OmniQO: An Adaptive Framework for Integrating ML and Traditional Query Optimizers

Junming Chen[1], Xu Chen[1], Zibo Liang[1], Jianbin Qin[2], Yan Zhao[3], and Kai Zheng[1](✉)

[1] University of Electronic Science and Technology of China, Chengdu, China
{junmingchen,xuchen,zbliang}@std.uestc.edu.cn
zhengkai@uestc.edu.cn
[2] Shenzhen University, Shenzhen, China
qinjianbin@szu.edu.cn
[3] Shenzhen Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen, China
zhaoyan@uestc.edu.cn

Abstract. Query optimization is essential for database performance and is primarily managed by the query optimizer. Over the years, various optimizer have been developed, each with unique strengths and limitations. These limitations, such as high tail latency and poor performance on complex queries, have hinder their applicability in production environments. To leverage the strengths of various optimizers while mitigating their weakness, we propose a lightweight, extensible, and adaptive framework called OmniQO (Omni Query Optimizer). OmniQO is designed to seamlessly integrate multiple query optimizers by dynamically analyzing query characteristics and intelligently selecting the optimizer best suited for the current query at negligible cost. By combining the strengths of different approaches and addressing their limitations, OmniQO enhances end-to-end query performance, prevents model degradation, and significantly improves system stability. Extensive experimental results confirm that OmniQO outperforms state-of-the-art methods in query latency, stability, and efficiency.

Keywords: Database · Query optimization · Machine learning.

1 Introduction

In modern Database Management Systems (DBMS), query optimization serves as a cornerstone that significantly impacts the overall system performance [7]. Over decades of development, diverse query optimization techniques have been introduced. Traditional cost-based Query Optimizers (QOs), such as the one in PostgreSQL (PG), rely on dynamic programming enumeration and an empirical cost model to identify optimal query execution plans. Recently, Machine Learning (ML)-based QOs have grown rapidly [3, 5, 9, 15, 18, 20, 21], leveraging historical data to train powerful models for query optimization. Some approaches, such as LEON [3], employ learning models as cost models to improve prediction

accuracy. Others, like Bao [15], enhances traditional approaches by generating candidate plans using query hints and selecting the optimal plan from them.

However, the "one size does not fit all" phenomenon is evident in QOs, as no single QO can achieve satisfactory results across all query scenarios. Existing QOs face inherent limitations due to their architecture and design choices, which prevent them from fully accommodating the diverse requirements of various workloads. A deeper analysis of these limitations is presented in Sec. 2.

To better validate our point, we selected three representative QOs — PG, LEON [3], and Bao [15] — and set up a simple experiment to analyze the specific defect scenarios of each. We chose some typical query templates[4] from JOB [10] and STACK [15] and generated a series of queries for each template, dividing them into training and test sets. For PG, which uses traditional methods, we directly ran the queries on the test set. For the learning-based methods, LEON and Bao, we evaluated their performance on the test set after training their models on the training set. We then recorded the average end-to-end (e2e) time[5] for each QO on every template. As shown in Fig. 1, the disadvantages of PG, LEON, and Bao are clearly evident.

Degradation Scenario of PG: High-Cardinality Queries. The PG query optimizer, representative of the Standard Query Optimizers (SQOs), generates execution plans based on cost estimation and heuristic formulas. However, when dealing with high-cardinality table queries, especially in cases of data skew, it may underestimate condition selectivity, leading to suboptimal plans. As illustrated in Fig. 1, PG performs well for JOB-q1, JOB-q6, and JOB-q28. However, for the more complex STACK-q2, its performance degrades significantly compared to LEON, with all query execution times exceeding 300 seconds.

Degradation Scenario of Bao: Lightweight and Multi-Table Queries. Bao processes queries by invoking the SQO multiple times under various database configurations to generate candidate plans, from which the final execution plan is selected. This process incurs significant computational overhead. In scenarios dominated by short and lightweight queries, the time spent on plan generation can exceed the actual query execution time, leading to substantial increases in e2e latency. As shown in Fig. 1, Bao performs well on JOB-q1, which involves fewer joins and is not a lightweight query. However, it exhibits poor performance compared to PG and LEON on the lightweight query JOB-q6 and the complex multi-table query JOB-q28.

Degradation Scenario of LEON: Unseen Queries. Learned Query Optimizers (LQOs) like LEON are inherently vulnerable to stability and generalization challenges, particularly in dynamic query workloads [11, 16]. These models are typically trained on historical data, which can render them insufficiently robust when encountering previously unseen query patterns. This may lead to the generation of suboptimal or significantly degraded execution plans, resulting in

[4] A query template is a query with placeholders for values, representing structurally similar queries.

[5] The End-to-end time is the sum of the planning time (time taken by the query optimizer to generate a plan) and the execution time (time to execute the plan).

unacceptable performance regressions [1]. As illustrated in Fig. 1, LEON outperforms the other two optimizers significantly on STACK-q3, but underperforms on JOB-q1 and JOB-q6. In addition, Fig. 2 showcases the execution times of plans selected by each optimizer on the JOB dataset. While LEON demonstrates strong performance on certain queries, it exhibits more pronounced performance fluctuations compared to PG and Bao on others.

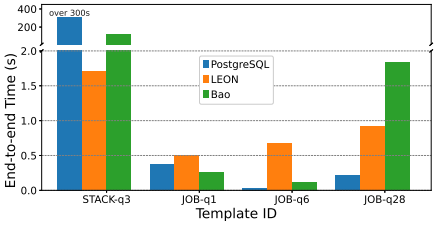

Fig. 1. Query templates where PG/LEON/Bao excel and struggle. Each template includes multiple queries and shows the average e2e time.

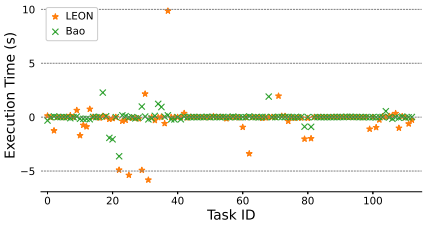

Fig. 2. Execution time difference on JOB (LEON/Bao execution time minus PostgreSQL execution time).

These degradation scenarios significantly limit their performance and applicability in real-world production environments. In addition to the degradation cases mentioned for the three optimizers, other optimizers exhibit a variety of additional issues (in Sec. 2). More importantly, beyond the constrained scenarios that human experts can identify, different QOs face even more unpredictable and unexplainable weaknesses in complex query workloads. 2 To this end, we introduce OmniQO (Omni Query Optimizer), a lightweight, highly extensible, and adaptive framework that seamlessly integrates multiple optimizers into a unified system. It leverages the strengths of individual query optimizers while mitigating their limitations. The foundation of OmniQO lies in its lightweight feature extractor, a highly scalable and dynamic optimizer selection mechanism, and training objectives and loss functions designed for improved generalization.

For each incoming query, OmniQO analyzes its lightweight key features and employs a fast machine learning model to evaluate any number of integrated QOs. The scores generated from these evaluations reflect the suitability of each optimizer for the specific query, enabling the framework to greedily assign the query to the optimizer with the highest score. During training, OmniQO capture the e2e latency of each QO, as this metric reflects the QOs's performance in both favorable and challenging scenarios. To guide training, we designed a novel objective function that converts e2e latency into a performance score for each QO. Additionally, OmniQO utilizes an example-dependent cross-entropy [2] loss function to enhance learning. Together, these techniques enable OmniQO to deeply understand each QO's strengths and effectively harness their capabilities.

Extensive experimental results demonstrate that OmniQO efficiently integrates multiple QOs at a negligible cost. On the JOB dataset, compared to individual optimizers, OmniQO reduced the end-to-end time by 13% to 80%. On the STACK dataset, the reduction is 5% to 90%. Notably, OmniQO demonstrates exceptional adaptability, maintaining consistent stability across different datasets. For instance, in subsequent experiments (detail in Sec. 5), optimizers that perform well on the JOB dataset often struggle on the STACK dataset, and vice versa. In contrast, OmniQO dynamically adjusts to the characteristics of each workload, effectively leveraging the strengths of its integrated optimizers.

Our contributions are summarized as follows:

- **Leveraging Strengths, Avoiding Weaknesses:** OmniQO intelligently collaborates with any number of QOs, bypassing those unsuitable for a given query and prioritizing the optimizers best suited to handle it. This adaptive selection ensures optimal query performance across diverse workloads.
- **Optimizing E2e Performance:** By dynamically assigning the best-suited optimizer to each query, OmniQO minimizes overhead and significantly reduces e2e latency, thereby enhancing query processing efficiency across varied scenarios.
- **Improved Stability**: OmniQO effectively mitigates the stability challenges inherent in LQOs. By coordinating across multiple optimizers, the framework mitigates performance degradation under adverse conditions, automatically switching to the most reliable optimizer to ensure system resilience.
- **Cost-Effective, Deployable, and Scalable:** OmniQO is designed for flexibility and ease of integration. It requires minimal hardware resources and incurs negligible computational overhead, and can effortlessly collaborate with any number of QOs, making it highly suitable for large-scale deployment in real-world systems.

2 Related Work

The Query Optimizer is a core component of database management systems, tasked with rapidly converting queries into efficient executable plans.

Standard Query Optimizer: Traditional optimizers primarily rely on rules and cost-based approaches to select execution plans. They estimate the cost of plans using statistical data and heuristic formulas, utimately choosing the plan with the lowest estimated cost. A representative example is PostgreSQL's query optimizer, which is generally efficient and stable in producing plans for most queries. However, the heuristic formulas used by PostgreSQL are based on simplifying assumptions, such as uniformity and independence. In real-world production environments, these assumptions frequently break down, resulting in sub-optimal and sometimes disastrous plans.

Learned Query Optimizer: As a key component in AI4DB area [4, 17, 24, 25], modern query optimizers increasingly incorporate machine learning to generate superior execution plans. We categorize these into several types. First,

some approaches [9, 14, 18, 21] leverage learning models to address cardinality and cost estimation issues in query optimization. While these methods improve prediction accuracy, they often face high overhead and lack of stability. Second, some research [12, 15] focuses on optimizing queries through DBMS tuning or introducing query hint sets. These method achieve good performance with rapid learning convergence but are limited in handling complex queries due to constrained search spaces. Additionally, there are studies [20] exploring reinforcement learning techniques to generate execution plans that can operate independently of expert knowledge and still achieve excellent performance, but the large search space poses challenges for convergence speed.

3 Overview

3.1 Problem Definition

To define our problem, we first clarify several related concepts. The role of a query optimizer QO is to generate the most efficient execution plan p for a given query q within the shortest possible time. In evaluating a query optimizer's performance, two main metrics are considered: the planning time, denoted as $planning\_time(QO, q)$, which represents the time taken by the query optimizer QO to generate the execution plan for query q; and the execution time, represented as $execution\_time(p)$, which is the time required to run the generated plan p. The sum of these two metrics is termed the end-to-end time or $e2e\_time(QO, q)$, indicating the total time taken for the query optimizer QO to handle query q, encompassing both planning and execution phases.

In our task setting, we have n query optimizers implemented through various approaches, denoted as $\{QO_0, \ldots, QO_{n-1}\}$. The objective is to train a classifier model, *OmniQO*, designed to predict the optimal index i ($i \in \{0, \ldots, n-1\}$) for any given query q, where the selected optimizer QO_i is expected to achieve the minimal $e2e\_time$. This mapping from query q to the optimal QO_i is defined by the following formula.

$$\text{OmniQO}(q) = \underset{i \in \{0,\ldots,n-1\}}{\arg\min} \ \text{e2e\_time}(QO_i, q) \tag{1}$$

3.2 Framework Overview

In this section, we present the framework of OmniQO, as illustrated in Fig. 3. OmniQO takes a query as input and selects the optimal Query Optimizer (QO) to generate an execution plan. The core steps of OmniQO include: (1) Collect Data, (2) Select Optimizer, and (3) Generate and Execute Plan.

Collect Data: Due to the limited information directly available from queries, we employ the SQO to generate a default execution plan, enriching the feature set for downstream model analysis [8, 23]. These plans, while not always optimal, include critical data about the query, such as primary joins and base tables, and provide essential approximate metrics on cardinalities and execution costs.

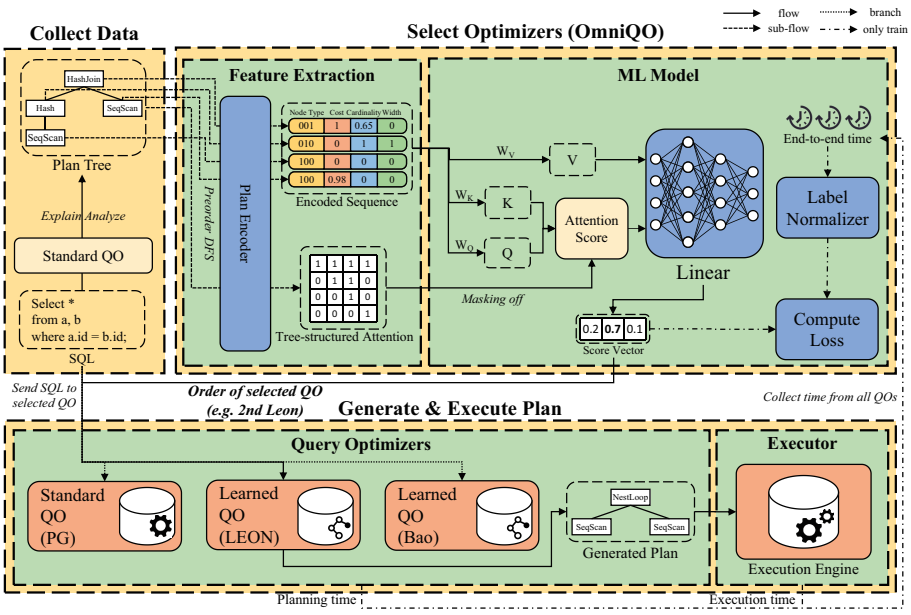

Fig. 3. OmniQO Framework.

Select Optimizers (OmniQO): In this phase, we employ a Depth-First Search (DFS) algorithm to traverse the default execution plan tree and extract critical information, including node attributes and structural details of the plan tree[6]. These features are then analyzed by a streamlined machine learning model to identify the optimal QO. Our model training is strategically designed to robustly emphasize both superior and inferior QOs, thereby facilitating the selection of the most effective QO while circumventing those that could impair query performance.

Generate and Execute Plan: Once the model identifies the best QO, the query is directed to that QO to generate an executable plan, which is then handed to the execution engine [7]. During training, we gather e2e latency data for each query in the dataset across all QOs. This data, comprising the planning time of each QO and the execution time in the execution engine, is used to train the classifier, enabling it to optimize QO selection with high precision.

4 Methodology

4.1 Collect Data

To overcome the constraints of limited data available from queries, we employ a expert QO (e.g. PostgreSQL) to generate preliminary execution plans, enhancing

[6] In this paper, we use the terms "plan", "query plan", "plan tree" and "query plan tree" to represent the execution plan of the DBMS

the analytical capabilities of our models. This plan delivers the SQO's insights into essential query elements and provides a reference execution path rich with informative details.

Although generating a default execution plan with the SQO introduces some computational cost, OmniQO can significantly optimize this overhead in practice. For example, suppose the SQO is ultimately chosen for plan execution. In that case, the pre-generated plan can be directly passed to the executor, eliminating redundant planning steps and streamlining overall execution efficiency.

4.2 Feature Extraction in OmniQO

This section focuses on extracting and encoding lightweight features from query plans, enabling the model to fully utilize this information for optimization. The feature extraction process revolves around a key encoding component—Plan Encoder—which generates two critical outputs: an encoded sequence and an tree-structured attention matrix.

Plan Encoder. The Plan Encoder's primary function is to transform a query plan into feature representations that the model can understand, enabling it to capture the detailed information embedded within the plan tree. Using a pre-order DFS algorithm, the encoder traverses the plan tree to extract two key types of information: the node-level details of the plan tree (Encoded Sequence) and its structural information (Tree-structured attention).

Encoded Sequence. OmniQO primarily extracts and encodes several key features from nodes. It encodes the NodeType feature, which represents the type of node in the plan tree, using one-hot encoding to differentiate node types. Additionally, it normalizes features such as estimated cost, estimated cardinality, and average width to ensure consistent scaling. These features are then integrated into a one-dimensional encoded sequence for each node, as depicted in Fig. 4. Subsequently, the encoded sequences from all nodes are combined to form a two-dimensional feature matrix.

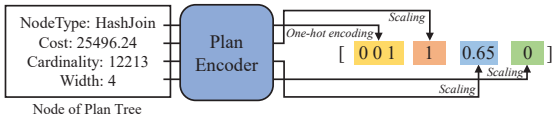

Fig. 4. Encode feature from node.

We omit features like filters and bitmap representations, which have been used in previous studies [8, 23], in order to preserve model efficiency. The chosen features—NodeType, Cost, Cardinality, and Width—are deemed sufficient for capturing the essential information.

Tree-structured Attention. Tree-structured attention models the hierarchical structure of a query plan by restricting each node's attention to its descendants, capturing parent-child relationships [13, 23]. We achieve this using an $n \times n$ matrix, where n represents the number of nodes in the query plan tree. The matrix is initialized to zeros, then for each node i, if node j is a descendant of node i, we set the value at position (i, j) to 1, indicating that node i will attend to node j. Finally, we set the diagonal to 1 to allow each node to attend to itself.

4.3 ML Model in OmniQO

Transformer. We draw on the DACE [13] framework, applying self-attention mechanisms [19] to the plan encoding output for feature extraction and incorporating tree-structured attention to support masked outputs, defined as follows:

$$\begin{cases} Q = SW_Q, \quad K = SW_K, \quad V = VW_V, \\ \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T \circ M}{\sqrt{d}}\right) V, \end{cases} \tag{2}$$

Let $S \in \mathbb{R}^{m \times d}$ represent the node encoding matrix, where each node is encoded as a feature vector, m denotes the total number of nodes, and d specifies the dimensionality of the node encoding. The parameters $W_Q \in \mathbb{R}^{d \times d_k}$, $W_K \in \mathbb{R}^{d \times d_k}$, and $W_V \in \mathbb{R}^{d \times d_v}$ are used to project the input features into queries $Q \in \mathbb{R}^{m \times d_k}$, keys $K \in \mathbb{R}^{m \times d_k}$, and values $V \in \mathbb{R}^{m \times d_v}$, respectively. Here, d_k and d_v refer to the dimensionalities of the query-key and value vectors. Additionally, $M \in \mathbb{R}^{m \times m}$ denotes the attention mask, constructed from the tree-structured attention matrix.

Unlike DACE, which predicts outcomes for all subplans, our approach targets only the full execution plan's result. We therefore isolate the output corresponding to the root node from $\text{Attention}(Q, K, V) \in \mathbb{R}^{m \times d_v}$, and pass it through a linear layer to extract further features. This layer outputs a sequence with a length matching the number of QOs, where each element reflects the performance score associated with each QO.

Label Normalizer. During the training phase, we collect e2e time data for the current query q across all n QOs, covering both planning and execution times, and label it as $\text{T} = \{\text{e2e\_time}(QO_0, q), \ldots, \text{e2e\_time}(QO_{n-1}, q)\}$, and apply a Label Normalizer to standardize these values as training targets. Inspired by Kepler [6]'s objective, our approach guides the model not only to identify a single optimal QO but to recognize a set of high-performing QOs. To further refine this, we introduce soft labels that provide a nuanced gradient [22], distinguishing strong QOs from weaker ones. Specifically, we process the e2e latency of each QO as follows.

$$\begin{cases} I = \{i \mid \text{e2e\_time}(QO_i, q) \leq C \cdot \min(T)\} \\ t_i = \begin{cases} 0 & i \notin I \\ \exp\left(-\frac{\text{e2e\_time}(QO_i, q)}{\max_{j \in I} \text{e2e\_time}(QO_j, q)} \cdot D\right) & i \in I \end{cases} \end{cases} \tag{3}$$

$$\text{label\_normalizer}(T) = \left\{ \frac{t_i}{\sum_j t_j} \;\middle|\; i = 0, 1, \ldots, n-1 \right\} \tag{4}$$

Formula (3) is designed to assign scores t_i based on the $e2e\_time$ of query plans, with the goal of emphasizing nodes with shorter e2e times while down-weighting those with longer e2e times. Specifically, we first identify query plans whose e2e times are within C times the minimum e2e time. For nodes outside this set, we assign a score of 0. For nodes within the set, we normalize their e2e times by dividing each by the maximum e2e times, scaling the values to the range $[0, 1]$. Next, we multiply by a constant D and take the negative value. Finally, an exponential function is applied to produce the final score.

Formula (4) refines each t_i into a normalized score by dividing each value by the total sum of t values, ensuring the scores collectively sum to 1. This yields the final output, $label\_normalizer(T)$, representing the normalized distribution of e2e times.

The constants C and D mentioned above are hyperparameters.

Compute Loss. We use example-dependent cross entropy [2] as the loss function, incorporating Kepler-inspired adaptive weighting [6] to adjust the calculated loss. This approach guides the model to prioritize selecting the optimal QO while remaining sensitive to suboptimal plans, effectively steering the model away from subpar QO choices. Specifically, for plans whose e2e times exceed E times $min(T)$, we multiply their weights by a factor W_{bad}. For near-optimal plans, we apply a soft weighting based on their empirical improvement in the e2e time ratio, i.e. $\exp\left((C - \frac{\text{e2e\_time}(QO_i, q)}{\min(T)}) \cdot W_{good}\right)$.

Additionally, we assign greater weights to queries with higher time costs in the dataset to mitigate long-tail effects. Based on the maximum value $max(T)$, we divide queries into k intervals $\{(0, \text{intv}_0], (\text{intv}_0, \text{intv}_1], \ldots, (\text{intv}_{k-2}, +\infty)\}$, applying a corresponding weight factor $\{W_0, W_1, \ldots, W_{k-1}\}$ to all plans within the same interval.

The constants mentioned above, E, W_{bad}, W_{good}, k, intv_i, W_i, are all hyperparameters.

4.4 Generate and Execute Plan

Query Optimizers. OmniQO's high scalability enables seamless collaboration with any number of QOs. By integrating QOs with diverse strengths, OmniQO ensures that a suitable optimizer is always selected to handle workloads effectively across various scenarios. As illustrated in Fig. 3, we demonstrate this capability using one SQO (PostgreSQL) and two LQOs (LEON and Bao) as examples working in conjunction with OmniQO.

Executor. In OmniQO framework, the database executor [7] executes the chosen physical plan, processes the data as instructed, and returns the results.

5 Experiment

5.1 Experiment Setup

Datasets and Queries. we conduct experiments on two public datasets: JOB and STACK.

JOB:

The JOB [10] dataset is a 3.6GB (11GB with indexes) real-world benchmark based on IMDB, consisting of 21 tables. The queries in this dataset involve between 4 to 17 table joins. We utilize the 113 original queries as templates, modifying their predicates to generate 4250 new query variations while preserving the original join structures.

STACK:

The Stack dataset consists of over 18 million Q&A from 170 Stack Exchange sites, totaling 100GB. We used a workload from [15] with 16 query templates, generating 640 queries. These queries vary in complexity, with join relationships ranging from 4 to 12.

Query Optimizers. In our experiments, we integrated OmniQO with a range of query optimizers, including PostgreSQL's native optimizer and two learned optimizers, LEON and Bao.

Baselines. To ensure a fair evaluation of OmniQO's performance, we established several comparison baselines:

Independent QO: Each QO was tested in standalone mode to gauge performance when deployed independently.

OmniQO Variants: We introduced adjustments to key components of OmniQO, such as simplifying its training objectives and loss functions and introducing more comprehensive encoded sequences as model inputs. These modifications will serve as an additional benchmark for evaluating system performance.

Implementation Details. The implementation details of our experiments are as follows:

Expert Engines: All learning-driven query optimization methods are deployed on PostgreSQL. Consistent with prior research [10, 20], PostgreSQL is configured with shared_buffers set to 32GB and work_mem to 4GB, and the Genetic Query Optimization (GEQO) is disabled.

Parameters Setting: We set the hyperparameters C and D mentioned in Formula (3) to 2 and 1.5, respectively. The hyperparameter settings for the weight matrix are as follows: E, W_{bad}, and W_{good} are set to 2, 5, and 15, respectively. The interval hyperparameters k, intv_i, and W_i are set to 5, $\{10, 20, 50, 100\}$, and $\{1, 2, 3, 5, 10\}$, respectively.

5.2 End to End Performance

In our experiments, we focused on end-to-end (e2e) query latency, which is the total time from query submission to result retrieval. This latency includes both the planning time, required to generate an execution plan, and the execution time, required to execute that plan. This metric provides a straightforward reflection of the optimizer's overall performance in a production environment.

Table 1. E2e performance of OmniQO and all baselines.

Methods \ Datasets	JOB E2E Time (sec)	STACK E2E Time (sec)
PG	186.66	5292.86
LEON	476.31	465.67
Bao	831.06	2493.17
OmniQO (PG + LEON)	181.66	**441.62**
OmniQO (PG + Bao)	166.70	2444.08
OmniQO (PG + LEON + Bao)	**162.74**	444.05

Table 1 compares the e2e latency performance of all baselines. Additionally, we included OmniQO that integrate only PG and LEON, as well as PG and Bao. Overall, all variants of OmniQO outperform individual QOs they integrate. Among them, the OmniQO integrating all QOs demonstrates significant advantages in e2e latency. On the JOB dataset, it achieves approximately a 13% performance improvement over the best-performing PostgreSQL (PG) method. On the STACK dataset, OmniQO outperforms the best method, LEON, by about 5%.

Leveraging Strengths, Avoiding Weaknesses. Fig. 5 illustrates the e2e performance on the JOB and STACK datasets. The results are divided into three regions, each representing the queries where PG, Bao, or LEON performs best. Within each region, the data is sorted by e2e time. These three regions highlight the query scenario where each QO excels. Clearly, OmniQO almost always selects the optimal strategy across all regions, demonstrating its ability to leverage the strengths of the three optimizers. OmniQO identifies their areas of expertise for each QO, and intelligently assign queries to the most suitable QO.

Furthermore, even in scenarios where OmniQO does not select the best-performing optimizer, it rarely selects the worst-performing one. This indicates that OmniQO is also capable of effectively recognizing the weaknesses of each optimizer and avoiding them, thereby enhancing overall performance stability.

Improved Stability. OmniQO is also capable of mitigating the degradation of various QO methods, particularly LQOs, to enhance system stability. Practitioners are often interested in tail latency when discussing degradation scenarios.

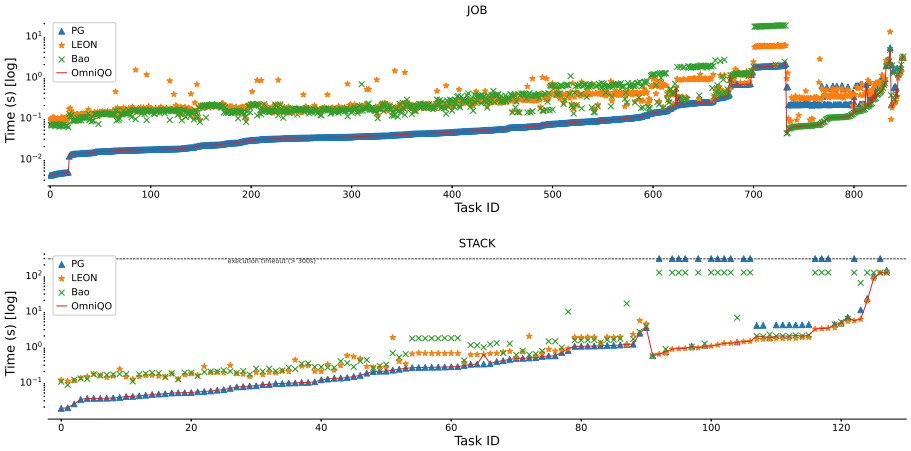

Fig. 5. E2e performance of OmniQO and all baselines on JOB and STACK datasets.

Thus, we present the 50%, 90%, 95%, 99%, and 99.9% e2e latencies of all methods on the JOB and STACK datasets.

As shown in Fig. 6, OmniQO consistently achieves the lowest tail latency across both datasets. On the JOB dataset, the tail latencies of learning-based methods LEON and Bao degrade significantly. On the STACK dataset, Bao exhibits noticeable degradation at the 90% and 95% levels. However, OmniQO, which integrates these methods, effectively avoids such degradation. Notably, the tail latency of PG on the STACK dataset is also extremely poor, exceeding 300 seconds starting from the 90% level. This indicates that even PG, known for its stability, can suffer severe degradation in certain scenarios. Our method successfully avoids these issues.

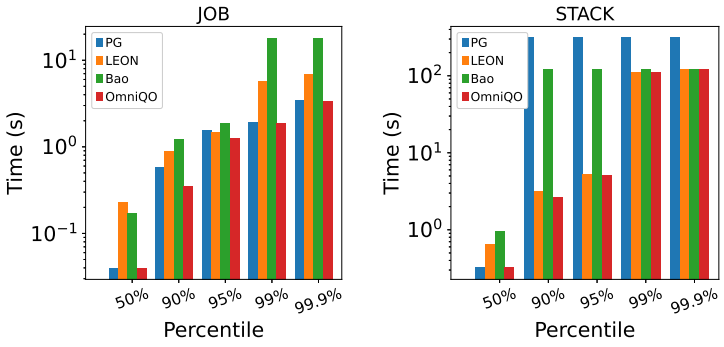

Fig. 6. Tail percentile e2e latency of OmniQO and all baselines.

5.3 Generalizability Analysis

This section investigates the generalization capabilities of OmniQO. Specifically, we employ a leave-one-out approach, in which each query template is iteratively excluded from the training set and subsequently used to test the model's performance on previously unseen templates. This methodology allows for a robust evaluation of OmniQO's performance against all baseline models.

Additionally, we assess the effectiveness of our soft-label training objective and the example-dependent cross-entropy approach through comparative analyses with additional baseline models.

Fig. 7 illustrates the e2e latency comparison of OmniQO and three baseline models across various query templates. OmniQO consistently outperforms all baselines for most templates, lagging behind PostgreSQL only in a few cases (e.g., JOB-q5, JOB-q7). Notably, OmniQO avoids latency drops on challenging templates (e.g., JOB-q18, JOB-q1) where PostgreSQL underperforms and leverages the strengths of all three baselines to achieve optimal latency on templates such as JOB-q15, JOB-q10, JOB-q12, and JOB-q4.

The robust generalization ability of OmniQO is largely attributed to its adoption of soft-label training and example-dependent cross-entropy loss. To validate the effectiveness of this approach, we compared OmniQO trained with standard one-hot encoding and cross-entropy loss. The results, illustrated in the Fig. 8, indicate that our approach not only matches but frequently surpasses the performance of traditional methods across all templates, with significant improvements on JOB-q10 and JOB-q5. This demonstrates the effectiveness of soft-label training and example-dependent loss in enhancing OmniQO's adaptability and performance on diverse query scenarios.

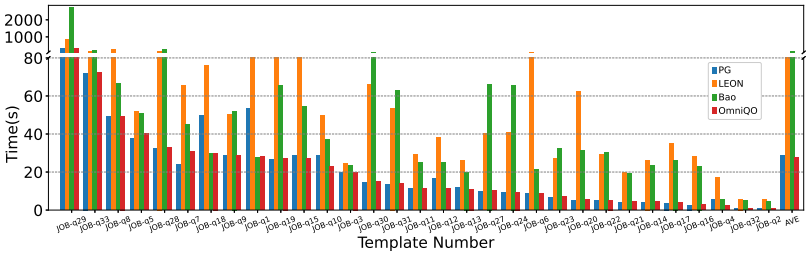

Fig. 7. Performance of OmniQO based on leave-one-out test on JOB.

5.4 OmniQO Efficiency

In this section, we discuss the efficiency of OmniQO, focusing on model size and inference performance. A highly efficient OmniQO ensures that it selects the optimal query optimizer without adding additional overhead on the workload.

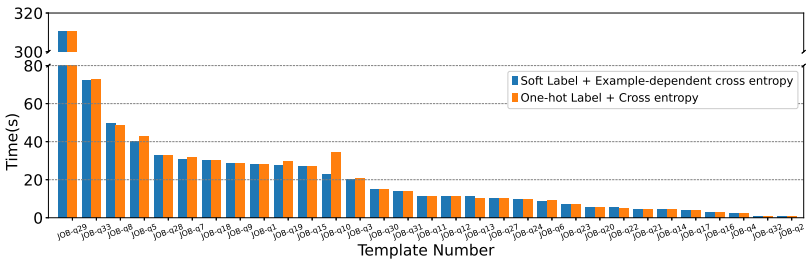

Fig. 8. Impact of soft-label training objectives and example-dependent cross-entropy loss on OmniQO's performance across query templates.

We further explore whether it is necessary to incorporate more features. While additional features can enrich the information available for model inference, they also increase encoding and inference time, ultimately reducing model efficiency.

We configured OmniQO with various encoding sequences as inputs and compared (1) model size, (2) e2e performance, and (3) inference efficiency. Alongside analyzing model efficiency, we assess the necessity of different input features for OmniQO's effectiveness.

Table 2. Efficiency analysis of the models on JOB.

Method	Model Size (MB)	E2E Time including OmniQO's overhead (s)	Inference Efficiency (s/query)
OmniQO	**0.3**	**167.286 (+1.289)**	**0.0014**
OmniQO + bitmap	3.1	169.489 (+199.987)	0.2352
OmniQO + histogram	1.6	167.946 (+183.350)	0.2157
OmniQO + bitmap + histogram	6.2	172.789 (+205.37)	0.2416

We introduced bitmap and histogram features into OmniQO to better capture filtering conditions in queries. The bitmap feature records filter distributions across base tables, while the histogram feature uses database statistics for finer-grained filtering descriptions.

Our experiments tested four model configurations: the basic OmniQO and versions with added bitmap and/or histogram features (see Table 2). Results showed that the basic OmniQO is extremely lightweight at just 0.3 MB, while incorporating bitmap and histogram features significantly increased the model size, expanding it by approximately 10, 5, and 20 times, respectively.

In evaluating e2e time and efficiency, we highlighted OmniQO's overhead in parentheses (see the third column in Table 2). The basic OmniQO model delivers excellent performance, optimizing e2e query time with negligible over-

head. It completes query optimizer selection within milliseconds, contributing to less than 1% of the total query time. In contrast, while adding bitmap and histogram features improves query representation, it does not yield significant performance gains and substantially increases inference time. In some cases, the overhead introduced by OmniQO exceeds the combined time for query planning and execution, making these additional features counterproductive.

In summary, OmniQO demonstrates outstanding efficiency as a lightweight, effective solution for query optimization, achieving optimizer selection within milliseconds. While additional features like bitmap and histogram enhance query representation, they significantly increase model size and inference time without offering proportional performance gains, making the basic OmniQO model the preferred choice for most practical applications.

6 Conclusion

In this paper, we propose OmniQO, a lightweight, extensible and adaptive framework. OmniQO collaborates with multi Query Optimizers (QOs) and intelligently selects the optimal QO based on query characteristics. OmniQO leverages the strengths of all integrated QOs while avoiding their individual weaknesses. Extensive experiment demonstrates OmniQO's advantages in end-to-end performance, system stability, model generalization, and efficiency, offering a scalable and effective solution for modern query optimization in database systems.

Acknowledgments. This work is partially supported by NSFC (No. 62472068), Shenzhen Municipal Science and Technology R&D Funding Basic Research Program (JCYJ20210324133607021), and Municipal Government of Quzhou under Grant (No. 2023D044).

References

1. Ammerlaan, R., Antonius, G., Friedman, M., Hossain, H.S., Jindal, A., Orenberg, P., Patel, H., Qiao, S., Ramani, V., Rosenblatt, L., et al.: Perfguard: deploying ml-for-systems without performance regressions, almost! Proc. VLDB Endow. **14**(13), 3362–3375 (2021)
2. Bahnsen, A.C., Aouada, D., Ottersten, B.: Example-dependent cost-sensitive logistic regression for credit scoring. In: 2014 13th International conference on machine learning and applications. pp. 263–269. IEEE (2014)
3. Chen, X., Chen, H., Liang, Z., Liu, S., Wang, J., Zeng, K., Su, H., Zheng, K.: Leon: A new framework for ml-aided query optimization. Proc. VLDB Endow. **16**(9), 2261–2273 (2023)
4. Chen, X., Liu, S., Yuan, T., Ye, T., Zeng, K., Su, H., Zheng, K.: Optimizing block skipping for high-dimensional data with learned adaptive curve. Proceedings of the ACM on Management of Data **3**(1), 1–26 (2025)
5. Chen, X., Wang, Z., Liu, S., Li, Y., Zeng, K., Ding, B., Zhou, J., Su, H., Zheng, K.: Base: Bridging the gap between cost and latency for query optimization. Proc. VLDB Endow. **16**(8), 1958–1966 (2023)

6. Doshi, L., Zhuang, V., Jain, G., Marcus, R., Huang, H., Altinbüken, D., Brevdo, E., Fraser, C.: Kepler: Robust learning for parametric query optimization. Proceedings of the ACM on Management of Data **1**(1), 1–25 (2023)
7. Hellerstein, J.M., Stonebraker, M., Hamilton, J., et al.: Architecture of a database system. Foundations and Trends® in Databases **1**(2), 141–259 (2007)
8. Hilprecht, B., Binnig, C.: Zero-shot cost models for out-of-the-box learned cost prediction. arXiv preprint arXiv:2201.00561 (2022)
9. Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., Kemper, A.: Learned cardinalities: Estimating correlated joins with deep learning. arXiv preprint arXiv:1809.00677 (2018)
10. Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really? Proc. VLDB Endow. **9**(3), 204–215 (2015)
11. Li, B., Lu, Y., Kandula, S.: Warper: Efficiently adapting learned cardinality estimators to data and workload drifts. In: Proceedings of the 2022 International Conference on Management of Data. pp. 1920–1933 (2022)
12. Li, G., Zhou, X., Li, S., Gao, B.: Qtune: A query-aware database tuning system with deep reinforcement learning. Proc. VLDB Endow. **12**(12), 2118–2130 (2019)
13. Liang, Z., Chen, X., Xia, Y., Ye, R., Chen, H., Xie, J., Zheng, K.: Dace: A database-agnostic cost estimator. In: 2024 IEEE 40th International Conference on Data Engineering (ICDE). pp. 4925–4937. IEEE (2024)
14. Liu, S., Chen, X., Zhao, Y., Chen, J., Zhou, R., Zheng, K.: Efficient learning with pseudo labels for query cost estimation. In: CIKM. pp. 1309–1318 (2022)
15. Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., Kraska, T.: Bao: Making learned query optimization practical. In: Proceedings of the 2021 International Conference on Management of Data. pp. 1275–1288 (2021)
16. Negi, P., Wu, Z., Kipf, A., Tatbul, N., Marcus, R., Madden, S., Kraska, T., Alizadeh, M.: Robust query driven cardinality estimation under changing workloads. Proc. VLDB Endow. **16**(6), 1520–1533 (2023)
17. Ni, J., Zhao, Y., Zeng, K., Su, H., Zheng, K.: Deepqt : Learning sequential context for query execution time prediction. In: DASFAA. pp. 188–203 (2020)
18. Sun, J., Li, G.: An end-to-end learning-based cost estimator. arXiv preprint arXiv:1906.02560 (2019)
19. Waswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I.: Attention is all you need. In: NIPS (2017)
20. Yang, Z., Chiang, W.L., Luan, S., Mittal, G., Luo, M., Stoica, I.: Balsa: Learning a query optimizer without expert demonstrations. In: Proceedings of the 2022 International Conference on Management of Data. pp. 931–944 (2022)
21. Yang, Z., Kamsetty, A., Luan, S., Liang, E., Duan, Y., Chen, X., Stoica, I.: Neurocard: one cardinality estimator for all tables. arXiv preprint arXiv:2006.08109 (2020)
22. Zhang, C.B., Jiang, P.T., Hou, Q., Wei, Y., Han, Q., Li, Z., Cheng, M.M.: Delving deep into label smoothing. IEEE Transactions on Image Processing **30**, 5984–5996 (2021)
23. Zhao, Y., Cong, G., Shi, J., Miao, C.: Queryformer: A tree transformer model for query plan representation. Proc. VLDB Endow. **15**(8), 1658–1670 (2022)
24. Zheng, K., Zhao, Y., Lian, D., Zheng, B., Liu, G., Zhou, X.: Reference-based framework for spatio-temporal trajectory compression and query processing. TKDE **32**(11), 2227–2240 (2020)
25. Zhou, L., Chen, X., Zhao, Y., Zheng, K.: Top-k spatio-topic query on social media data. In: DASFAA. pp. 678–693 (2019)