reach the maximum number of hops. If the newly generated subgraph has been seen before, we retry until we either find a new subgraph or exhaust the retry limit.

F.3 Predicate Generation

Given a join signature we generate a list of predicates that don't result in an empty answer set. We take as input a subgraph of the foreign key graph, and a list of desired predicates (range, equality and IN-list); we generate the amount of desired predicates while guaranteeing that each individual predicate returns a non-empty answer set. For each predicate we start by randomly selecting a column to add a predicate on. If we are to generate a range predicate we have a parameter of the minimum selectivity, and thus select a range that guarantees that selectivity based on a previously computed histogram. If we are to generate an equality predicate we use a parameter of minimum relative frequency and use most common values to find a valid candidate; if no candidate is available we skip the generation of this predicate. A similar process to equality is done for IN-list predicate with the addition of extra elements taken from the histogram boundary to fill a desired list length. Finally to make sure that a set of predicates don't have an empty answer set we run the query to check the size.

F.4 LLM augmentation

In LLM augmentation we choose one query generated by the snowflake query generation pipeline. As swon in Listing 1 we then use an LLM to add one of the following extensions:

(1) Group by
(2) Group by rollup
(3) Group by order by
(4) Nested join exists
(5) Window function
(6) Self join
(7) Nested join with in predicate
(8) Inequality join
(9) Outer join

We manually add `Union` queries by using queries that derive from the same join signature but vary in predicates.

Listing 1: LLM Prompt Template

```
System: "You are a database expert writing
TPC-DS queries to test your current database
with challenging and diverse queries."

User: "Modify this query by adding
a self-join, preserving all existing
predicates: {snowflake_query}"
```

F.5 Generated Datasets

Using both of our pipelines (LLM and Snowflake) we generate datasets for the TPCDS and IMDB datasets to boost the performance of models in the TPCDS and JOB benchmark respectively.

Dataset	# of Tables	Templates	# Queries
TPC-DS	1–13	99	48793
TPC-DS Star join filtered	1–4	4821	30000
TPC-DS Star join complete	1–9	5518	55180
TPCDS Snowflake	2–14	2043	9741
LLM 1	2–15	19	5016
LLM 2	2–14	18	3310

Table 15: TPCDS query statistics

Dataset	# of Joins	Templates	# Queries
JOB Full	5–16	31	113
JOB light	1–4	23	70
JOB train	0–2	21	100000
CEB	5–15	15	13646
JOB synthetic	0–2	20	5000
JOB starjoin	3–4	180	27371
JOB snowflake	1–10	859	20311

Table 16: JOB dataset signature and query statistics

Dataset	EQUALITY	RANGE	IN LIST	LIKE
JOB Full	Yes	Yes	Yes	No
JOB light	Yes	Yes	No	No
JOB train	Yes	Yes	No	No
CEB	Yes	Yes	Yes	Yes
JOB synthetic	Yes	Yes	No	No
JOB starjoin	Yes	Yes	No	No
JOB snowflake	Yes	Yes	Yes	No

Table 17: Predicate supported across JOB and CEB datasets. Additionally CEB is the only one that has Group by and Order by.

F.5.1 JOB SNOWFLAKE. We generate dataset containing 20k queries with 859 unique join signatures. For the IMDB benchmark we consider only the `title` table as the only fact table.

F.5.2 TPCDS SNOWFLAKE 10k. We generate 9741 queries with 2043 distinct join signatures.

F.5.3 TPCDS LLM 5k. Using a sample of 2000 distinct joins We generate 5016 queries that contain the operations shown in Table 18. We add three different predicates with different probabilities, equality (61% of queries), range predicates (91% of queries) and IN-list predicates (25% of queries).

Count	Operation
734	group by order by
717	group by
576	inequality join
617	outer join
565	self join
454	window function
270	nested join with exists
372	nested join with IN
463	group by rollup
248	union

Table 18: LLM Operation frequencies

F.5.4 TPCDS LLM2 3.5k. We generate an LLM dataset from join signatures not used in TPCDS SNOWFLAKE 10k picking the top 500 signatures which embeddings are the furthest away and thus offer the most valuable information to the model. The distribution is shown in Table 19.

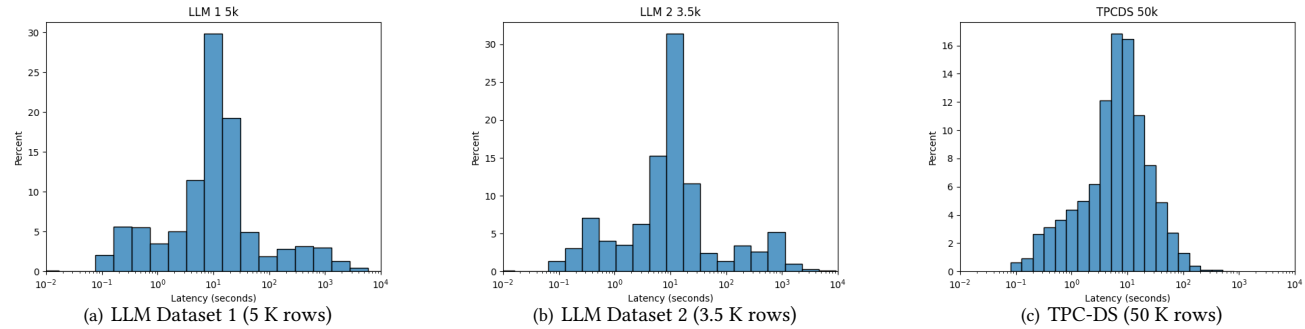
Count	Operation
674	group by order by
489	group by
247	inequality join
397	outer join
151	self join
224	window function
414	nested join with exists
412	nested join with IN
328	group by rollup

Table 19: LLM2 Operation frequencies

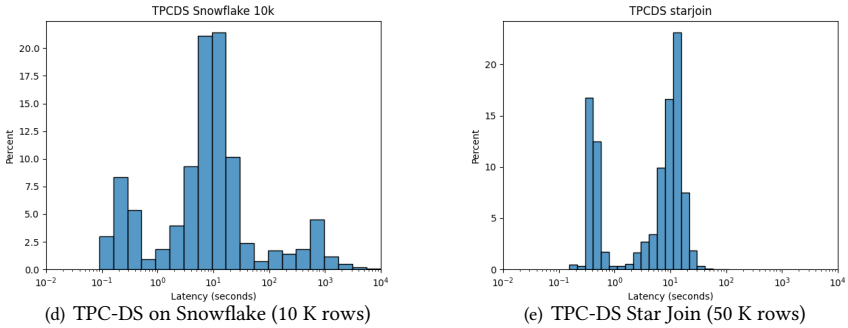
F.5.5 TPCDS. Our test data set is composed of TPCDS and JOB. The actual distribution of operators in TPCDS is shown in Table 20.

Probability	Operation
0.718	Equality
0.372	Range
0.388	IN-list
0.834	Group by
0.883	Order by
0.174	Union
0.019	Anti join
0.136	Inequality join
0.214	Self join
0.097	Outer join
0.029	Many-to-many join
0.175	Nested join

Table 20: Operation frequencies



(a) LLM Dataset 1 (5 K rows) (b) LLM Dataset 2 (3.5 K rows) (c) TPC-DS (50 K rows)



(d) TPC-DS on Snowflake (10 K rows) (e) TPC-DS Star Join (50 K rows)

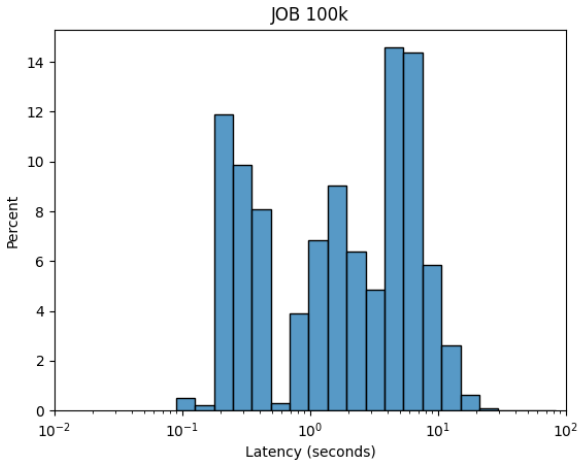
Figure 12: Query latencies for the datasets in TPCDS.

G QUERY FILTERING USING GTN EMBEDDING SPACE

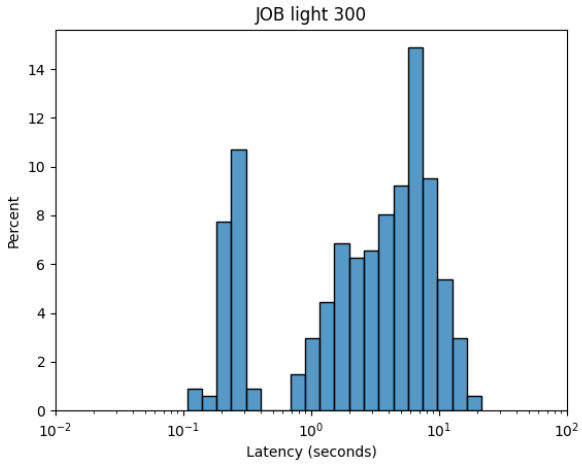
We filter the queries based on the following procedure:

(1) PCA is applied on the training set to determine the number of dimensions to keep with UMAP. We set the threshold at the minimum number of dimensions such that the cumulative sum of the explained variance is above 90%.
(2) A UMAP embedder [56] is fitted to the training data (training data GTN embeddings).
(3) The queries to be filtered are passed through the GTN to extract their embeddings.
(4) The embeddings are passed through the fitted UMAP.
(5) The pairwise distance between every synthetic query and the training set is computed.
(6) For each synthetic query, select the 100 closest queries in the training set, and compute the average distance.
(7) (Optional) When trained using a k-fold strategy, the metrics are aggregated across the folds (for instance, by taking the median)

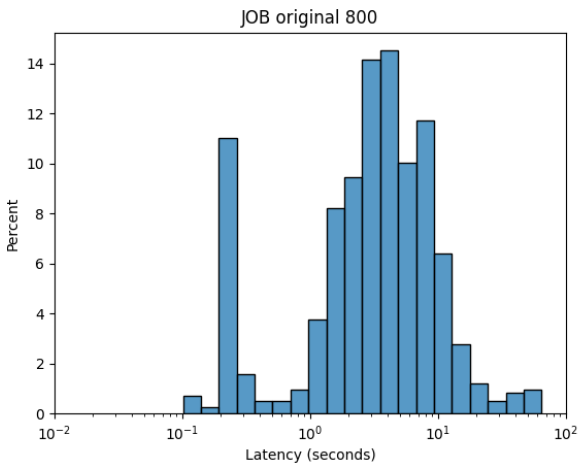
The threshold applied can be determined based on the number of queries to be kept (e.g., 30% of the queries or 10,000 queries).



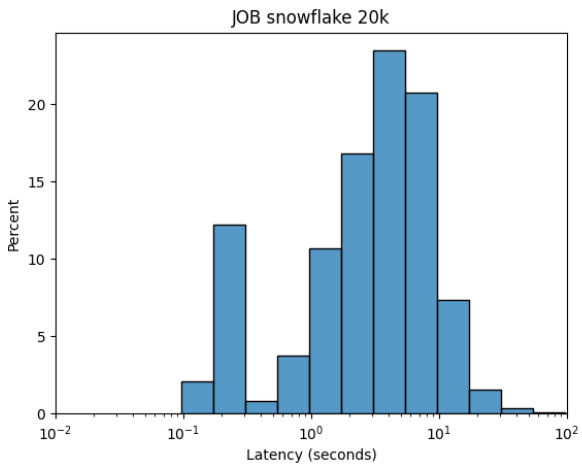
(a) JOB (100 K rows)



(b) JOB-light (300 rows)

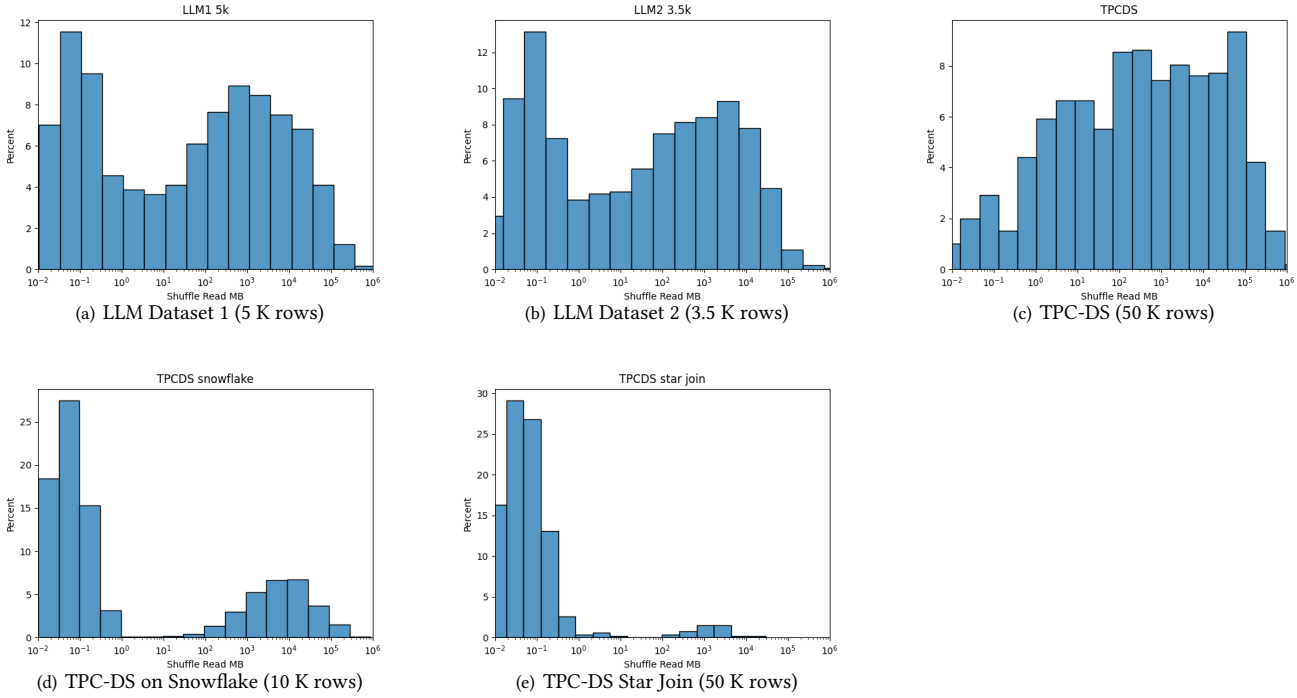


(c) JOB-original (800 rows)



(d) JOB on Snowflake (20 K rows)

Figure 13: Query latencies for the datasets in IMDB dataset.



(a) LLM Dataset 1 (5 K rows)
(b) LLM Dataset 2 (3.5 K rows)
(c) TPC-DS (50 K rows)
(d) TPC-DS on Snowflake (10 K rows)
(e) TPC-DS Star Join (50 K rows)

Figure 14: Query shuffle read for datatasets in TPCDS.