Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries

SONGSONG MO, Nanyang Technological University, Singapore
YILE CHEN, Nanyang Technological University, Singapore
HAO WANG, Nanyang Technological University, Singapore
GAO CONG, Nanyang Technological University, Singapore
ZHIFENG BAO, RMIT University, Australia

With the expansion of modern database services, multi-user access has become a crucial feature in various practical application scenarios, including enterprise applications and e-commerce platforms. However, if multiple users submit queries within a short time frame, it can result in potential issues such as redundant computation and query concurrency. Unfortunately, most existing multi-query optimization methods, which aim to enhance query processing efficiency, have not adequately addressed these two problems, especially in the setting where multiple queries are being executed concurrently. To this end, we propose a novel method named Lemo for the multi-query optimization problem. Specifically, we propose a novel value network to predict latencies of concurrent queries as the foundation model for query plan generation. Furthermore, we introduce a shared buffer manager component to cache the intermediate results of sub-queries. The shared buffer manager applies a novel replacement policy to maintain the cached buffer with the objective of maximizing the opportunity for the reuse of the cached sub-queries. Based on the shared buffer, our proposed value network can incorporate the cached results into cost estimation to further guide Lemo in generating query plans, thus avoiding redundant computation. Lemo has been integrated into PostgreSQL and experiments conducted on real datasets with PostgreSQL show that it outperforms all the baselines in efficiency.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: databases, concurrent query optimization, machine learning

ACM Reference Format:
Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 247 (December 2023), 26 pages. https://doi.org/10.1145/3626734

1 INTRODUCTION

As the number of users and volume of data in database services continue to increase, multi-user access has become increasingly prevalent in real-world application scenarios, such as enterprise and e-commerce operations. Taking the smartphone rebate event on an e-commerce platform like Taobao as an example, many users trigger consecutive queries on smartphone products, each with individually tailored but slightly different filtering criteria, like brand and memory size. Consecutive queries from the same user in a query session or queries submitted by different users frequently contain identical subqueries. This is particularly common when the filtering conditions vary, but

Authors' addresses: Songsong Mo, Nanyang Technological University, Singapore, songsong.mo@ntu.edu.sg; Yile Chen, Nanyang Technological University, Singapore, yile.chen@ntu.edu.sg; Hao Wang, Nanyang Technological University, Singapore, cshaowang@gmail.com; Gao Cong, gaocong@ntu.edu.sg, Nanyang Technological University, Singapore; Zhifeng Bao, RMIT University, Melbourne, Australia, zhifeng.bao@rmit.edu.au.

2836-6573/2023/12-ART247
https://doi.org/10.1145/3626734

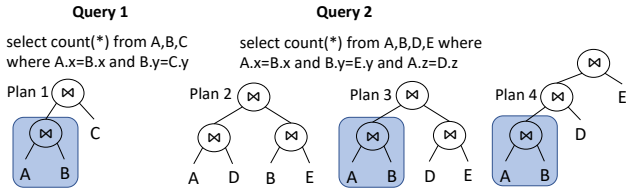


Fig. 1. An Illustration Example of Motivation

the queries target the same type of products. Another case is that many users may search for products within the same time frame (e.g., after office hours), resulting in many concurrent queries. In both cases, identical subqueries can lead to unnecessary resource consumption and potential system slowdowns. Moreover, these concurrent queries may engender resource contention and lock conflicts that can significantly impact the overall efficiency of a database system. In a nutshell, processing queries submitted by multiple users would result in two potential issues: *redundant computation* and *query concurrency*.

Optimizing the processing of multiple queries (or simply *multi-query optimization*) has been studied extensively in recent years. Existing methods can be grouped into two broad categories: batch query optimization [4–6, 12, 19, 21–23, 25, 33] and real-time query optimization [1, 3, 7, 8, 11, 18]. The former focuses on analytical tasks (e.g., sales data analysis), while the latter concentrates on real-time queries for individual users. We follow the real-time query setting, and *tackle the real-time multi-query optimization based on the tuple-at-a-time*[1] executor, which is used in most Database Management Systems (DBMS). Specifically, we aim to generate a query plan for an incoming query in the setting where multiple queries are being executed concurrently from the submissions of multiple users, as opposed to organizing them in batches to process. Existing studies on real-time multi-query optimization mainly adopt two solutions: one is to use traditional optimizers to generate query plans (e.g., Recycler [18] and QPipe [7]). Taking Recycler as an example, it first generates query plans through traditional DBMS and caches the intermediate results of subqueries from previous queries, and then utilizes these results to accelerate the execution of subsequent queries. Although Recycler can reduce redundant computation in such a way, it does not consider the resource contention and lock conflicts of concurrent executions for multiple queries. The other is to employ manually designed rules to guide the generation of query plans (e.g., DataPath [1], CJoin [3] and CACQ [8, 11]). Taking DataPath as an example, it maintains a global plan for concurrent queries. When a new query is submitted, it integrates the new query into the global plan by reusing the common sub-plans and minimizing the sum of cardinalities for each node in the plan. The main difference between the problems addressed by these studies and our problem lies in the execution approach. In these studies, every node in the global plan can be reused by incoming queries as they rely on an operator-at-a-time[2] executor. In contrast, *our problem employs a tuple-at-a-time executor where only nodes that have been proactively cached or materialized are eligible for reuse.*

Figure 1 illustrates the motivation behind real-time multi-query optimization. Specifically, we consider a scenario where a database system is executing plan 1 for query 1 and a new query 2 arrives. In this scenario, a traditional DBMS (e.g., PostgreSQL) would treat each query independently without considering identical subqueries that appeared previously (e.g., A⋈B). Since it does not utilize any context information, it may generate plan 2 for query 2. However, considering identical subqueries, it is highly likely that plan 3 and plan 4 will outperform plan 2 as both plan 3 and plan 4 reduce the redundant computation by reusing the shared identical subqueries from plan 1 (highlighted in blue). Moreover, when the identical subqueries were reused, there also exist different

[1]Each operator calls next on their child to get the next tuple to process.
[2]Each operator materializes their entire output for their parent operator.

query plans such as plan 3 and plan 4 for query 2. Therefore, it is essential to estimate which candidate is optimal for query 2 when being executed concurrently with plan 1.

In this work, we aim to generate an effective plan for a new query for real-time multi-query optimization, by taking the following two into consideration for the first time: *minimizing redundant computation* and *supporting concurrent query execution*. That leads to two major challenges to be resolved.

Challenge 1: Maximizing the reuse of common subqueries, which are shared among multiple queries, is crucial to prevent redundant computation and expedite the execution of query plans. Specifically, two research questions need to be addressed: how to choose a subset of nodes to cache for reuse; how to select a subset of the cached nodes to answer the incoming query.

Challenge 2: Generating an optimal query plan is nontrivial in the setting of concurrent query execution. Its complexity stems from the vast search space involved in generating a query plan. Furthermore, cost estimation of the query plan in a concurrent execution environment introduces an additional layer of complexity, considering the impact of other concurrently executed queries.

To address these two challenges, we propose a **le**arned **m**ulti-query **o**ptimizer (Lemo). Different from existing work that utilizes traditional optimizers to generate query plans or employs hand-crafted rules to boost query plan generation, we propose a transformer-based value network that serves as the foundation for query plan generation. This network aims to predict the best-possible latency of a partial plan[3] in the scenario of concurrent query execution. Here, the best-possible latency is the best-expected runtime that could result from completing this partial plan. To enhance the efficiency of the value network, we saddle the value network with two novel attention mechanisms to reduce its time complexity. To address Challenge 1, we design a shared buffer manager component to cache the intermediate results of sub-queries, as the tuple-at-a-time executor does not preserve intermediate results. Specifically, we propose a workload-driven replacement strategy in the shared buffer manager of Lemo to reduce repeated computation. This replacement strategy considers both the reuse probability of a plan and the benefit of reusing this plan, where the benefit is estimated by the proposed value network. To address Challenge 2, we design a two-stage plan search algorithm in the plan search component of Lemo for plan generation. In the selection stage, we propose to assess the benefit of cached nodes in the shared buffer manager using our value network and then select a subset of these nodes as a sharing node set. In the search stage, starting from the sharing node set, we perform a best-first search guided by our value network to find an optimal query plan for the incoming queries.

In summary, we make the following contributions:

- We propose Lemo, a novel framework for multi-query optimization that considers both minimizing redundant computation and concurrent query execution. To the best of our knowledge, this work is the first to tackle the real-time multi-query optimization problem using a value network for predicting the cost of concurrent queries.
- On the basis of this value network model, we propose a new replacement policy to manage the intermediate results of subqueries to meet memory budget constraints. We design a value network-based plan search algorithm to generate query plans for an incoming query, which leverages the intermediate results of common subqueries to alleviate redundant computations.
- We integrate Lemo into PostgreSQL. Extensive experiments demonstrate that Lemo outperforms baselines markedly.

[3]Throughout, the term "plan" refers to both complete plan and partial plan unless specified otherwise.

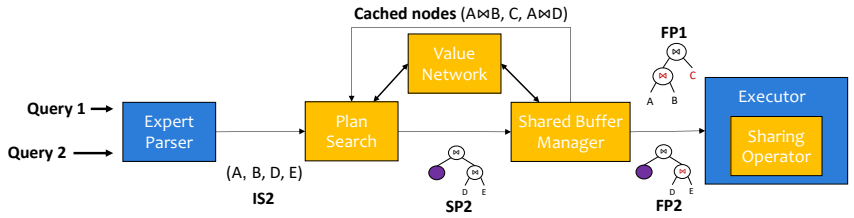


Fig. 2. Workflow for Query Processing

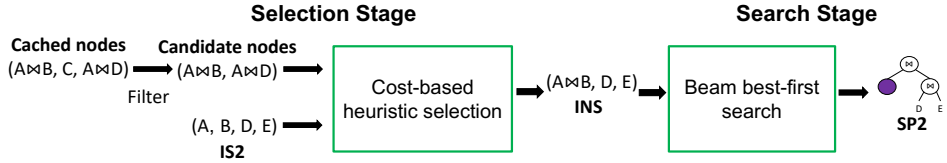


Fig. 3. The Workflow for Plan Search

2 SYSTEM OVERVIEW

Figure 2 presents an overview of Lemo, which includes four newly designed components (highlighted in yellow): *value network*, *plan search*, *shared buffer manager*, and *sharing operator*. The workflow of query processing is also exemplified in Figure 2. Before describing each newly designed component in this overview, we introduce some notational conventions. In our problem, queries are assumed to be submitted consecutively in a streaming manner. Query 1 (Q1) and Query 2 (Q2) come from two different streams, respectively. When Q2 arrives, Lemo has generated a query plan (FP1) for Q1 and is executing this plan. Meanwhile, intermediate results have been stored as cached nodes in the shared buffer of Lemo, including A⋈B, C, and A⋈D, where A⋈B and C are derived from the subqueries of Q1, and A⋈D is collected from the subqueries of previously executed queries. Next, we briefly introduce the essential components in our Lemo. The details of each component will be clear shortly.

Value network. We train a value network model to estimate the best-possible query latency for the plan of an incoming query as well as the query latencies for the concurrent query plans. However, two challenges arise for the latency estimation in concurrent query executions. First, the tree structure of the query plan is a directed acyclic graph due to the addition of sharing nodes (see Section 5.2 for details). In this case, previous models that encode the query plan for a single query such as TreeCNN [17] and TreeLSTM [26] are not applicable to concurrent queries. Second, since the query latencies of concurrent queries could be affected by the correlations among concurrent queries (e.g., global buffer sharing, lock conflict, and resource conflict) [34], it is essential to effectively capture these correlations. GPredictor [34] has four hand-crafted rules to model the correlations among concurrent queries manually and encodes the concurrent queries by GNN [2]. However, such methods with hand-crafted rules cannot capture all kinds of correlations among concurrent queries. Moreover, it is not applicable to our setting of predicting the best possible latency for the partial plans because GPredictor aims to predict the execution time of complete query plans. To tackle these two challenges, inspired by the success of QueryFormer [31] in single query representation, we devise a Transformer-based [27] value network, called MQFormer, for the latency prediction of multiple queries (Section 3). It is worth noting that Queryformer is not applicable to our problem due to the high time complexity it encounters when being extended from a single query to multiple queries.

Plan search. Guided by the value network, the plan search component performs a two-stage search algorithm for plan generation. To better demonstrate the query plan generation process of the learned optimizer, we give a toy example as shown in Figure 3 (from IS2 to SP2). When generating a plan for Q2, Lemo first employs an expert parser (e.g., the parser in PostgreSQL) to generate an

initial state (IS2). Then both IS2 and the cached nodes flow into the plan search component to select a subset of the cached nodes to answer Q2. Specifically, it initially filters the cached nodes using the relations and conditions in IS2 as candidate nodes. The relation refers to a table, and the condition refers to a filtering criterion applied to the data in the table. To this end, we employ a cost-based heuristic algorithm to choose a subset of candidate nodes to be included in the final query plan for Q2 at the selection stage. As shown in Figure 3, it selects a subset (A⋈B) of the candidate nodes to be included in the final query plan for Q2. Subsequently, it combines this subset (A⋈B) with the remaining relations (D, E) in IS2 to form an intermediate state INS (A⋈B, D, E). Next, the plan search component needs to search over the rest space of the query plan, starting from INS until an optimal plan is found. To deal with this issue, we perform a beam best-first search and generate the query plan taking a step forward until it outputs a complete query plan (SP2). As illustrated in Figure 3, guided by the estimated cost that is provided by the model, the best-first search in each round chooses from the set (A⋈B, D, E) to join two tables together. The plan search algorithm will terminate once a complete query plan (SP2) is generated (Section 4.1).

Shared buffer manager. Shared buffer manager aims to maintain a shared buffer where the cached nodes can be reused to reduce repeated computations. To this end, we propose a workload-driven replacement strategy, which considers both the probability of cached nodes being reused (*hit probability*) and their reuse benefits (*hit benefit*). Here, the benefit refers to the time saved by reusing this node. Lemo estimates the benefit using the proposed value network again. In Figure 3, it takes SP2 as the input and updates the shared buffer within the memory budget. The shared buffer manager incorporates all subqueries of SP2 into the shared buffer and executes the replacement strategy to determine which parts of the subquery results of SP2 should be cached and which parts of the current cached results should be removed. Finally, it outputs the query plan (FP2) marked with the subquery results to be written into the shared buffer (highlighted in red) (Section 4.2).

Sharing operator. It is integrated into the executor for writing and reading the shared buffer during the execution. Specifically, we use an expert executor (e.g., the executor of PostgreSQL) with the sharing operator component to run FP2.

In summary, we design a shared buffer manager with a workload-driven replacement strategy to avoid repeated computation (Challenge 1) and a two-stage plan search algorithm for plan generation (Challenge 2). As both the workload-driven replacement strategy and the plan search algorithm build upon the primary value network component to estimate the plan's cost, we next delve into the details of the value network (Section 3). Then, we discuss the design of the plan search (Section 4.1) algorithm and the shared buffer manager (Section 4.2) based on the value network. Finally, we extend a matching-based multi-query optimization algorithm in sharing optimization component (Section 5.2) as a baseline to collect training data for our value network (Section 5).

3 VALUE NETWORK

In this section, we start from a brief description of the Transformer encoder architecture and then elaborate our value network model called MQFormer. As mentioned in Section 2, there are two challenges for the cost estimation of concurrent queries. First, the tree structure of a query plan would be a directed acyclic graph (DAG) as the additional edges are added due to the sharing nodes. Second, it is essential to capture the correlations among concurrent queries, such as global buffer sharing, lock conflicts, and resource conflicts, because the correlations have an influence on the query latencies of concurrent queries. To this end, we develop a Transformer-based value network for multi-query latency prediction.

In detail, to address the above two issues, we propose two novel types of self-attention mechanisms: data flow attention for encoding the DAG structure of concurrent queries, and correlation attention for capturing the correlations among queries. Note that the DAG structure here represents

the flow of data during the execution process. So, we refer to it as the data flow information. Next, we introduce the design of MQFormer and the details of these two novel self-attention mechanisms. And then, in the end, we present the training process of MQFormer.

3.1 Overview of the Transformer encoder

The Transformer encoder [27] is constituted by a stack of identical layers, with each layer consisting of two primary components: a self-attention mechanism and a position-wise fully connected feed-forward network.

Self-attention mechanism. The self-attention mechanism, also known as scaled dot-product attention, calculates the attention score for each node in the input sequence with respect to every other node, allowing it to capture long-range dependencies between nodes irrespective of their positions in the given sentence. In our problem, we encode each node in the query plan into a vector and concatenate all nodes into a sequence to serve as the input vector H (See section 3.2 for more details).

Given a set of queries Q, keys K, and values V, which are derived by projecting the input vector H with learned weight matrices W_Q, W_K, and W_V respectively, the self-attention mechanism can be defined as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

In this equation, d_k is the dimension of the keys, and the division by $\sqrt{d_k}$ is a scaling factor that prevents the dot product from growing too large as the dimensionality increases, which could lead to difficulties with gradient descent.

The softmax function is applied to the dot products of the queries with all keys, ensuring that the attention scores sum to 1 and thus can be interpreted as probabilities. These scores determine how much each node in the input sequence contributes to the final output of the self-attention mechanism for a given node. In our problem, the attention scores represent either the data transfer relationships or the correlations among concurrent query nodes.

Position-wise fully connected feed-forward network. Another essential component in the Transformer architecture is the position-wise feed-forward network (FFN). The FFN, applied identically to each position, consists of two linear transformations with a ReLU activation function in between:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

Here, x represents the input, which is specifically the output derived from the self-attention layer for a particular position in the sequence; W_1, b_1, W_2, and b_2 are learned parameters. The function $\max(0, x)$ represents the ReLU activation function. This position-wise FFN introduces additional non-linear transformation, thereby aiding in capturing complex patterns in the data.

3.2 Model design

The architecture of MQFormer is shown in Figure 4. The model consists of three parts: *feature extraction and encoding*, *multi-query embedding*, and *cost estimation*. We next present the model architecture in a bottom-up manner.

Feature extraction and encoding. The input data fed into the model is a two-tuple: (a set of concurrent queries with their complete query plans, an incoming query with its partial or complete query plan). As shown in Figure 4, we apply two kinds of encoding schemes, namely query-level encoding and plan-level encoding. This approach is adopted because it effectively encodes information from both SQL and plan aspects simultaneously. Similar to [14, 28], query-level encoding is to extract join graph (e.g., A.x = B.x) and predicate information (e.g., A.x<10), and

then transform these features into a 32-dimensional vector via a multi-layer perceptron (MLP) for query information embedding.

Plan-level encoding is to produce the representation of the partial or complete plan(s) for both the incoming query and its concurrent queries that are being executed. There are three types of features to be considered for each node in a query plan: operation type, relation information, and position. For operation information, we aggregate the operations on nodes into seven typical operations, i.e., hash join, merge join, loop join, index scan, seq scan, aggregate, and share. Each type of operation is encoded as a seven-dimensional one-hot vector. For relation information, we use one-hot encoding, where each bit indicates whether the node touches the corresponding table. We use a two-dimensional vector to embed the position of one node. One dimension represents the sequence number of the plan to which the node belongs among concurrently executed queries, and the other dimension represents the node's height within the query. Here, the height of a node refers to the maximum possible distance from the node to a leaf node, with leaf nodes having a height of 0. As illustrated at the bottom of Figure 4, the plan-level encoding of node 8 is $N_8 + P_{31}$. The first seven dimensions represent operator information, dimensions 8-11 correspond to relation information, and the final two dimensions signify position information.

Following our feature extraction and encoding, the representation vector of each node is formulated in the form of [operation types, relations, position, query information] $\in \mathcal{R}^N$, where $N = 7 + |R| + 2 + 32$ and $|R|$ denotes the total number of relations in the workloads.

Multi-query embedding. Taking the representation vectors from the previous stage as input, multi-query embedding aims at capturing the data flow information and the correlations among concurrent queries to generate a single embedding vector for each query. As shown in the middle of Figure 4, multi-query embedding consists of a multi-query encoder and a pooling layer.

As shown in Figure 5, the architecture of our multi-query encoder follows the backbone of Transformer encoder architecture [27]. Specifically, it consists of an attention layer and a position-wise feed-forward network. Instead of using the original full self-attention module, we redesign an attention layer that combines two attention patterns to efficiently capture both data flow information and correlations among concurrent queries (see more details in Section 3.3).

Regarding our pooling layer, it takes the outputs of the multi-query encoder as inputs and summarizes all the nodes from the same query into a vector by a mean-pooling layer.

Cost estimation. Our cost estimator aims to predict the best-possible query latencies for each query plan. We adopt a three-layer MLP for cost estimation. Specifically, the fully connected layer further integrates and condenses the features outputted from the multi-query embedding into a fixed-size output for cost prediction.

Remark. In detail, QueryFormer differs from MQFormer, the existing Transformer-based value network for single query embedding, in two aspects: (1) Query encoding – MQFormer uses two-level (query level and plan level) information, but QueryFormer only uses plan level information; also, the position information used in MQFormer is different from that used in QueryFormer. (2) Attention pattern – QueryFormer adopts full attention pattern, while MQformer adopts two new attention patterns, data flow attention and sparse correlation attention, for efficiency improvement.

3.3 Attention layer

We present the details of the proposed attention layer equipped with two novel attention mechanisms to capture the DAG structure (data flow) of concurrent queries and the correlations among them. Before that, we first briefly review the self-attention mechanism used in the original Transformer [27].

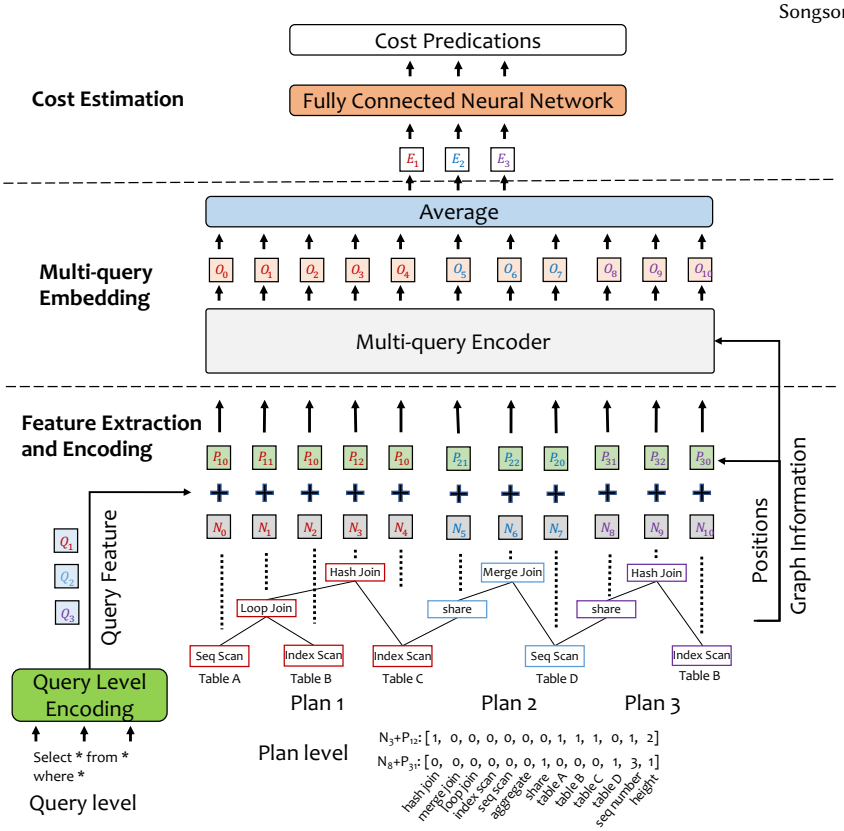


Fig. 4. Value network architecture

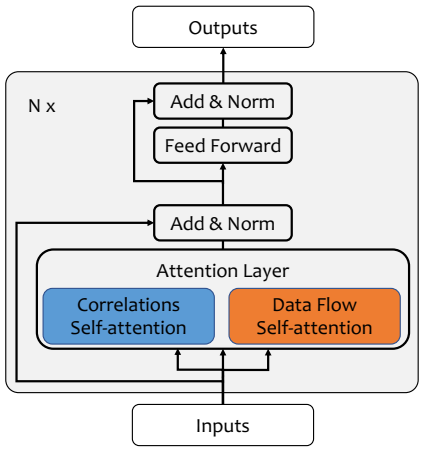
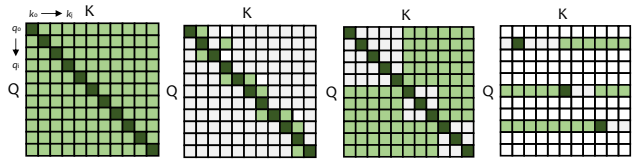


Fig. 5. Multi-query encoder

Self-attention in Transformer. Given the queries[4] $Q \in \mathbb{R}^{L_q \times d_q}$, keys $K \in \mathbb{R}^{L_k \times d_k}$, and values $V \in \mathbb{R}^{L_v \times d_v}$, which are derived from the multi-query encoding with linear projection, respectively. For a query vector q_i and a key vector k_j (i.e., a row in the query and the key matrices), we have a

[4]Note that the query here only refers to the term in self-attention and not to the user query in a database.



(a) FA (b) DA (c) CA (d) SCA

Fig. 6. Attention patterns for determining which QK (Query-Key) values need to be calculated.

scalar attention score as follows:

$$a_{ij} = \text{softmax}\left(\frac{q_i k_j^\top}{\sqrt{d_k}}\right) = \frac{\exp\left(q_i k_j^\top\right)}{\sqrt{d_k}\sum_{r=0}^{L_k}\exp\left(q_i k_r^\top\right)} \tag{3}$$

where L_k denotes the total number of key vectors. This full self-attention mechanism, depicted in Figure 6a, exhibits quadratic time and memory complexity, denoted as $O(n^2)$. Here, n signifies the length of the input sequence. This is attributable to each input node attending to all other nodes within the sequence.

We denote that it is not optimal to apply the full attention mechanism to our problem for two reasons:

- It is not necessary to compute the attention score for all pairs of nodes. For example, data flow does not exist between all the nodes (i.e., no data flow between unreachable nodes in the DAG), and correlations among concurrent queries do not exist for nodes from the same query.
- Our system runs in the setting of concurrent queries, which calls for high efficiency, and thus the full attention mechanism is not adequate to satisfy this demand.

To this end, we propose two types of attention mechanisms: *data flow attention* and *correlation attention*.

Data flow attention. To capture the structure information in the data flow, we extract data flow attention scores from the DAG of concurrent query plans. For a node pair n_i and n_j, since the edges in a DAG represent the direction of data flow, we only compute the attention score if there is an edge between nodes n_j and n_i in the DAG. For example, the DAG of query plans in Figure 4 corresponds to the data flow attention scores as shown in Figure 6b. Since there is an edge between nodes n_0 and n_1 in the DAG, only the attention score $q_0 k_1^\top$ between node n_0 and n_1 needs to compute, and thus the grid (1,0) is marked in green color in Figure 6b.

It is worth noting that the data flow attention is different from the tree-bias attention in QueryFormer. QueryFormer first computes the full attention score of all nodes. Then tree-bias attention uses a learnable scalar bias for all reachable nodes with different distances. Finally, QueryFormer adds the tree-bias to the full attention score. Differently, the data flow attention only computes the attention score if there is an edge between two nodes.

Correlation attention. Correlation attention aims to capture the degree of correlations among concurrent queries. Similar to our data flow attention, since the correlation patterns (e.g., resource contention and lock conflicts) only exist between different queries, we only calculate the attention scores of a node with other nodes which are not in the same query plan. For example, in Figure 6c, we illustrate the correlation attention scores for the query plans in Figure 4. However, as shown in Table 1, it still suffers from high time complexity. Therefore, we further speed up the calculation of correlation attention using a sparse attention strategy [32].

The sparsity metric measure of q_i is formulated as follows:

$$M(q_i, K) = \ln \sum_{j=1}^{L_k} e^{\frac{q_i k_j^\top}{\sqrt{d_k}}} - \frac{1}{L_k} \sum_{j=1}^{L_k} \frac{q_i k_j^\top}{\sqrt{d_k}} \tag{4}$$

This formula evaluates the information gained for each query by employing the Kullback-Leibler (KL) divergence between the attention distribution of q_i across all keys and a uniform distribution. Specifically, the KL divergence quantifies how much the attention distribution of a query deviates from a uniform distribution. A uniform distribution represents a state of maximum uncertainty, where every key is equally likely to be attended. When the attention distribution significantly deviates from this state (i.e., certain keys are attended to more than others), the KL divergence is large, implying that the query q_i is highly informative. In other words, the larger the KL divergence from the uniform distribution, the more specific or selective the information the query q_i contains. We select top-u query vectors with the highest sparsity measures and set all the others to be zeros in queries Q as a sparse matrix $\overline{Q}$, where $u = \ln(L_q)$. Given the sparse matrix $\overline{Q}$, we formulate our sparse correlation attention as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{\overline{Q}K^T}{\sqrt{d_k}}\right)V \tag{5}$$

As shown in Figure 6d, we obtain a sparser matrix of correlation attention scores compared to Figure 6c. We can see that sparse correlation attention has low time complexity.

Complexity analysis. We use FA, DA, CA, and SCA to denote full attention, data flow attention, correlation attention, and sparse correlation attention, respectively. Given a DAG graph G(V, E) generated from x concurrent query plans, where $|V| = n = \sum_{i=1}^{x} n_i$ represents the number of nodes, $m = |E|$ represents the number of edges, each node is encoded into a d-dimensional vector, and n_i denotes the node number of i-th query plan. The input sequence length is $L_k = L_q = n$ in MQFormer and each node embedding is a $d_k = d_q = d$ dimensional vector. Then we summarize the time complexity and space complexity of each type of attention mechanism in Table 1.

Table 1. Complexity analysis ($SSD = n^2 - \sum_{i=1}^{x} n_i^2$ represents the sum of squared differences)

Attention Pattern	Time Complexity	Space Complexity
FA	$O(n^2 \cdot d)$	$O(n^2 + nd)$
DA	$O((n+m) \cdot d)$	$O(n^2 + nd)$
CA	$O((n + SSD) \cdot d)$	$O(n^2 + nd)$
SCA	$O(\ln(n)(1 + \frac{SSD}{n}) \cdot d)$	$O(n^2 + nd)$

Attention layer design. In each attention layer, we employ two sets of projections (Q_f, K_f, V_f and Q_s, K_s, V_s) generated from the multi-query encoding using linear projection. These sets are used to calculate the attention scores for data flow attention and sparse correlation attention, respectively. Ultimately, the final attention scores are compiled by aggregating these two types of attention scores as dictated by Equation 6. In this case, the time complexity and space complexity of our attention layer is $O((n + m + \ln(n)(1 + \frac{SSD}{n})) \cdot d)$, which approximates to $O((m + n\ln(n)) \cdot d)$, and $O(2n^2 + nd)$.

$$\text{Attention} = \text{softmax}\left(\frac{Q_f {K_f}^T}{\sqrt{d_k}}\right)V_k + \text{softmax}\left(\frac{Q_s {K_s}^T}{\sqrt{d_k}}\right)V_s \tag{6}$$

3.4 Model Training

In this section, we present the loss function of MQFormer, model pre-training, and model fine-tuning.

Loss function. Since the target of Lemo is to minimize the overall execution time, we train MQFormer by minimizing the following loss function: $Loss = \frac{1}{m}\sum_{i=1}^{m}(\hat{l}_i - l_i)^2$, where l_i is the ground-truth latency of plan i in training data and $\hat{l}_i$ is the predicted i-th latency by MQFormer(CoQ, CoP). Here, CoQ and CoP denote the m concurrent execution queries and plans, respectively.

Model pre-training. To facilitate the effectiveness of our model, we initially pre-train MQFormer using the training data (denoted as *Experience*) that are gathered from a matching-based multi-query optimization technique (see *Training Data Collection* in Section 5) in an offline manner. In a nutshell, we collect m concurrent queries and plans with their latencies $\{(q_1, p_1, l_1), (q_2, p_2, l_2), \cdots, (q_m, p_m, l_m)\}$ as training data, where m is number of concurrent queries. To enable the model to predict the best possible latency of a subquery plan, for any a record $\{(q_1, p_1, l_1), (q_2, p_2, l_2), \cdots, (q_m, p_m, l_m)\}$ in *Experience*, we first enumerate all subqueries of p_m as $\{p_{m1}, p_{m2}, \cdots, p_{mi}\}$. Then, we set the best possible latency l_{mi} of p_{mi} as $min(l_j | p_{mi} \subset p_j \wedge p_j \subset Experience)$. Lastly, for each (p_{mi}, l_{mi}) pair, we generate the record $\{(q_1, p_1, l_1), (q_2, p_2, l_2), \cdots, (q_m, p_{mi}, l_{mi})\}$ as the final training data.

Model fine-tuning. In the online query processing stage, all the training queries consist of m streams. The queries in each stream are from a user/customer and each user submits a new query only after the previous query. Lemo generates a query plan for each query based on the submission order[5] and executes them concurrently. We gather the plans and their latencies as a training data sample if these plans are executed concurrently (i.e., their execution times overlap) and then add this data sample to the Experience. We fine-tune the MQFormer using a batch of newly acquired *Experience* after processing every 100 queries.

Model re-training. In our existing implementation, we assume a static system state, which encompasses database configuration, optimizer implementation, and data distribution. If any of these factors undergo slow or infrequent changes, we need to periodically gather new execution data and concentrate on retraining with data reflective of the new system state.

4 COMPONENT DESIGN BASED ON MQFORMER

In this section, we demonstrate how to utilize the proposed value network model MQFormer for query plan generation. Specifically, we design (1) a novel plan search component to generate optimal query plans, and (2) a shared buffer manager component to maximize the reuse of common sub-query.

4.1 Plan Search

Based on the learned value network, we develop a two-stage plan search algorithm for Lemo to generate an optimal query plan for a recently submitted query. The process of our plan search algorithm is summarized in Algorithm 1. In the selection stage, guided by the value network, it selects a subset of the cached nodes as the sharing nodes in the shared buffer manager to optimize an incoming query. In the search stage, based on the selected set, it performs a best-first search to find an optimal query plan for the incoming query. We next introduce the detailed processes in each stage.

4.1.1 The selection stage. In this stage, we formulate the sharing node selection problem as an Integer Linear Programming (ILP) problem and employ a greedy-based algorithm to solve the ILP problem. While it is a standard routine used in many similar problems [16, 29], the essence of a

[5]In the case of the queries from different users submitted simultaneously, the queries will be processed in a random sequence.

Algorithm 1: Plan Search(q, s, b)

Input: q: an incoming query, s: a link list of cached sharing nodes, b: the beam width
Output: p: a query plan

State ← $Parser(q)$; Candidate set S ← filter(State, s);
Compute $score(n_j) = v'_j/w'_j$ for each node in S;
Compute conflict edge E according to S;
Rank nodes in S by the score; $c \leftarrow |State|$; $c' \leftarrow 0$; $R \leftarrow \emptyset$;
for *each* $n_j \in S$ **do**
    if $c' + w(n_j) \leq c$ *and conflict_check(c,R,E)* **then**
        $R \leftarrow R \cup n_j$; State ← State - {leaf nodes of n_j};
newState ← R∪State;
Initialize priority queue *queue*; *queue*.add(*newState*, 0);
while *queue is not null* **do**
    curState ← *queue*.get();
    if $|curState| = 1$ **then**
        $p = curState$; break;
    Enumerate all next states of *curState* as *neighbors*;
    Evaluate all states in *neighbors* by *MQFormer* as *costs*;
    for *each nextState* ∈ *neighbors, cost* ∈ *costs* **do**
        queue.add(*nextState*, *cost*);
    Keep top-b states in *queue*.
return p;

greedy algorithm lies in its decision-making process. This process involves choosing the option that seems the best at the current step with the goal of finding a local or global optimum. The novelty of our work lies in employing MQFormer as a cost estimator to direct this decision-making process.

Problem fomulation. Given a query q and a candidate sharing node set S, we select the optimal sharing node subset $S_q \subseteq S$ under the constraint of common subqueries. Let m be the number of nodes cached in $S = \{n_1, n_2, ..., n_m\}$. For a given node $n_i \in S$, we denote the corresponding weight and value as $w'_i := |n_i.relations|$ and $v'_i := be_i$, respectively. Here, $n_i.relations$ indicates the relations on node n_i. be_i is estimated by our MQFormer as follows:

$$be_i = MQFormer(CoQ \cup q, CoP \cup n_i) - MQFormer(CoQ \cup q, CoP \cup \overline{n_i}) \tag{7}$$

where CoQ and CoP denote the concurrent execution queries and plans, respectively, q is an incoming query, and n_i is the subplan of the incoming query plan. We calculate the benefit be_i of reusing node i by taking the difference between the predicted latency when node i is not reused (n_i) and the predicted latency when node i is reused ($\overline{n_i}$).

In the finalized query plan, each relation appears only once on the leaf nodes. Consequently, conflicts arise among cached nodes when they contain the same relation and the aggregate weight of all chosen nodes is less than $c = |q.relations|$, where $q.relations$ represents the relations in the incoming query q. We then add pair (i, j) into the conflict edge set E if $n_i.relations \cap n_j.relations \neq \emptyset$.

In this case, the sharing node selection problem can be formulated as an ILP problem as follows:

$$\max \sum_{j=1}^{n} v'_j x_j \quad \text{s.t.} \sum_{j=1}^{n} w'_j x_j \leq c \quad \text{and} \quad x_i + x_j \leq 1, \forall (i,j) \in E. \tag{8}$$

where the first constraint ensures that the relations of q are only used once in the complete query plan, and the second constraint controls the conflict between the sharing nodes. Formally, we resolve $X(x_j)$ for using the sharing node n_j to optimize query q, where X is a binary decision variable vector in this ILP problem and $x_j \in \{0,1\},\ j = 1,\ldots,m$.

However, ILP problem is an NP-hard problem. To demonstrate this, we perform a reduction from the knapsack problem with conflict graphs (KCG) [20], which is known to be NP-hard. In a KCG, we are given a set of items, each with a weight and value, a knapsack with a capacity, and a conflict graph that has nodes representing the items and edges connecting pairs of items that are in conflict. We respectively map the items in the KCG to cached nodes, the knapsack capacity to the weight constraint c, and the edges in the conflict graph of the KCG to the conflict edge set E. After that, the objective remains the same: to maximize the total value of the items in the knapsack without exceeding its weight capacity and without including conflicting items.

Since the KCG problem is NP-hard, our sharing node selection problem is also NP-hard. A heuristic or approximation algorithm is promising to tackle the problem. Therefore, we propose a greedy-based selection approach to solve this NP-hard problem.

Greedy-based selection. Based on v'_j and w'_j for each node j, we devise a greedy-based selection for this sharing node selection problem, as shown in lines 1-8 of Algorithm 1. It gets a leaf node set as the original state by an expert parser and selects the candidate set S according to the definition of S in line 1. Then, it computes $score(n_j) = v'_j / w'_j$ for each node in S, adds the conflict edge according to S, and ranks the nodes with their scores in line 2. Next, it selects the node n_j into set R to optimize the incoming query if there is enough space and no conflicts between n_j and the nodes in R (lines 6-8). Lastly, it generates a new state by combining the reuse set R and the part of the original state that was not included in the leaf nodes of R (line 9).

Recall the toy example in Figure 3, during the selection stage, let us assume that the plan search algorithm initially calculates the scores of nodes A⋈B and A⋈D as 5 and 3, respectively. It would initially select A⋈B (If both nodes A⋈B and A⋈D share an identical score of 5, it would make a random selection between them). However, since both A⋈B and A⋈D contain A, this creates a conflict. Consequently, {A⋈B} is chosen as the final reuse set.

4.1.2 The search stage. In the second stage, starting from the subset of cached nodes, we conduct a best-first beam search guided by the value network for the remaining parts of the query plan. This approach is also employed in generating single query plans [14, 28]. The key novelty of our MQFormer is that our MQFormer is trained to consider both the shared nodes and the concurrent queries, and hence it makes the generated plans more suitable for concurrent execution.

As shown in lines 9-18 of Algorithm 1, it keeps all states in a priority queue sorted by their predicted costs. For each step, it first pops the best state in line 12. From this state, it enumerates all next states by adding a specific scan or joining two sub-plans as *neighbors* and evaluates all states in *neighbors* by MQFormer to get their predicted costs (lines 15-16). Then it adds all states with their predicted costs into the priority queue and keeps only top-b states for the next step (lines 17-19). This algorithm keeps running until a complete plan is generated (lines 13-14).

Building upon the toy example in Figure 3, during the search stage, and based on the reuse set {A⋈B}, the plan search algorithm needs to concatenate A⋈B with the remaining leaf nodes D and E. In each round, MQFormer is used to evaluate which two nodes should be joined until a complete query plan is formed.

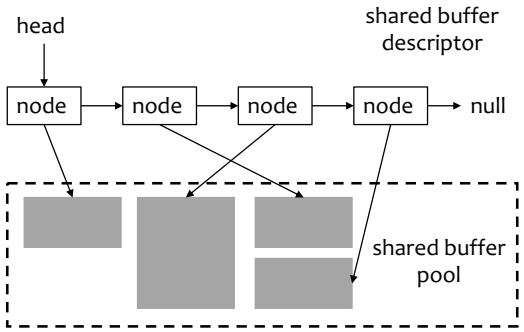


Fig. 7. Data Structure of Shared Buffer Manager

4.2 Shared Buffer Manager

In this subsection, we present the data structure design of the shared buffer manager component and introduce our proposed replacement policy regarding how to update the cache when a new query plan is added.

Data structure design. As shown in Figure 7, the shared buffer manager component contains two data structures: a shared buffer descriptor and a shared buffer pool. The shared buffer descriptor is a linked list of the cached nodes of partial intermediate results of the past workload. For each node in this linked list, it records the node id, relations, conditions, reference count, cached flag, size, gain, usage count with timestamps, and the plan tree rooted at this node. The node id is used as the key for linking the results in the shared buffer pool. Relations and conditions are used for equality checking. Reference count records how many executing queries use this node, and a query will not be replaced unless the reference count is 0. The cached flag is a boolean variable that marks whether this node links a cache block in the share buffer pool. The size, gain, and usage count with timestamps of this node is used as parameters in our replacement policy (more details in the following paragraph). The share buffer pool is implemented by the shared memory, where multiple processes can access through a given key (node id).

Replacement policy. Managing the shared buffer pool under a size budget is essentially a knapsack problem (maximizing the benefit with a size budget restriction), which is a typical NP-hard problem [15]. To this end, we propose a greedy-based heuristic algorithm as the shared buffer replacement policy.

Since the buffer size budget can be treated as the capacity of the knapsack, the challenge is how to estimate the weight and value of each node before execution. To tackle this, we employ an expert cardinality estimator, such as the one in PostgreSQL, to estimate the weight w_i of node n_i. After the execution of the node, we update w_i with its actual memory size. For estimating the value v_i of a node n_i, we consider both the hit probability and hit benefit.

The hit probability is the probability that node n_i will be shared by the subsequent queries (pr_i). Formally, we employ the least recently and frequently used policies (i.e., LFRU model [9]) to estimate this probability formulated as: $pr_i = \sum_{t \in T_i} \left(\frac{1}{2}\right)^{(t_{base}-t)\lambda}$, where T_i is the timestamps of the usage of node n, t_{base} is the current timestamp, and λ is a parameter to trade off the least recently used policy and the least frequently used policy. It is worth noting that we only need to have the relative size of pr_i, and thus we don't normalize it to [0,1].

The hit benefit denotes the benefit when the subsequent queries reuse node n_i. We also estimate the benefit of node n_i (be_i) according to Equation 7. Stemming from the hit probability and the hit benefit, we finally define the value of node n_i in the shared buffer manager as $v_i = pr_i * be_i$.

Algorithm 2: Replacement Policy(p, s, b)

Input: p: a tree of incoming query plan, s: a link list of cached share nodes, b: a budget of shared memory
Output: p: the query plan that marks which nodes need to be cached

```
1  AddNode(p, s);
2  Compute the greedy score for n_i ∈ s by v_i/w_i;
3  Rank link list s first by the reference count, then by the score;
4  s' ← s; b' ← 0; C ← ∅;
5  while s' is not null do
6      if b' + s'.size ≤ b then
7          b' ← b' + s'.size;
8          if s'.cached is False then
9              Create cache block for s' the in share buffer pool;
10             C ← C ∪ s'.id; s'.cached ← True;
11         s' ← s'.next;
12     else
13         if s'.cached then
14             Delete cache block for s' the in share buffer pool;
15         Delete node s';
16 TagNode(p, C);
17 return p;
18 ---------------------------------------------------------------
19 Function AddNode(p, s):
20     if p is null then
21         return;
22     Create a link node n_i and copy information from p;
23     Estimate the w_i and be_i of n_i; Insert n_i into s;
24     AddNode(p.left, s); AddNode(p.right, s);
25 ---------------------------------------------------------------
26 Function TagNode(p, C):
27     if p is null or C is null then
28         return;
29     if p.id ∈C then
30         p.toCache ← True; C ← C\p.id;
31     TagNode(p.left, C); TagNode(p.right, C);
```

Remark. The primary distinction between our replacement policy and the classical cache replacement strategies (FIFO, LRU, LFU, and LRFU [9]) is that we consider both hit probability and hit benefit, whereas traditional strategies focus solely on hit probability.

Based on the weight and value of the node in the shared buffer manager, the replacement policy for managing the shared buffer pool under a size budget is shown in Algorithm 2. It first adds all the nodes in the incoming query plan p into the shared buffer descriptor s and computes the greedy score (value/weight) for all the nodes in s (lines 1-2). Then it ranks these nodes firstly by their reference counts, and then by their scores. This is because we cannot remove the node that is shared by the executing queries (line 3). Next, it checks the node in s one by one (lines 5-15). For each node s', it adds s' to the share buffer pool if there is enough space to accommodate the weight

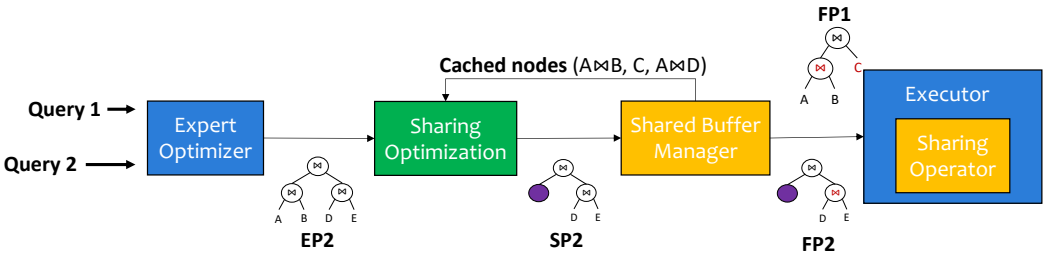


Fig. 8. Workflow for Collecting the Training Data

of s'. If s' does not link a cache block in the shared buffer pool, it creates a cache block for s' and collects the node id into set C. Otherwise, it removes s' and deletes the cache block for s' if exists. Last, we mark the boolean variable *toCache* of the nodes in an incoming query plan p as true if the node id is in C (line 16). This means we need to cache the results of this node into the shared buffer pool during execution.

5 TRAINING DATA COLLECTION

In this section, we describe how we collect the training data for the value network pre-training. Specifically, we augment a matching-based multi-query optimization method, which serves as the core technique in most existing studies on the sharing optimization problem for multi-query optimization. We adapt it to our system as a baseline for training data collection. Next, we present the details of the workflow of collecting the training data and the matching-based multi-query optimization method.

5.1 The Workflow of Training Data Collection

As illustrated in Figure 8, we extend a matching-based multi-query optimization algorithm in our system for collecting pre-training data for the value network. This idea has been widely used in the sharing optimization problem in existing multi-query optimization studies [18] and is implemented in our sharing optimization component. Subsequently, we demonstrate the workflow for pre-training data collection by utilizing the sharing optimization, shared buffer manager, and sharing operator components, as presented through the example in Section 2.

Sharing optimization. Sharing optimization is responsible for identifying common subquery plans between the incoming query plan and the cached nodes in the shared buffer manager (introduced later). It rewrites sub-plans of the incoming query as sharing nodes, allowing the reuse of intermediate results cached in the shared buffer manager. In the example of Q2, we first utilize the expert optimizer (e.g., optimizer in PostgreSQL) to generate an initial query plan (EP2). Then the initial query plan and the cached nodes are sent to the sharing optimization component for further processing. Specifically, this component runs a matching-based algorithm to identify common subquery plans between EP2 and the cached nodes and replace the common subquery plans in EP2 with the detected sharing nodes. In our case, since EP2 and the cached nodes have the same subquery plan A⋈B, the subquery plan A⋈B in EP2 would be replaced by the sharing nodes (denoted as a purple node). Finally, this component outputs a modified query plan SP2 (Section 5.2).

Shared buffer manager. Continuing the above example, the shared buffer manager processes SP2 within the memory budget constraint, incorporating its subqueries and executing the replacement strategy to determine which results to cache and remove. It then outputs FP2, marking some nodes (highlighted in red) that need to be cached in the shared buffer.

Sharing operator. We utilize an expert executor (e.g., PostgreSQL's executor) with the sharing operator component to run FP2.

Algorithm 3: MMO(p, s)

Input: p: a tree of incoming query plan, s: a link list of cached share nodes
Output: p: the optimized query plan

```
1  Collect(p); Match(p, s);
2  return p;
3  ---------------------------------------------------------------
4  Function Collect(p):
5      if p is null then
6          return;
7      Collect(p.left); Collect(p.right);
8      if p.type = scan then
9          p.R ← p.relation; p.C ← p.filter_condition;
10     else
11         p.R ← p.left.R ∪ p.right.R; p.C ← p.join_condition;
12         p.C ← p.C ∪ p.left.C ∪ p.right.C;
13 ---------------------------------------------------------------
14 Function Match(p, s):
15     if p is null or s is null then
16         return;
17     s' ← s;
18     while s' is not null do
19         if p.R=s'.R then
20             if p.C=s'.C then
21                 Modify p to share node; Copy the node id of s' to p;
22                 return;
23         s' ← s'.next;
24     Match(p.left, s); Match(p.right, s);
```

5.2 Matching-based Multi-query Optimization

The sharing optimization module aims to identify common subtrees between the incoming query plan and the cached intermediate results in the shared buffer manager and then rewrite the incoming query plan.

To this end, we implement a matching-based multi-query optimization (MMO) algorithm which is a core technique applied in [18], and the pseudo-code of MMO is listed in Algorithm 3. MMO operates in two stages: a collection stage and a matching stage. In the collection stage (lines 4-16: Function **Collect**), it collects information on each node in the query plan in a bottom-up manner, and postorders through the tree of the query plan. For each node, it collects all relations and conditions of the query tree rooted at this node. In the matching stage (lines 17-29: Function **Match**), it checks the equality between the nodes of the query plan and the intermediate results in a top-down manner, and it preorders through the tree of the query plan. Specifically, for each node p, it checks the equality between p and the intermediate results in the share manager (more details in section 4.2) one by one. Once an equal intermediate result s is obtained, it modifies this node as a shared node and replaces the node id of p with that of s.

Time Compelxity. We denote n as the number of nodes in the query plan and m as the number of nodes in the shared buffer. In the collection stage, each node in the query plan is traversed, resulting in a time complexity of $O(n)$. In the matching stage, n sub-plans are compared with m

cached nodes one by one. This could, in the worst-case scenario, lead to a time cost of $O(mn)$. Therefore, the overall time complexity of the MMO is $O(mn)$.

5.3 Collecting the Training data

Based on the aforementioned sharing optimization, shared buffer manager, and sharing operator, we can collect sufficient training data to train the value network MQFormer. To simulate the concurrent execution of multiple queries for a given concurrency level m, we submit the training workload in m query streams. For the m concurrently executing queries, we collect their SQLs (q), plans (p), and latencies (l) as the training data (i.e., *Experience*). Here, plans are generated by MMO and latency is the running time. Formally, each query is represented as a 3-tuple (q_i, p_i, l_i), where i signifies the submission order of the query among the m concurrent queries. If the queries are submitted simultaneously, i represents the order in which the system generates the query plans. Hence each sample of the training data is denoted as $\{(q_1, p_1, l_1), (q_2, p_2, l_2), \cdots, (q_m, p_m, l_m)\}$.

6 ADAPTATION TO OTHER EXECUTORS

Since the proposed Lemo is originally designed for tuple-at-a-time setting, to illustrate the compatibility of Lemo to other forms of DBMS, we present how to extend it to vector-at-a-time and operator-at-a-time DBMS executors.

6.1 Operator-at-a-Time Adaptation

As described in Section 1, in the tuple-at-a-time model, the query operates on one tuple at a time. Differently, in an operator-at-a-time executor, an entire operation is applied to all tuples in a set before moving to the next operator. To this end, we can extend our method to the operator-at-a-time executors in two ways by trading off the hit possibility and the memory usage.

Hit-possibility-first strategy. To improve the hit possibility, we treat all the nodes of concurrently executing queries as candidate nodes. In this way, our method can be directly applied to this scenario. Firstly, we select a subset of nodes from all nodes of concurrently executing queries to cache with the given shared buffer size. Then, when constructing a query plan for a newly arrived query, we first select a portion of cached nodes for reuse, followed by the construction of a complete query plan via heuristic search.

Memory-usage-first strategy. Given that the operator-at-a-time model passes all tuples from one node to the parent node at once, to save memory usage, we can avoid using extra shared memory specifically for caching intermediate nodes of concurrently executing queries. When a new query arrives, we can consider only the unexecuted nodes as the candidate set, and select a subset from them for reuse. Based on this subset, a complete query plan is then constructed using the heuristic search algorithm.

6.2 Vector-at-a-Time Adaptation

The vector-at-a-time executor makes a balance between tuple-at-a-time and operator-at-a-time in the sense that, it operates a group of tuples (a vector) at a time, as opposed to all tuples or one at a time.

Given that a new query can be submitted at any point during the execution of concurrent queries, in order to fully reuse the intermediate data of nodes in concurrent queries, we also need to employ a shared buffer to cache data. Consequently, our method can be directly applied to vector-at-a-time-based DBMS. In a manner similar to our earlier approach, we choose a subset of nodes from all concurrently executing queries for caching according to the constraints of the shared buffer size. Subsequently, when a new query arrives, we initially opt to reuse a portion of the previously cached nodes. This is followed by the use of a heuristic search strategy to finalize the construction of a complete query plan.

7 EXPERIMENTS

7.1 Experimental Setup

Datasets. We use three datasets to evaluate the performance of our proposed framework in the experiments.

- **Join Order Benchmark (JOB)**. It contains 113 queries designed by Leis et al. [10] over the Internet Movie Data Base (IMDB). Each query involves 3-16 complex joins (8 joins on average) for testing the query optimizer. We randomly choose 80% of the queries as the training set and use the other 20% of the queries as the testing set.
- **Join Order Benchmark Extension (JOBE)**. To test out how redundant computations are handled, we extend 113 queries to 873 queries as a new workload by manually modifying the predicates of each raw query. Then we split the training set and test set in the same way as the JOB dataset.
- **TPCH**. TPCH is designed to test the response time of database systems for complex queries. It contains eight tables, and the scale factor is set to 10 in our experiments. We follow the setting in [28] to use 70 queries as a training set and ten queries for the testing.

Obviously, queries with different submission orders have different reuse patterns. For a given concurrency level m, we create m copies of each dataset, and each copy is submitted according to the following order: (1) all the queries are submitted randomly (denoted as JOB-R, JOBE-R, and TPCH-R); (2) the query template is randomly picked, and queries from the same template are arranged as a queue to be submitted in order sequentially (denoted as JOB-T, JOBE-T, and TPCH-T). We use JOB-R as the default dataset unless specified otherwise.

Baselines. We compare Lemo against one mature expert system, PostgreSQL, and implement a matching-based multi-query optimization algorithm (MMO), which is the core technique used in exiting multi-query optimization work [18]. We present our multi-query optimization method in the context of tuple-at-a-time execution, which forms the foundation for widely used systems like MySQL and PostgreSQL. We do not implement a comparison with other multi-query optimization methods [1, 3, 7, 8, 11] that are based on vector-at-a-time or operator-at-a-time executors. Moreover, we also compare Lemo with the recently proposed learned query optimizers, Balsa [28] and Lero [35]. As reported in [28, 35], Balsa and Lero have demonstrated their superiority over Bao [13] and Neo [14], we do not further compare them in the rest of the experiments.

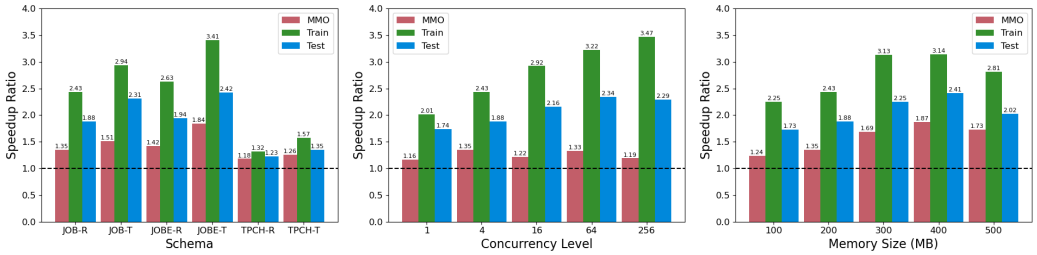
Metrics and default parameter setting. To evaluate the performance of different methods, we use normalized runtimes, which are calculated as the runtimes of all solutions divided by the expert system's runtimes, or the speedup ratio, which is the expert system's runtimes divided by the runtimes of all solutions. The default concurrency level is set to 4, the default memory size is 200MB, the default beam size is 5, and the default value of parameter λ is set to 0.01. For the Transformer backbone, we set the number of heads to 12, attention layers to 8, and attention dropout to 0.1. We train all the tasks using Adam optimizer with a learning rate of 0.001 until convergence.

Setup. All experiments are conducted on a machine with Intel(R) Xeon(R) Gold 6148 CPU, 20 cores, 512GB RAM, and Tesla V100 32GB GPU. PostgreSQL version is 11.2.

7.2 Overall Performance

In this subsection, we present our comprehensive evaluation of the overall performance for Lemo, MMO, and PostgreSQL.

Varying schema type. Figure 9a shows the performance of Lemo on different schema types, and we make several observations: (1) Lemo outperforms MMO and PostgreSQL across all schemas. This is because Lemo first learns from MMO and PostgreSQL and then further learns from real execution through trial and error. (2) For the same dataset, the performance of Lemo is affected



(a) Varying schema type (b) Varying concurrency level (c) Varying size of share memory

Fig. 9. Overall performance

Table 2. Pre-training efficiency

Dataset	Size	Collection Time (min)	Train Time (min)
JOB	3.1K	10.6	12.3
JOBE	21.3K	75.6	25.9
TPCH	2.4K	25.4	6.4

by different submission orders. Specifically, Lemo works better on JOB-T, JOBE-T, and TPCH-T than on JOB-R, JOBE-R, and TPCH-R, respectively. This is because different submission orders would influence the sharing opportunity, and the sharing opportunity between queries stemming from the same template is greater than that between random submissions. (3) The performance of Lemo varies across datasets, since the query complexity of different datasets is different, leading to different sizes of optimization space. Specifically, JOB has a higher average number of joins per query compared to TPCH, making it more complex and resulting in a larger optimization space.

Varying concurrency level. We test Lemo's advantage in handling concurrent queries by varying the concurrency level, and the results are presented in Figure 9b. It can be observed that with the growth of the concurrency level, the performance gap of Lemo over MMO and PostgreSQL increases. This is because MMO and PostgreSQL do not consider other concurrently executed queries when generating query plans.

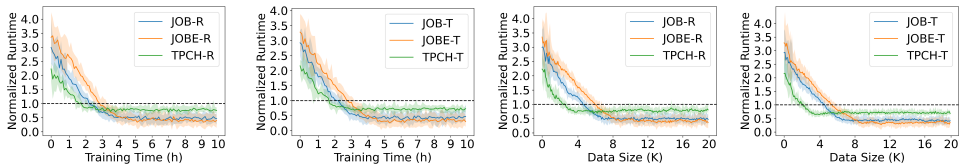
Varying size of shared memory. Figure 9c shows the performance of all the methods by varying the size of the shared memory. When the size of the shared memory increases, the performance of Lemo and MMO first increases and then decreases. This is because when the size of the shared memory increases from 100MB to 400MB, the cached intermediate results also increase, which improves the sharing opportunities. However, when the size of the shared memory further increases, the time of writing more intermediate results into the cache dominates the total cost, and it also takes more time to select the subset of the cached intermediate results for optimizing the incoming query.

7.3 Learning Efficiency

In this subsection, we evaluate the learning efficiency of Lemo by illustrating how long it takes to surpass the expert performance and reach its peak performance.

Pre-training efficiency. In the pre-training phase, we collect training data from MMO for different concurrency level. Taking the concurrency level of 4 as an example, Table 2 displays the time required during the pre-training phase. For all three datasets, both data collection and training time can be completed within several dozen minutes.

Wall-clock efficiency. Figure 10a displays the wall-clock efficiency of Lemo during the execution stage. After pre-training, Lemo initially performs about 3 times slower than the experts. However, with just a few hours of learning, Lemo matches the performance of PostgreSQL. Lemo converges after approximately 2 hours in TPCH, 4 hours in JOB, and 5 hours in JOBE. This is due to the nature



(a) Time efficiency (b) Data efficiency

Fig. 10. Learning efficiency

Table 3. The planning time and execution time (s)

	Planning			Execution			Total
PG	2.84			263.74			266.58
MMO	PT	MT	SBMT	RSBT	WSBT	Other	192.41
	2.93	8.84	0.7	3.01	8.69	168.24	
Lemo	PST	AT	SBMT	RSBT	WSBT	Other	126.65
	2.54	12.23	0.62	4.82	9.71	96.73	

of TPCH queries and their smaller search space, while JOBE queries are more complex and have a larger search space.

Data efficiency. Figure 10b showcases the data efficiency curves. Lemo requires a few thousand training samples to reach peak performance (6.3K for JOB, 9.1K for JOBE, and 3.8K for TPCH). The number of query plans needed increases for datasets as the complexity of the dataset grows.

7.4 Overhead Analysis

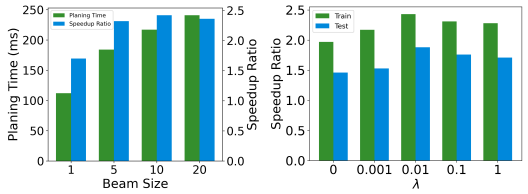
In this subsection, we discuss the impact of the extra plan search time and the buffer maintenance time brought by Lemo on the latency. Table 3 shows the total running time of each part in PostgreSQL (PG), MMO, and Lemo in finishing JOB-R workload.

Unlike PG, MMO and Lemo introduce additional planning time and shared buffer maintenance time. Specifically, the planning time within MMO encapsulates the time required for invoking PG to generate a query plan (PT), the time taken to match subqueries with previously cached intermediate results (MT), along with the shared buffer maintenance time for earmarking the release and introduction of cache nodes (SBMT). Similarly, Lemo planning time consists of plan search time (PST), attention computation time (AT), and SBMT. During execution, both MMO and Lemo account for the time needed to read from the shared buffer (RSBT), the time spent writing to the shared buffer (WSBT), along with the time expended on other execution tasks (Other).

Compared with PostgreSQL's 2.84s, MMO and Lemo respectively utilize 2.93 + 8.84 + 0.7 = 12.46s and 2.54 + 12.23 + 0.62 = 15.39s in planning, amounting to 4.3 times and 5.4 times longer than PostgreSQL's planning duration. However, the planning time for MMO and Lemo only constitutes 4.6% and 5.7% of PostgreSQL's total latency, respectively. In the execution phase, the new overhead introduced is the reading and writing to the shared buffer. MMO and Lemo respectively spent 3.01 + 8.69 = 11.7s and 4.82 + 9.71 = 14.53s on these operations. However, the results reveal that by accessing the intermediate results in the shared buffer, the overall execution time can be effectively reduced.

7.5 Parameter Sensitivity Analysis

In this subsection, we evaluate the sensitivity of different parameters on Lemo, including the beam size, which is used in the second stage of the plan search, and parameter λ which controls the relative importance of the least recently used policy and the least frequently used policy.



(a) Beam size (b) λ

Fig. 11. Impacts of parameters on the performance

Table 4. Results of ablation study

Methods	FA	DA	CA	SCA	DA + CA	DA + SCA
Train	1.82	2.13	1.25	1.41	2.27	2.43
Test	1.45	1.61	1.13	1.22	1.81	1.88

The impact of beam size. The impact of beam size on the per-query planning time and the speedup ratio is reported in Figure 11a. The beam size has a negative impact on the per-query planning time. This is because the beam size directly affects the number of states which are kept in the priority queue of plan search. As the beam size becomes larger, Lemo needs to process a larger number of candidate states, thus resulting in an extended planning time. On the other hand, as the beam size increases, Lemo is able to find better query plans due to its larger search space. We set the beam size to 5 as the default setting because it achieves a high speedup ratio with a reasonable planning time.

The impact of parameter λ. As λ increases, the performance of Lemo increases first and then decreases, as reported in Figure 11b. This shows that when estimating the probability of a cache node being reused, it is more effective to consider both aspects of the least recently used and least frequently used than either aspect alone. We set $\lambda = 0.01$ as the default setting due to its best performance.

7.6 Ablation Study

We conduct an ablation study by using different attention patterns for Lemo to demonstrate their respective contributions to the performance. Specifically, we compare the following variants of Lemo: (1) FA: only the full attention is used; (2) DA: only the data flow attention is used; (3) CA: only the correlations attention is used; (4) SCA: only the sparse correlations attention is used; (5) DA+CA: both the data flow attention and the correlations attention are used; (6) DA+SCA: both the data flow attention and the sparse correlations attention are used.

The speedup ratios of each variant are summarised in Table 4. From Table 4, we make several observations: (1) DA+SCA outperforms other attention strategies, which demonstrates the effectiveness of these two attention patterns. (2) DA outperforms both SCA and CA, indicating that the data flow information has a more significant impact on execution time than the correlations between different queries. This also suggests that the structure of a query plan (i.e., data flow information) is a primary factor affecting the execution time. (3) The performance of SCA is superior to CA, demonstrating the effectiveness of the sparse attention mechanism.

7.7 Comparison with Learned Query Optimizer

In this subsection, we compare Lemo with the existing learned query optimizers, Balsa and Lero, which are designed for single query optimization. All results shown in this subsection are speedup ratios.

Comparison on different schemas. We compare Lemo with Balsa and Lero on JOB-R and JOB-T in the context of non-concurrent execution scenarios. That is to say, we set the concurrency level

Table 5. Comparison on different schemas

Schema	JOB-R		JOB-T	
	Train	Test	Train	Test
Lemo	2.01	1.74	2.94	2.12
Balsa	2.13	1.72	1.93	1.58
Lero	2.76	1.88	2.61	1.62

Table 6. Comparison on different concurrency levels

Concurrency levels	1		4		64	
	Train	Test	Train	Test	Train	Test
Lemo	2.01	1.74	2.43	1.88	3.22	2.24
Balsa	2.13	1.72	1.89	1.52	1.53	1.33
Lero	2.76	1.88	2.52	1.73	2.12	1.46

to 1, which is the experimental configuration used in both Balsa and Lero. The motivation for choosing JOB-R and JOB-T is that they are representative of two common situations where the same subquery occurs less frequently (JOB-R) or more frequently (JOB-T). From the results in Table 5, we make two observations: (1) Lero performs the best on JOB-R while Lemo excels on JOB-T. (2) Lemo would be more influenced by schema variations than Lero and Balsa. This is attributable to the fact that different schemas have different probabilities of containing the same subquery. The higher the probability of containing the same subquery is, the better Lemo performs.

Comparison on different concurrency levels. We further evaluate the performance on different concurrency levels on JOB-R, and Table 6 shows the evaluation results. From the results, we can see that as the concurrency level increases, Lemo outperforms Lero and Balsa more significantly. This is because Lemo considers the concurrent execution when generating the query plan, while Lero and Balsa could not.

8 RELATED WORK

There are two closely related research topics: *multi-query optimization* and *learned query optimizer*.

Multi-query optimization. Multi-query optimization has received considerable attention. Existing methods can be categorized into two types based on their application scenarios: *batch query* [4–6, 12, 16, 19, 21–25, 33] and *real-time query* [1, 3, 7, 8, 11, 18, 29]. In the batch query scenario, multiple queries are submitted as a batch. The optimization process for batch queries aims to find the best global plan for the entire batch by seeking opportunities for the sharing among queries and maximizing the reuse of their results. The earlier methods of multi-query optimization [19, 22, 23] detect common subexpressions within query execution plans during the optimization process to produce globally optimal plans. Shared-workload Optimizers (e.g., SWO [6]) generate global plans by leveraging shared operators. SharedDB [5] uses a batched execution model to optimize query sharing. MQJoin [12] further improves the batched model by incorporating a high-throughput join. To execute global plans efficiently, several shared-work systems utilize materialized views [21, 33], caching [16, 24] and pipelining [4]. RouLette [25] scales multi-query execution by utilizing reinforcement learning. These methods only exploit sharing possibilities within a single query batch.

For the real-time query scenario, multiple queries are submitted as a sequence and then executed concurrently. Most of the existing methods [1, 3, 7, 8, 11] focus on optimizing concurrently executing queries. Specifically, QPipe [7] and DataPath [1] identify sharing opportunities at the query level, while QPipe concerns on sharing common sub-plans and DataPath extends a global query plan

for incoming queries at minimally additional cost. CJOIN [3] and CACQ [8, 11] detect sharing opportunities at the operator level, and meanwhile, they reorder operators at runtime. Therefore, their sharing opportunities are only limited to concurrent queries. In addition, to exploit more sharing possibilities, the cache of intermediate results from past workloads has been investigated in [18, 29]. These intermediate results are later used for optimizing the query plans for subsequent queries.

Remark: Our problem differs from other real-time multi-query optimization problems in at least one of two main aspects: first, since we are based on a tuple-at-a-time executor, we need to cache the intermediate results ourselves. Second, when generating query plans, we take into account the concurrent execution of other concurrent queries.

Learned query optimizer. Learned query optimizers are also related to our research. The learned query optimizers mainly adopt two architectures: one involves utilizing cost estimation models combined with model-based query plan generation, and the other relies on traditional optimizers to generate query plan candidates for selection using a cost estimation model. Neo [14] and Balsa [28] adopt the first architecture. Neo initializes its cost estimation model using traditional optimizers and continuously learns from incoming queries. In contrast, Balsa begins by learning the cost estimation model from a simple simulator, followed by safe learning in real execution scenarios.

On the other hand, Bao [13], HybridQO [30], and Lero [35] utilize the second architecture. Bao employs a hint set to guide traditional optimizers in generating different query plans and then uses a TreeCNN-based cost estimation model to select the optimal query plan. HybridQO utilizes Monte-Carlo Tree Search to efficiently explore better hints in a large hint space, modeling them as a tree. It employs an uncertainty-based optimal plan selection model that predicts execution time and measures the confidence of each candidate plan, considering both factors for plan selection. Lero utilizes expression-level cardinality estimation to effectively explore diversified plans and prioritize more promising ones. It then applies a pairwise learning approach for plan selection by comparing two plans and predicting the better option.

Remark: These learned query optimizers focus on optimizing query plans for a single query without considering the concurrent execution of multiple queries or reducing redundant computations for common subqueries.

9 CONCLUSIONS

In this paper, we proposed Lemo, a learned multi-query optimizer, to tackle a challenging multi-query optimization problem that aims at generating query plans for new queries when multiple queries are being executed. The key novelties of Lemo are that it minimizes redundant computation and performs concurrent query execution for the efficiency and effectiveness of query plan generation. To this end, we proposed a transformer-based value network to predict the best-possible latencies of concurrent plans. Moreover, we proposed a novel replacement policy to manage the intermediate results of subqueries for minimizing redundant computation and designed a two-stage plan search algorithm for query plan generation when performing concurrent query execution. Extensive experimental results on real-world datasets demonstrate both the efficiency and the effectiveness of the proposed Lemo.

ACKNOWLEDGMENTS

This research is supported in part by MOE Tier-2 grant MOE-T2EP20221-0015, National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-TC-2021-001) and ARC DP220101434.

REFERENCES

[1] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Leopoldo Perez. 2010. The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 519–530.

[2] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *Proceedings of the 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

[3] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *Proc. VLDB Endow.* 2, 1 (2009), 277–288.

[4] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. 2001. Pipelining in Multi-Query Optimization. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM.

[5] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries with One Stone. *Proc. VLDB Endow.* 5, 6 (2012), 526–537.

[6] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared Workload Optimization. *Proc. VLDB Endow.* 7, 6 (2014), 429–440.

[7] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. ACM, 383–394.

[8] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. 2004. The Case for Precision Sharing. In *Proceedings of the 30th International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Morgan Kaufmann, 972–986.

[9] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Computers* 50, 12 (2001), 1352–1361.

[10] 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.

[11] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2002, Madison, Wisconsin, USA, June 3-6, 2002*. ACM, 49–60.

[12] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2016. MQJoin: Efficient Shared Execution of Main-Memory Joins. *Proc. VLDB Endow.* 9, 6 (2016), 480–491.

[13] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2021, Virtual Event, China, June 20-25, 2021*. ACM, 1275–1288.

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

[15] Silvano Martello and Paolo Toth. 1987. Algorithms for Knapsack Problems. *North-Holland Mathematics Studies* 132 (1987), 213–257.

[16] Pietro Michiardi, Damiano Carra, and Sara Migliorini. 2021. Cache-based multi-query optimization for data-intensive scalable computing frameworks. *Information Systems Frontiers* 23 (2021), 35–51.

[17] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence, AAAI 2016, Phoenix, Arizona, USA, February 12-17, 2016*. AAAI Press, 1287–1293.

[18] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. 2013. Recycling in Pipelined Query Evaluation. In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 338–349.

[19] Jooseok Park and Arie Segev. 1988. Using Common Subexpressions to Optimize Multiple Queries. In *Proceedings of the 4th International Conference on Data Engineering, ICDE 1988, Los Angeles, California, USA, February 1-5, 1988*. IEEE Computer Society, 311–319.

[20] Ulrich Pferschy and Joachim Schauer. 2009. The Knapsack Problem with Conflict Graphs. *J. Graph Algorithms Appl.* 13, 2 (2009), 233–249.

[21] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2000, Dallas, Texas, USA, May 16-18, 2000*. ACM, 249–260.

[22] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (1988), 23–52.

[23] Kyuseok Shim, Timos K. Sellis, and Dana S. Nau. 1994. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data Knowl. Eng.* 12, 2 (1994), 197–222.

[24] Yasin N Silva, Paul-Ake Larson, and Jingren Zhou. 2012. Exploiting common subexpressions for cloud query processing. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1337–1348.

[25] Panagiotis Sioulas and Anastasia Ailamaki. 2021. Scalable Multi-Query Execution using Reinforcement Learning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2021, Virtual Event, China, June 20-25, 2021*. ACM, 1651–1663.

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

[27] 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: NIPS 2017, Long Beach, CA, USA, December 4-9, 2017*. 5998–6008.

[28] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2022, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 931–944.

[29] Zhengyu Yang, Danlin Jia, Stratis Ioannidis, Ningfang Mi, and Bo Sheng. 2018. Intermediate data caching optimization for multi-stage and parallel big data frameworks. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 277–284.

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

[31] 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.

[32] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. 2021. Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI 2021, Virtually, February 2-9, 2021*. AAAI Press, 11106–11115.

[33] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. 2007. Efficient Exploitation of Similar Subexpressions for Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2007, Beijing, China, June 12-14, 2007*. ACM, 533–544.

[34] 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.

[35] 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.* (2023). https://arxiv.org/abs/2302.06873

Received April 2023; revised July 2023; accepted August 2023