LIMAO: A Framework for Lifelong Modular Learned Query Optimization

Qihan Zhang
University of Southern California
Los Angeles, USA
qihanzha@usc.edu

Shaolin Xie
University of Southern California
Los Angeles, USA
shaolinx@usc.edu

Ibrahim Sabek
University of Southern California
Los Angeles, USA
sabek@usc.edu

ABSTRACT

Query optimizers are crucial for the performance of database systems. Recently, many learned query optimizers (LQOs) have demonstrated significant performance improvements over traditional optimizers. However, most of them operate under a limited assumption: a static query environment. This limitation prevents them from effectively handling complex, dynamic query environments in real-world scenarios. Extensive retraining can lead to the well-known catastrophic forgetting problem which reduces the LQO generalizability over time. In this paper, we address this limitation and introduce LIMAO (Lifelong Modular Learned Query Optimizer), a framework for lifelong learning of plan cost prediction that can be seamlessly integrated into existing LQOs. LIMAO leverages a modular lifelong learning technique, an attention-based neural network composition architecture, and an efficient training paradigm designed to retain prior knowledge while continuously adapting to new environments. We implement LIMAO in two LQOs, showing that our approach is agnostic to underlying engines. Experimental results show that LIMAO significantly enhances the performance of LQOs, achieving up to a 40% improvement in query execution time and reducing the variance of execution time by up to 60% under dynamic workloads. By leveraging a precise and self-consistent design, LIMAO effectively mitigates catastrophic forgetting, ensuring stable and reliable plan quality over time. Compared to Postgres, LIMAO achieves up to a 4× speedup on selected benchmarks, highlighting its practical advantages in real-world query optimization.

PVLDB Reference Format:
Qihan Zhang, Shaolin Xie, and Ibrahim Sabek. LIMAO: A Framework for Lifelong Modular Learned Query Optimization. PVLDB, 18(11): 4546 - 4559, 2025.
doi:10.14778/3749646.3749712

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Tsihan/LIMAOLifeLongRLDB.

1 INTRODUCTION

Query optimizers are crucial components of database systems, responsible for selecting efficient execution plans for queries. Traditional query optimizers have been the backbone of database systems for decades (e.g., [20, 55]). However, their inherent limitations and challenges, such as reliance on heuristics (e.g., attribute independence [32, 59]) and inaccurate cost models especially with complex workloads and diverse data distributions, have spurred the development of learned query optimizers (LQOs) (e.g., [15, 36, 37, 65, 70]), where machine learning (ML) models have been used to replace or improve different components of the query optimizer including cardinality estimation [28, 57, 67], cost prediction [38, 56], and plan search [36, 37, 65, 70].

While effective in certain scenarios, existing LQOs struggle in dynamic environments with shifting data distributions and query workloads. Recent efforts (e.g.,[10, 48]) have primarily focused on managing controlled and infrequent shifts. However, addressing significant and frequent changes remains a challenge, as the predominant solution still relies on *complete retraining from scratch*. This approach is computationally expensive and susceptible to performance degradation due to *catastrophic forgetting*[26], where previously learned knowledge is lost when adapting to new workloads, particularly in cases of temporary changes or workload reversions.

In this paper, we address this limitation by introducing LIMAO, a framework for Lifelong Modular Learned Query Optimization. LIMAO focuses on the *learned cost prediction* (LCP) problem, where ML models (e.g., tree convolution networks [46] and transformers [44]) are employed to predict (sub-)plan costs. In particular, LIMAO aims at promoting existing LCPs (e.g., [36, 37, 65, 70]) in LQOs to be *lifelong* learners; capable of continuously learning and adapting over time while retaining and leveraging previously acquired knowledge to ensure stable performance. LIMAO is based on the idea of learning reusable knowledge in a modular manner [40, 41]. The core intuition is that, to develop a *versatile* LCP capable of accurately predicting the costs of unseen queries, it is essential to learn reusable knowledge of queries' sub-plans, i.e., tasks, that can be recombined for new queries, rather than tailoring LCPs to specific query types or workloads. This approach draws inspiration from advancements in robotics and mirrors human problem solving, where accumulated, reused, and recombined skills enable efficient handling of novel challenges over time.

Realizing the idea of learned reusable knowledge in the query optimization context requires addressing three key challenges: (1) identifying a modular decomposition approach suited to the tree structure of query plans, (2) ensuring the LCP effectively reuses existing knowledge about tasks (i.e., sub-plans) before incorporating new knowledge to prevent redundant learning, and (3) developing a learning strategy that preserves previously acquired knowledge when integrating new knowledge from tasks in new queries. To address these challenges, LIMAO introduces several core components. To tackle the first challenge, LIMAO employs an

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749712

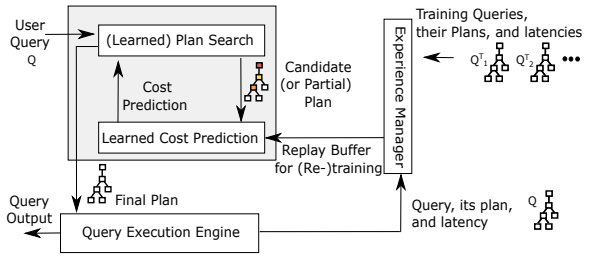


Figure 1: A typical learned query optimizer architecture.

efficient plan decomposer that splits query plans into sub-plans (i.e., tasks) using a new notion of "break" operators that define modular points in the plan tree. For the second challenge, LIMAO builds and maintains *Module Hubs*, a shared repository of specialized neural network modules corresponding to specific task types, using an efficient variation of the K-prototype algorithm [25]. LIMAO further introduces a new *attention-based neural network composition* architecture that dynamically integrates the selected modules from the hub and assigns appropriate weights to their contributions to the final plan cost estimation. To overcome the third challenge, LIMAO proposes a novel two-phase training approach grounded in the lifelong learning paradigm that avoids overriding previous knowledge. This approach leverages *experience replay* buffers along with a lightweight episodic updates mechanism to facilitate frequent, low-overhead model updates during the online phase while eliminating the need for full retraining in the offline phase.

LIMAO is a generic framework designed to seamlessly integrate with almost all existing LQOs. In this paper, we showcase its potential by integrating it with Balsa [65], a state-of-the-art reinforcement learning-based query optimizer, and Bao, a state-of-the-art hint-based query optimizer, as system prototypes. We tested on IMDB [32] and TPC-H [11] using Postgres and designed challenging dynamic situations for them. Our experimental evaluation across various dynamic workloads and data distribution shifts shows that LIMAO not only has better results on **execution time** but also outperforms the prototype by orders of magnitude in **query stability** (variance of plan quality over the same query). For IMDB queries, LIMAO improves plan execution time up to 40% and query stability by 60%, effectively addressing catastrophic forgetting. LIMAO also achieves up to 400% gains in execution time compared to Postgres. For TPC-H queries, 2× speedup is achieved, and the query stability gain is more than 100×, demonstrating significant performance improvements and resilience under dynamic situations. LIMAO can reduce the number of bad plans to 0, while Balsa has a few hundred. We also test LIMAO's resilience to overcome the fluctuation in different dynamic levels on different workloads for IMDB, the result shows that LIMAO can achieve up to 6× speedup than Balsa.

2 LCP AS A LIFELONG LEARNING PROBLEM

2.1 LCP in Learned Query Optimizers

Plan cost prediction is a critical operation in query optimization that determines the efficiency of an execution plan. Traditional cost models rely on formulas that take cardinality estimates of subqueries as inputs and use manually-tuned constants and heuristics refined over years to align estimated costs with actual performance (e.g., [19, 20, 55]). While effective, these traditional models often struggle to capture the complexity of varying data distributions and query workloads [38, 56].

Many LQOs (e.g., [36, 37, 65, 70]) adopt *learned cost prediction* (LCP) techniques, where ML models, such as tree convolution networks [46] and attention mechanisms [60], are used to predict the cost of (sub-)plans based on execution statistics from diverse datasets and workloads (Unlike approaches that focus solely on learning cardinality estimation [17, 27, 47, 62, 67, 68, 71], our focus is on learned end-to-end cost prediction techniques). In these LQOs, LCPs guide the plan search algorithms to find the best execution plan. For instance, the plan search process can be modeled as a deep reinforcement learning problem, with LCP as value networks to guide searches that complete partial plans into full execution plans [65]. Alternatively, other plan search algorithms learn how to select among candidate plans generated by traditional optimizers while leveraging LCPs to estimate the cost of these candidate plans (e.g., [36]) or their relative rankings (e.g., [70]). Figure 1 shows a typical architecture of an LQO employing LCP with its plan search.

2.2 LCP with Lifelong Learning

Performance Degradation in Dynamic Environments. While existing LCP approaches (e.g., [36–38, 56, 65]) have shown promise, they face significant challenges in dynamic environments of data and query workloads. Some LQOs have attempted to address the dynamic environment challenges by emulating the data and workload drift scenarios during training, such as randomly removing [10] or partially masking query information [48] to force ML models to make predictions with missing information. However, these approaches are only effective for slight and infrequent changes in query or data distributions. Most LCP approaches still require *complete* (re-)training from scratch (i.e., ML model parameters are completely re-learned) when faced with *significant* and *frequent* changes in the environment. For example, an LCP trained on a workload dominated by JOB queries [32] will generate suboptimal plans if the workload is frequently altered with CEB queries [48], which involve more complex join patterns than JOB. While robust learned cardinality estimators (e.g., [34, 48]) might adapt to such changes, the LCP still needs to be retrained to reflect this change in the plan cost prediction. Extensive retraining of LCPs in response to frequent changes in dynamic environments can lead to the *catastrophic forgetting* problem [26], where the learned model fails to retain prior knowledge when retrained on new ones, especially in cases of *temporary changes* or *workload reversions*. To better understand this failure behavior, we conducted a simple experiment that simulates frequent workload drift using a TPC-H query workload [11] (scale factor of 10). We divided the workload into two sets, namely set1 and set2, based on query templates, and alternated their execution every 5 iterations over a total of 100 iterations. In each iteration, all queries from the active set were executed (e.g., set1 ran during iterations 1–5, set2 during iterations 6–10, and so on, with each set running for a total of 50 iterations). We evaluated two variations of Balsa [65], a typical RL-based LQO: vanilla Balsa, which suffers catastrophic forgetting, and our LIMAO-based Balsa,

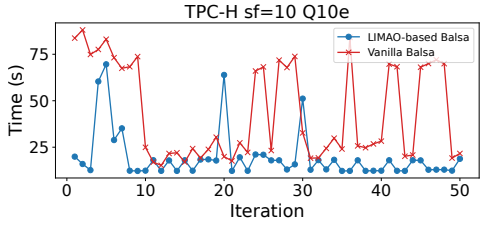


Figure 2: Catastrophic forgetting on the LQOs performance.

which addresses this issue based on our proposed LIMAO framework. Both variations incrementally retrained their LCPs at the end of each iteration. This means that for iterations 2 to 4 within any 5-iteration period, queries used an LCP that had been (re)-trained on their current query set, while in iteration 1, they relied on an LCP trained on the other query set from the previous period. Figure 2 shows the execution time of one query in set1 across its 50 iterations. After iteration 20, the performance of both variations started to stabilize, where vanilla Balsa encounters latency jumps at the beginning of the 5-iterations period due to the catastrophic forgetting of its LCP that was re-trained on the other query set in the previous 5-iterations period. In contrast, the LIMAO-based variation was more stable despite frequent workload changes.

Our Goal. Our ultimate goal is to develop a *lifelong* LCP that can be useful for long stretches of time. The LCP can continuously learn and adapt over time, *integrating new knowledge* from changing queries and data while *retaining and leveraging the previously acquired knowledge* to ensure that performance in prior tasks does not degrade. While single-task learning approaches incorporating techniques like transfer learning [6] and mechanisms to avoid catastrophic forgetting [29] seem to be a natural fit for our problem, they are inherently task-specific and struggle to generalize to significantly different tasks. Multi-task learning (e.g., [53]) offers another direction, where a single model can be learned to share knowledge across diverse types of queries and data scenarios, but this approach is impractical since future queries and data patterns cannot be fully anticipated. Following the same direction, large-scale pre-training on diverse query workloads and data distributions (e.g., [22, 23]) can provide a foundation, yet assumes stationary distributions and fails to adapt to the dynamic nature of real-world database environments. Furthermore, pre-trained models are inherently imperfect and require continual feedback from the execution environment to correct unexpected behaviors and maintain effectiveness over time.

3 LIMAO OVERVIEW

Main Observation and Challenges. We observe a key limitation hindering existing LCPs from being suitable for a lifelong setup: they treat each query as a standalone task, which is inadequate for knowledge retention. Queries often comprise fine-grained patterns that are hard to learn with a single ML model. Moreover, these patterns can be shared across queries to help in predicting their costs. For example, in a multi-table join query, it is typical to combine different types of join operators (e.g., nested-loop join, hash join, sort-merge join) to run the query, and such combinations often recur in other queries. Thus, it is valuable to learn these patterns as independent ML *modules*. Learning compositional modules enables the recombination of these modules for unseen queries and new patterns without requiring a complete update of the LCP's model. However, achieving this presents three main challenges. First, we need to identify a modular decomposition approach that is suitable for the plan tree structure (**Challenge 1**). Second, we need a strategy that *ensures* that the LCP really reuses knowledge from existing modules to predict the cost of new queries before incorporating any specific new knowledge to these new queries (**Challenge 2**). Third, when incorporating such new knowledge, we must devise a learning strategy that simultaneously prevents the forgetting of knowledge already stored in existing modules (**Challenge 3**).

Framework Overview. Here, we propose LIMAO, a framework (shown in Figure 3) that tackles the above-mentioned challenges and allows existing LCPs to be efficient lifelong learners. Inspired by recent advances in robotics and human learning processes [39–42] that focus on learning reusable knowledge, LIMAO formulates the learning process of LCP as a *functional compositional* learning problem [40]. The core idea is to decompose query plans into smaller "tasks" (i.e., sub-plans), each handled by specialized neural network "modules" (e.g., Tree-CNN [46], Transformer [60]), which are then composed into a higher-level network to predict the overall cost. For example, a query plan with nested-loop joins and hash joins can reuse pre-trained modules for these operators, recombining them to handle unseen queries efficiently.

To address Challenge 1, LIMAO introduces a Plan Decomposer (Section 4) that divides the plan into sub-plans (i.e., tasks) based on a new notion of "break" operators. These operators efficiently define points in the plan tree where modular composition can occur. To tackle Challenge 2, LIMAO maintains a shared compositional knowledge base called *Module Hubs*. Each hub contains a set of specialized neural network modules that correspond to a specific task type, allowing the model to dynamically select the most relevant modules for a given task. LIMAO further employs a composable neural network (Section 7) designed to predict the cost of any plan based on the modules chosen for the tasks inside it, dynamically weighting their contributions to refine the final estimate. To overcome Challenge 3, and inspired by prior works [40], LIMAO adopts an efficient two-phase training paradigm (Section 8) that encourages knowledge reuse rather than redundantly learning patterns. The training process of LCP is divided into *online* and *offline* phases. During the *online* phase, LIMAO attempts to form a cost prediction model for the LCP and train it using only existing *close enough* modules from the Module hubs. To ensure that LCP wisely reuses previously acquired knowledge *without overriding it*, LIMAO modifies only a copy of the selected modules using a lightweight training mechanism. In this approach, each training iteration is divided into smaller sections called *episodes*, which allow for prompt model updates and avoid disastrous plans. The original module parameters remain unchanged, and any newly discovered knowledge is stored in a buffer for later use in the offline phase. During the *offline* phase, the LCP incorporates any new knowledge from the buffer by updating the original parameters of existing modules.

Figure 3 shows the components and workflow of LIMAO. First, a candidate or a final plan is obtained from an LQO. For example, an LQO like Bao [36] would rely on complete candidate execution plans generated by a traditional optimizer to select from, whereas

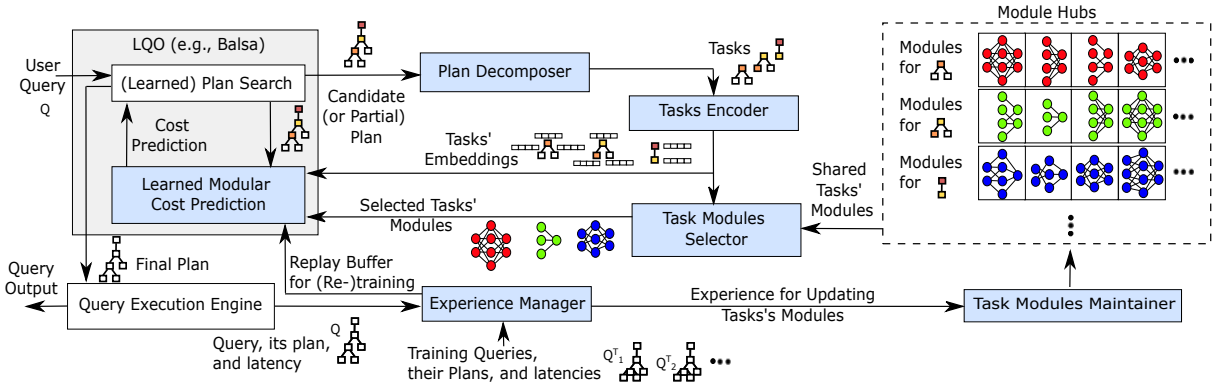


Figure 3: LIMAO framework overview.

Neo [37] and Balsa [65] use a step-by-step plan searcher that builds on partial plans at each iteration. Then, the *Plan Decomposer* breaks the plan into sub-plans (i.e., tasks) (Section 4). Next, the *Tasks Encoder* obtains efficient embeddings for each task using plan-level and operator-level features (Section 5), which are used by the *Task Modules Selector* and *Modular Plan Cost Predictor* components. The Task Modules Selector (Section 6) retrieves the most relevant modules from the *Module Hubs* based on embeddings, adopting a *K*-prototype clustering algorithm [25]. These hubs are created and frequently updated by the *Task Modules Maintainer* (Section 6). The *Modular Plan Cost Predictor* assembles the selected modules into a compositional neural network. Their output query representations are weighted and merged using the attention mechanism [7, 12, 21] to generate a final cost estimate for the plan. When the LQO provides a final plan based on the cost estimates, the plan is executed in the query execution engine and feedback is sent to *Experience Manager* for (re-)training the *Modular Plan Cost Predictor* and the selected modules using our two-phase training approach.

4 PLAN DECOMPOSITION INTO TASKS

In this section, we illustrate the process of decomposing a query plan into a set of tasks to address **Challenge 1**. This process is an important step in enabling the functional composition of LIMAO. However, it is very challenging because decomposing a plan into a set of combinable tasks is inherently complex. This complexity arises from the variable structure of the query plans, where the number, types, sequence of operators, and tree depth are not fixed.

Approach. A straightforward idea for decomposing a query plan is to divide it into pipelines based on "blocking" operators [43]. However, this often results in suboptimal tasks from the functional composition perspective, as blocking operators may create pipelines that are too coarse-grained, resulting in large, monolithic tasks where diverse execution patterns (e.g., various join types or scan methods) are grouped together. Such coarse decomposition makes it challenging to construct reusable modules for other queries. Determining appropriate *breakpoints* in the plan tree for task decomposition is, in general, a non-trivial problem. Sub-trees that are too deep result in ineffective decomposition, as they closely resemble the original query plan, while overly frequent breakpoints create

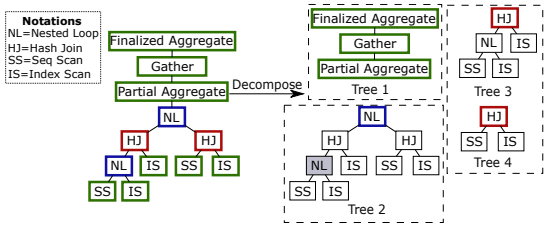


Figure 4: Example on plan decomposition into tasks. Grey operators are those break operators that are ignored because their ancestral nodes already include at least one instance of the same break operator type.

shallow sub-trees that fail to capture plan complexity and add computational overhead due to the proliferation of sub-trees. To address this, we introduce the concept of "break" operators to guide the plan decomposition. A break operator is an operator that splits the plan tree at its first occurrence during a top-down traversal. Specifically, we identify the first occurrence of a break operator in both the *left* and *right* sub-trees of the root, and extract the corresponding sub-trees *rooted* at these operators. We stop the traversal upon encountering the first instance of each break operator type to avoid creating structurally overlapping sub-trees rooted at the same operator type. Continuing the decomposition beyond the first occurrence would lead to multiple sub-trees that share the same break operator type at their roots—i.e., they would be *functionally overlapping*—resulting in unnecessary tasks.

Joins as Break Operators. In principle, break operators can be any type of query plan operator (e.g., scan, join, aggregate, sort). To be effective, they should satisfy two key criteria: 1) *Performance-critical*: the break operator should significantly influence the performance, and 2) *Decomposition-effective*: splitting around this operator should produce sub-plans that are neither too fine-grained nor too coarse, enabling meaningful modular learning. We chose the join operators because they best satisfy both criteria. First, join operators are typically the most computationally expensive components in query plans. Second, decomposing around join operators results in sub-plans that are rich in structure and execution variability.

Example. Figure 4 shows an example of our plan decomposition approach using two example break operators: hash join (HJ) and nested-loop join (NL). The left side shows the original plan tree, while the right side shows the resulting four sub-trees (i.e., tasks): three sub-trees that are generated by the break operators and one sub-tree that spans the remaining operators (i.e., aggregations, scans, non-break join operators) from the root to the first encountered break operator or leaf nodes for completeness. The plan tree is traversed independently for each break operator type to generate their corresponding tasks. For the NL operator, only one sub-tree (Tree 2) is generated as the first encountered NL node has no sibling and already covers its grandchild NL node, which is colored in grey. Conversely, for the HJ operator, two HJ-rooted sub-trees (Trees 3 and 4) are produced, as the traversal continues through the left and right sub-trees of the NL node, marked in purple. Note that, since Tree 2 is NL-rooted and Tree 3 (or Tree 4) is HJ-rooted (i.e., functionally non-overlapping), they can overlap structurally.

5 TASK ENCODING

Efficient encoding of decomposed tasks is crucial to capture their characteristics during the module selection and cost prediction steps. Below, we provide the details of encoding a single task:

Table Selectivity (Feature *A*). This captures the selectivity of tables involved in the task. Following the approach in [48], we use traditional cardinality estimators to compute these selectivities. These estimators add minimal overhead to the inference time of the learned model while providing valuable information. If there are n tables in a workload, then we represent this feature with a numerical vector of length n, where each entry contains a normalized value between 0 and 1 (i.e., table selectivity divided by table size) if the task involves the corresponding table, and 0 otherwise.

Operator Mapping (Feature *B*). This encodes the join and scan properties. A bottom-up traversal of the sub-tree corresponding to the task is performed and the count of each type of join operator (hash join (HJ), merge join (MJ), and nested-loop (NL) join) and scan operator (sequential scan (SS) and index scan (IS)) encountered is recorded. If there are n tables in a workload, then we represent this feature with a numerical vector of length $3 + 2n$, where the first three entries correspond to the counts of the join types, and the remaining entries represent the counts of scan types for each table.

Preorder Index (Feature *C*). This encodes the tree structure of the sub-tree associated with the task in a numerical vector. Similar to [37, 65], we generate this vector using a pre-order traversal of the sub-tree, where each entry represents the index of a node in the original query plan to which the task belongs. For nodes with no children, the corresponding child entries in the vector are set to 0.

Query Feature (Feature *D*). This captures whether the SQL query associated with the task possesses any of the following four properties: (1) contains a sub-query, (2) includes an aggregation function, (3) contains a GROUP BY clause, or (4) contains an ORDER BY clause. It is represented as a binary 0-1 vector of length 4.

Example. Figure 5 depicts the encoding of the three break-operator-based tasks in Figure 4 (we omitted the encoding details of Tree 1 to avoid cluttered visualization). The upper left part shows the SQL query corresponding to the decomposed plan. Features *A* and *D* are defined at the query-level and then used with all tasks. In contrast,

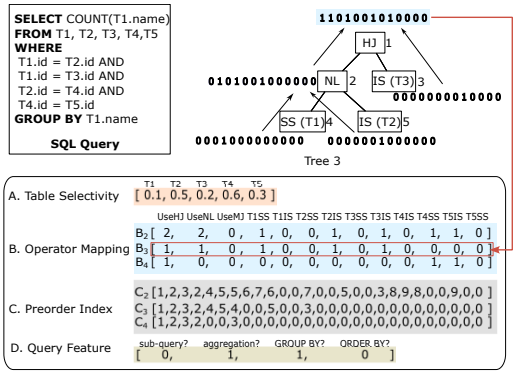


Figure 5: Encoding for the tasks represented with trees 2, 3, and 4 in Figure 4.

features *B* and *C* are task-specific, with B_i and C_i representing the features for the task associated with Tree i ($i \in \{2, 3, 4\}$). Additionally, we detail the process of generating Feature *B* for Tree 3, demonstrating a bottom-up traversal that incrementally counts the join and scan operators encountered.

6 MODULES MAINTENANCE AND SELECTION

Identifying the appropriate neural network modules for the decomposed tasks is a critical step in constructing an accurate LCP. A straightforward approach is to treat each unique combination or sequence of operators in a query plan as an independent task and assign a dedicated neural network module to it in the module hub. For example, a task that involves SS followed by NL would require a different module from a task that involves SS followed by "indexed" NL. Similarly, a task executing HJ before NL would require a distinct module from the one executing them in reverse order. However, this approach is impractical, as it would lead to an explosion in the number of modules, significantly increasing storage and search overhead while limiting the benefits of modularization by over-specializing each module to a highly specific task. To address this, our idea is to maintain a *representative* module for each group of *similar enough* tasks using a clustering algorithm. These representative modules serve as a carrier of knowledge, which is the prerequisite for solving **Challenge 2**. All representative modules built for an LQO integrated with LIMAO share the same neural network architecture, inherited from the base LCP model used by this LQO (e.g., the Tree-CNN architecture in Balsa [65]). However, they differ in their parameters, which are specialized through training on their respective tasks (i.e., sub-plans). These parameters are then dynamically adapted as tasks evolve over time.

Building Each Module Hub as a Cluster. We adopt a variation of the K-prototype algorithm [25] to maintain K representative neural network modules for each type of task decomposed using a specific break operator type (i.e., a set of K representatives for NL-rooted tasks, and another set of K representatives for HJ-rooted tasks, etc). We choose the K-prototype algorithm for two key reasons. First, it is an extension of K-means designed to handle mixed data types (numerical and categorical features), which is compatible with our numerical and categorical task encodings (Section 5). Second, it is

computationally efficient, making it practical for maintaining and updating module clusters dynamically over time. Our algorithm for constructing clusters for each module hub operates in four steps during the training phase of any LQO. First, initialize K module representatives (i.e., centroids) with random values for Feature A as the numerical feature, and Features B, C, D as categorical ones, based on the training set queries. Second, for each task x_i decomposed from the training examples, assign it to the cluster with the smallest dissimilarity value, calculated according to this formula:

$$d = \sum_{j \in \{\text{A}\}} \text{dis}_{\text{num}}(x_{ij}, c_{kj}) + \gamma \sum_{j \in \{\text{B, C, D}\}} \text{dis}_{\text{cat}}(x_{ij}, c_{kj}) \quad (1)$$

where x_{ij} is a feature vector (numerical or categorical), and c_k is the representative for cluster $k \in \{1, \ldots, K\}$, γ is a weighting factor to balance the influence of numerical and categorical features, and dis_{num} and dis_{cat} are *vector-wise* numerical (e.g., Euclidean distance) and categorical (e.g., mismatch count) dissimilarity functions, respectively. Third, for each cluster representative c_k, update its numerical c_{kj}^{num} and categorical c_{kj}^{cat} features using the numerical mean $c_{kj}^{\text{num}} = \frac{1}{|S_k|} \sum_{x_i \in S_k} x_{ij}$ and most frequent value $c_{kj}^{\text{cat}} = \arg\max_v \sum_{x_i \in S_k} 1(x_{ij} = v)$ functions, respectively, where S_k is the set of tasks assigned to the cluster. Fourth, repeat the second and third steps until convergence.

Selecting Modules and Updating Hubs. During the testing phase, we simply use Equation 1 to determine the appropriate representative cluster for any incoming task that minimizes the dissimilarity value and then retrieve its corresponding neural network module.

7 MODULAR PLAN COST PREDICTION

Our primary objective in LIMAO is to enable LQOs to construct a customized LCP neural network policy for each query by selecting and combining modules from several shared module hubs, thereby promoting knowledge reuse to address **Challenge 2**. To achieve this, we propose to use *neural composition* techniques (e.g.,[2, 18]) to assemble a complete LCP neural network from the selected modules to handle the query. Neural composition has been extensively applied to build modular architectures in supervised learning (e.g.,[2, 52]) and reinforcement learning (e.g.,[18, 64]) and has recently been adapted for functional compositional problems [13, 40].

Attention-based Neural Composition. A large class of existing neural composition methods assumes a linear dependency structure between decomposed tasks (i.e., a chaining or graph structure), which is common in robotic tasks [4, 40] (e.g., a pick-and-place robotic task is decomposed into sequential steps such as "grasp", followed by "lift", and then "place"). This chaining structure, however, creates brittle dependencies among modules, where changes to one module can have cascading effects on others. Moreover, this assumption does not align well with our break-operator-based decomposition, which can produce non-temporally related tasks. For instance, a multi-table join query may be decomposed into two parallel HJ tasks, as shown in Figure 4. Although parallel neural composition approaches [3, 51] seem well-suited for such scenarios, they often fail to capture the inherent inter-dependencies between query tasks, particularly as the outputs of these tasks must be combined to produce a unified cost prediction for the entire query plan. Additionally, not all tasks contribute equally to the final cost prediction, which these methods typically overlook.

To address these limitations, we propose an *attention-based* variation of parallel neural composition. This variation utilizes the composition of cost prediction tasks by leveraging attention mechanisms [7, 12, 21] to capture task relationships and adjust their contributions accordingly. The core idea is to assign importance to the outputs of different modules using attention weights, dynamically adjusting their contributions based on their potential impact on the final cost. Attention mechanisms have shown effectiveness in other database tasks [34, 54]. Formally, assume that there are k modules with importance $w_1, ..., w_k$, then the assigned weight w_i' for $w_i, i \in [1, k]$ is in a softmax form: $w_i' = \exp(w_i)/\sum_{j=1}^{k} \exp(w_j)$. Furthermore, we train the combination of modules corresponding to one query plan as a whole unit. The modules are not only trained on their tasks but also learn to collaborate with other modules under different combinatorial setups.

Network Architecture. Figure 6 shows the neural network architecture of our LCP. After a plan is decomposed into K tasks and their corresponding modules are retrieved (see Section 6), the table selectivity encoding (i.e., Feature A) is passed through a series of fully-connected layers, each progressively reducing in size. The output of the third fully connected layer is then concatenated with the operator mapping encoding (Feature B) of each decomposed task, resulting in a new vector representation P for each task. Next, using the pre-order index encoding (Feature C), the tree representation of encodings is reconstructed for each task. Each node in this tree corresponds to the previously concatenated vectors from Features A and B, resulting in $1 \times f'$ dimension. This tree representation is then processed by its corresponding module to generate a module-specific plan representation w_i for each task (e.g., Tree-CNN-based [36, 48, 70] or Tree-LSTM-based [8, 69] plan representation). Afterward, the attention merger is applied to these module-specific plan representations. It dynamically assigns weights to each representation and combines them to send to the output MLP layers, producing the final cost estimation of the plan.

8 LIMAO TRAINING

Algorithm 1 Training Mechanism in *LIMAO*

```
1:  T ← 0; B_episode ← []; B_all ← []                          ▷ Initialization
2:  M ← Init(input & output MLP, Module Hubs)
3:  while T < iterations do
4:      if T > 0 then
5:          M ← DeepCopy(M')                                   ▷ Offline updates
6:          if Drift Detected then
7:              M.CompletelyTrainWith(B_all)
8:          else
9:              M.CompletelyTrainWith(B_last)
10:     M' ← DeepCopy(M), B_last ← []
11:     e_1 ... e_n ← iteration queries                        ▷ Split queries into episodes
12:     for each episode e_i do                                ▷ Online exploration phase
13:         for each query q_j ∈ e_i do
14:             modulesIds ← ModuleSelector(q_j)
15:             feedback ← PredictAndExecute(M', q_j, modulesIds)
16:             B_episode.Add(feedback, modulesIds)
17:         M'.LightlyTrainWith(E)
18:         B_last.Add(B_episode) and Reset B_episode
19:     B_all.Add(B_last)
20:     Evaluate(M, testSet)                                   ▷ Evaluation
21:     T ← T + 1
```

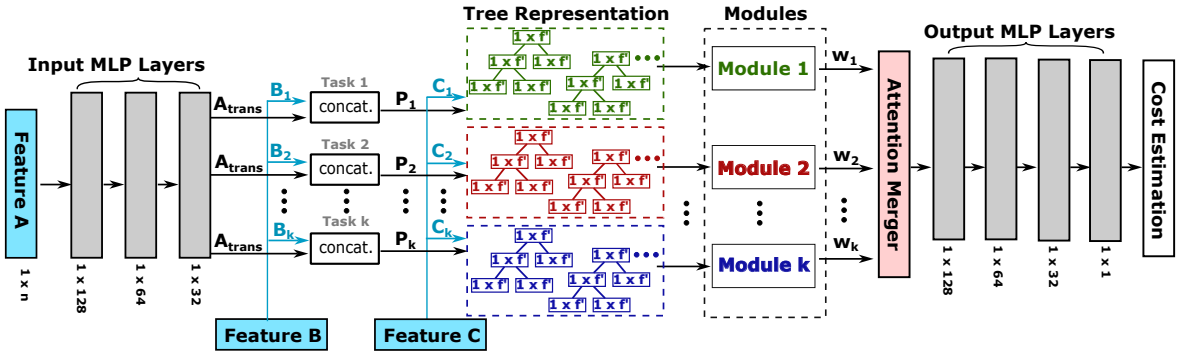


Figure 6: Neural network architecture of modular LCP.

LIMAO introduces a novel training mechanism that follows the lifelong learning paradigm, including experience buffers and episodic updates to improve the quality of LCP and reduce training costs. The training mechanism effectively addresses **Challenge 3** by: 1) taking advantage of reusable modules to eliminate the time needed for retraining from scratch when query or data pattern drifts; 2) using experience replay to quickly reclaim previous knowledge; and 3) dividing training queries within each iteration into subsets of episodes and performing lightweight model training after each episode (training iterations and their corresponding episodes use different sets of queries in LIMAO). In such a way, LIMAO generates better plans in shorter learning periods while effectively avoiding bad plans and stabilizing plan quality. We have two main phases: an *online phase* where we episodically update a copy of the LCP model and populate the experience buffer, and an *offline phase* where we update the original LCP model and perform experience replay. The details of our method (Algorithm 1) are as follows:

Initialization. The *Experience Manager* maintains an episode buffer $\mathcal{B}_{episode}$ that is refreshed per training episode, and an experience buffer $\mathcal{B}_{all}$ that contains experience from every iteration (line 1). The experience from the previous iteration is defined as $\mathcal{B}_{last}$. We create the LCP model $\mathcal{M}$ by initializing the neural network's MLP input and output layers with random parameters, as well as all the modules in Module Hubs and Attention Merger (line 2). To avoid catastrophic damage to the model $\mathcal{M}$ while (re-)training it, we maintain a copy $\mathcal{M}'$ that is updated frequently in our *episodic* setup, whereas $\mathcal{M}$ is updated once per iteration (Details are below).

Online Phase. An under-trained LCP model, particularly in early iterations, could lead to a sequence of inefficient plans due to infrequent model updates. To avoid this, we introduce an episode-based paradigm that updates the LCP model more frequently to stabilize performance. In each episode, the current LCP model is used to predict query plans for the episode's queries, which are then executed. The resulting runtime feedback, along with the corresponding module combinations, is stored in the episode buffer $\mathcal{B}_{\text{episode}}$ (lines 14–16). At the end of the episode, $\mathcal{M}'$ is quickly trained based on the buffer, with only a few epochs of neural network updates (line 17). This lightweight training process introduces minimal latency to the online exploration phase, typically completes in a few seconds, while avoiding damage to the model $\mathcal{M}$.

Offline Phase. In this phase, we perform comprehensive training on the model $\mathcal{M}$ to ensure it remains up-to-date in dynamic environments (lines 5-9). We start by replacing $\mathcal{M}$ with the model copy $\mathcal{M}'$ which was episodically trained in the previous iteration. Following this, $\mathcal{M}$ is retrained using an experience replay buffer, determined by whether a drift is detected or not. If a drift in query patterns or data distribution is identified [1, 5, 16, 24, 33, 35], then $\mathcal{M}$ is retrained using the experience buffer $\mathcal{B}_{all}$, which contains experiences from all prior iterations, allowing the model to retain historical knowledge while integrating new patterns. If no drift is detected, then $\mathcal{M}$ only requires light adaptation. In this case, we limit the retraining of $\mathcal{M}$ to the most recent experiences stored in $\mathcal{B}_{last}$ (lines 6-9) only to reduce overhead and prevent overfitting. Note that, regardless of whether retraining uses $\mathcal{B}_{all}$ or $\mathcal{B}_{last}$, $\mathcal{M}$ converges quickly, as it is initialized from $\mathcal{M}'$, which has already been lightly exposed to recent workload and data shifts.

9 EXPERIMENTAL EVALUATION

Here, we conduct experiments to answer some key questions: Can LIMAO help generate better plans in dynamic environments (Section 9.2.1)? How does LIMAO training paradigm prevent catastrophic forgetting (Section 9.2.2)? How do the design components of LIMAO contribute to its performance (Section 9.3.3)? How effectively can LIMAO avoid bad plans (Section 9.3.4)?

9.1 Experiment Setup

Benchmarks. We perform our evaluation using TPC-H [11] and IMDB-based [32] benchmarks. For IMDB, we evaluate on the JOB [32] and CEB [48] query sets, as well as a set of queries from [36], which we refer to as BaoQs. We construct six benchmarks for IMDB and three benchmarks for TPC-H, where we randomly split each benchmark into training and test sets, as shown in Table 1. The first three IMDB workloads consist of distinct query templates, while the last three reuse the same templates as the first three but apply different filtering conditions to create different queries.

Static and Dynamic Environments. We conducted experiments under four different environments: ***Static***, ***Workload Switch***, ***Volume Switch***, and ***Both Switch***. For *Workload Switch*, the experiment iteratively switches between multiple query sets. We refer to the collection of these query sets as a *combined workload*. For *Volume Switch*, the experiment iteratively switches between two databases

Table 1: Evaluation workloads and their templates.

Workload	training	test	Template
IMDB_set1	94	18	Most from JOB, 2 from BaoQs
IMDB_set2	71	14	Most from BaoQs, 3 from JOB
IMDB_set3	30	5	Most from CEB, 3 from JOB
IMDB_set4	94	18	Most from JOB, 2 from BaoQs
IMDB_set5	14	5	Most from CEB, 3 from JOB
IMDB_set6	14	5	Most from CEB, 3 from JOB
TPC-H_set1	34	6	TPC-H template 3, 8, 10, 13
TPC-H_set2	34	6	TPC-H template 5, 7, 12, 14
TPC-H_set3	40	12	Other TPC-H template except 15

with different data volumes. For the IMDB dataset, this involves switching between the full database and a subset containing only data from the year 2000 onward. For TPC-H, we switch between databases with scale factors of 10 and 1. The *Both Switch* scenario combines changes in both query patterns and data volume simultaneously. Unless otherwise specified, dynamic scenarios involve periodic switching every 5 iterations over a total of 100 iterations, resulting in 50 combined iterations to evaluate for each switch side. For example, in *Workload Switch*, IMDB_set1 is used for iterations 1–5, IMDB_set2 for iterations 6–10, then back to IMDB_set1 for iterations 11–15, and so on. We analyze the performance of LIMAO under non-periodic switching in Section 9.2.3 and 9.2.4.

We compare LIMAO to the following LQOs and Postgres:
Baselines: We evaluate the performance of LIMAO against three baselines: *Balsa* [66], an LQO that uses deep reinforcement learning to construct query plans step-by-step from scratch; *Bao* [36], an LQO that employs a multi-armed bandit model to steer the traditional optimizer toward more efficient plans via ranked hint sets, without fully replacing it; and *Postgres* [49], the traditional rule-based and cost-based optimizer (Version 12.5). We integrated LIMAO with both Balsa and Bao, denoted as **LIMAO-Balsa** and **LIMAO-Bao**, respectively. We also extended Balsa's source code to support non-equality joins and multiple join conditions between two tables, and applied these improvements to both Balsa and LIMAO-Balsa. Our experimental evaluation primarily uses LIMAO-Balsa to assess LIMAO, while LIMAO-Bao is used to explore the performance under non-periodic and cross-schema switching situations in Section 9.2.3 and Section 9.2.4, respectively.
Performance Metrics. We evaluate performance using several key metrics, including query *execution time*, the *number of bad query plans*, and *execution stability*. Execution stability is used to quantify the degree of catastrophic forgetting and is measured using two indicators: the *variance* and the *derivative* of the total workload execution time across iterations. Variance captures fluctuations in execution time, while the derivative is computed from a smoothed curve of execution time to reflect the rate of change over time. Better stability implies better handling of catastrophic forgetting.
Default Settings. Unless otherwise specified, we adopt the following default configurations. To construct the *Module Hubs* in LIMAO, we select HJ and NL as break operators for the IMDB benchmark, with module hub sizes (i.e., K representative modules) of 2 and 3 respectively. For the TPC-H benchmark, we use HJ and MJ as break operators, each with a module hub size of 1. The sizes of *Module Hubs* are determined by the complexity of the workload and the occurrence frequency of the corresponding operator types. In both benchmarks, we include an additional module hub, referred to as *OTH*, which stores modules corresponding to sub-plans not rooted at a break operator (e.g., Tree 1 in Figure 4). For training LIMAO, we divide the queries in each training iteration into episodes of 10 queries. In both the Bao and LIMAO-Bao implementations, we use 49 hint sets and train using the most recent 500 execution records.
Machine Settings. Unless otherwise specified, experiments are conducted on CloudLab [50], using c240g5 which is equipped with dual Intel Xeon Silver 4114 CPUs (2.20 GHz, 20 cores total), 192 GB DDR4 RAM, a 480 GB SATA SSD, and NVIDIA P100 GPU.

9.2 End-to-End Experimental Results

In this section, we evaluate the performance of LIMAO across four dynamic scenarios. Section 9.2.1 presents the performance of LIMAO-Balsa under a basic *Workload Switch* scenario, where two workloads are alternated periodically. In this setting, LIMAO-Balsa performs comparably to Balsa and shows advantages in certain cases only. Therefore, in Section 9.2.2 and Section 9.2.3, we devise more complex *Workload Switch* scenarios which involve periodic switching among three workloads and the other using a non-periodic switch pattern. In Section 9.2.4, we shift to LIMAO-Bao to evaluate performance under a non-periodic, cross-schema switching scenario involving five different workloads. Given the stronger performance of LIMAO-Balsa, we further evaluate its behavior in Sections 9.2.5 to provide a deeper analysis of the benefits introduced by integrating LIMAO into Balsa.

9.2.1 Comparison with Balsa under a simple Workload Switch. In a simple *Workload Switch* scenario, we evaluate the performance of LIMAO-Balsa and Balsa by alternating between IMDB_set1 and IMDB_set2 every 5 iterations over a total of 100. Table 2 shows the speedup ratios of the best query plans generated by LIMAO-Balsa and Balsa in this setup. Both systems outperform Postgres by up to 2.4× on IMDB queries. The performance gain in IMDB workloads is largely due to the ability of LQOs like Balsa and LIMAO-Balsa, which generate plans from scratch, to better handle complex join conditions commonly found in these queries. In IMDB_set1, LIMAO further enhances performance by decomposing complex queries into simpler sub-queries and effectively preserving historical knowledge from the workload, giving it an edge over Balsa. In IMDB_set2, the BaoQs workload is more complex than JOB, featuring more join operations and longer average query execution time. Under such conditions, the limited number of training iterations and queries may not provide enough examples per module combination to train a fully effective model in LIMAO-Balsa, resulting in a slight performance degradation in IMDB_set2 compared to IMDB_set1.

For TPC-H, the advantage of LIMAO-Balsa and Balsa over Postgres is less obvious, particularly due to the relatively simple join conditions. This observation aligns with prior findings [15, 66, 70], which show that LQOs typically offer limited improvements over traditional query optimizers for TPC-H. Despite this, LIMAO-Balsa still achieves improved performance under the TPC-H_set1 and TPC-H_Combined workloads. The lack of performance gains in TPC-H_set2 for both Balsa and LIMAO-Balsa can be attributed to operator selection behavior. In TPC-H_set1, using only HJ is

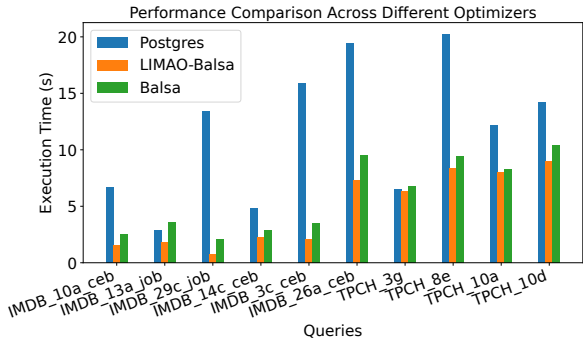


Figure 7: Sample execution time in simple *Workload Switch*.

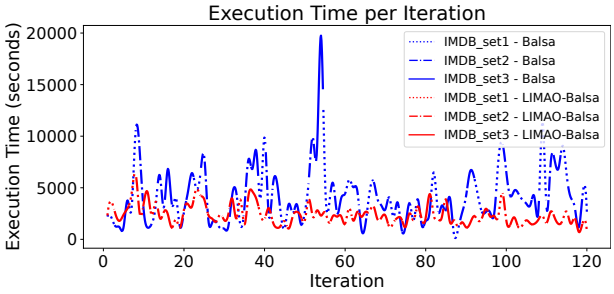


Figure 8: Execution time throughout 120 iterations in the challenging *Workload Switch*.

generally sufficient to generate optimal plans. However, in TPC-H_set2 which includes templates such as 5, 7, 12, and 14, the use of NL would be more optimal. However, when Balsa becomes biased toward always selecting HJ during training, it fails to consider NL, which may be critical for optimal performance in certain queries. This bias can result in performance that is even worse than Postgres. Increasing the number of training iterations could help mitigate this issue by enabling the model to learn when using NL is beneficial.

In Figure 7, we provide further insights by analyzing several slow queries: those with execution times exceeding a few seconds, drawn from different workloads to highlight where performance advantages emerge. Consistent with the results in Table 2, LIMAO-Balsa generally outperforms Balsa on IMDB queries. TPC-H templates 3, 8, and 10 do not involve complex aggregation functions or nested structures and contain complex join orders. In such cases, a build-from-scratch LQO like Balsa performs well. However, for TPC-H queries with relatively simple join structures, the benefits of using Balsa or LIMAO-Balsa over Postgres are minimal.

Table 2: Execution speedup ratio of LIMAO-Balsa vs Balsa in simple *Workload Switch*.

Workload	LIMAO-Balsa	Balsa	Postgres
IMDB_set1	**2.44**	1.96	1.00
IMDB_set2	2.66	**2.68**	1.00
IMDB_Combined	**2.59**	2.40	1.00
TPC-H_set1	**1.31**	1.30	1.00
TPC-H_set2	0.76	0.75	**1.00**
TPC-H_Combined	**1.11**	1.10	1.00

9.2.2 Challenging Periodic Workload Switch. Here, we move to a more complex scenario, where we alternate between IMDB_set1, IMDB_set2, and IMDB_set3 every 5 iterations, over a total of 120 iterations, and evaluate LIMAO-Balsa against Balsa. Table 3 shows the overall speedup and the variance of plan execution times across iterations. LIMAO-Balsa consistently outperforms Balsa in both metrics, indicating that it produces faster and more stable query plans. Specifically, LIMAO-Balsa requires as little as 30% of Postgres's execution time and 60% of Balsa's, while achieving up to 8.8× greater stability compared to Balsa. As the environment becomes increasingly dynamic, the stability of LIMAO-Balsa becomes more prominent, thanks to its episode-based training mechanism, which promptly trains the model to prevent generations of bad plans.

Table 3: Speedup ratio and variance of execution time of LIMAO-Balsa vs Balsa in challenging *Workload Switch*.

Metric	LIMAO-Balsa	Balsa	Postgres
IMDB_set1 Speedup	**2.49**	1.53	1.00
IMDB_set2 Speedup	**4.08**	2.92	1.00
IMDB_set3 Speedup	**1.53**	1.31	1.00
Combined Speedup	**2.90**	2.07	1.00
IMDB_set1 Variance	$\mathbf{4.73 \times 10^5}$	4.20×10^6	NA
IMDB_set2 Variance	$\mathbf{1.18 \times 10^6}$	1.05×10^7	NA
IMDB_set3 Variance	$\mathbf{5.21 \times 10^5}$	2.29×10^6	NA

Figure 8 visually depicts the execution time of Balsa and LIMAO-Balsa, summed across all training and testing queries within each iteration, with different line segments indicating the active workload at each point. LIMAO-Balsa exhibits significantly greater stability and shows steady performance improvements over time. In contrast, Balsa's performance curve is highly erratic, with frequent spikes that correspond to instances of catastrophic forgetting, where the model generates poor plans upon switching back to a previously seen workload. This figure highlights LIMAO-Balsa's ability to effectively mitigate catastrophic forgetting, demonstrating its robustness and practical viability for real-world dynamic environments.

9.2.3 Challenging Non-periodic Workload Switch. In this experiment, we use all six IMDB workloads, with workloads switching randomly: each executed for 2 to 8 consecutive iterations. To ensure reproducibility, we use fixed random seeds. The total number of iterations is set to 100. As shown in Figure 9, Balsa requires 405 seconds to complete all workloads optimally, while LIMAO-Balsa achieves the same in just 307 seconds. Balsa encounters 464 timeouts (defined as queries exceeding 60 seconds in execution time), compared to only 219 timeouts with LIMAO-Balsa. These results demonstrate that LIMAO significantly improves Balsa's performance in highly unpredictable, non-periodic environments.

9.2.4 Non-periodic Workload Switch with Schema Changes. In this section, we evaluate LIMAO-Bao under a non-periodic *Workload Switch* scenario involving schema changes. We randomly switch

Table 4: Speedup ratio of LIMAO-Bao vs Bao in non-periodic *Workload Switch* for LIMAO-Bao experiment.

Workload	LIMAO-Bao	Bao	Postgres
IMDB_set1	**1.17**	**1.17**	1.00
IMDB_set2	**1.48**	1.25	1.00
IMDB_Combined	**1.34**	1.22	1.00
TPC-H_set1	1.98	**2.04**	1.00
TPC-H_set2	**3.20**	**3.20**	1.00
TPC-H_set3	**1.97**	1.58	1.00
TPC-H_Combined	**2.15**	1.91	1.00
All 5 Workloads	**1.85**	1.66	1.00

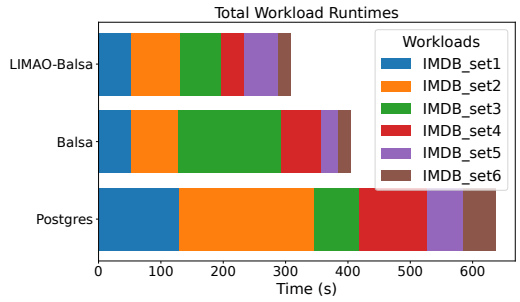


Figure 9: Total optimal execution time of different workloads in non-periodic cross-schema *Workload Switch*.

between five workloads: IMDB_set1, IMDB_set2, TPC-H_set1, TPC-H_set2, and TPC-H_set3, where each workload is executed consecutively for 2 to 10 iterations. Table 4 reports the speedup ratios of Bao and LIMAO-Bao under this dynamic setting. Overall, LIMAO-Bao shows either promising improvements or performance comparable to Bao across most workloads. LIMAO-Bao achieves an average speedup of 1.85× over Postgres, compared to 1.66× achieved by Bao. Both Bao and LIMAO-Bao perform particularly well on the TPC-H benchmarks, which can be attributed to Bao's hint-based design that is especially effective for handling simpler, join-naive queries.

9.2.5 Performance of LIMAO-Balsa and Balsa in All Situations. Figure 10 presents the execution time of static workloads and three types of dynamic switching scenarios across iterations, highlighting the overall performance stability of LIMAO-Balsa and Balsa. While both achieve comparable overall performance, LIMAO-Balsa demonstrates a significantly more stable execution pattern, whereas Balsa suffers from noticeable fluctuations.

In the *Static* setting (Figures 10a and 10b), LIMAO-Balsa performs comparably to Balsa, but with consistently lower execution time in most iterations. Notably, in the TPC-H *Static* scenario, LIMAO-Balsa exhibits much lower execution time at the beginning, whereas Balsa produces a continuous segment of poorly performing plans. This trend indicates that LIMAO-Balsa's episodic training mechanism helps prevent the repeated generation of suboptimal plans, even in static environments. In the *Both Switch* scenarios (Figures 10g and 10h), Balsa's performance is highly unstable, especially in the IMDB environment, where performance even deteriorates over time. In contrast, LIMAO-Balsa maintains low execution latency and stable performance across all iterations. These results demonstrate that LIMAO-Balsa is more robust than Balsa across both static and dynamic scenarios. This improvement can be attributed to the *lifelong* LCP, which stabilizes the training process and reduces the occurrence of performance spikes—an issue commonly observed in conventional reinforcement learning approaches.

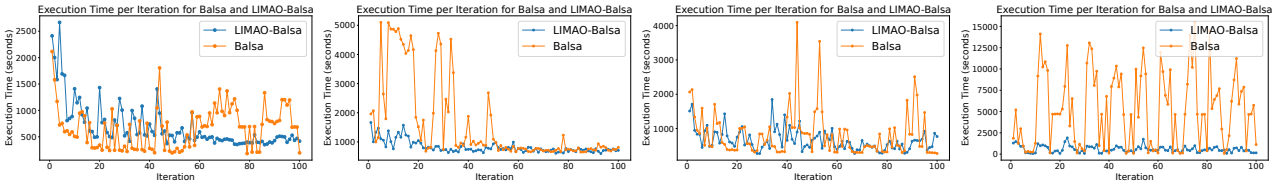
9.3 Micro Benchmarks

9.3.1 Performance Stability. During training iterations, LIMAO-Balsa demonstrates notable performance stability. In this experiment, we collect the total execution time at each iteration, smooth the resulting time series into a continuous curve, and compute its derivative to assess stability. Figure 11 shows the derivatives of the fitted curves for both Balsa and LIMAO-Balsa. In dynamic environments, Balsa exhibits substantial instability, as reflected by its large and highly variable derivative values. Such instability often results in severe performance degradation, as discussed in detail in Section 9.3.4. Moreover, consistently lower derivative values suggest that a model is converging, as the differences in execution time between consecutive iterations become minimal. For instance, in the IMDB *Workload Switch* scenario, LIMAO-Balsa's derivative values remain below 1500 after 30 iterations, compared to 11,000 for Balsa. Similarly, in the TPC-H *Both Switch* scenario, LIMAO-Balsa's values stay below 1000 after 20 iterations, whereas Balsa's rise to 4400. These observations indicate that LIMAO-Balsa likely converges significantly faster, potentially requiring only slightly more than half the number of iterations compared to Balsa. Table 5 further supports this observation by presenting the variance in execution time across different scenarios. In every case, LIMAO-Balsa exhibits lower variance than Balsa, often by orders of magnitude. Notably, in the IMDB *Both Switch* scenario, LIMAO-Balsa achieves 22× greater stability than Balsa. These results highlight that LIMAO substantially improves Balsa's robustness particularly when both query workloads and data distributions shift.

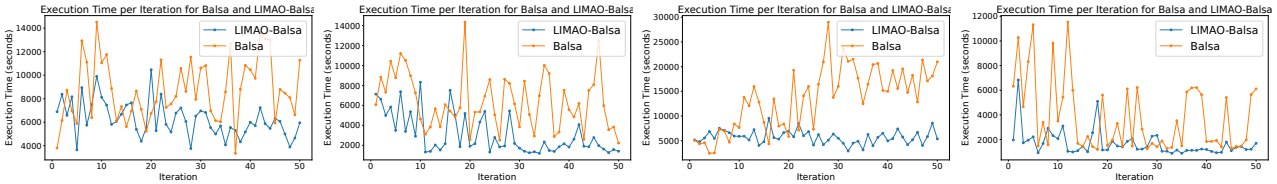
Table 5: Variances for Balsa and LIMAO-Balsa. The results are based on each iteration's combined execution time.

Workload	pattern	LIMAO-Balsa	Balsa
IMDB	*Both Switch*	$\mathbf{1.77 \times 10^6}$	4.00×10^7
IMDB	*Workload Switch*	$\mathbf{2.23 \times 10^6}$	6.96×10^6
IMDB	*Volume Switch*	$\mathbf{9.43 \times 10^4}$	4.32×10^5
TPC-H	*Both Switch*	$\mathbf{1.10 \times 10^6}$	8.30×10^6
TPC-H	*Workload Switch*	$\mathbf{3.93 \times 10^6}$	7.70×10^6
TPC-H	*Volume Switch*	$\mathbf{1.44 \times 10^5}$	1.88×10^7

9.3.2 Choice of Module Hub's Length. To evaluate the impact of different Module Hub sizes, we test three settings on the IMDB workload by varying the number of modules assigned to the HJ, NL, and OTH hubs. In setting S1, we assign 1 module each to HJ, NL, and OTH. In setting S2, we use 2 modules for HJ, 3 for NL, and 1 for OTH. In setting S3, we assign 3 modules to each of HJ, NL, and OTH. These three settings are evaluated over 20 iterations in the IMDB *Workload Switch* scenario to compare their minimum execution time per iteration. The results for S1, S2, and S3 are 4067*s*, 3652*s*,

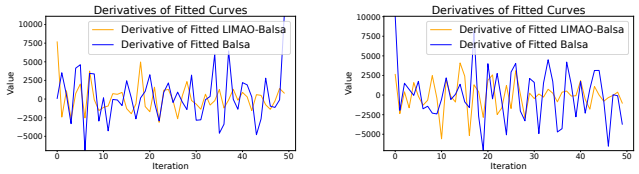


(a) IMDB *Static Workload.* (b) TPC-H *Static Workload.* (c) IMDB *Volume Switch.* (d) TPC-H *Volume Switch.*

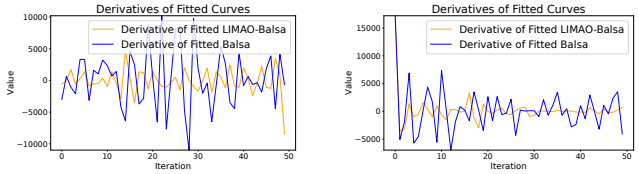


(e) IMDB *Workload Switch.* (f) TPC-H *Workload Switch.* (g) IMDB *Both Switch.* (h) TPC-H *Both Switch.*

Figure 10: Performance of Balsa and LIMAO-Balsa throughout iterations in all 8 types of dynamic environments.



(a) IMDB *Workload Switch.* (b) TPC-H *Workload Switch.*



(c) IMDB *Both Switch.* (d) TPC-H *Both Switch.*

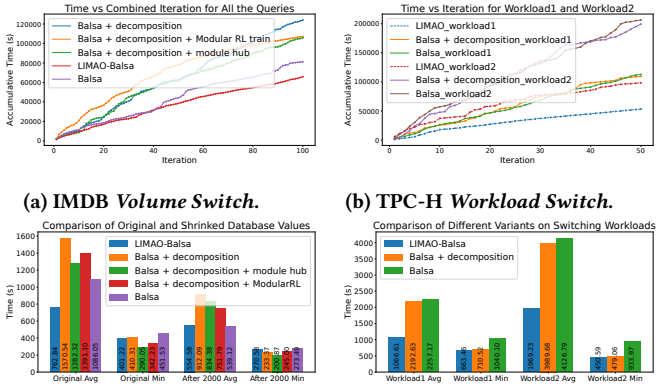
Figure 11: Derivatives of execution time from each iteration.

and 4826*s*, respectively (S2 has the best performance). This result highlights how the choice of Module Hub size can significantly influence the performance of LIMAO. In the IMDB workload, NL joins tend to exhibit more complex behavior than HJ and OTH operators, so allocating more modules to NL tasks improves model performance. However, assigning too many modules can lead to underfitting, as the number of training tasks per module becomes insufficient, ultimately degrading overall performance.

9.3.3 Ablation Study. To evaluate the impact of different components in LIMAO, we conduct ablation studies in the IMDB *Volume Switch* and TPC-H *Workload Switch* environments. For the IMDB workload, we compare the following variations: (1) Balsa, (2) *Balsa + decomposition*, (3) *Balsa + decomposition + Module Hub*, (4) *Balsa + decomposition + Modular RL training*, and (5) the full LIMAO-Balsa. As described in Section 9.1, each Module Hub in the TPC-H workload is limited to one module due to the simpler join patterns characteristic of TPC-H. Accordingly, we evaluate only three configurations for TPC-H: Balsa, *Balsa + decomposition*, and *Balsa + decomposition + Modular RL training* (i.e., LIMAO-Balsa without the Module Hub component). Figure 12 shows the performance of Balsa and the various LIMAO-Balsa variants. As shown in Figures 12a and 12b, LIMAO-Balsa achieves the lowest cumulative execution time in both IMDB and TPC-H workloads, demonstrating its robustness and effectiveness in dynamic environments. In Figures 12c and 12d, we observe that while some variants occasionally discover better individual query plans than Balsa and LIMAO-Balsa, they suffer from higher average execution times overall. This is primarily because, without the use of a replay buffer, these variants tend to overfit to a narrow subset of queries. A specific module combination may be trained on only a limited set of queries and thus fail to generalize when presented with new ones, resulting in performance collapse. Similarly, using decomposition alone can lead to overfitting and poor generalization. This issue is particularly evident in the variant combining tree decomposition with Module Hubs. In contrast, LIMAO-Balsa achieves an optimal balance between short-term peak performance and long-term stability.

9.3.4 Bad Performance Plans. A query plan is considered bad if its execution time exceeds 512 seconds. Table 6 reports the number of such timeouts across different scenarios for both Balsa and LIMAO-Balsa. Balsa experiences significantly more timeouts during model training, while LIMAO-Balsa effectively avoids them. When LIMAO-Balsa encounters a bad plan, it is less likely to generate a similar one in subsequent iterations, due to its episodic training mechanism that enables prompt model tuning. This enhanced stability stems from the *lifelong* LCP training paradigm, which allows LIMAO-Balsa to quickly recognize the characteristics of poor-performing plans and adapt accordingly in future episodes. This highlights the importance of training the model to identify and avoid bad plans as early as possible.

9.3.5 Extra Training Overhead. As discussed in Section 8, LIMAO introduces additional training overhead due to experience buffer



(a) IMDB *Volume Switch*. **(b) TPC-H *Workload Switch*.**

(c) IMDB Average and MIN Running Time. **(d) TPC-H Average and MIN Running Time.**

Figure 12: Accumulated, average, and minimum execution performance of LIMAO-Balsa variations.

Table 6: Number of time-outs in different situations.

Database	Environment	Balsa	LIMAO-Balsa
IMDB	*Static Workload*	54	**5**
TPC-H	*Static Workload*	127	**0**
IMDB	*Workload Switch*	1128	**120**
TPC-H	*Workload Switch*	230	**26**
IMDB	*Volume Switch*	67	**2**
TPC-H	*Volume Switch*	949	**0**
IMDB	*Both Switch*	605	**120**
TPC-H	*Both Switch*	322	**98**

replay and episodic updates. Although this adds cost per iteration, the overall training time remains competitive due to faster convergence. With access to high-performance GPUs such as the NVIDIA 4090 or A100, this overhead is further mitigated and does not hinder practical deployment. In our IMDB experiments, episodic training in LIMAO-Balsa takes approximately 3 seconds per episode, with up to 10 episodes per iteration. The post-iteration training phase in both Balsa and LIMAO-Balsa (including buffer replay) ranges between 4 and 9 seconds per iteration, depending on the workload (e.g., JOB, CEB, BaoQs). While LIMAO-Balsa incurs slightly higher per-iteration training costs, it converges significantly faster, typically within 30 iterations (as shown in Figure 11)—resulting in a total training overhead of 160–310 seconds. In contrast, Balsa often requires at least 50 iterations, leading to a total overhead of 200–450 seconds. Thus, despite the added per-iteration cost, LIMAO-Balsa achieves lower overall training time. For simpler workloads like TPC-H, the training overhead becomes minimal for both systems.

10 RELATED WORK

Learned Query Optimizers (LQOs). Existing LQOs like Bao [36], LEON [9], and Lero [70], leverage neural networks to refine and enhance query plans generated by traditional optimizers. It minimally interferes with the underlying optimizer logic, reducing the risk of generating bad plans. However, these LQOs remain capped with the recommendations of traditional optimizers. On the other hand, LQOs like Balsa [66], Neo [37], and Lemo [45] control the plan generation process, allowing them to explore a broader range of execution strategies. While this leads to the discovery of superior plans, it also introduces greater variability and potential instability in performance. LIMAO can be integrated with both categories of LQOs and allow their LCPs to operate in a lifelong setup, improving their training stability, and ensuring more consistent performance across varying workloads and data distributions.

Query Execution in Dynamic Environments. Bao [36] and HybridQO [69] evaluate their performance under dynamic workloads, data, and schema by periodically introducing new query templates, data updates, and schema normalization. Similarly, Lero [70], LEON [9], and ERASER [61] test their models by incrementally adding new data to the database. However, they provide limited insights into the LQOs' performance stability, as the updates are neither extensive nor frequent enough to fully capture the complexities of real-world, fluctuating environments. In contrast, LIMAO introduces a comprehensive benchmarking framework for dynamic scenarios, simulating more intricate changes and showing how it can enhance performance stability. Meanwhile, recent research in learned cardinality estimation has addressed challenges related to workload drifts during query execution by partially masking query information to promote generalization [48], utilizing rapid re-training with replay buffers [63], and applying attention mechanisms to capture relationships between queries and dynamic underlying data [34]. Conversely, LIMAO takes a more holistic approach, focusing on learned cost prediction for entire query plans.

11 CONCLUSION AND FUTURE WORK

In this paper, we introduce LIMAO, the first modular lifelong learning framework for query optimization designed to handle significant and frequent shifts in workloads and data distributions. LIMAO provides a lifelong learned plan cost predictor that can adapt over time while retaining and leveraging previously acquired knowledge to ensure stable performance. LIMAO has several contributions, including a modular plan decomposition, an attention-based neural composition, and an efficient two-phase training approach. We integrated LIMAO with two LQOs, and evaluated them with IMDB and TPC-H queries under dynamic situations. Our evaluation shows that LIMAO can improve query speed up to 40% and the variance in running times of the same query by 60% for IMDB. For TPC-H, LIMAO achieves more than 2× speedup and 100× query stability gain. For future work, we plan to 1) integrate LIMAO with additional LQOs, 2) automatically identify workload-customized break operators for query decomposition and sizes for module hubs, and 3) explore knowledge transfer across different databases. In this study, we only focused on query optimization, a task with inherent modularity that aligns well with the principles of lifelong modular learning. However, we believe that our proposed framework can be extended to other learned database tasks, such as knob tuning [31, 58] and learned indexes [14, 30], where modular representations and continual adaptation are also essential.

ACKNOWLEDGMENTS

This work is partially funded by Google Data to Insights and Google ML and Systems Junior Faculty research awards.

REFERENCES

[1] Ahmad Abbasi, Abdul Rehman Javed, Chinmay Chakraborty, Jamel Nebhen, Wisha Zehra, and Zunera Jalil. 2021. ElStream: An ensemble learning approach for concept drift detection in dynamic social big data stream learning. *IEEE Access* 9 (2021), 66408–66419.

[2] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural Module Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 39–48.

[3] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural Module Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 39–48. https://doi.org/10.1109/CVPR.2016.12

[4] Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The option-critic architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1726–1734.

[5] Ishwar Baidari and Nagaraj Honnikoll. 2021. Bhattacharyya distance based concept drift detection method for evolving data stream. *Expert Systems with Applications* 183 (2021), 115303.

[6] Yoshua Bengio. 2012. Deep Learning of Representations for Unsupervised and Transfer Learning. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*.

[7] Gianni Brauwers and Flavius Frasincar. 2021. A general survey on attention mechanisms in deep learning. *IEEE Transactions on Knowledge and Data Engineering* 35, 4 (2021), 3279–3298.

[8] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans. *Proc. VLDB Endow.* 16, 7 (March 2023), 1777–1789.

[9] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. Leon: A new framework for ml-aided query optimization. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2261–2273.

[10] Xue Chen, Rui Xi, Yiheng Tang, and Mengshu Hou. 2023. Robust Listwise Learning-to-Rank Approach for Database Query Optimizer. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. 1165–1172.

[11] Transaction Processing Performance Council(TPC). 2021. TPC-H Vesion 2 and Version 3. http://www.tpc.org/tpch/

[12] Alana de Santana Correia and Esther Luna Colombini. 2022. Attention, please! A survey of neural attention models in deep learning. *Artificial Intelligence Review* 55, 8 (2022), 6037–6124.

[13] Coline Devin, Abhishek Gupta, Trevor Darrell, Pieter Abbeel, and Sergey Levine. 2017. Learning modular neural network policies for multi-task and multi-robot transfer. In *ICRA*. 2169–2176.

[14] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 969–984.

[15] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Parametric Query Optimization. In *SIGMOD*.

[16] Quang-Huy Duong, Heri Ramampiaro, Kjetil Nørvåg, Philippe Fournier-Viger, and Thu-Lan Dam. 2018. High utility drift detection in quantitative data streams. *Knowledge-Based Systems* 157 (2018), 34–51.

[17] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently approximating selectivity functions using low overhead regression models. *Proc. VLDB Endow.* 13, 12 (July 2020), 2215–2228. https://doi.org/10.14778/3407790.3407820

[18] Anirudh Goyal, Alex Lamb, Jordan Hoffmann, Shagun Sodhani, Sergey Levine, Yoshua Bengio, and Bernhard Schölkopf. 2021. Recurrent Independent Mechanisms. In *International Conference on Learning Representations*.

[19] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[20] Goetz Graefe and William J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. 209–218.

[21] Meng-Hao Guo, Tian-Xing Xu, Jiang-Jiang Liu, Zheng-Ning Liu, Peng-Tao Jiang, Tai-Jiang Mu, Song-Hai Zhang, Ralph R Martin, Ming-Ming Cheng, and Shi-Min Hu. 2022. Attention mechanisms in computer vision: A survey. *Computational visual media* 8, 3 (2022), 331–368.

[22] Benjamin Hilprecht and Carsten Binnig. 2022. One Model to Rule them All: Towards Zero-Shot Learning for Databases. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*.

[23] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *Proc. VLDB Endow.* 15, 11 (July 2022), 2361–2374.

[24] Songqiao Hu, Zeyi Liu, and Xiao He. 2022. CADM: Confusion model-based detection method for real-drift in chunk data stream. In *International Conference on Sensor Systems and Software*. Springer, 191–201.

[25] Zhexue Huang. 1997. Clustering Large Data Sets With Mixed Numeric And Categorical Values,". *Proceedings Of 1st Pacific-Asia Conference on Knowledge Discovery And Data Mining Conference*, 21–34.

[26] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L. Hayes, and Christopher Kanan. 2018. Measuring catastrophic forgetting in neural networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence* (New Orleans, Louisiana, USA) *(AAAI'18/IAAI'18/EAAI'18)*. AAAI Press, Article 415, 9 pages.

[27] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study. In *SIGMOD*. Association for Computing Machinery, 1214–1227.

[28] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.

[29] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3521–3526.

[30] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[31] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (2024), 1939–1952.

[32] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[33] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently adapting learned cardinality estimators to data and workload drifts. In *Proceedings of the 2022 International Conference on Management of Data*. 1920–1933.

[34] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *VLDB* 17, 2 (Oct. 2023), 197–210.

[35] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. 2022. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems* 4 (2022), 77–94.

[36] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *SIGMOD*. 1275–1288.

[37] Ryan 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 (jul 2019), 1705–1718.

[38] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.* 12, 11 (July 2019), 1733–1746.

[39] Jorge A. Mendez and Eric Eaton. 2023. How to Reuse and Compose Knowledge for a Lifetime of Tasks: A Survey on Continual Learning and Functional Composition. *Trans. Mach. Learn. Res.* (2023).

[40] Jorge A. Mendez, Harm van Seijen, and Eric Eaton. 2022. Modular Lifelong Reinforcement Learning via Neural Composition. In *ICLR*.

[41] Jorge Mendez Mendez. 2023. *Embodied lifelong learning for decision making: Opportunities brought on by modularity*.

[42] Jorge Mendez-Mendez, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. 2023. Embodied Lifelong Learning for Task and Motion Planning. In *Proceedings of The 7th Conference on Robot Learning (Proceedings of Machine Learning Research)*, Jie Tan, Marc Toussaint, and Kourosh Darvish (Eds.), Vol. 229. PMLR, 2134–2150.

[43] Yabin Meng, Paul Bird, Pat Martin, and Wendy Powley. 2007. An approach to managing the execution of large SQL queries. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. 268–271.

[44] Artem Mikhaylov, Nina S Mazyavkina, Mikhail Salnikov, Ilya Trofimov, Fu Qiang, and Evgeny Burnaev. 2022. Learned Query Optimizers: Evaluation and Improvement. *IEEE Access* 10 (2022), 75205–75218.

[45] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[46] 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 Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 1287–1293.

[47] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2019–2032.

[48] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust query driven cardinality estimation under changing workloads. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1520–1533.

[49] PostgreSQL Global Development Group. [n.d.]. https://www.postgresql.org/.

[50] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *login*

Usenix Mag. 39, 6 (2014), 36–38.

[51] Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. 2017. Routing Networks: Adaptive Selection of Non-linear Functions for Multi-Task Learning. *ICLR* abs/1711.01239 (2017).

[52] Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. 2018. Routing Networks: Adaptive Selection of Non-Linear Functions for Multi-Task Learning. In *International Conference on Learning Representations*.

[53] Paul Ruvolo and Eric Eaton. 2013. ELLA: an efficient lifelong learning algorithm. In *ICML*. JMLR.org, I–507–I–515.

[54] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *SIGMOD*. ACM, 1228–1242.

[55] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[56] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 307–319.

[57] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and a Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (sep 2021), 85–97.

[58] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.

[59] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *Proc. VLDB Endow.* 4, 11 (Aug. 2011), 852–863.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*. Curran Associates Inc., 6000–6010.

[61] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2024. Eraser: Eliminating performance regression on learned query optimizer. *Proceedings of the VLDB Endowment* 17, 5 (2024), 926–938.

[62] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD* (Virtual Event, China). Association for Computing Machinery, 2009–2022.

[63] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proc. ACM Manag. Data* 2, 1, Article 38 (March 2024), 27 pages.

[64] Ruihan Yang, Huazhe Xu, Yi Wu, and Xiaolong Wang. 2020. Multi-task reinforcement learning with soft modularization. In *NIPS*. Curran Associates Inc., Article 400, 11 pages.

[65] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 931–944.

[66] 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 2022 International Conference on Management of Data*. 931–944.

[67] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 61–73.

[68] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (nov 2019), 279–292.

[69] 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 (Sept. 2022), 3924–3936.

[70] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A learning-to-rank query optimizer. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1466–1479.

[71] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: fast, lightweight and accurate method for cardinality estimation. *Proc. VLDB Endow.* 14, 9 (May 2021), 1489–1502.