How Good are Learned Cost Models, Really? Insights from Query Optimization Tasks

ROMAN HEINRICH, Technical University of Darmstadt & DFKI Darmstadt, Germany
MANISHA LUTHRA, Technical University of Darmstadt & DFKI Darmstadt, Germany
JOHANNES WEHRSTEIN, Technical University of Darmstadt, Germany
HARALD KORNMAYER, Duale Hochschule Baden-Württemberg (DHBW) Mannheim, Germany
CARSTEN BINNIG, Technical University of Darmstadt & DFKI Darmstadt, Germany

Traditionally, query optimizers rely on cost models to choose the best execution plan from several candidates, making precise cost estimates critical for efficient query execution. In recent years, cost models based on machine learning have been proposed to overcome the weaknesses of traditional cost models. While these models have been shown to provide better prediction accuracy, only limited efforts have been made to investigate how well *Learned Cost Models* (LCMs) actually perform in query optimization and how they affect overall query performance. In this paper, we address this by a systematic study evaluating LCMs on three of the core query optimization tasks: *join ordering*, *access path selection*, and *physical operator selection*. In our study, we compare seven state-of-the-art LCMs to a traditional cost model and, surprisingly, find that the traditional model often still outperforms LCMs in these tasks. We conclude by highlighting major takeaways and recommendations to guide future research toward making LCMs more effective for query optimization.

CCS Concepts: • **Information systems** → **Query optimization**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: Cost Estimation, Learned Cost Models

ACM Reference Format:
Roman Heinrich, Manisha Luthra, Johannes Wehrstein, Harald Kornmayer, and Carsten Binnig. 2025. How Good are Learned Cost Models, Really? Insights from Query Optimization Tasks. *Proc. ACM Manag. Data* X, X, Article XX (February 2025), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cost Estimation is Crucial for Databases. Accurate cost prediction of a query plan is essential for optimizing the query performance in a database. During query optimization, estimated costs of various candidate plans guide the plan selection for execution [21]. Consequently, accurate cost estimates are pivotal in query optimization. In the worst case, incorrect cost estimates can lead to the selection of highly unfavorable plans with a runtime that is several factors higher than the optimal one. However, it is well known that providing accurate cost estimates is difficult and inaccurate estimates influence the results of finding an optimal plan significantly [12, 19].

Authors' addresses: Roman Heinrich, Technical University of Darmstadt & DFKI Darmstadt, Darmstadt, Germany; Manisha Luthra, Technical University of Darmstadt & DFKI Darmstadt, Darmstadt, Germany; Johannes Wehrstein, Technical University of Darmstadt, Darmstadt, Germany; Harald Kornmayer, Duale Hochschule Baden-Württemberg (DHBW) Mannheim, Mannheim, Germany; Carsten Binnig, Technical University of Darmstadt & DFKI Darmstadt, Darmstadt, Germany.

2836-6573/2025/02-ARTXX $15.00
https://doi.org/XXXXXXX.XXXXXXX

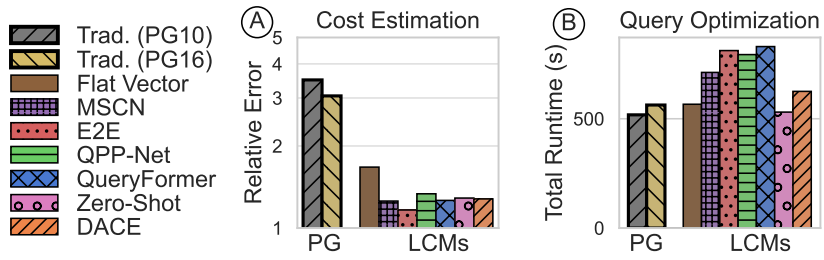

Fig. 1. Ⓐ LCMs outperform traditional approaches in terms of cost estimation on an unseen IMDB dataset. Ⓑ However, when optimizing a workload (JOB-Light) for join order, the traditional PostgreSQL (PG) model still performs best.

The Need for Accurate Cost Models. In recent decades, classical rule- or heuristic-based cost estimation methods have been the backbone for query optimization in commercial DBMSs. However, since they rely on heuristics and simple analytical models, traditional methods struggle with accuracy and provide estimates that often deviate by orders of magnitude from the actual execution costs [21], leading to sub-optimal query execution plans with prolonged runtimes. Thus, much research has been dedicated to making cost models more precise and improving overall performance [11, 15].

The Rise of Learned Cost Models. Supported by the high potential of Machine Learning (ML), many *Learned Cost Models* (LCMs) have been proposed over the last years [1, 2, 5, 8, 10, 12, 13, 18, 22, 23, 25, 27, 39, 40, 44, 46]. These models typically leverage actual costs from executing training queries to learn patterns and predict the execution cost of new queries. The main promise of LCMs lies in the fact that they can better capture complex data distributions and inter-dependencies with workloads and thus potentially lead to more efficient query processing and shorter query runtimes. As a result, many recent papers about LCMs [13, 23, 31] show that they can significantly outperform classical cost models that are employed in traditional database systems such as PostgreSQL in terms of cost prediction.

Do LCMs Really Help in Query Optimization? However, while LCMs have led to improvements in accuracy, a core question is if they lead to improvements for query optimization and to what extent. Surprisingly, most existing evaluations primarily focus on the accuracy of the cost estimation task and largely neglect a deeper analysis of how these cost models actually improve query optimization [2, 10, 13, 18, 23, 25, 27, 31, 40, 44, 46]. In this paper, we argue that accuracy alone is not meaningful, as it cannot reflect important tasks in query optimization, such as the ranking and selection of plans. We thus aim to close this gap and conduct a systematic study to assess *how good learned cost models really are for query optimization?* Unfortunately, the results of our study are rather grim, as shown by Figure 1, which highlights some of the findings of our study. In Figure 1 Ⓐ, we compare the accuracy of a broad spectrum of recent LCMs. Here, in terms of prediction error, the traditional approach PostgreSQL (PG, black bar) is outperformed by all LCM competitors (colored bars) on the IMDB dataset[1] However, the picture is very different when using the cost models for finding optimal join orders on the JOB-Light benchmark [18]. Figure 1 Ⓑ shows that the total query runtime is *not* improving when using LCMs for join ordering. Here, *none* of the LCMs is able to provide better selections than PostgreSQL, resulting in a higher total runtime of selected plans on the JOB-Light benchmark of up to 832s, whereas PostgreSQL achieves 510s.

[1]We report the median Q-error as standard metric for cost models. It defines the relative deviation of the predicted from the actual cost. A perfect prediction has a Q-error of 1. See Section 3.3 for more details on the setup.

A Novel Evaluation Study. From these results, it becomes clear that focusing alone on the accuracy of cost estimation is not sufficient. As such, in this paper, we provide a systematic study to shed some light on the question of why LCMs fail to enable better optimizer decisions. To answer this question, we have chosen *three* of the most important tasks of query optimization (*join ordering*, *access path selection*, and *physical operator selection*) and analyze whether or not LCMs are able to improve plan selection. As a main contribution, we evaluate a set of recent LCMs that cover a broad spectrum of approaches proposed in the literature and compare their impact on query optimization against the traditional cost model of PostgreSQL. We suggest a task-specific, fine-grained evaluation strategy for each downstream task that goes beyond prediction accuracy, assessing how LCMs affect query optimization.

Key Insights of Our Study. Our evaluation reveals three key insights that we summarize in the following:

(1) *High accuracy in cost is not sufficient*: Across all analyzed query optimization tasks, we observed that it is insufficient to focus solely on the prediction accuracy of plan costs. Instead, LCMs need to fulfill other properties such as reliable ranking and selection of plans. Moreover, existing LCMs majorly only optimize for median prediction errors while possessing high errors in the tail, leading to large over- and underestimation and making LCMs prone to select non-optimal plans.

(2) *Training data matters*: As LCMs are typically trained on queries that have been pre-optimized by a traditional query optimizer, their training data is biased towards near-optimal plans. However, during query optimization, LCMs must predict costs for both optimal and non-optimal plans. Moreover, training data quality can also introduce other biases, especially when timeouts are used during query execution for training data collection, distorting the LCMs understanding of "bad" plans. For instance, query plans with nested-loop joins may often timeout before completion and thus, only the cases where nested-loop joins are beneficial are included in the training data.

(3) *Don't throw away, what we know*: Traditional cost models often deviate significantly from actual costs in their estimates. However, they incorporate extensive expert knowledge based on years of experience. In our paper, we found that using their estimates as input to LCMs is highly beneficial as it significantly improves the cost estimates for query optimization tasks.

Consequences for LCMs. Overall, we believe that the results of our study can guide future research and development efforts toward more reliable ML-based cost estimation that makes informed decisions about query optimization tasks. We discuss some directions based on the evidence of this paper that will help to provide LCMs, which actually benefits query optimization. Furthermore, to enable the research community to build on our results, we made the source code, models, and all the evaluation data publicly available[2]

Outline. We first provide a background in cost estimation in Section 2. Next, we present our evaluation methodology in Section 3, including a taxonomy of recent LCMs. In the subsequent sections, we then evaluate the downstream tasks of join ordering (Section 4), access path selection (Section 5), and physical operator selection (Section 6). We provide our recommendations for LCMs in Section 7 and summarize this paper in Section 9.

[2]**Source Code**: `https://github.com/DataManagementLab/lcm-eval`;
Experimental data & trained models: `https://osf.io/rb5tn/`

2 BACKGROUND OF COST ESTIMATION

This section first gives a brief overview of classical and learned cost estimation. Afterwards, we describe the learning procedure of LCMs and provide a taxonomy that guides our selection of recent LCMs for this study in Section 3.

2.1 Traditional & Learned Cost Estimation

Traditional Cost Estimation. Precise cost estimates for different plan candidates in a database are crucial for the query optimizer to select optimal plans from a large search space. Thus, a lot of engineering effort has been spent since the beginning of database development to estimate the execution costs of a query plan. Most database systems such as MySQL [37], Oracle, PostgreSQL, or System R [3] use hand-crafted cost models to reason about the execution costs of a query plan. These models typically provide a cost function for each physical operator in a query plan that estimates its runtime costs according to CPU usage, I/O operations, memory consumption, expected tuples, and random or sequential page accesses. However, due to the wide variety of data, queries, and data layouts, traditional cost models need to make simplifying assumptions (e.g., independence of attributes). These often lead to incorrect predictions of the execution cost. Consequently, the query optimizer makes sub-optimal decisions that degrade the query performance by increasing its runtime [21].

Learned Cost Estimation. The need to improve prediction accuracy and the rise of machine learning motivated the idea of LCMs. The main idea is to approximate the complex cost functions with a learned model. Generally, a typical model learns from previous query executions to predict execution costs like runtime. In contrast to traditional cost models, the promise of LCMs is that they can better learn arbitrarily complex functions. Thus, improved prediction accuracy can be expected in contrast to traditional approaches based on simplifying assumptions. Overall, the higher accuracy is expected to lead to a selection of query plans with improved query performance.

2.2 Learning Procedure of LCMs

For our study, we look at effects that also result from the learning procedure of LCMs. As such, we briefly review the traditional procedure as depicted in Figure 2 to provide the necessary background: Ⓐ At first, a workload generator is used to create a large set of randomized, synthetic SQL-Strings that involve a variety of representative query properties such as filter predicates, joins, or aggregation types. Ⓑ These queries are executedses (e.g., an airline or movie database) to collect the actual costs of queries. An important aspect here is that training procedures of many LCMs leads to biases in the dataset due to timeouts and pre-optimized queries, as discussed later. Ⓒ Next, various information is extracted from the workload execution. Most importantly, the physical query plans are extracted, which serve as input to cost models. In addition to physical plans, LCMs require different information, such as data characteristics like histograms or sample bitmaps. Ⓓ+Ⓔ Finally, the workload (i.e., plans and runtime) is then split for training and testing the LCMs.

2.3 Taxonomy of LCMs

LCMs developed in the last years differ in various dimensions. This section provides a brief taxonomy of recent LCMs to structure the different methodological approaches. This taxonomy will guide the selection of LCMs that we use in this study and ensure that we cover the different methodologies to analyze how they affect the ability of LCMs to support query optimization.

Input Features. The first crucial dimension is the input features that a LCM learns from. The input features are extracted from the executed workloads (cf. Figure 2Ⓒ). The query plan and the underlying data distribution need to be modeled so that a LCM is informed to make reasonable

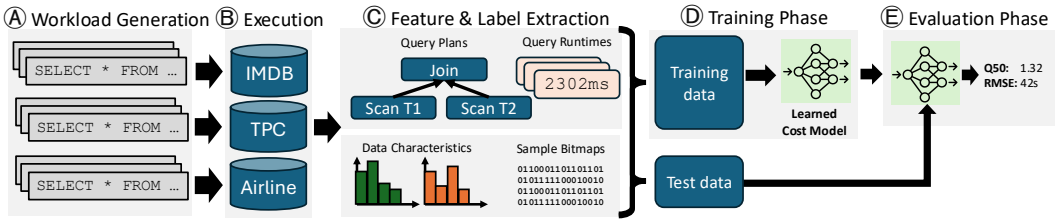

Fig. 2. Learning procedure of LCMs. Ⓐ Generation of synthetic training queries. Ⓑ Query execution on training databases. Ⓒ Feature (query plans, data characteristics, and sample bitmaps) and label (query runtimes) extraction to generate the training and test dataset. Ⓓ Training of the LCM with supervised learning. Ⓔ Evaluation of the LCM against unseen test data.

predictions about the execution costs, which in turn affects query optimization, as we will show. However, LCMs make use of different information for cost estimation.

(1) **SQL-String vs. Query Plans**: Some of the first models rely on the SQL string to describe a query, as it gives insights about the tables, predicates, and joins. However, details of the execution plan, such as physical operators or the order of joins, are not described there. Thus, most LCMs utilize the physical query plan, which includes the operators (e.g., scans, joins) and physical operator types (e.g., nested loop vs. hash join). As we will see later, this is fundamental for query optimization.
(2) **Cardinalities**: Intermediate cardinalities are an important input signal for the overall cost of a plan as they denote the number of tuples an operator needs to process [21]. Thus, many LCMs leverage intermediate cardinalities as input features, which are either annotated by the databases' cardinality estimator or obtained through an additional learned estimator from related work [14, 18, 42]. While some LCMs also ignore cardinalities as input for cost prediction, we show in our study that they, in fact, improve the usefulness of cost estimates from LCMs for various query optimization tasks.
(3) **Data Distribution**: Another helpful factor in estimating cost is understanding the data distribution in the base tables, especially if no cardinalities are used. For instance, the fact of how many distinct values exist in a column might influence the efficiency of physical operators (e.g., hash join). As such, some LCMs use data distribution represented as database statistics and histograms or sample bitmaps (which we explain later) from the base tables as inputs. However, as we will show in our study, their effect on query optimization tasks remains unclear.
(4) **Cost Estimates**: Finally, some of the most recent LCMs even leverage the cost estimates provided by a classical cost estimator as an input feature, which serves as a strong input signal. This idea renders these LCMs to *hybrid* as they combine a traditional cost model with a learned approach. The study shows that this provides significant benefits.

Query Representation. Many LCMs use model architectures use a graph-based representation to encode query plans as input to the models[3]. These approaches thus explicitly leverage information about the order (parent-child relationships) of operators in plans. However, other LCMs [2, 18] represent a query plan (or the SQL string) as a flat vector of fixed size without modeling the operator dependencies, which we refer to as *flat* representation in this paper. While intuitively, capturing the structure and not using a flat representation should be beneficial for LCMs, the results of using graph structure in this study are not that clear.

[3]The graph-based representation of the queries refers to the fact whether a model leverages the query graph structure and not to the model learning architecture itself.

Model	Input Features: SQL String	Physical Plan	Cardinalities	DB Cost Estimates	DB Statistics	Sample Bitmaps	Query Representation	Database-Dependency	Model Architecture
Flat Vector [10]		✓	✓				Flat	DB-agnostic	Regression Tree
MSCN [18]	✓					✓	Flat	DB-specific	Deep Sets
End-To-End [31]		✓			✓	✓	Graph	DB-specific	Tree Structured NN
QPP-Net [27]		✓	✓	✓		✓	Graph	DB-specific	Neural Units
QueryFormer [44]		✓			✓	✓	Graph	DB-specific	Transformer
Zero-Shot [13]		✓	✓		✓		Graph	DB-agnostic	Graph Neural Networks
DACE [23]		✓	✓	✓			Graph	DB-agnostic	Transformer

Table 1. Selection and main dimensions of LCMs for our study

Database Dependency. Furthermore, an important aspect is whether LCMs can generalize to unseen databases (i.e., a new set of tables) or not. *Database-agnostic* LCMs were designed [13, 23] to enable cost predictions for unseen databases that were not part of the training data. This approach has the advantage of directly providing results without requiring database-specific training data. In contrast, *database-specific* [27, 31, 44] models cannot generalize for unknown databases. For this study, an interesting question is if one of these classes is better suited to support query optimization tasks as database-specific can better adapt to one single database while database-agnostic models can generalize better.

Model Architecture. Finally, the presented LCMs differ largely in their learning approach. Various learning architectures were proposed, including decision trees, tree-structured neural networks, neural units, graph neural networks, and transformer architectures. While different architectures show different results on the cost estimation tasks, it is still open to see which architecture provides the best results for query optimization.

3 EVALUATION METHODOLOGY

In this section, we discuss the selection of LCMs for this study and then explain our evaluation strategy. Afterwards, we discuss which downstream tasks we include in our study and why we select those tasks. Finally, we explain the experimental setup.

3.1 Selection of LCMs

Existing LCMs differ in various dimensions, which might affect query optimization. For this paper, we thus select a representative, broad set of recent state-of-the-art LCMs covering various approaches as shown in Table 1. Moreover, we focus on models where artifacts were available to make the results of this study reproducible. Below, we discuss the model selection briefly.

(1) `Flat Vector` [10]: This is one of the earliest approaches published more than 15 years ago and serves as a simple baseline for more recent and complex approaches. This paper aims to represent the physical query plan as a fixed-size vector with one entry per operator type (e.g., hash join, nested loop join, sort-based aggregate, hash aggregate). Each entry then contains the sum of the intermediate cardinalities per operator type. To predict the runtime of a plan, a state-of-the-art regression model LightGBM [16] is trained, which uses such a flat vector as input.

(2) `MSCN` [18]: As a second model, we choose `MSCN` as one of the first models of the more recent generation. While initially developed for cardinality prediction, the model has also been used for cost estimation [13, 31, 40]. We include this model as the only one that uses the SQL-String as input instead of the physical query plan. To represent the SQL query, `MSCN` uses one-hot encoded flat feature vectors to describe tables, join conditions, and predicates used in a query. Moreover, to learn from the data distribution, `MSCN` uses *sample bitmaps* as model input, which

indicates for the filter conditions of a given query which rows of the base tables qualify for selection.

(3) `End-To-End` [31]: As the third approach, we select `End-To-End`, as this was the first proposed LCM which explicitly models the plan structure. As a model architecture, a tree-structured neural network is used for which it combines different *Multi-Layer Perceptron* (MLP) for encoding input features and aggregating them over the query graph.

(4) `QPP-Net` [27]: This approach is also aware of the plan structure but uses a more modular approach of so-called *neural units*, which are MLP, trained per operator type (e.g., one for hash joins, one for nested loop joins). Each neural unit considers a set of operator-related features and predicts a per-operator runtime and hidden states, passed along the query graph as input signal for the following MLP. In addition, it learns from estimated cardinalities and estimated operator costs.

(5) `QueryFormer` [44]: Different from `QPP-Net` and `End-To-End`, which are based on simple MLP, this model employs a learning architecture using *transformers* to estimate query costs. Most importantly, it introduces a tree-based self-attention mechanism, which is designed to learn from long query plans with many operators. Furthermore, it also incorporates bitmap samples and a richer feature set compared to other LCMs, including histograms on base tables.

(6) `Zero-Shot` [13]: This is the first *database-agnostic* model proposed that can generalize across databases. To achieve this, it learns from so-called transferable features such as table size and does not encode database-specific features (e.g., attribute & table names) as the models did before. As a model architecture, variations of *graph neural networks* are used.

(7) `DACE` [23]: Finally, `DACE` represents one of the most recent models, which combines transformers with a database-agnostic approach. As it is based on transformers, it uses self-attention [35] and *tree-structured attention* mechanisms. In contrast to other models, it uses a reduced feature set, mainly learning from the operator tree and the cost estimates provided by a traditional cost model (i.e., PostgreSQL cost).

3.2 Tasks of Query Optimization

In this study, we evaluate the selected LCMs against *three* query optimization tasks that each rely on precise cost estimates but stress different abilities of cost models for plan selection.

Join Ordering. The join order describes the sequence in which base tables are joined in a query plan. It is decisive for the query runtime as joins are the most expensive operations in a query plan, and a sub-optimal join order can significantly increase the runtime as the intermediate cardinalities explode. To solve this task during query optimization, LCMs are combined with plan enumeration techniques such as dynamic programming to select the plan with the estimated lowest cost for execution.

Access Path Selection. In addition to the join order, the correct choice of access path (index vs. table scan) is another decisive factor for the runtime of the final query plan [29]. Using indexes like B-trees for accessing data in queries can massively accelerate access when selected correctly by a cost model. However, the cost of index-based access vs. scan-based access depends on several factors, such as selectivity or data distribution. LCMs must fundamentally understand these to provide reliable cost estimates for access paths.

Physical Operator Selection. Another critical task for query optimization is to select a physical implementation of query operators. For instance, most database systems support various join implementations like hash join, sort-merge join, or nested-loop join. Again, the optimal selection of physical operators depends on several factors, such as intermediate sizes or the sortedness of data, which are not all explicitly included in many LCMs.

3.3 Experimental Setup

In the following, we explain the experimental setup of this paper that is common for all these tasks.
Training and Evaluation Data. To train and test the selected LCMs, we leverage the benchmark proposed by [13] that contains 20 real-world databases and a workload generator that provides SPJA-queries supported by all presented LCMs. The generator reflects the state-of-the-art workload generation used by recent LCMs to create training queries. We generate and execute 10,000 queries per database to learn from a broad range of query patterns and set the timeout to 30s, which is sufficient to show the effects of selecting good plans. As we will demonstrate, this runtime can show already significant trade-offs for query optimization. Moreover, this is a realistic scenario for cloud providers [33, 34]. We use the same dataset and workloads for all cost models. For the execution of queries, we use PostgreSQL v10.23 on bare-metal instances of type `c8220` on CloudLab as an academic resource [9] to generate both our training and evaluation data. Each query is executed three times, and the runtime is averaged for a stable ground truth. For some experiments in our study, we enforce the plan selection with `pg_hint_plan`. We make all models, evaluation results and training data publicly available[2].
Model Training. For the model training, we need to differentiate between database-specific and database-agnostic LCMs:

Database-specific models are trained for a specific database (i.e., set of tables). For these models, we ran 10,000 queries on a single database and divided them into a training, evaluation, and test set (80%, 10%, 10%). To ensure that database-specific models are sufficiently trained, we provided all of them with 10.000 training queries and validated that adding more training data does not improve the quality of their predictions.

Database-agnostic models are trained across various databases. Thus, they typically require more training data but then generalize out-of-the-box to unseen datasets. We trained all database-agnostic LCMs on 19 training databases, each with 5,000 queries, and evaluated on an unseen target database, according to the strategies proposed in [13, 23]. Training with more training data also did not yield significant improvements in terms of accuracy. Finally, each LCM is trained three times with different seeds on the weights initialization and the train/test-split. We average the model predictions across the seeds for all evaluation results reported.

Traditional Baselines. As a traditional model, we use PostgreSQL's cost model in this study[4]. Here, we included two different versions (10.23 and 16.4) to allow a broader comparison and also analyze how traditional cost models evolved over time In PostgreSQL, the cost of an operator is determined by a weighted sum of the number of disk pages accessed and the amount of data processed in memory. Note that the cost estimates from PostgreSQL do not represent actual execution times. Still, they are designed to represent the execution cost and can thus be used for all query optimization tasks. As such, to make the predictions comparable with the LCMs, we scale the logical costs to the actual runtime with a linear regression model and refer to this approach as `Scaled PG10` and `Scaled PG16`, which is similarly used in other papers [13, 23, 40, 44].

LCM Implementations. For all LCMs, we relied on published source code, which we refer to in our repository[2]. However, we needed to re-implement some details, such as their ability to work with the same training datasets. For instance, `QueryFormer` was hard-coded to work with IMDB only, as it assumed a fixed set of tables and filters. We addressed this by first standardizing the various inputs of LCMs (cf. Table 1), like physical plans, database statistics and sample bitmaps. In addition, we maintain feature statistics for each dataset that help to normalize the inputs and to create one-hot encodings (e.g., to encode table or column names) that are required for some LCMs.

[4]While other even more sophisticated traditional models in commercial DBMS such as Microsoft SQL Server exist, we already see as a result of this study that LCMs cannot (yet) improve over PostgreSQL

We further unified the LCM training and evaluation pipelines to collect consistent metrics for all models. However, it is important that all our changes did not change the internal behavior of these models.

3.4 The Need for New Metrics

Most works focus on the prediction accuracy over an unseen test dataset by typically reporting the median Q-Error to evaluate how well LCMs predict the execution costs. Often, the median as well as percentiles are reported. It is defined as follows:

Definition 1. *Q-Error (Q_{50})*: The Q-Error is defined as the relative, maximal ratio of an observed label y and its prediction $\hat{y}$ with $Q = \max(\hat{y}/y, y/\hat{y})$ where $Q = 1$ indicates a perfect prediction.

However, we argue that this strategy is *not sufficient* to evaluate the applicability of LCMs in query optimization due to two reasons:

(1) **Focus on Single Plan Candidates.** The traditional strategy typically only evaluates *one plan per query* in a workload. However, the typical task in query optimization is to select one plan out of multiple candidates. As such, for a study that aims to understand the quality of plan choices, an evaluation methodology needs to enumerate multiple plans for the same query and report metrics that show us how well LCMs can pick the best plan (see next).

(2) **Focus on Accuracy as only Metric.** Traditional strategies focus mainly on the overall prediction accuracy of runtimes. However, the ability of the model to *select the right plan* and to *rank the plan candidates* is much more critical. Thus, we introduce novel metrics for the corresponding tasks to evaluate the ranking and selection properties of LCMs in the later sections.

4 TASK 1: JOIN ORDERING

Optimizing join order is a crucial task in query optimization as it improves query performance, especially in complex queries involving joins over multiple tables. In the following, we analyze *how reliably current LCMs reason about join orders* by first describing the detailed experimental setting and additional metrics before presenting the results of several experiments.

4.1 Evaluation Setup

Experimental Setting. Unlike previous evaluation strategies that evaluate the prediction accuracy on a single plan candidate, we compare how well LCMs can be used to pick a join order. For this, we exhaustively generate *all* possible join permutations for a given workload and let all LCMs predict their execution costs (as a traditional query optimizer would do). We decided to use exhaustive enumeration to separate concerns and avoid a bad plan being picked based on the enumeration strategy (e.g., by only enumerating left-deep plans). That way, we analyze how well LCMs provide costs (i.e., runtime) that enable an optimizer to pick good join orders, not how well the enumeration strategy works. As a workload in this study, we use the JOB-light benchmark [18, 21], which operates on the IMDB dataset consisting of 70 SPJA-Queries, as it is specifically designed to evaluate the task of join ordering. In contrast to other datasets like TPC-H, this dataset contains diverse correlations and non-uniform data distributions, increasing the difficulty of the task.

Experimental Metrics. As explained before, we define new metrics to evaluate LCMs for the join ordering task as follows:

(1) To evaluate how LCMs affect the outcome of query optimization in terms of runtime, we introduce the *selected runtime*:

Definition 2. *Selected Runtime (r)*: Given a set of plan candidates, the selected runtime, r, is defined as the actual runtime y of the plan that the LCM would choose, i.e., where the prediction $\hat{y}$ is minimal. Formally, it is: $r = y_{\arg\min_i \hat{y}}$

(2) To report how well LCMs are able to select the optimal out of multiple plans, we introduce *surpassed plans*.
Definition 3. *Surpassed Plans (s_p)*: In relation to a selected plan with actual runtime y, surpassed plans refer to the proportion of query plans that actually have a longer actual runtime. $s_p = 100\%$ therefore means that the optimal plan was selected, while $s_p = 0\%$ stands for the worst choice. Formally, it is defined as: $s_p = 100\% * (1/n) * \sum_{i=1}^{n} \text{bool}(y_i > r)$ with total n plans with $s_p \in [0\%, 100\%]$

(3) Moreover, it's crucial for LCMs to order query plans by their costs, as the optimizer needs to select from different candidates. Thus, we report the correlation of actual and predicted runtimes of different query plans with the *rank correlation* that asses the ranking ability of the LCM towards the query plans [30].
Definition 4. *Rank Correlation (ρ)*: For our study, we use Spearman's Correlation which is given by: $\rho = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2-1)}$, with $\rho \in [0\%, 100\%]$, where d_i is the difference between the ranks of corresponding runtimes, and n is the number of plans.

(4) Finally, to evaluate the likelihood of LCMs to pick a non-optimal plan, we report under- and overestimation. For instance, if the span of LCM under- and overestimation is high, it is more likely to select a plan that is, in fact, sub-optimal.
Definition 5. *Maximal Relative Under- /Overestimation(m_u, m_o)*: These metrics indicate by which factor a query plan candidate is over- or underestimated in the worst case. For a set of predictions $\hat{y}$ and labels y, they are defined as $m_u = min_i(y_i/\hat{y}_i)$ and $m_o = max_i(\hat{y}_i/y_i)$.

Metric Discussion: As we show for all experiments, the focus on accuracy alone is not sufficient to evaluate cost models for query optimization, and therefore, we propose various other metrics. However, it is important to note that each metric has limitations in what it can show, and only the combination of multiple metrics can help to understand the overall behavior. For example, for join ordering, correlation metrics are good in helping to understand how well cost models rank plans. Still, they fail to analyze the severity of the effect if plans are not ranked well, which can lead to sub-optimal plan selections. For this reason, its is crucial to look at the percentage of surpassed plans and the total selected runtime.

4.2 Example Query & Metrics

We first report the results of an example query (Query 33 from JOB-Light) in Figure 3 (at the top), as its results are representative of the full study that we conduct in Section 4.3 and to illustrate our metrics introduced above. As this query has four tables, there are 120 possible join permutations. We discuss the results of each subplot (Ⓐ - Ⓖ) in the following:

Ⓐ Model Predictions. We sort the plans with the different join permutations by their actual runtime, as presented by the black curve. Intuitively, the leftmost plan is optimal, with the shortest runtime of 2.20s, while the worst plan requires 11.17s. Moreover, we show the predictions of all LCMs in the same plot to get a qualitative impression of their predictions. On the first view, the predictions of LCMs often show a behavior where predicted costs do not grow regularly with the actual costs. In general, this is undesirable behavior, as it reinforces the choice of unfavorable plans since undesirable, false minima can occur in the cost predictions. In contrast, both `Scaled PG10` (gray curve) and `Scaled PG16` (gold curve) show that the predicted cost grows with the actual cost.

Ⓑ Median Q-Error. To assess the overall prediction accuracy, we report the Q-Error (Q_{50}) of the predictions over all join permutations. Interestingly, for this particular query, `Scaled PG16`

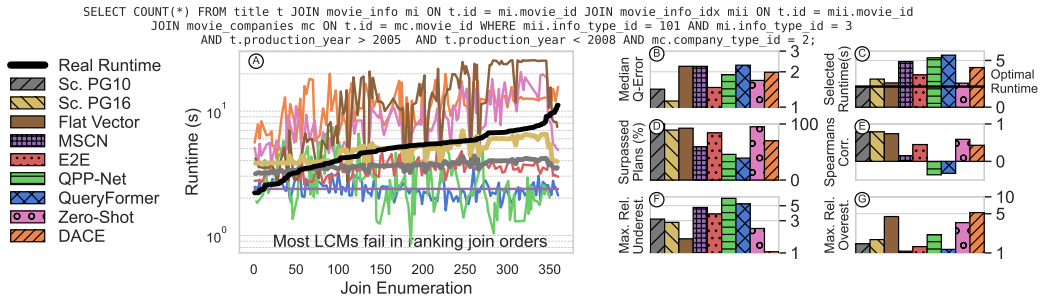

Fig. 3. Example Query Nr. 33 from JOB-Light for join ordering. We report model predictions (Ⓐ), overall accuracy (Ⓑ), outcome on query optimization (Ⓒ), model optimality (Ⓓ), model ranking (Ⓔ) as well as under- and overestimation (Ⓔ & Ⓕ). In this query, `Scaled PG10` picks the optimal plan and outperforms all LCMs for most of the metrics.

shows the best results with Q_{50}=1.23, followed by `Scaled PG10` with Q_{50}=1.42. Moreover, another interesting observation is when looking at `MSCN`. While the Q-error is not that bad (with $Q_{50} = 2.23$), the predictions are not helpful for an optimizer at all, as they predict the very same runtime value for all plans. Due to its SQL-based featurization, `MSCN` cannot reason about the join order of the query plan or the query operators and thus, it cannot be used for join ordering or physical plan selection. Still, we included `MSCN`, which is often used as a naive baseline for cost predictions [13, 24, 44].

Ⓒ **Selected Runtime.** To evaluate how much LCMs affect query optimization, we report the selected runtime (r), as this reflects the query runtime of the chosen plan when relying on the cost estimates of a given LCM. Here, the best cost model is `Scaled PG10`, where a query runtime of r=2.20s is achieved. In contrast, the worst case is when using `QueryFormer`; an optimizer would lead to a plan with a much longer runtime of r=5.56s, which is more than twice as optimal. All other LCMs end up between `Scaled PG10` and `QueryFormer`; i.e., no LCM provides a better plan choice than `Scaled PG10`. Interestingly, `Scaled PG16` selects a slightly worse plan than `Scaled PG10`, which is, however, still near-optimal.

Ⓓ **Surpassed Plans.** Next, we evaluate the surpassed plans (s_p). This shows the fraction of plans outperformed by the selected plan from a given LCM and thus shows the relative rank of the plan a model picks; i.e., if a model picks a plan on rank 5 of 10 plans, it surpasses $5/10 = 50\%$ of the plans. Consistent with the previous results, `Scaled Postgres` beats the other models as it selects the optimal plan in this example (s_p=100%). In contrast, `QueryFormer` (worst LCM) only surpasses s_p=39% of the plan candidates.

Ⓔ **Rank Correlation.** As LCMs crucially need to determine the *rank* of query plans by their costs, it is important to analyze the rank faithfulness of a cost model[5]. We report the ranking correlation (ρ) between the actual and predicted runtimes to evaluate how well the models perform in the ranking task. In the given example query, both `Scaled PG10` and `Scaled PG16` have the best correlation ($\rho = 0.79/\rho = 0.72$) indicating a successful ranking. In contrast, all LCM competitors are worse, down to $\rho = -0.23$ for `QueryFormer`. Critically, the negative value means that as the actual runtime increases, the predicted runtime tends to decrease, indicating a failure of the ranking task. This shows that cost predictions of traditional models provide a better ranking of query plans than LCMs for this query.

Ⓕ & Ⓖ **Under- and Overestimation.** We now report underestimation and overestimation of LCMs. Overall, underestimation and overestimation indicate the probability of a query optimizer

[5]This observation led to *ranking-based cost models* as an interesting alternative [4, 6, 47].

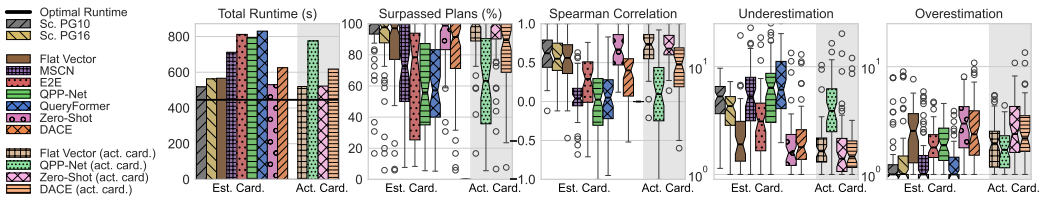

Fig. 4. Full Join Ordering Results on JOB-Light benchmark. Traditional models often still outperform LCMs in terms of join ordering across all metrics. Using actual cardinalities helps to improve the results of LCMs significantly.

not picking the optimal plan. For example, when significantly underestimating the runtime, a plan might be chosen that is, in fact, a long-running plan. When looking at Ⓕ & Ⓖ, we make two interesting observations: (1) Under- & overestimation is generally larger with LCMs than for `Scaled PG10` and `Scaled PG16`. (2) Generally, PostgreSQL tends towards systematic underestimation of execution costs, which similarly has been shown by [21]. A typical reason for this is the assumption of filter attribute independence, leading to underestimates. However, LCMs are prone towards both under- and overestimation. Overall, this wider range of over- and underestimation of LCMs, in fact, increases the likelihood of picking a non-optimal plan.

4.3 Full Results on Join Order

To report more representative results, we aggregate the previously discussed metrics over the full JOB-Light benchmark. The results can be seen in Figure 4. Most interestingly, when looking at the total runtime of selected plans, we see that `Scaled PG10` still provides the shortest total selected runtime of $r = 518s$, followed by `Zero-Shot` with $r = 530s$. The optimal runtime for the workload is at r=446s, which assumes a perfect plan selection for every query. The worst model again is `QueryFormer`, where the selected runtime for all queries is $r = 830s$. When looking at surpassed plans, `Scaled PG10` and `Scaled PG16` also outperform their LCM competitors. Regarding the ranking of plans, as shown by ranking correlation ρ, `Zero-Shot` as LCM actually returns the best results, outperforming traditional approaches. This is surprising, as `Zero-Shot` still does not lead to plans that outperform them in total runtime. However, when looking at under- and overestimation, we can see that in line with the anecdotal result, `Zero-Shot` tends to both under- and overestimate at the same time. This leads to worse plan selections. Interestingly, the best three performing LCMs are all *database-agnostic* and `Flat Vector` shows a good (often third place) performance, although it uses a simplistic model structure.

4.4 Impact of Improved Cardinalities

The optimal order of joins is impacted significantly by intermediate cardinalities, which in turn influence query costs. So far, many of the proposed LCMs use *estimated cardinalities* as input derived from PostgreSQL to provide cost estimates for a plan. However, the PostgreSQL cardinality estimator frequently produces inaccuracies, sometimes deviating by several orders of magnitude. It was shown that better cardinality estimates substantially improve the cost estimation results [21]. This raises the question of their impact on cost estimates of LCMs to make better optimizer decisions. To investigate this, we train and evaluate all LCMs, that utilize cardinalities as input features (i.e., `Flat Vector`, `QPP-Net`, `Zero-Shot`, `DACE`, cf. Table 1), with actual, observed cardinalities rather than estimated ones. This aims to isolate the effects of cost estimation from that of cardinality estimation. We repeat the same evaluation from the previous experiment from Section 4.3 with

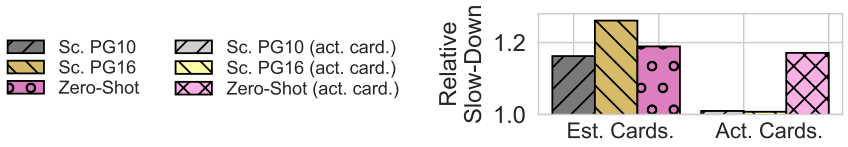

Fig. 5. Relative runtime slow-down vs. the optimal execution. PostgreSQL models are closer to the optimum than LCMs, especially when using perfect cardinalities

actual cardinalities instead and report the results in Figure 4, where the new model variants are represented by the lighter-colored bars. It can be seen that, indeed, perfect cardinalities have a positive impact on the overall results. For instance, the total runtime of `QPP-Net` decreases from r=765s to r=644s. Still, the best LCM variant, which is `Flat Vector` here, is outperformed by `Scaled PG10`, which still relies on estimated cardinalities. However, as a promising result, we would like to highlight ranking correlation and the underestimation, where adding perfect cardinalities significantly improves all LCM variants.

4.5 Accurate Cardinalities for PostgreSQL

Finally, we analyze how optimal the selected plans of LCMs are in relation to the total runtime when providing accurate cardinality estimates for classical cost models. We compare `Scaled PG10`, `Scaled PG16` and the best performing LCM, from the previous experiment (i.e., `Zero-Shot`) and report the relative slow-down of runtime that they achieve compared to the optimal workload runtime for the JOB-Light benchmark. The results are shown in Figure 5 where we see that overall, the execution is 18.9% slower than the optimum when using `Zero-Shot`. In contrast, `Scaled PG10` achieves a slow-down of 16.23% and `Scaled PG16` 24.9%. Importantly, when providing actual cardinalities to the PostgreSQL models, their performance improves drastically towards near-optimal plan selections (1.0% and 0.8% slow-down). In contrast, `Zero-Shot` only gets slightly better, indicating that it cannot yet make full use of information provided by cardinality estimates; i.e. `Zero-Shot` still over- and underestimates costs for individual plans, leading to non-optimal plan selections.

4.6 Summary & Takeaways

The analysis of join ordering indicates that the traditional model, `Scaled Postgres`, continues to outperform LCMs in selecting plans with low runtimes. In fact, only in terms of the ability of *ranking* of plans, the traditional models are outperformed by `Zero-Shot`, which demonstrates that LCMs have the ability to improve query optimization. However, traditional models are not without flaws, as they also still fail to identify faster plans in some cases. For this task, the three best performing LCMs are all DB-agnostic, and the simple model `Flat Vector` performed relatively well. Interestingly, LCMs tend to be more precise in predicting costs for query plans close to the optimal plan, while they have more significant prediction errors (over-and under-estimates) for less optimal plans with higher actual runtimes. One reason is that training data typically contains more optimal than sub-optimal plans, which comes from the biased training strategy that we discuss in Section 7. As such, in the future, we suggest that LCMs should focus on mitigating over-and under-estimates for such plans by systematically including signals from these plans in the training of LCMs. Another direction is to extend LCMs to predict not only runtime but also their confidence.

5 TASK 2: ACCESS PATH SELECTION

As a second task in our study, we look at access path selection and analyze how much existing LCMs improve over classical models on this task. An access path determines *how* a database engine retrieves the requested data, with standard methods being sequential scans or index accesses. A key issue is that using indexes is not always faster; for small tables or queries that return a large portion of the table, the overhead of accessing the index can outweigh the benefits, making a sequential scan more efficient. This makes the selection of access paths a crucial task in query optimization due to its direct impact on database performance [29]. Selecting an access path is typically done by estimating the cost and determining the path with the lowest cost. In the following, we study whether LCMs improve over classical models by a set of experiments. Like before, we first report anecdotal results from individual cases and then provide an analysis with a broader set of workloads.

5.1 Evaluation Setup

Experimental Setting. In our study, we look at the ability of LCMs to select between sequential scan and index scan using B+-trees that are both supported in PostgreSQL (internally denoted as `SeqScan`, and `IndexScan`/`IndexOnlyScan`). There are also other access options, such as bitmap index scans or hash scans, but these are out of the scope of our study. We will show that even the selection between these two access paths is hard to solve for all models. Note that the training data contains indexes on the primary keys (PKs) of *all* tables, which is a common setup in databases.

Experimental Metrics. For this task, in addition to the previous metrics, we introduce a new metric to describe how accurately a LCM selects the access path when one class in the decision-making is under- or over-represented (which is often the case for access path selection as we will see in our evaluation).

Definition 6. *Balanced Accuracy (B)*: Balanced accuracy B assesses the performance of classification models on imbalanced outcomes as it ensures equal performance consideration for both classes. It represents the arithmetic mean of true positive rate (TPR) and true negative rate (TNR) as follows: $B = \frac{1}{2}\,(\text{TPR} + \text{TNR})$ with $\text{TPR} = TP/(TP + FN)$ and $\text{TNR} = TN/(TN + FP)$ and $B \in [0, 1]$.

5.2 Example Query & Metrics

To get an intuition about the access path selection of LCMs, we first analyze how LCMs predict execution costs for either high or low selectivity. In the following, we will first present an intuitive example to demonstrate our approach and the new metrics before we later present a broader study across various columns. We select a query that filters on the column `production_year` of the table `title` of the IMDB dataset. This column contains the production year of different movies and ranges from 1880 to 2019. Note that the data distribution is skewed, as more movies were produced over a certain period of years. We perform two corresponding base table accesses on `production_year`, each with either `IndexScan` and `SeqScan`. To achieve a low selectivity, we use the attribute `>=1880` (which basically selects the whole table), and for high selectivity, we use `>=2011`, which contains just $\approx$ 20% of the entries. We report the results in Figure 6, where we mark correct (✓) and incorrect (X) access path selections based on the LCM predictions. Moreover, we present the actual runtimes and the model predictions for both scenarios. For the low selectivity query Ⓐ, we can see, based on the actual runtime (left-most two bars), that the sequential scan is indeed faster (0.71s) than the index scan (1.78s). In contrast, the high selectivity query Ⓑ is the reverse. As we can also see, the cost predictions of the LCMs is often the opposite and lead to wrong access path selection. In fact, five of nine models (`Flat Vector`, `MSCN`, `Zero-Shot`, `QueryFormer`, `QPP-Net`) select the 2.5× slower index scan for the high selectivity (shown in Ⓐ). In contrast, for the low selectivity query (shown in Ⓑ), the sequential scan is actually 5× slower (0.35s) than the

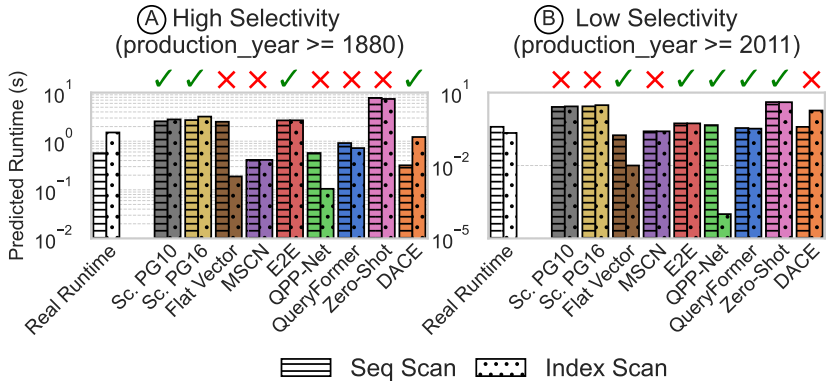

Fig. 6. Predictions for table scans on `movie.production_year`. We show the real runtimes (white bars) against LCM predictions and indicate correct (✓) and incorrect (X) selections for a high selectivity Ⓐ and low selectivity Ⓑ scenario.

index scan (0.07s). Only two LCMs (`MSCN`, `DACE`) select the wrong access pass. However, if we look at the cost estimates, we can see that estimates are for many LCMs not very indicative since the estimated runtime for scan vs. index are very close, which means that access path selection is highly unstable for this query. Overall, this indicates that LCMs are struggling with the correct access path selection and their accuracy changes over the selectivity.

5.3 Access Path Selection over Selectivities

Next, we show the results on a broader set of selectivities over different predicates. To achieve this, we generate appropriate filter literals for the same query as in the previous section. More precisely, we scan all movies by `production_year` with the `>=` operator and vary the predicates to ensure equal steps in the domain of selectivities. We analyze the runtime for each query by selecting the access path according to the cost estimates provided by the LCM. The results are shown in Figure 7. The real runtimes of the access methods over the selectivity for the sequential and index scans are reported on the very left. In green, we show the runtime if a cost model selects the optimal path. As expected, for small selectivities, it is beneficial to select an index scan (dashed line). When the selectivity gets larger than 0.3, the sequential scan (solid line) is a better choice. Next, we show in Figure 7 which access paths PostgreSQL selects (second & third plot) and contrast this with the access path selection of LCMs (fourth to last plot). The selected access paths of the cost models are shown by a cross per query we executed. When looking at the model selections, we see that actually *no single model* always selects the correct access method. In fact, many models always select the same access method regardless of the selectivity; more details later. For instance, `Flat Vector`, `Zero-Shot`, `QPP-Net` and `QueryFormer` always select the index scan while other models select different access paths depending on the selectivity. However, other models like `MSCN` and `End-To-End` show a highly unstable behavior, especially for higher selectivities, where they randomly switch between access paths. While `Scaled PG10` and `Scaled PG16` show the desired behavior, they also mispredict access paths for some queries having lower selectivities. This behavior is followed by `DACE`, which learns from PostgreSQL costs.

To aggregate results over the access path choices, we report the balanced prediction accuracy B for each model in Figure 7 (right). As analyzed before, both PostgreSQL models and `DACE` achieve the best overall balanced accuracy that reports classification performance with $B = 0.62$ and outperform

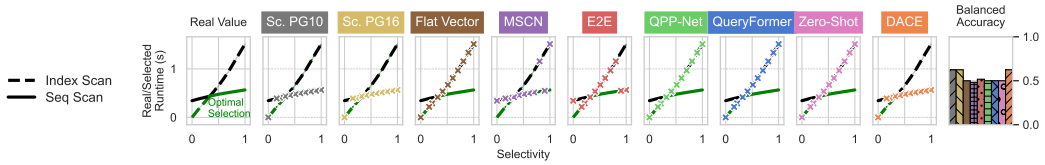

Fig. 7. Real scan costs and selected runtimes over different selectivities for scanning the column `title.production_year` of IMDB dataset with either sequential or index scan. Many LCMs would select the `IndexScan` regardless of the selectivity.

Dataset	Number of Tables	AVG NaN ratio (%)	Columns per Table (#)		Table Length (#)		Distinct Column Values (#)	
			min	max	min	max	min	max
Baseball	25	9.70	25	48	520	1.38×10^6	2	1.64×10^4
IMDB	15	20.91	4	49	4	1.48×10^7	2	3.62×10^7
TPC-H	8	0.00	2	12	5	1.50×10^6	2	1.50×10^6

Table 2. Statistics of the datasets Baseball, IMDB and TPC-H used to evaluate access path selection of LCMs

the remaining LCMs. Overall, these numbers are, however, still not satisfying, as they indicate that all cost models (even classical ones) still struggle to select the best access method.

5.4 Access Path Selection across Queries

Here, we show the results of a broader study, which confirms similar issues as observed in the previous experiment when selecting access paths for tables and columns with different sizes and data characteristics. In this evaluation, we use three different datasets (Baseball, IMDB and TPC-H) from [13], as they differ in their data characteristics by various dimensions, as shown in Table 2. For this experiment, we focus on columns that do not have an index according to the training data (i.e., PK columns) and ask LCMs about access path costs if an index were available. Moreover, we exclude columns that have more than 70% of missing values as this leads to large cardinality errors, which we want to isolate from this study. Finally, we focus on numeric data types to obtain reliable percentiles, as they allow range filter predicates to vary the selectivities precisely. The overall results are in Table 3, where we report average accuracy B over all columns (last column in gray). As seen previously, `Scaled PG10`, `Scaled PG16`, and `DACE` again perform best with $B \approx 0.64$ on average across all columns and tables in this broader study. Overall, the simple `Flat Vector` model performs second best. Although some LCMs (e.g., `QPP-Net`) learn specifically from statistics and histograms, they do not show better performance. It is, therefore, questionable whether LCMs can derive meaningful information from these artifacts. Another interesting observation is that results across columns vary for LCMs, ranging from low to high accuracy (marked in bold) across different columns. Moreover, some LCMs are not better than randomly guessing the access path, which would result in an accuracy of 0.5, e.g., `QueryFormer` and `QPP-Net`. Overall, this is far from satisfactory for robustly solving the task of access path selection.

5.5 Access Path Preferences

To analyze why many LCMs often choose the wrong access paths, we analyze their selection preferences across all queries. For this, we use all queries of the previous experiment and show in Figure 8Ⓐ the averaged ratio of selected table scans broken down by selectivities. The index scan ratio is reflected in this experiment as a $1 -$ [ratio of table scans]. In Figure 8Ⓐ, the black line

LCM \ Tab./Col.	Baseball							IMDB						TPC-H								Total AVG
	batting AB	batting G_batting	halloffame needed	managers L	managers W	mgrhalf L	mgrhalf W	aka_name person_id	cast_info movie_id	cast_info nr_order	person_inf person_id	title episode_nr	title prod_year	lineitem l_ext.price	lineitem l_partkey	part p_retailprice	part p_size	partsupp ps_availqty	partsupp ps_partkey	partsupp ps_suppkey	partsupp ps_supply-cost	
Scaled PG10	**0.67**	**0.67**	0.50	**0.6**	0.60	0.50	0.50	0.62	0.75	0.57	**0.6**	**0.58**	**0.62**	**0.67**	**0.75**	0.6	0.5	0.75	**1.0**	0.67	0.67	**0.64**
Scaled PG16	**0.67**	**0.67**	0.50	**0.6**	0.60	0.50	0.50	0.62	0.75	0.57	**0.6**	**0.58**	**0.62**	**0.67**	**0.75**	0.6	0.5	0.75	0.88	0.67	0.67	**0.64**
Flat Vector	0.58	0.46	0.50	0.50	0.40	**0.83**	**1.0**	0.38	0.50	0.50	0.40	0.33	0.5	0.23	0.36	0.4	0.38	**0.94**	0.71	**1.0**	**1.0**	0.57
MSCN	0.46	0.69	0.48	0.30	**0.73**	0.50	0.50	0.61	0.25	0.32	0.33	0.5	0.48	0.54	0.39	**0.72**	0.46	0.39	0.25	0.48	0.25	0.46
End-To-End	0.31	0.50	0.50	0.50	0.50	0.58	0.50	**0.75**	0.25	0.50	0.28	0.5	0.52	0.5	0.5	0.4	0.5	0.5	0.5	0.33	0.5	0.47
QPP-Net	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	**1.0**	**0.79**	0.50	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.54
QueryFormer	1.0	0.50	0.50	0.50	0.60	0.50	0.50	0.38	0.56	0.36	0.50	0.25	0.5	0.38	0.22	0.5	0.5	0.5	0.5	0.5	0.5	0.50
ZeroShot	0.42	0.60	0.50	0.50	0.50	0.50	0.50	0.38	0.56	0.36	0.50	0.5	0.5	0.5	0.25	0.63	0.36	0.25	0.5	0.33	0.46	0.46
DACE	**0.67**	**0.67**	**0.62**	**0.6**	0.60	0.50	0.50	0.62	0.75	0.50	**0.6**	**0.58**	**0.62**	**0.67**	**0.75**	**0.6**	**0.62**	0.75	0.86	0.67	0.67	**0.64**

Table 3. Balanced accuracy B of LCMs when selecting access paths for different workloads, tables, and columns.

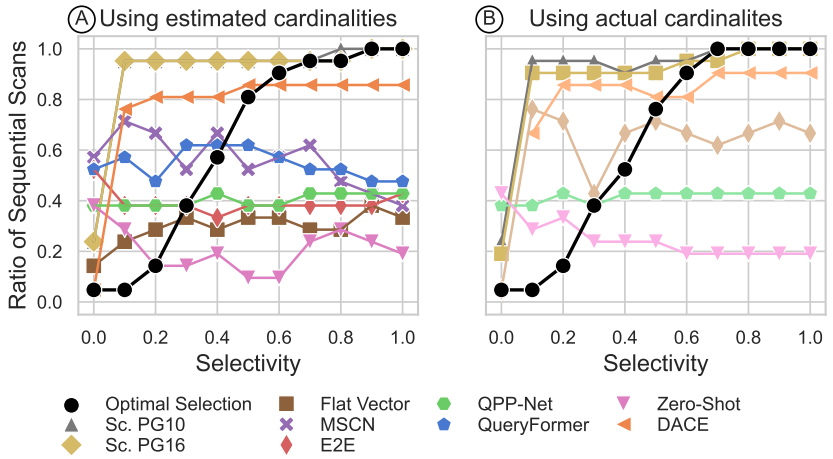

Fig. 8. Scan preference of LCMs over selectivity using either estimated Ⓐ or actual cardinalities Ⓑ

shows the optimal selection. For the LCMs (colored lines), the observations can be divided into two groups: (1) Overall, `Scaled PG10`, `Scaled PG16` and DACE are closest to the optimal selection and follow the trend across the selectivities, i.e., they select more index scans for small selectivities and more table scans for large selectivities. However, they select sequential scans far too early for too small selectivities. (2) The other LCMs, on the other hand, seem not to understand the effect of selectivity at all, as the ratio of table scans is almost constant across varying selectivities. Moreover, there seems to be some (static) preference towards index scans for many LCMs. To explain why many LCMs have this bias, we analyzed the training data. Surprisingly, it contained most often sequential scans ($\approx$ 90%) and a few index scans ($\approx$ 10%). Moreover, index scans were only used when they were really beneficial for queries, and no negative examples were included. Therefore, LCMs learn that index scans seem highly promising without understanding the downsides for high selectivities.

5.6 The Effect of Improved Cardinalities

Finally, as for the join order task, we want to see the effect of cardinality estimates, which are input to many cost models, and see if more accurate estimates lead to better decisions for cost models. For this purpose, we repeat the previous experiment with perfect cardinalities to estimate their effect on the access path selection. Interestingly, as shown in Figure 8Ⓑ, the access path selection

improves slightly for some LCM which now better follow the optimal choice, e.g., `Flat Vector`. However, other LCMs are still not able to catch the trend. We analyzed this behavior and found that the cardinality estimates for these simple scan queries are already highly accurate (with a Q_{50} <= 1.05). Thus, further improving cardinalities does not help this task very much.

5.7 Summary & Takeaways

Overall, no single LCM delivers convincing results in access path selection. Different from join ordering, the main reason is still that the costs of access paths are not really understood. However, incorporating database statistics and sample bitmaps into LCMs was not beneficial. Interestingly, LCMs are biased in their selection towards index scans across the selectivity range, which indicates that they learned that index scans are always beneficial due to bias in the training data. To mitigate this bias, LCMs need to learn from execution costs for both access paths across selectivity. We present strategies and initial results for this in Section 7.

6 TASK 3: PHYSICAL OPERATOR SELECTION

The task of physical operator selection refers to the process of choosing the most efficient algorithm to execute a given query operator. This selection is crucial, as it directly impacts the query performance and resource utilization. The complexity of this task arises from understanding the combined effects of algorithm complexity and various other factors, such as the data distribution, available indexes, hardware capabilities, and the specific characteristics of the workload. In this section, we report how well recent LCMs are able to select physical operators using physical operator selection for joins as an important example.

6.1 Evaluation Setup

Experimental Setting. For this experiment, we generate various queries that join two tables with join predicates on the foreign key relationships as these are the most common joins, like: `SELECT COUNT(*) FROM title,movie_info_idx WHERE title.id=movie_info_idx.movie_id AND title.production_year=2009`. We use `COUNT`-expressions so as not to deviate too much from the training data, which also makes typical use of them. We execute each query three times with three different join algorithms to get the true runtimes, i.e. `Hash Join` (HJ), `Sort Merge Join`, (SMJ) `Indexed Nested Loop Join` (INLJ)[6] which are algorithms that PostgreSQL supports. For all queries, we also obtain the predictions of LCMs.

Experimental Metrics. For this study, we introduce a new metric that determines how often the LCM picks the plan with the optimal physical operator algorithm.

Definition 7. *Pick Rate (p)*: The pick rate reports the percentage of p out of n query plans that gives how often a cost model would pick the optimal plan, e.g., where the plan with the minimal prediction has the lowest actual runtime.

6.2 Example Query & Metrics

To first get an impression of how LCMs select physical operators and how our novel metrics are applied, we illustrate in Figure 9 the runtimes for a representative example query of the IMDB dataset when using three different physical operators (left) and the model predictions (right). For the given query, SMJ is the most optimal selection according to the runtime of around 6.47s, while INLJ and HJ have longer runtimes of up to 13.19s (left-most bars). Interestingly, neither the PostgreSQL cost models nor a single LCM select the optimal physical operator. In fact, most models prefer the

[6]We only used INLJ joins, as primary keys always used an index in our setup.

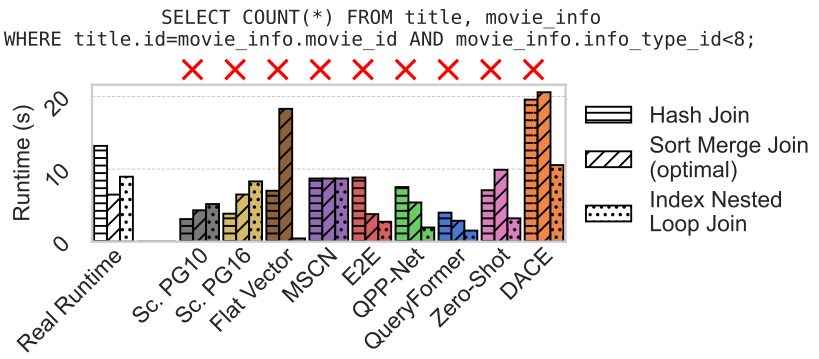

Fig. 9. Predictions for physical operators for a two-way join on the IMDB dataset. We show the real runtimes against LCM predictions and indicate correct (✓) and incorrect (X) selections. No single LCM would pick the fastest join (SMJ).

INLJ, although it has a longer runtime of 8.94*s*. Note that `Scaled PG10` and `Scaled PG16` also fail here, as they select the worst operator (HJ) with 13.19*s*.

6.3 Full Results on Physical Operator Selection

We now validate these initial findings for a broader set of queries and datasets. For this, we repeat the previous experiment with each 100 queries on the three different datasets IMDB, Baseball, and TPC-H (as described in Table 2).

All Predictions: We visualize the predictions vs. the actual runtimes in Figure 10. As each of the 300 queries has three different plan candidates with different joins, each subplot contains 900 predictions. Overall, for many LCMs, we observe huge differences between the operator type and the predicted costs. For example, `Zero-Shot` overall shows good correlations between the actual and predicted runtime for each operator type. However, while Hash Joins are precisely estimated, Index Nested Loop Joins are systematically underestimated, and Sort-Merge-Joins are overestimated roughly by a constant factor. Similarly, `Scaled PG10` and `Scaled PG16` show a linear trend between the predicted and actual runtime but overall tend to underestimate. In contrast, other LCMs like `End-To-End` or `DACE` show a noisy behavior that does not show any linear function of the operator type. However, both types of behavior lead to a sub-optimal operator selection. Interestingly, for some LCMs, we observe a consistent ranking within an operator type (e.g. INLJ for `Flat Vector`) but not across operator types, which makes it particularly hard to select the optimal operator.

Aggregated Results: To provide a more quantitative evaluation, we report aggregated results in Figure 11. We report the pick rate p (upper row) for each workload and the selected total runtime r vs. the runtime when optimal operators are chosen (lower row). Interestingly, when looking at the pick rate on IMDB, LCMs perform comparably well to `Scaled PG10` and `Scaled PG16`, and some even outperform them on IMDB and TPC-H. For instance, `DACE` as a database-agnostic model achieves a pick rate of $p = 82\%$ on IMDB, whereas `Scaled PG10` only achieves $p = 60\%$. A similar trend is observed for the TPC-H dataset. However, for baseball, `Scaled PG16` achieves the best pick rate with $r = 74\%$. A slightly different trend is observed when looking at the overall selected runtime. For IMDB and Baseball, `Scaled PG10` and `Scaled PG16` achieve the lowest runtime, closely followed by `DACE`. On TPC-H, `DACE` outperforms the other models slightly. These results show that LCMs already have the potential to be competitive with traditional models on this task while again not providing significant benefits.

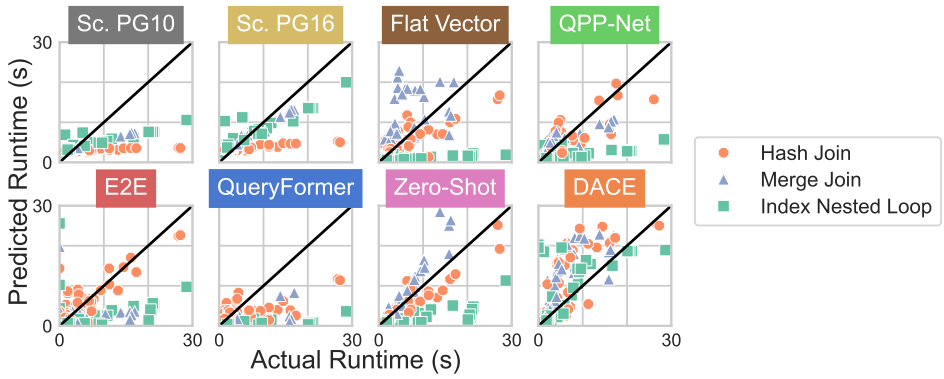

Fig. 10. Predicted vs. actual runtimes for different join types on 300 queries (on IMDB, TPC-H and baseball datasets)

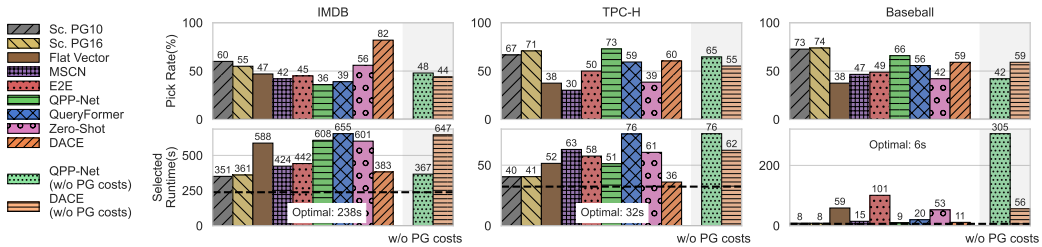

Fig. 11. Pick rate and selected runtime for physical operator selection over 100 queries of three datasets (IMDB, TPC-H, Baseball). LCMs are able to achieve a better performance than `Scaled Postgres`. However, the performance deteriorates when removing estimated PostgreSQL costs from the input features of `DACE` and `QPP-Net`.

6.4 Learning From PostgreSQL Costs

`DACE` and `QPP-Net` use estimated PostgreSQL costs as input features to predict the cost of queries as shown in Table 1. They, therefore, function as *hybrid* approaches, as they build on PostgreSQL estimates. That way, these models utilize the expert knowledge inherent in the PostgreSQL cost model. In this ablation study, we thus want to explore the contribution of PostgreSQL costs to the estimates. Thus, we train a variant of these two models `QPP-Net` and `DACE` without PostgreSQL costs as input, i.e., we removed them from the featurization. The results can be seen in Figure 11 (light bars at the right in each plot) As expected, the pick rate of `DACE` on the IMDB dataset decreases from $p = 82\%$ to $p = 44\%$, and the selected runtime increases from $r = 383$ to $r = 647$s. A similar observation also holds on the other datasets and also for `QPP-Net`– except for one case where surprisingly `QPP-Net` gets better on IMDB. Thus, it becomes clear that the PostgreSQL costs typically are an important signal and should be included in future LCMs, as we will discuss.

6.5 Operator Breakdown & Preferences

In this experiment, we want to understand where LCMs make mistakes when selecting physical operators. For this, we show the distributions of the selected operator types for the 100 queries on the IMDB dataset compared to the optimal distribution in Figure 12. As shown by the optimal distribution (left), in 43% of the queries, an INLJ is best, while in 41%, it is the HJ, and in 16%, the

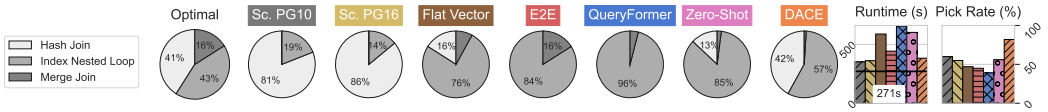

Fig. 12. Breakdown of selected operators for physical operator selection on the IMDB test queries.

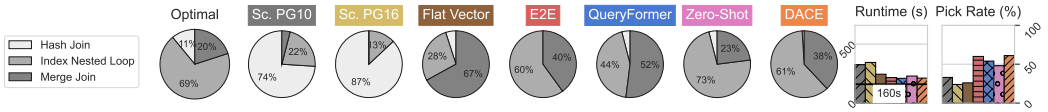

Fig. 13. Breakdown with additional indexes on the filter columns on the IMDB test queries.

SMJ. However, we see fundamentally different operator selections when looking at the selections when using the cost models. First, `Scaled PG10` and `Scaled PG16` both show a strong tendency to over-select HJs. Looking at LCMs, we see a very different picture: Most LCMs have an over-preference for INLJs. For example, `QueryFormer` chooses an INLJ even in 95%, and `Zero-Shot` in 85% of the queries. This is probably due to a similar effect for access path selection, as INLJs are represented in the training set as beneficial. We additionally report the pick rate and selected runtime in Figure 12, which shows similar observations.

6.6 Additional Indexes on Filter Columns

In the previous experiments, only indexes on the primary keys are used. In addition, we now add indexes on the filter columns, which open up additional ways of using INLJs because they also enable faster look-ups for non-primary key columns. This shifts the distribution of which physical operators are optimal, as we discuss next. For this, we repeat the previous experiment with additional indexes on the filter columns and show the breakdown in Figure 13[7]. As we can see, the use of INLJ when selecting operators optimally increases to 69%. Interestingly, the performance of `Scaled PG10` and `Scaled PG16` drops significantly as it still prefers HJ (74% and 87%), and it is now outperformed by most LCMs in terms of runtime and pick rate. Thus, it seems that the cost model of PostgreSQL is not well calibrated and still prefers hash operations. On the other hand, the benefit of LCMs might stem from their over-preference for INLJ, which, in this experiment, luckily, is often the better choice.

6.7 Summary & Takeaways

Overall, no cost model could achieve a near-optimal runtime when selecting physical query operators. Still, the classical models performed best and `DACE` as a database-agnostic model is close according to the selected runtime. However, as mentioned, `DACE` learns from PostgreSQL costs, which is highly beneficial for `DACE`, as shown in our ablation study. Furthermore, another observation is that most LCMs showed a strong over-preference for INLJ, which supports the findings of the access path selection study that bias in training data is a potential problem one needs to tackle for LCMs. However, this is also non-trivial as INLJs can cause high runtimes and, thus, often timeout during training data collection (as they need hours or even days). Thus, an interesting avenue of research is to include negative signals in training the models without the need actually to run these costly negative examples.

[7] As `QPP-Net` cannot predict the cost for additional indexes due to its fixed featurization using one-hot encoding, we excluded it from this experiment.

7 RECOMMENDATIONS FOR COST MODELS

In this paper, we comprehensively evaluated LCMs against traditional cost models for query optimization. In the following, we summarize our results and recommendations. As a main outcome, in *none* of the tasks we analyzed, LCMs can significantly beat traditional approaches for cost modeling – even though LCMs provide higher accuracy over query workloads. The total execution time of plans selected by the LCMs in query optimization was, in fact, often even higher than with traditional models. However, we still believe that LCMs have a high potential, but the focus *only* on their accuracy has resulted in the position we are in today. In the following, we distill recommendations based on our main findings to provide future directions to unlock the full potential of LCMs.

R1: Consider Model Architectures and Features. As shown in the classification of models in Section 2.3, LCMs largely vary in input features, query representation and model architecture. We summarize the most critical learnings: (1) Learning from the query plan is absolutely necessary, but using the SQL string as input alone is unsuitable (2) Simple model architectures like `Flat Vector` often perform relatively well, making it questionable whether very complex architectures are necessary. (3) DB-agnostic LCMs often outperformed DB-specific models, as they were trained on a larger variety of query workloads and data distributions. (4) While precise cardinality estimates help solve the downstream tasks, it remains unclear to what extent statistics and sample bitmaps help.

R2: Use Appropriate Metrics. Another key finding of this paper is that traditional evaluation strategies are insufficient to assess how good LCMs are for query optimization. Therefore, we recommend using the metrics presented in this paper that show how well an LCM selects from multiple plan candidates, how well it can rank these plans, and what speed-up it provides for a given query optimization task. Ideally, these metrics influence the design and learning approach of future LCMs. For example, one direction could be to apply ranking-based approaches, which have been used recently for end-to-end learned optimizers [4, 6, 47] to cost models.

R3: Diversify Your Training Data. The third key finding of this study is that traditional training strategies for LCMs induce fundamental biases stemming from how training data is collected. Specifically, LCMs are typically trained on *pre-optimized queries*, as these have already been executed by a database system (cf. Figure 2Ⓑ) to generate the corresponding labels. Another reason for biases in training data is timeouts, which are necessary to execute training queries within a limited, reasonable time. Query plans with expensive operators like nested-loop joins are thus likely to timeout before completion. Consequently, only the cases where these operators are beneficial are included in the training data, leading to a bias in the models, as for access path selection (Section 5). A similar bias was observed for learned cardinality estimation in [28]. Overall, the traditional training strategies are thus particularly ill-suited for query optimization, as they lead to a substantial divergence between the training data distribution and non-optimized queries that occur during query optimization. To resolve this bias, training data should be diversified so that both good and bad query plans can be learned from.

We demonstrate the effect of how diversification can help with the task of access path selection in a small experiment. In particular, we fine-tune LCMs with additional training data consisting of 500 randomized scan queries with different selectivities. For each query, we enforce two executions: One with an `IndexScan` and one with a `SeqScan`, and we do not enforce timeouts. We report the balanced accuracy B over the columns from IMDB of Table 3 *before* and *after* fine-tuning in Figure 14 Ⓐ. As we can see, using diversified training data is highly beneficial in most cases as accuracy improves. Even more importantly (see Figure 14 Ⓑ), this also leads to improvements of up to 45% in the total runtime across all LCMs. Interestingly, `Zero-Shot` is the only cost model that

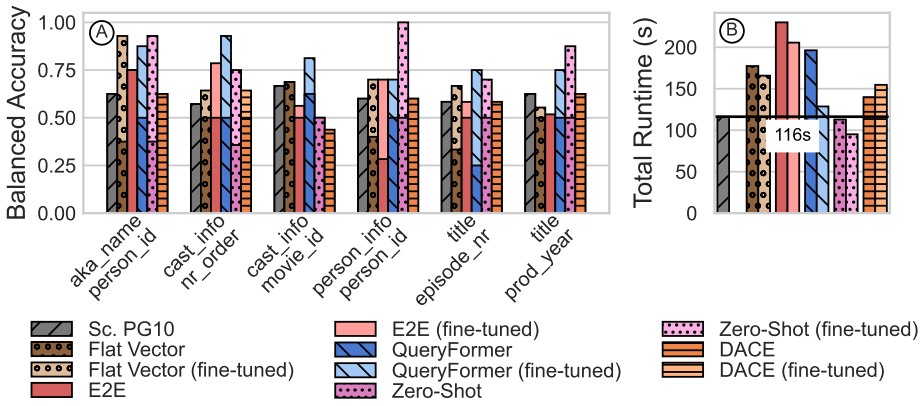

Fig. 14. Fine-tuning for access path selection on IMDB columns. Ⓐ For most columns, balanced accuracy increases after fine-tuning. Ⓑ Also the total runtime improved, so that `Zero-Shot` outperforms `Scaled Postgres`.

outperforms the runtime of `Scaled PG10` (95s vs. 116s), which shows that LCMs can outperform traditional approaches for downstream tasks and training data plays a crucial role. The results for `DACE` typically do not improve because it seems to learn mainly from PostgreSQL costs as a signal, as discussed in Section 5.3.

However, while this diversification of training data has already been leveraged in the domain of learned query optimization (e.g., by cardinality injection [7, 47] or exploration [41]), its application for LCMs is not trivial. One challenge is that each query plan would result in many possible candidates (e.g., join order permutations). Sophisticated strategies are required to select meaningful training queries to maximize information gain and still keep the training data execution costs reasonable. Related promising ideas are data-efficient training strategies [1], pseudo-label generation [24], geometric learning [28], or simulation [41]. Another challenge is extremely long-running query plans, which cannot be executed in a practical sense as one plan would take days. For this, other strategies are needed to reflect also timeout queries in the training procedure. This is again non-trivial, as simply using a high constant value representing timeout queries leads to difficulties, as the LCM cannot really learn meaningful costs from this.

R4: Do Not Throw Expert Knowledge Away. We observed that using estimates from PostgreSQL is beneficial for LCMs, as demonstrated by the comparably good results of `DACE` and `QPP-Net`. Such a hybrid approach combines both the expert knowledge incorporated in traditional approaches and the strength of ML, which can learn arbitrarily complex functions. While `DACE` and `QPP-Net` inherently use these estimates as training features, recent work explicitly combines traditional cost functions with learned query-specific coefficients as a hybrid approach [40]. This hybrid approach is particularly promising when looking at situations where LCMs still face major challenges, and traditional models are more reliable, like string-matching operations or user-defined functions.

8 RELATED WORK

In the following, we structure and discuss related work.

Learned Cost Models (LCMs). As discussed in Section 1, existing works on LCMs do not evaluate against query optimization tasks. Only a few evaluations exist, that in contrast to our evaluation lack either the consideration of diverse query optimization tasks or analyze only a

limited number of learned approaches. For example, an early analysis [38] also compares traditional to ML-based cost models. However, only simple ML approaches were examined. Moreover, [21] analyzes the quality of cardinality estimates and cost models of traditional query optimizers, but they do not evaluate learned approaches. Another line of work evaluates individual aspects, such as the impact of different query plan representation methods and featurizations on cost estimation [5, 45].

Learned Cardinalities (LCE). The idea of learning cardinalities has been widely studied [14, 18, 42, 43] and often evaluated. [36] showed that LCE approaches are often more accurate than traditional methods but come with high training and inference efforts. [32] revisits learned cardinalities and proposes a unified design space. However, neither study evaluates the effects of query optimization. Different is [17], which showed that the overall runtime using LCE approaches can significantly reduce against PostgreSQL in many cases. Another recent work [28] showed that a similar bias occurs for LCE as we show in our work for LCMs, as they typically rely on training samples that are provided by the PostgreSQL optimizer and thus near-optimal. They address this bias by proposing a novel geometric LCE approach that requires fewer training samples. However, these approaches cannot be easily transferred to LCMs.

Learned Query Optimization (LQO). The idea of LQO is to directly predict an optimal query execution plan or execution hints given a SQL query without the use of a cost model [6, 26, 41, 47]. While these approaches improve the overall query performance, the generality of these results is limited, as shown by [20]. Their work found that PostgreSQL still outperforms recent LQO approaches in many cases, especially for the end-to-end execution time, including inference and plan selection. Similar to our work, they point out that learned database components often do not behave as expected and highlight biased evaluation strategies. However, they focus on reinforcement learning approaches and the effect of different sampling strategies for obtaining splits of the training and test data.

9 CONCLUSION

In this paper, we analyzed how good recent LCMs really are for query optimization tasks. Particularly, we experimentally evaluated seven recent LCMs for join ordering, access path selection, and physical plan selection. Although LCMs are basically capable of learning complex cost functions, they are often inferior to traditional approaches in selecting possible execution plans. To improve future LCMs for query optimization, we recommend using appropriate metrics, diversification of training data, and hybrid models that incorporate estimates from traditional cost models.

ACKNOWLEDGEMENTS

This work has been supported by the LOEWE program (Reference III 5 - 519/05.00.003-(0005)), hessian.AI at TU Darmstadt, the IPF program at DHBW Mannheim, as well as DFKI Darmstadt.

REFERENCES

[1] Pratyush Agnihotri, Boris Koldehofe, Paul Stiegele, Roman Heinrich, Carsten Binnig, and Manisha Luthra. 2024. ZEROTUNE: Learned Zero-Shot Cost Models for Parallelism Tuning in Stream Processing. In *IEEE 40th International Conference on Data Engineering (ICDE 2024), Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 2040–2053. https://doi.org/10.1109/ICDE60146.2024.00163

[2] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 390–401. https://doi.org/10.1109/ICDE.2012.64

[3] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger,

Bradford W. Wade, and Vera Watson. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1, 2 (1976), 97–137. https://doi.org/10.1145/320455.320457

[4] Henriette Behr, Volker Markl, and Zoi Kaoudi. 2023. Learn What Really Matters: A Learning-to-Rank Approach for ML-based Query Optimization. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings (LNI, Vol. P-331)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.). Gesellschaft für Informatik e.V., 535–554. https://doi.org/10.18420/BTW2023-25

[5] Baoming Chang, Amin Kamali, and Verena Kantere. 2024. A Novel Technique for Query Plan Representation Based on Graph Neural Nets. In *Big Data Analytics and Knowledge Discovery - 26th International Conference, DaWaK 2024, Naples, Italy, August 26-28, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14912)*, Robert Wrembel, Silvia Chiusano, Gabriele Kotsis, A Min Tjoa, and Ismail Khalil (Eds.). Springer, 299–314. https://doi.org/10.1007/978-3-031-68323-7_25

[6] 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

[7] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Parametric Query Optimization. *Proc. ACM Manag. Data* 1, 1 (2023), 109:1–109:25. https://doi.org/10.1145/3588963

[8] Jennie Duggan, Ugur Çetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 337–348. https://doi.org/10.1145/1989323.1989359

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[10] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings of the 25th International Conference on Data Engineering (ICDE 2009), March 29 2009 - April 2 2009, Shanghai, China*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 592–603. https://doi.org/10.1109/ICDE.2009.130

[11] Zhen He, Byung Suk Lee, and Robert R. Snapp. 2005. Self-tuning cost modeling of user-defined functions in an object-relational DBMS. *ACM Trans. Database Syst.* 30, 3 (2005), 812–853. https://doi.org/10.1145/1093382.1093387

[12] Roman Heinrich, Carsten Binnig, Harald Kornmayer, and Manisha Luthra. 2024. Costream: Learned Cost Models for Operator Placement in Edge-Cloud Environments. In *40th IEEE International Conference on Data Engineering (ICDE 2024), Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 96–109. https://doi.org/10.1109/ICDE60146.2024.00015

[13] 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

[14] 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

[15] Zisis Karampaglis, Anastasios Gounaris, and Yannis Manolopoulos. 2014. A bi-objective cost model for database queries in a multi-cloud environment. In *Proceedings of the 6th International Conference on Management of Emergent Digital EcoSystems*. 109–116.

[16] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. 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.). 3146–3154. https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html

[17] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. [n. d.]. Learned Cardinality Estimation: An In-depth Study. In *Proceedings of the 2022 International Conference on Management of Data* (New York, NY, USA, 2022-06-11) *(SIGMOD '22)*. Association for Computing Machinery, 1214–1227. https://doi.org/10.1145/3514221.3526154

[18] 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

[19] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering* 6 (2021), 86–101.

[20] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proc. VLDB Endow.* 17, 7 (2024), 1565–1577. https://doi.org/10.14778/3654621.3654625

[21] 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

[22] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, Bolong Zheng, and Zhiyong Peng. [n. d.]. A learned cost model for big data query processing. 670 ([n. d.]), 120650. https://doi.org/10.1016/j.ins.2024.120650

[23] Zibo Liang, Xu Chen, Yuyang Xia, Runfan Ye, Haitian Chen, Jiandong Xie, and Kai Zheng. 2024. DACE: A Database-Agnostic Cost Estimator. In *40th IEEE International Conference on Data Engineering (ICDE 2024), Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 4925–4937. https://doi.org/10.1109/ICDE60146.2024.00374

[24] Shuncheng Liu, Xu Chen, Yan Zhao, Jin Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Learning with Pseudo Labels for Query Cost Estimation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022*, Mohammad Al Hasan and Li Xiong (Eds.). ACM, 1309–1318. https://doi.org/10.1145/3511808.3557305

[25] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. 2021. Pre-training Summarization Models of Structured Datasets for Cardinality Estimation. *Proc. VLDB Endow.* 15, 3 (2021), 414–426. https://doi.org/10.14778/3494124.3494127

[26] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *SIGMOD Rec.* 51, 1 (2022), 6–13. https://doi.org/10.1145/3542700.3542703

[27] Ryan 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

[28] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proc. VLDB Endow.* 17, 4 (2023), 740–752. https://doi.org/10.14778/3636218.3636229

[29] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[30] C. Spearman. 1904. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 15, 1 (1904), 72–101. http://www.jstor.org/stable/1412159

[31] 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

[32] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97. https://doi.org/10.14778/3485450.3485459

[33] 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

[34] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. 16, 6 (2023), 1413–1425. https://doi.org/10.14778/3583140.3583156

[35] 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

[36] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654. https://doi.org/10.14778/3461535.3461552

[37] Michael Widenius, Davis Axmark, and Paul DuBois. 2002. *Mysql Reference Manual* (1st ed.). O'Reilly & Associates, Inc., USA.

[38] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *29th IEEE International Conference on Data Engineering (ICDE 2013), Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899

[39] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2022. A Unified Transferable Model for ML-Enhanced DBMS. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p6-wu.pdf

[40] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch? *Proc. ACM Manag. Data* 1, 4, 255:1–255:27. https://doi.org/10.1145/3626769

[41] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. [n. d.]. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia PA USA, 2022-06-10) *(SIGMOD '22)*. ACM, 931–944. https://doi.org/10.1145/3514221.3517885

[42] 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

[43] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi 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

[44] 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://doi.org/10.14778/3529337.3529349

[45] 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://doi.org/10.14778/3636218.3636235

[46] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428. https://doi.org/10.14778/3397230.3397238

[47] 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