Athena: An Effective Learning-based Framework for Query Optimizer Performance Improvement

RUNZHONG LI*, Southern University of Science and Technology, China and The Hong Kong Polytechnic University, Hong Kong
QILONG LI*, Southern University of Science and Technology, China
HAOTIAN LIU, Southern University of Science and Technology, China
RUI MAO, Shenzhen University, China
QING LI†, The Hong Kong Polytechnic University, Hong Kong
BO TANG†, Southern University of Science and Technology, China

Recent studies have made it possible to integrate learning techniques into database systems for practical utilization. In particular, the state-of-the-art studies hook the conventional query optimizer to explore multiple execution plan candidates, then choose the optimal one with a learned model. This framework simplifies the integration of learning techniques into the database system. However, these methods still have room for improvement due to their limited plan exploration space and ineffective learning from execution plans. In this work, we propose Athena, an effective learning-based framework of query optimizer enhancer. It consists of three key components: (i) an order-centric plan explorer, (ii) a Tree-Mamba plan comparator and (iii) a time-weighted loss function. We implement Athena on top of the open-source database PostgreSQL and demonstrate its superiority via extensive experiments. Specifically, We achieve 1.75x, 1.95x, 5.69x, and 2.74x speedups over the vanilla PostgreSQL on the JOB, STATS-CEB, TPC-DS, and DSB benchmarks, respectively. Athena is 1.74x, 1.87x, 1.66x, and 2.28x faster than the state-of-the-art competitor Lero on these benchmarks. Additionally, Athena is open-sourced and it can be easily adapted to other relational database systems as all these proposed techniques in Athena are generic.

CCS Concepts: • **Information systems → Query optimization**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Learning-based Query Optimization, Learned Optimizer Enhancer

ACM Reference Format:
Runzhong Li, Qilong Li, Haotian Liu, Rui Mao, Qing Li, and Bo Tang. 2025. Athena: An Effective Learning-based Framework for Query Optimizer Performance Improvement. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 132 (June 2025), 24 pages. https://doi.org/10.1145/3725395

*Both authors contributed equally to this work.
†Corresponding authors: Prof. Bo Tang and Prof. Qing Li.

Authors' Contact Information: Runzhong Li, lirunzhong2021@mail.sustech.edu.cn, Southern University of Science and Technology, Shenzhen, China and The Hong Kong Polytechnic University, Hong Kong; Qilong Li, liql2022@mail.sustech.edu.cn, Southern University of Science and Technology, Shenzhen, China; Haotian Liu, liuht2022@mail.sustech.edu.cn, Southern University of Science and Technology, Shenzhen, China; Rui Mao, mao@szu.edu.cn, Shenzhen University, Shenzhen, China; Qing Li, csqli@comp.polyu.edu.hk, The Hong Kong Polytechnic University, Hong Kong; Bo Tang, tangb3@sustech.edu.cn, Southern University of Science and Technology, Shenzhen, China.

ACM 2836-6573/2025/6-ART132
https://doi.org/10.1145/3725395

Table 1. Summary of learned optimizer enhancer

Method	Plan Space		Representation Method	Training Objective	
	Direct	Diversity	Global structure	Time	Rank
Bao	No	Medium	No	Yes	No
Lero	No	Low	No	No	Yes
Athena	**Yes**	**High**	**Yes**	**Yes**	**Yes**

1 Introduction

Query optimizer is one of the core components in database management systems (DBMSs) for efficient query processing. The conventional cost-based query optimizer [21] consists of three core subroutines: (i) plan enumerator, (ii) cardinality estimator, and (iii) cost model. However, the performance of these query optimizers can be significantly improved as their cardinality estimator fails to capture the complex correlations among multiple tables, which results in inaccurate cardinality/cost estimations of the query plans.

In recent years, dozens of learning-based techniques have been utilized to improve the performance of the query optimizer from various aspects. We classify existing learning-based techniques into three categories by considering how the proposed techniques work. *(I) Learned estimator*: it improves the conventional query optimizer by utilizing learning-based models to provide more accurate estimates of intermediate cardinalities [4, 20, 22, 32, 40, 51, 55] or plan costs [29, 40]. *(II) Learned optimizer*: it supersedes conventional query optimizer in the DBMS via an end-to-end learned query optimizer, e.g., Neo [27], Balsa [50], and LOGER [6]. In particular, the learned optimizer consists of a search-based plan generator and a learned value network. *(III) Learned optimizer enhancer*: the general idea of the learned optimizer enhancer is that it generates multiple query plan candidates, which probably have good performance, then selects the best one among them via a learned model. It enhances the query optimization performance by utilizing both the existing query optimizer and learning-based models [26, 54].

In this work, we focus on the learned optimizer enhancer as it is more practical to improve the query optimization performance [26]. The major reasons are two-fold: (i) it combines both the advantages of conventional query optimizer and learned models. Thus, it not only improves the optimization performance but also is robust to the schema, data, and workload changes; (ii) it shows its superiority in terms of both tail performance and training cost over *learned estimator* and *learned optimizer* [26].

Bao [26] and Lero [54] are two representative solutions of learned optimizer enhancers. In particular, Bao [26] utilizes hints to generate more plan candidates and employs a Tree-CNN model to choose the best one to execute. Lero [54] generates more plan candidates by swinging the estimated cardinalities and selects the executed one via a learning-to-rank model. Even though both Bao and Lero use different approaches to enhance the performance of the query optimizer, their proposed solutions try to overcome three fundamental challenges, which are from three aspects of the learned optimizer enhancer framework: (i) plan exploration space, (ii) plan representation method, and (iii) model training objective. Subsequently, we first briefly elaborate on the existing solutions in each aspect of the framework, and then highlight the technical challenges in them.

Plan Exploration Space. The plan exploration space of a given query is the set of all its possible plan candidates. It is impractical to enumerate all of the plan candidates as the number of them is exponential to the number of joins and operator types of the query. Returning to the general idea of the learned optimizer enhancer, the first task of it is to generate multiple plan candidates for a given query from its plan exploration space. To achieve that, Bao uses hints to indicate the available operator types to generate various plan candidates. Lero swings the estimated cardinalities of

Table 2. The end-to-end latency (in seconds) on 4 benchmarks

Workload	JOB	STATS	TPC-DS	DSB
PostgreSQL	1666	8181	3773	1979
Bao	>40611	>14605	2089	2399
Lero	1661	7841	1106	1649
LOGER	1207	12095	–	–
Athena	**954**	**4185**	**663**	**722**

sub-plans to generate different plans. However, both approaches can be significantly improved as (i) they change the plan structure by affecting the join orders indirectly (e.g., hints or cardinalities adjustments) and (ii) the diversity of the generated plan candidates is limited as they do not explore different join orders explicitly, as summarized in Table 1. Thus, the first technical challenge of an ideal learned optimizer enhancer is how to generate a subset of plan candidates from plan exploration space with good diversity.

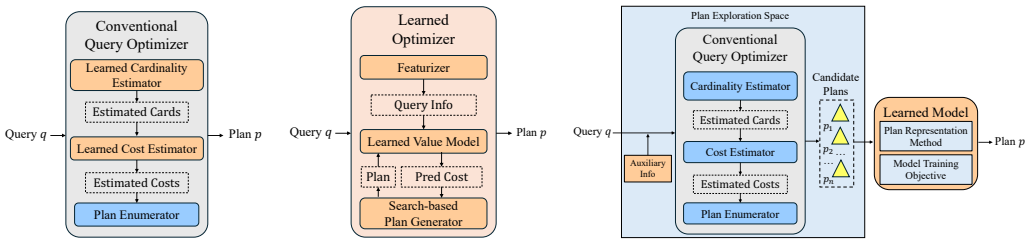
Suppose the above generated plan candidates are good. The second task of the learned optimizer enhancer is to select the best one among them for execution via learned models. Thus, the key to success of this task includes two parts: plan representation method and model training objective. We elaborate on them as follows.

Plan Representation Method. Every plan candidate is embedded into a hidden representation, i.e. a single vector that represents the plan, via the plan representation method. A good plan representation method should be able to capture both the local and global structural information of the candidate plan. Both Bao and Lero employ Tree-CNN [31] to embed the candidate plan. Tree-CNN excels at capturing local structural information. However, it integrates this local information using an approach that does not consider global structure, as shown in Table 1. Thus, the second technical challenge in the learned optimizer enhancer is to design an effective and efficient representation method that explicitly captures both local and global structural information of the query plan.

Model Training Objective. Given a good plan representation method, how to define the model training objective, i.e., the loss function, is essential to the learned optimizer enhancer as it affects the quality of the selected plan. Bao employs the mean squared error of the plan execution time as the loss function. Lero uses cross-entropy loss to learn the pair-wise rank of the plans as selecting the best one from a set of plans is finding the top-1 plan by ranking among them (see Table 1). However, both training objectives of Bao and Lero do not consider the influence of different samples on the overall performance of a workload. Hence, the third technical challenge is to propose a loss function whose training objective aligns well with the task goal, e.g. improving the query processing performance of the workload.

In this work, we propose Athena, an effective learning-based framework of query optimizer enhancer, to overcome the above three technical challenges. First of all, we devise an order-centric plan explorer to address the first challenge, which explicitly changes the root node of the join plan to generate a set of high-quality plan candidates. Moreover, the explorer can be seamlessly integrated into the conventional query optimizer and generate a set of plan candidates during a single traditional optimization procedure. For the second challenge, we propose Tree-Mamba, a new tree structure representation method to capture both local and global structures in a plan tree. Tree-Mamba includes a newly designed Tree-CNN and a carefully designed method to fuse states from child nodes. For the third challenge, we devise a time-weighted loss function, which both considers the exact execution time and leverages the learning-to-rank method.

We implement Athena on top of PostgreSQL [39]. It is open-sourced at [2]. Additionally, it is worth pointing out that Athena is generic and can be adapted to other RDBMSs, e.g. MySQL. Table 2



(a) Learned estimator (b) Learned optimizer (c) Learned optimizer enhancer

Fig. 1. The architectures of learning-based methods for query performance optimization

shows the performance of vanilla PostgreSQL, the representative learned optimizer (LOGER [6]), two learned optimizer enhancers (Bao [26] and Lero [54]), and Athena on four widely used benchmarks, i.e., JOB, STATS, TPC-DS, and DSB. Athena achieves up to 5.69x speedups over the vanilla PostgreSQL. Athena outperforms the learned optimizer enhancer competitors. In particular, it is up to 42.57x and 2.28x faster than Bao and Lero, respectively. Moreover, Athena is significantly better than the learned optimizer LOGER in all tested benchmarks. '-' indicates that LOGER does not support the complex queries (e.g., nested subqueries) in these benchmarks.

In summary, the technical contributions of Athena are:

- We propose a learning-based framework for the learned optimizer enhancer, and we architect Athena by addressing the challenges from the plan exploration space, the plan representation method, and the model training objective in it.
- We propose an order-centric plan explorer to generate high-quality plan candidates by explicitly exploring diverse join orders of each query.
- We introduce Tree-Mamba, a novel plan representation method that precisely captures both local and global information in execution plans.
- We devise a time-weighted loss function that incorporates both the latency and rank of plan candidates to improve the overall performance of workloads.
- We integrate Athena into the open-sourced database PostgreSQL, and conduct extensive experiments to demonstrate the superiority of Athena over the competitors.

The remainder of this paper is organized as follows. Section 2 summarizes related work, Section 3 overviews the architecture of Athena. Sections 4 to 6 introduce the details of our technical contributions in Athena. We present the experimental evaluation in Section 7, and conclude this work in Section 8.

2 Related Work

Figure 1 depicts the architectures of learning-based methods for query performance optimization. In this section, we briefly introduce the most relevant studies in three categories: (i) learned estimator, (ii) learned optimizer, and (iii) learned optimizer enhancer.

Learned Estimator. A learned estimator employs machine learning models and is trained on historical query execution data to deliver more accurate estimates, as shown in Figure 1(a). Specifically, it consists of two types as follows.

Learned Cardinality Estimator: The cardinality estimator in the conventional query optimizer is replaced by the learned cardinality estimator [7, 19, 38, 42, 47, 55], which can be divided into three classes according to the model training methodologies: query-driven, data-driven and hybrid methods. Query-driven estimators [10, 13, 20, 32, 33, 35, 38, 40, 42] treat cardinality estimation

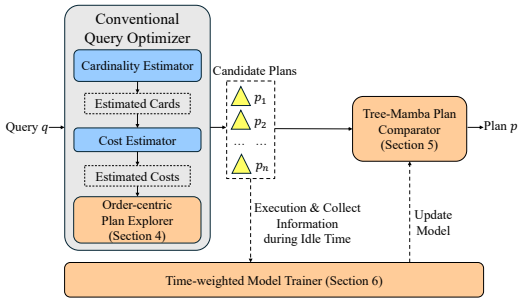


Fig. 2. System overview of Athena

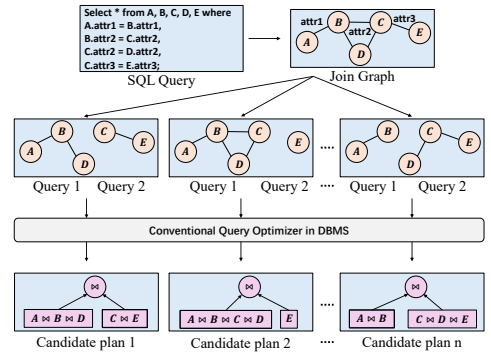


Fig. 3. Plan exploration procedure in Athena

as a regression problem and learn a mapping from query to cardinality. For example, MSCN [20] encodes a query into a feature vector and employs a multi-set convolution network to map it to cardinality. Data-driven estimators [14, 16, 19, 23, 24, 43, 46–48, 51, 52, 55] try to capture the correlation and distribution of data across multiple tables. Specifically, DeepDB [16] and FLAT [55] adopt the sum-product network (SPN) to model the joint probability distribution of the relations. Hybrid estimators [22, 44, 45] combine both query-driven and data-driven methods. For example, ALECE [22] utilizes a histogram to represent the data distribution of each column in the tables and utilizes a query-driven approach to learn the complex correlations among these columns.

Learned Cost Estimator: It predicts the cost of execution plans through learned models. Specifically, T-LSTM [40] employs a Tree-LSTM network to learn the execution time of executed plans. QPPNet [29] designs a neural network for each physical operator and combines them to estimate the final cost of a query plan.

Learned Optimizer. As illustrated in Figure 1(b), it supersedes conventional query optimizer by an end-to-end learned query optimizer [6, 27, 30, 50], which consists of a learned value model and search-based plan generator. Specifically, Neo [27] employs the reinforcement learning paradigm to learn to construct an optimized plan iteratively through a value network. However, Neo still relies on a conventional query optimizer to pre-train the value network. In contrast, Balsa [50] introduces a simple cost model to pre-train the value network, which makes it available for DBMSs without a query optimizer. LOGER [6] uses ϵ-greedy beam search to balance exploration and exploitation in reinforcement learning. Moreover, the conventional query optimizer has also been used to make several optimization decisions in LOGER.

Learned Optimizer Enhancer. Figure 1(c) depicts the architecture of learned optimizer enhancer [9, 26, 54]. The general idea is that it augments the conventional query optimizer by providing auxiliary information. Specifically, it generates multiple execution plan candidates from the plan exploration space and then selects the best one among them via learned models. As discussed in Section 1, both Bao [26] and Lero [54] are representative solutions in this category. In addition, Kepler [9] is a learned optimizer enhancer specifically designed for parametric query optimization.

3 Overview of Athena

In this work, we propose Athena, a novel and effective learned optimizer enhancer framework. Figure 2 depicts the system overview of Athena. Specifically, it generates multiple candidate plans $p_1, \cdots, p_n$ of a query q via the augmented conventional query optimizer, and then selects the best

plan p^* to execute via learned models. Athena achieves excellent performance by devising three key components: (i) order-centric plan explorer (in Section 4), (ii) Tree-Mamba plan comparator (in Section 5), and (iii) time-weighted model trainer (in Section 6). In particular, Athena is built upon the conventional query optimizer, i.e., with traditional cardinality estimator and cost estimator. The benefits are threefold: (i) the traditional cardinality/cost estimator enables us to explore diversified execution plans effectively; (ii) the traditional cardinality/cost estimator could easily adapt to schema, data, and workload changes; (iii) it can be easily integrated to existing DBMSs.

4 Order-centric Plan Explorer

In this section, we first present the existing methods to generate multiple plans and highlight their limitations in Section 4.1, then elaborate on our proposed approach in Athena, in Section 4.2.

4.1 Existing Methods

Existing learned optimizer enhancers utilize auxiliary information to generate multiple candidate plans from the plan exploration space. In particular, these methods change the join orders (or plan structure) and operator types. Bao [26] changes the operator types by providing a series of hints, which are boolean flags to enable or disable specific optimization rules. For example, if the native query optimizer initially generates a plan for the input query that uses the hash join algorithm, then Bao uses hints to disable the hash join. Thus, the native optimizer will yield an alternative plan that might use merge join. However, this approach may miss good candidate plans when different parts of a query can be optimized with different strategies. For example, if two sub-queries in a query should use two different join algorithms (e.g., one uses hash join and the other uses merge join), the coarse-grained hints in Bao cannot generate the above plan.

Lero [54] swings the estimated cardinalities of sub-queries to generate different candidate plans in its plan explorer. To avoid too many enumerations, it proposes a heuristic approach to explore the neighbors of the original plan. In particular, it applies the same multiplier to the cardinality estimates of all sub-queries that have the same number of tables in them. Considering a query that joins tables A, B, and C, the initial cardinality estimates of $A \bowtie B$ and $B \bowtie C$ are 3000 and 5000, respectively. When a multiplier of 2 is applied, the estimates will change to 6000 and 10000. Similarly, applying a multiplier of 1/10 would change the estimates to 300 and 500. Lero uses these different estimates to generate different candidate plans. However, the limitation of Lero is that it lacks the ability to generate diverse plan structures. In the above example, $B \bowtie C$ will never be generated because the changed estimate of $A \bowtie B$ is consistently smaller than that of $B \bowtie C$.

Kepler [9] uses an evolutionary algorithm to randomly perturb the cardinality estimates of the conventional optimizer. In particular, the initial generation of query plans in Kepler is generated by the conventional query optimizer in the DBMS. It then perturbs the cardinalities of sub-queries to generate the next generation of query plans. However, the perturbation method in the evolutionary algorithm may be inefficient with respect to new plan generation because it cannot guarantee that each perturbation will generate a new query plan. For example, Kepler generates only an average of 20 plans for each query after 10,000 random perturbations in JOB workload. Moreover, the cost of perturbations cannot be ignored as they invoke multi-pass optimization procedures in the traditional query optimizer. We will experimentally evaluate it in Section 7.

4.2 Our Proposed Method

To overcome the limitations of existing work and investigate diverse plan structures in the plan exploration space, we introduce an order-centric plan explorer in Athena, which is inspired by the key role that join order plays during query optimization in previous studies [6, 21, 27, 28, 36, 50]. In particular, the order-centric plan explorer focuses on exploring diverse join orders as much as

possible. However, evaluating all potential join orders through the brute force method is computationally prohibitive. The reason is that the number of possible join orders grows exponentially as the number of tables grows. For example, there are 3,628,800 possible join orders for a query with 10 tables. Thus, it is obviously impractical to enumerate all of them. We next describe the general idea of our proposed method via the illustrated example in Figure 3.

The intuition behind our order-centric plan explorer in Athena is to explicitly change the join orders to obtain a diverse set of candidate plans. For the input query, the join relationship among the tables can be represented by a graph, in which the nodes are the tables and the edges are the join conditions between the connected tables. For the given query, table A joins with table B on *attr1*, tables B, C, and D are joined on *attr2*, and tables C and E are joined on *attr3*, as the join graph in Figure 3 shows. Next, the join graph can be divided into two sub-graphs and each corresponds to a part of the original query. Thus, the executed result of the original query can be derived by joining the executed results of these two queries, where the join condition corresponds to that between the two sub-queries in the input query. For the candidate plan generation task in Athena, we employ the conventional query optimizer to generate the execution plans for these two queries, then connect these two obtained plans as a candidate plan for the input query, see Figure 3.

Through the above approach, we can directly change the structures of the candidate plans for the input query. The advantages of our proposed method are two-fold: (i) it can be seamlessly integrated into the conventional query optimizer as it does not incur extra overhead to generate the execution plan of the divided queries, as every divided query is a subquery of the original query enumerated by the conventional optimizer; and (ii) the diversity of the generated candidate plans by our order-centric plan explorer is obviously better than the existing methods in Bao, Lero, and Kepler as they change the join orders or the plan structures implicitly by auxiliary information (e.g., hints, cardinality adjustments).

Integration into the conventional query optimizer. Our order-centric plan explorer can be integrated into both top-down optimizer (e.g., TiDB [17], GreenPlum-Orca [25]) and bottom-up optimizer (e.g., System-R [3], PostgreSQL [39]).

In particular, the top-down optimizer begins with the entire query and recursively breaks it down into smaller sub-queries or operations. This approach applies heuristics and rules to optimize the query plan from the highest level down to the detailed operations. To integrate our plan explorer into a top-down optimizer, we can hook the intermediate results from rules related to join orders of the entire query. The bottom-up optimizer starts with the smallest components of the query, such as base relations, and builds up the query plan incrementally. It evaluates and combines these components step-by-step, considering various join orders and access methods to construct a query plan. To integrate our plan explorer into a bottom-up optimizer, we can hook the intermediate results when deciding the last join operation. By integrating directly with the existing optimizer, our method generates a set of multiple diverse candidate plans within a single pass of the traditional query optimization procedure. This results in faster exploration time, when compared with Bao and Lero as both of them trigger multi-pass query optimization procedures.

Discussion of join order enumeration. Enumerating the join order at other levels (e.g., bottom level) is also possible to generate candidate plans. However, Athena only enumerates the top-level (i.e., root node) as described above. The reason is that enumerating join order at top level affects the whole join order of the candidate plans. To verify that, we internally evaluate the benefits of join order enumeration at different levels. The experimental results confirm that enumerating the join order at the top level generates the highest quality candidate plans among these alternatives.

Discussion of hints assignment solution. One possible alternative solution to our proposed order-centric plan explorer is to replace it through a DBMS that supports subquery hints [1, 39], in

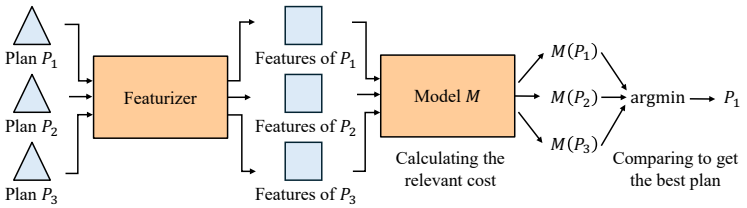


Fig. 4. Selecting procedure of candidate plans

which our above exploration heuristics could be used as hints. However, it is not easy to semantic equivalently convert our proposed heuristics to the hints, and use them in the DBMS. The core reason is that the supported hints in existing DBMSs should be specified in advance. For example, for the SQL query in Figure 3, the specified join order `/*+ JOIN_ORDER(A, B, C, D, E) */` can be used as hints in MySQL, and the specified nested join order `/*+ Leading(((A (B D)) (C E))) */` can be used as PostgreSQL hints. On the contrary, the proposed plan exploration heuristics are specifying the join order at the top level and ask the query optimizer to decide the join orders at other levels. In addition, our proposed order-centric plan explorer will be more efficient than the alternative hints assignment solution even if DBMS supports our heuristic hints as it needs to assign different hints to the optimizer and invokes multiple optimization procedures.

5 Tree-Mamba Plan Comparator

After generating execution plans via the proposed order-centric plan explorer, Athena selects the best plan from these candidates using learned models. The selection procedure is depicted in Figure 4. Initially, the generated plans P_1, P_2, and P_3 are input into the featurizer. The featurizer converts a query plan into a tree of vectors, where each vector is an encoded representation of the corresponding node. Subsequently, the tree of vectors will be used by a learned selective model M to predict their relative costs $M(P_1)$, $M(P_2)$, and $M(P_3)$. Finally, the plan with the lowest cost, i.e., P_1, is selected as the final execution plan. In the selective model, a plan representation network embeds the tree of vectors into a hidden representation of the plan. An MLP is used subsequently to predict the cost of the plan using its hidden representation.

The crux of the plan comparator is the plan representation network in the selective model M, while the featurizer plays a supportive role. The state-of-the-art learned optimizer enhancers use Tree-CNN [26, 54] to represent plans. Meanwhile, many learned estimators and learned optimizers use Tree-LSTM [6, 40, 53].

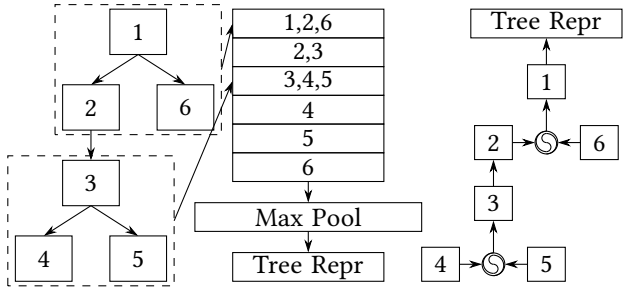
5.1 Existing Tree Representation Network

The tree representation network embeds a tree of feature vectors into a hidden representation for cost or cardinality estimation. In particular, the tree representation network can be classified into two categories: Tree-CNN and Tree-RNN.

Tree-CNN. Tree-CNN is a variant of CNN, which is designed to handle tree-structured data. The key designs of it are:

- **Tree Convolution Operations.** It convolutes the vectors of a parent node and its two children nodes. As depicted in Figure 5(a), node 1 and its two children, node 2 and 6, are convoluted into node 1. Given 3 vectors $\boldsymbol{x}_p$, $\boldsymbol{x}_l$ and $\boldsymbol{x}_r$ and learnable parameters $\boldsymbol{W}_p$, $\boldsymbol{W}_l$ and $\boldsymbol{W}_r$ of a convolution kernel, Tree-CNN can be formalized as:

$$\text{TreeCNN}(\boldsymbol{x}_p, \boldsymbol{x}_l, \boldsymbol{x}_r) = \boldsymbol{x}_p\boldsymbol{W}_p + \boldsymbol{x}_l\boldsymbol{W}_l + \boldsymbol{x}_r\boldsymbol{W}_r$$



(a) Tree-CNN procedure (b) Tree-RNN procedure

Fig. 5. Tree-CNN and Tree-RNN procedure

- **Pooling Operations.** In Tree-CNNs, max pooling is used to capture the global information in trees. It pools the maximum elements of vectors into a single vector, as depicted in Figure 5(a). Given n vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$ and their maximum elements $m_i = \max(x_{1i}, x_{2i}, \ldots, x_{ni})$ in dimension i, a MaxPooling layer can be defined as:

$$\text{MaxPooling}(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n) = (m_1, m_2, \ldots, m_d)$$

Tree-RNN. Tree-RNN is a variant of RNN, which is also designed to handle tree-structured data. RNN can handle sequential data by storing a hidden state h_i representing the history at each time step i. The hidden state h_{i-1} is updated by the input x_i at time step i.

$$h_i = \text{RNN}(x_i, h_{i-1})$$

Given a binary tree, each node is processed in a time step. The challenge is that each node may have at most 2 children, hence two hidden states $h_{i-1,l}$ and $h_{i-1,r}$ to be processed at each time step. These two states need to be merged as depicted in Figure 5(b). Different solutions to this challenge make different versions of Tree-RNN. The three most prevalent variants are listed as follows:

- **Mean Average.** The average of these two hidden states $h_{i-1} = (h_{i-1,l} + h_{i-1,r})/2$ is used.
- **Project.** The linear projection of these two hidden states $h_{i-1} = W(h_{i-1,l}, h_{i-1,r})$ are used, where W is a learnable parameter.
- **Modified Cell.** The RNN cell is modified to accept two hidden states, namely $h_i = \text{RNN}(x_i, h_{i-1,l}, h_{i-1,r})$.

As Figure 5(a) shows, in the Tree-CNN procedure, the local structural information of nodes 1, 2, and 6, as well as nodes 3, 4, and 5, is pooled into the tree representation. This is irrelevant to the structure of that tree and may lose global structure information. Tree-RNN iterates over the tree structure, as shown in Figure 5(b), which is more favorable for capturing the global structure. To present the plan tree accurately, the desired plan representation network should capture both local and global tree structures and store larger hidden states as the space of trees is larger than that of sequences. Recently, Mamba [11] uses a short causal CNN to improve the performance of its state space model (SSM), which is a linear RNN, except for its much larger hidden state. Specifically, the Mamba network (i) uses a linear RNN to capture the global sequence information, (ii) uses a causal CNN to capture the local sequence information, and (iii) has a larger hidden state than traditional RNN. Motivated by the nice properties of Mamba, we adapt Mamba to Tree-Mamba for the plan representation in Athena. However, it is not trivial to devise Tree-Mamba as it has three questions to be addressed.

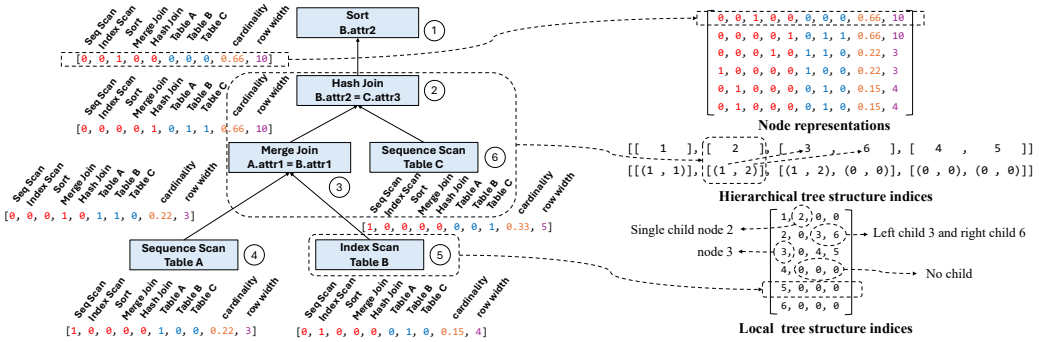


Fig. 6. Plan featurization in Athena

- How to encode nodes in a tree to fit the Tree-Mamba network?
- How to use Tree-CNN in Tree-Mamba?
- How to merge the large hidden states (w.r.t the traditional RNN) in Tree-Mamba?

To provide the answers to the above three questions in Athena, we present the proposed featurizer in Athena which generates the input of the selective model in Section 5.2, and introduce our Athena's Tree-Mamba plan representation network in Section 5.3.

5.2 Featurizer in Athena

It converts a plan tree to a set of node representations and tree structures indices, as illustrated in Figure 6.

Node Representation. The node encoding method converts each node in the execution plan into a vector. In particular, the encoded vector of each node in the execution plan has 3 parts.

- ***Operator***: One-hot of operator type. Taking node 1 as an example, the first 5 red numbers are [0, 0, 1, 0, 0]. It means that the operator type of node 1 is Sort.
- ***Tables***: Multi-hot of the included tables. Taking node 2 as an example, the 6th to 8th blue numbers are [0, 1, 1]. It means that tables B and C are included in the join condition of node 2. Taking node 4 as an example, the 6th to 8th blue numbers are [1, 0, 0]. It means that node 4 scans table A.
- ***Statistics***: Cardinality estimate and row width. Taking node 5 as an example, the 7th number [0.15] is the normalized cardinality estimate, and the 8th number [4] is the row width of table B.

The representation of a node in a query execution plan is the concatenation of the operator, the tables, and the statistics. Taking node 6 as an example, the encoding of its operator is [1, 0, 0, 0, 0]. The encoding of its included tables is [0, 0, 1]. The encoding of its statistics is [0.33, 5]. Concatenating all these 3 parts together, the representation of node 6 is [1, 0, 0, 0, 0, 0, 0, 1, 0.33, 5].

The node representation method described above basically follows the node representation method in Lero [54], except for a key difference. Specifically, our encoding of the operator and statistics in a node is the same as that of Lero, while our encoding of tables included in a node is different. Taking node 2 as an example, it joins (A, B) and C. Lero's encoding of tables included in node 2 is [1, 1, 1]. It means that A, B, and C are all included in the sub-plan rooted at node 2. If the join condition of node 2 is `A.attr3 = C.attr3`, Lero's encoding will still be the same [1, 1, 1]. Unlike Lero, Athena only encodes the tables used in the join condition, i.e. [0, 1, 1] for `B.attr2 = C.attr2` and [1, 0, 1] for `A.attr3 = C.attr3`.

Compared with Lero, we choose to encode the join condition because it is related to important properties, such as the index selection and the selectivity of the join. Though the encoding loses the information of the tables included in the sub-plan, our Tree-Mamba can compensate for this loss. Taking node 2 as an example, the lost information is already encoded in nodes 4, 5, and 6. And the information of included tables can be carried to node 2 by the hidden states in Tree-Mamba.

Tree Structure Indices. The structure encoding method converts the tree structures of the plan into structural indices, as depicted in Figure 6. These indices are utilized in the model to encode tree structure information, as will be discussed in detail in Section 5.3. The structural indices consist of two indices: (i) local tree structure indices and (ii) hierarchical tree structure indices.

Local structure indices, or local indices, are represented as a matrix $\mathcal{I}^l \in \mathbb{N}^{n\times 4}$. Each row of this matrix, containing four numbers, encodes a node by specifying the IDs of its children and itself. The first number in the row corresponds to the node's own ID, followed by the IDs of its children. If a node has only one child, the second number in its row corresponds to the child's ID. For nodes with two children, the third and fourth numbers represent the IDs of the left and right children, respectively. For example, in Figure 6, the second row of the matrix, [2, 0, 3, 6], corresponds to node 2. This indicates that node 2 has two children, i.e. nodes 3 and 6.

Hierarchical tree structure indices, referred to as hierarchical indices ($\mathcal{I}^h$), are represented as a list of matrices. The length of this list corresponds to the maximum level of the tree. For a given level l with n_l nodes, the l-th matrix $\mathcal{I}^h_l$ in $\mathcal{I}^h$ has a shape of $\mathbb{N}^{n_l\times 3}$. Each row in this matrix, composed of three numbers, encodes a node by specifying its own ID and the level IDs of its children. The level ID of a node corresponds to its position within its level, ordered from left to right. For example, the level IDs of nodes 4 and 6 are 1 and 2, respectively. This is because node 4 is the first node in the 4th level, while node 6 is the 2nd node in the 3rd level. Note that level IDs in different levels can be repeated. The first number in a row represents the node's own ID, followed by the level IDs of its children. For nodes with two children, the 2nd and 3rd numbers indicate the level IDs of the left and right children, respectively. In cases where a node has only one child, the 2nd and 3rd numbers in the row both correspond to the child's level ID.

Taking node 2 as an example, its two children, nodes 3 and 6, are the first and second nodes in the subsequent level. Consequently, the first row of the second matrix (corresponding to node 2, which is the first node in the second level) is represented as [2, 1, 2]. Here, the first number (2) denotes the node's own ID, while the second and third numbers (1 and 2) indicate the level IDs of its left and right children, respectively.

5.3 Tree-Mamba Architecture in Athena

Figure 7 depicts the structure of Tree-Mamba plan comparator, which takes a plan encoding as the input and predicts the cost of the plan. First, the node representations are passed through a linear layer (positioned below the Tree-Mamba network in Figure 7). The linear layer converts each node's representation into its corresponding hidden representation. Subsequently, the hidden representations are processed by the Tree-Mamba network, which utilizes the structural indices from the input. The Tree-Mamba network transforms them into the hidden representation of the input plan. Finally, the hidden representation of the plan is processed by the prediction network, resulting in the predicted cost, which is output as a scalar value.

Throughout this process, the most critical component is the Tree-Mamba network. It is composed of three sequential layers: a Mamba layer, followed by a Feed Forward layer, and another Mamba layer. Each Mamba and Feed Forward layer incorporates standard techniques widely adopted in modern deep learning, i.e. skip connections and layer normalization [49]. Specifically, the output

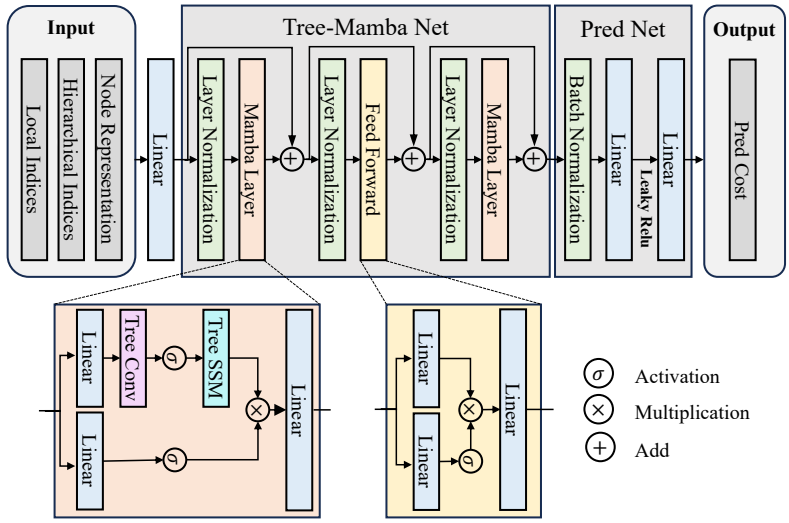


Fig. 7. Tree-Mamba structure in Athena

of each layer is computed as $\boldsymbol{y} = \boldsymbol{x} + \mathrm{F}(\mathrm{LayerNorm}(\boldsymbol{x}))$, where $\boldsymbol{x}$ represents the input to the layer, and F corresponds to either a Feed Forward or a Mamba layer.

The Feed Forward layer is a Gated Linear Unit (GLU [37]). Let $\boldsymbol{x} \in \mathbb{R}^{1\times d}$ represent the hidden representation of a node before entering the Feed Forward layer. The output of the layer is computed as follows: $\boldsymbol{y} = (\boldsymbol{x}\boldsymbol{W}_u \odot \mathrm{SiLU}(\boldsymbol{x}\boldsymbol{W}_v))\,\boldsymbol{W}_o$, where $\boldsymbol{W}_u, \boldsymbol{W}_v \in \mathbb{R}^{d\times d_i}$, and $\boldsymbol{W}_o \in \mathbb{R}^{d_i\times d}$ correspond to the three linear layers within the Feed Forward layer, as illustrated in Figure 7. Here, d_i is a model hyperparameter, $\odot$ denotes the element-wise product, and SiLU represents the Sigmoid Linear Unit activation function [15]. The use of the GLU in our Tree-Mamba is the same as the original Mamba.

The Mamba layer retains the same structure as the original Mamba architecture [11]. However, we introduce a novel Tree-Conv layer and a new Tree-SSM layer to replace the original Conv and SSM layer, enabling the processing of tree-structured data. Let $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ denote the hidden node representations before the Mamba layer, where n is the number of nodes and d is the hidden dimension. Additionally, let $\mathcal{I}^l$ represent the local indices and $\mathcal{I}^h$ represent the hierarchical indices. The output $\boldsymbol{Y}$ of the Mamba layer is then defined as follows:

$$\begin{aligned} \boldsymbol{X}_c &= \mathrm{SiLU}\left(\text{Tree-Conv}\left(\boldsymbol{X}\boldsymbol{W}_x, \mathcal{I}^l\right)\right) \\ \boldsymbol{X}_s &= \text{Tree-SSM}\left(\boldsymbol{X}_c, \mathcal{I}^h\right) \\ \boldsymbol{Y} &= \left(\boldsymbol{X}_s \odot \mathrm{SiLU}\left(\boldsymbol{X}\boldsymbol{W}_z\right)\right)\boldsymbol{W}_o, \end{aligned} \tag{1}$$

where $\boldsymbol{W}_x, \boldsymbol{W}_z \in \mathbb{R}^{d\times d_i}$, and $\boldsymbol{W}_o \in \mathbb{R}^{d_i\times d}$ represent the 3 linear layers within the Mamba layer, as depicted by Figure 7. Next, we will provide a detailed explanation of the Tree-Conv and Tree-SSM layers in Equation (1).

Tree-Conv Layer. The Tree-Conv layer applies convolution to the hidden representations of nodes $\boldsymbol{X} \in \mathbb{R}^{n\times d_i}$, guided by the local tree structure indices $\mathcal{I}^l \in \mathbb{R}^{n\times 4}$. Specifically, assuming $\boldsymbol{Y} = \text{Tree-Conv}\left(\boldsymbol{X}, \mathcal{I}^l\right)$, the i-th row of the output, $\boldsymbol{y}_i$, is computed as follows:

$$\boldsymbol{y}_i = \sum_{j=1}^{4} \boldsymbol{w}_j \odot \boldsymbol{x}_{\mathcal{I}^l_{i,j}}, \tag{2}$$

Algorithm 1 Tree-SSM Layer

1: **function** TREE-SSM($\boldsymbol{X} \in \mathbb{R}^{n \times d_i}$, $\mathcal{I}^h$)
2: $\quad \boldsymbol{S}_{\text{out}} \leftarrow$ []
3: $\quad l_{\max} \leftarrow$ the maximum level of the tree.
4: $\quad$ **for** $l \leftarrow l_{\max}$ to 1 **do**
5: $\quad\quad \boldsymbol{S}'_{\text{out}} \leftarrow$ []
6: $\quad\quad n_l \leftarrow$ the number of nodes in level l
7: $\quad\quad \mathcal{I} \leftarrow \mathcal{I}^h[l] \in \mathbb{R}^{n_l \times 3}$
8: $\quad\quad$ **for** $i \leftarrow 1$ to n_l **do**
9: $\quad\quad\quad \boldsymbol{x}_i \leftarrow \boldsymbol{X}[\mathcal{I}_{i,1}]$
10: $\quad\quad\quad \boldsymbol{S}_i \leftarrow \boldsymbol{S}_{\text{out}}[\mathcal{I}_{i,2}] \odot \boldsymbol{W} + \boldsymbol{S}_{\text{out}}[\mathcal{I}_{i,3}] \odot (1 - \boldsymbol{W})$
11: $\quad\quad\quad \boldsymbol{y}_{\mathcal{I}_{i,1}}, \boldsymbol{S}_i \leftarrow$ SSM-Step $(\boldsymbol{x}_i, \boldsymbol{S}_i)$
12: $\quad\quad\quad \boldsymbol{S}'_{\text{out}}.\mathsf{append}(\boldsymbol{S}_i)$
13: $\quad\quad$ **end for**
14: $\quad\quad \boldsymbol{S}_{\text{out}} \leftarrow \boldsymbol{S}'_{\text{out}}$
15: $\quad$ **end for**
16: $\quad Y \leftarrow [\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_n]$
17: $\quad$ **return** Y
18: **end function**

where $\boldsymbol{w}_{1\ldots4} \in \mathbb{R}^{d_i}$ are four learnable convolution kernels. Note that $\boldsymbol{x}_{\mathcal{I}^l_{i,j}} = \boldsymbol{0}_{d_i}$ if $\mathcal{I}^l_{i,j} = 0$ in the above equation.

There are 2 key differences between the Tree-Conv layer described in Equation 2 and the traditional Tree-CNN outlined in Section 5.1. First, the Tree-Conv layer eliminates the matrix multiplication required in traditional Tree-CNN, which reduces the number of parameters and accelerates computation. Second, the Tree-Conv layer employs 4 kernels instead of the 3 kernels used in traditional Tree-CNN. In traditional Tree-CNN, a single child of a node uses the same 2nd kernel as the left child of a join node. In contrast, our Tree-Conv layer assigns the 2nd kernel to a single child, while the left and right children of a join node use the 3rd and 4th kernels, respectively. This design allows the Tree-Conv layer to better distinguish between join nodes and other nodes.

Tree-SSM Layer. Algorithm 1 depicts the processing procedure of the Tree-SSM layer. In particular, it processes each level of the tree sequentially, starting from the bottom and moving upward to the root node, as described in the for loop spanning Lines 4 to 15 in Algorithm 1. For each level, the layer retrieves and merges the hidden states from the children of each node within that level, then performs an SSM step to update the hidden states of the node, as outlined in the for loop from Lines 8 to 13 in Algorithm 1.

Its core component is the newly proposed learned average layer (see Line 10 in Algorithm 1), which merges two states from the children of a node without performing matrix multiplication. Additionally, if a node has only one child, the learned average layer performs no operation, as $\boldsymbol{W} + 1 - \boldsymbol{W} = 1$, where $\boldsymbol{W} \in \mathcal{R}^{d_i \times d_s}$ is a learnable weight with the same shape as the hidden state, and d_i, and d_s are two hyper-parameters. After merging the states, an SSM step is applied to update the state (see line 11 in Algorithm 1), which is consistent with the SSM step used in the original Mamba [11].

Returning to the plan in Figure 6, the layer first processes nodes 4 and 5, followed by nodes 3 and 6. Next, node 2 is processed, and finally, the root node (node 1) is addressed. Specifically, when

processing node 3 at the third level, the Tree-SSM layer retrieves the hidden states of its children, nodes 4 and 5, and merges them to form the hidden state input for node 3.

Let $\boldsymbol{X} \in \mathbb{R}^{n \times d_i}$ represent the hidden representations of nodes before the Tree-SSM layer, and let $\mathcal{I}^h$ denote the hierarchical indices. The Tree-SSM layer is detailed in Algorithm 1. Notably, in Line 10 of Algorithm 1, $\boldsymbol{S}_{\text{out}}[\mathcal{I}_{i,j}] = \boldsymbol{0}$ if $\mathcal{I}_{i,j} = 0$, where j can be 2 or 3. If $\boldsymbol{S}_{\text{out}}$ is empty, the only valid index is zero.

6 Time-weighted Model Trainer

The model proposed in Section 5 will be trained with the collected execution latency. We denote the execution latency of a plan p as $L(p)$. For a certain workload, the purpose of the training is to reduce the overall execution latency of plans chosen by the model. In this section, we first introduce the existing solutions and their limitations in Section 6.1. Then, we elaborate on the implementation of our time-weighted model trainer in Section 6.2.

6.1 Existing Methods

Bao[26] learns to predict execution latency. However, predicting the execution latency with inaccurate cardinality estimates from the conventional query optimizer is error-prone. Lero[54] argues that learning the accurate latency is impractical and unnecessary. Instead, it learns the rank of the plans of each query, which simplifies the task of selecting the optimal plan.

In particular, Lero adopts the pair-wise learning-to-rank loss function proposed by RankNet[5]. Its loss function can be stated as follows: Given a pair of plans (p_1, p_2) where $L(p_1) > L(p_2)$ from a query q, the predicted costs are $M(p_1)$ and $M(p_2)$. The loss function of the pair (p_1, p_2) is the cross entropy loss l

$$l = -\log P(p_1 \triangleright p_2) \tag{3}$$

where $P(p_1 \triangleright p_2)$ is the predicted probability that $L(p_1) > L(p_2)$, which is calculated by $M(p_1)$ and $M(p_2)$. Specifically,

$$\begin{aligned} P(p_1 \triangleright p_2) &= \text{sigmoid}\left(M\left(p_1\right) - M\left(p_2\right)\right) \\ P(p_1 \triangleleft p_2) &= \text{sigmoid}\left(M\left(p_2\right) - M\left(p_1\right)\right) \\ &= 1 - P(p_1 \triangleright p_2) \end{aligned} \tag{4}$$

Minimizing this loss function will maximize the probability of choosing the better plan p_2. However, it mismatches the general purpose to optimize the overall execution latency of a workload. In particular, it treats all pairs equally, ignoring the latency difference between pairs.

6.2 Time-weighted TaiLr Loss in Athena

To address the limitations of existing methods, we propose a time-weighted loss function, which is based on the overall latency. The overall latency over a batch $\mathcal{B}$ composed of n pairs (p_1^i, p_2^i) is

$$p^i = \arg \min_{p \in \{p_1^i, p_2^i\}} M(p) \tag{5}$$

$$L(\mathcal{B}) = \sum_{i=1}^{n} L(p^i) \tag{6}$$

However, $L(\mathcal{B})$ in Equation 6 uses $\arg\min(\cdot)$ and is not derivable. To get a derivable approximation of Equation 6, we leverage the $\text{softmax}(\cdot)$ as an approximation of $\text{onehot}(\arg\max(\cdot))$. Then, $L(p^i)$ and can be approximated as:

$$\begin{aligned} L(p^i) &= \left(L(p_1^i), L(p_2^i)\right) \cdot \text{onehot}\left(\arg\min\left(M(p_1^i), M(p_2^i)\right)\right) \\ &\approx \left(L(p_1^i), L(p_2^i)\right) \cdot \text{softmax}\left(M(p_2^i), M(p_1^i)\right) \\ &= \frac{\exp\left(M(p_2^i)\right) L(p_1^i) + \exp\left(M(p_1^i)\right) L(p_2^i)}{\exp\left(M(p_2^i)\right) + \exp\left(M(p_1^i)\right)} \\ &= P(p_1^i \triangleleft p_2^i) L(p_1^i) + P(p_1^i \triangleright p_2^i) L(p_2^i) \end{aligned} \tag{7}$$

where the onehot(arg min(·)) and the softmax(·) can be stated as:

$$\begin{aligned} \text{onehot}(\arg\min(a, b)) &= \begin{cases} (1, 0) & \text{if } a < b \\ (0, 1) & \text{if } a \geq b \end{cases} \\ \text{softmax}(a, b) &= \left(\frac{\exp(a)}{\exp(a) + \exp(b)}, \frac{\exp(b)}{\exp(a) + \exp(b)}\right) \end{aligned} \tag{8}$$

Interestingly, the approximated latency of p^i is also its expectation. By substituting Equation 7 into Equation 6, we can get a differential approximation of $L(\mathcal{B})$:

$$\begin{aligned} L(\mathcal{B}) &\approx \sum_{i=1}^{n} \left(P(p_1^i \triangleleft p_2^i) L(p_1^i) + P(p_1^i \triangleright p_2^i) L(p_2^i)\right) \\ &= \sum_{i=1}^{n} L(p_1^i) - \sum_{i=1}^{n} \left(L(p_1^i) - L(p_2^i)\right) P(p_1^i \triangleright p_2^i) \\ &= k + L'(\mathcal{B}) \end{aligned} \tag{9}$$

In Equation 9, the term $k \equiv \sum_{i=1}^{n} L(p_1^i)$ represents a constant that is not subject to optimization. Consequently, the objective reduces to minimizing the loss function $L'(\mathcal{B}) \equiv -\sum_{i=1}^{n} \left(L(p_1^i) - L(p_2^i)\right) P(p_1^i \triangleright p_2^i)$. Returning to Equation 3, to maximize $P(p_1 \triangleright p_2)$, the loss function is conventionally defined as the cross-entropy loss, $l = -\log P(p_1 \triangleright p_2)$. Following this principle, we similarly adopt the cross-entropy formulation to optimize $L'(\mathcal{B})$. Specifically, the loss function can be expressed as:

$$\mathcal{L}(\mathcal{B}; \tau) = -\frac{\sum_{i=1}^{n} w_i^{\frac{1}{\tau}} \log P(p_1 \triangleright p_2)}{\sum_{i=1}^{n} w_i^{\frac{1}{\tau}}} \tag{10}$$

where $\tau \in [1, \infty)$ is the temperature coefficient and the weight w_i is defined as follow

$$w_i = L(p_1^i) - L(p_2^i) \tag{11}$$

This weight brings the latency difference information of the two plans into the training procedure. For example, we have 3 pairs with latency $(10, 2)$, $(10, 1)$ and $(2, 1)$, this loss function makes the model focus more on the first two pairs, while potentially neglecting the third pair. In this way, the model will distinguish between plans with substantial differences in latency but can lead to confusion when comparing plans with similar latencies. To balance the influence of the time weights, we introduce the temperature coefficient τ. Specifically, when $\tau = 1$, the loss function mainly optimizes overall latency, whereas when $\tau = \infty$, it reduces to the standard cross-entropy loss, focusing on maximizing the accuracy of selecting the better plan.

Further Optimization. The equation $\mathcal{L}(\mathcal{B})$ matches $L'(\mathcal{B})$ closely in form, differing only in that it uses $\log P(p_1 \triangleright p_2)$ instead of $P(p_1 \triangleright p_2)$. The gradient of this log form $\nabla \log P = \nabla P / P$ helps the model to converge fast by scaling ∇P by a factor of $1/P$ when P is small. However, this also brings

the over-confidence problem. For example, when $P(p_1 \triangleright p_2)$ is close to zero while $L(p_1)$ is only slightly larger than $L(p_2)$, the gradient of $\log P(p_1 \triangleright p_2)$ will be large. Thus, $P(p_1 \triangleright p_2)$ will increase by a large scale, which is contrary to the fact that $L(p_1)$ is only slightly larger than $L(p_2)$. To solve this problem, we adopt the TaiLr loss proposed by [18]. By replacing the $\log P$ with $\text{TaiLr}(P)$ in Equation 10, we adopt the new loss function $\mathcal{L}'(\mathcal{B}; \tau, \gamma)$ as our final loss function:

$$\mathcal{L}'(\mathcal{B}; \tau, \gamma) = -\frac{\sum_{i=1}^{n} w_i^{\frac{1}{\tau}} \text{TaiLr}\left(P(p_1 \triangleright p_2); \gamma\right)}{\sum_{i=1}^{n} w_i^{\frac{1}{\tau}}} \tag{12}$$

$$\text{TaiLr}(P; \gamma) = \frac{\log(\gamma + (1-\gamma)P)}{1-\gamma} \tag{13}$$

where $\text{TaiLr}(P; \gamma)$ is an interpolation of P and $\log P$, and $\gamma \in [0, 1)$ is a hyper-parameter to balance between the fast convergence provided by $\log P$ and the alignment degree to $L'(\mathcal{B})$ provided by P. Specifically, when $\gamma = 0$, $\text{TaiLr}(P) = \log P$; whereas when $\gamma \to 1$, $\text{TaiLr}(P) \to P$.

7 Experimental Evaluation

In this section, we thoroughly evaluate the performance of Athena through comprehensive experimental studies.

7.1 Experimental Setup

Experimental Configuration. We deploy Athena on a system with an Intel(R) Xeon(R) Gold 5122 CPU (3.6 GHz, 16 cores), 480 GB of DDR4 RAM. Additionally, the machine is equipped with two NVIDIA TITAN Xp GPUs for model training and inference. Our experiments are conducted on Ubuntu Linux 20.04 LTS, with Python 3.10 and PyTorch 12.1. Athena is implemented in PostgreSQL 16.1, and parameter configuration is handled using PGTune [34]. In addition, we preload all database tables and indexes into memory before running the experiments to minimize performance fluctuations.

Compared Competitors. We compare our methods with (i) two state-of-the-art optimizer enhancers, Bao [26] and Lero [54]; (ii) the representative learned optimizer LOGER [6], as it outperforms other learned optimizers, such as Neo and Balsa; and (iii) the conventional query optimizer in PostgreSQL. Furthermore, we include the plan explorer proposed in Kepler [9] to evaluate the effectiveness of different plan exploration methods. The reported end-to-end latency measures the wall-clock time of the query processing in these systems. In particular, it includes (i) plan exploration time, (ii) model inference time, and (iii) plan execution time in all these learned optimizer enhancers.

Tested Benchmarks. We evaluate the performance of all systems on four widely used benchmarks, i.e., JOB [21], STATS [12], TPC-DS [41] and DSB [8]. We follow [26, 54] to extend their queries by generating additional queries from the templates of the original queries, as JOB and STATS only contain a limited number of queries. For training-testing split, we use the original queries as the training set and the extended queries as the testing set in both JOB and STATS. For TPC-DS and DSB, we generate a set of queries via their provided templates and split them into training set and testing set. Next, we introduce the details of each benchmark.

- **JOB [21].** The JOB benchmark consists of a workload of 33 query templates and 113 specific queries based on the IMDB database, which includes 21 tables related to movies. We take the

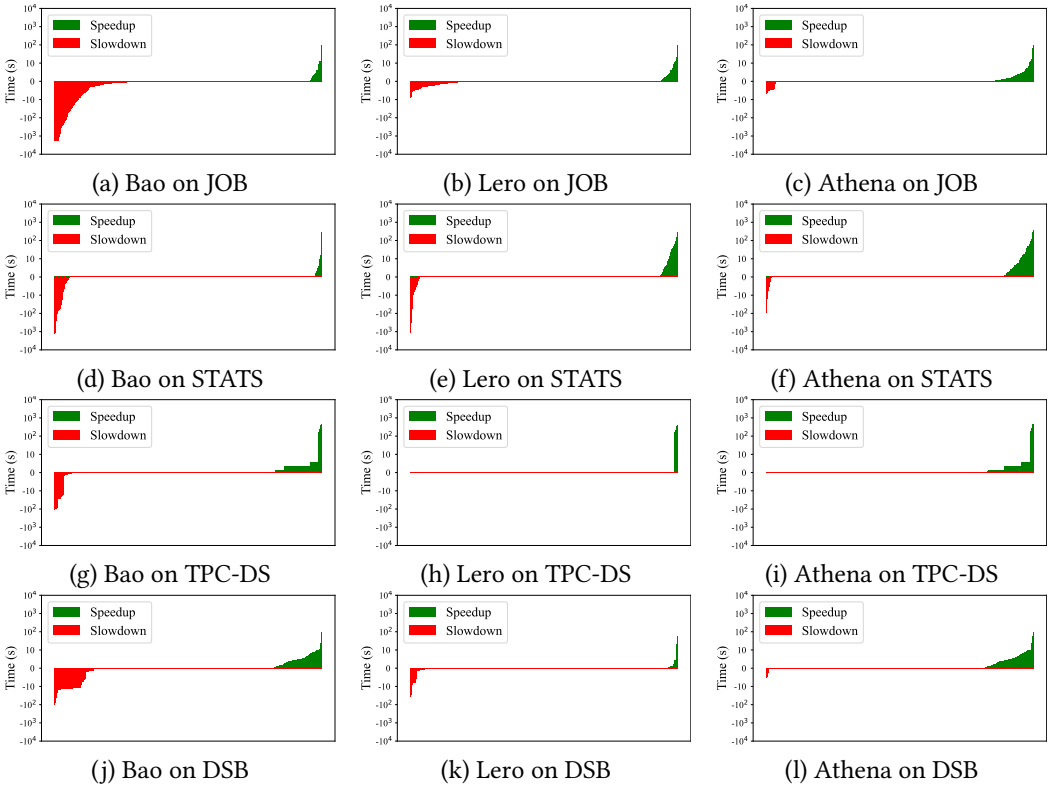


Fig. 8. Latency improvement and regression of each query in learned optimizer enhancers w.r.t PostgreSQL

original queries for model training and generate 983 more queries based on its 33 query templates for testing.

- **STATS [12].** The STATS benchmark consists of a dataset derived from user-contributed content on Stats Stack Exchange, encompassing 8 tables, and a workload STATS-CEB with 146 query templates, where each query template corresponds to a specific query. We take its original queries for model training and generate 1406 more queries for testing.
- **TPC-DS [41].** The TPC-DS benchmark includes 99 query templates for analytical processing. We exclude the templates containing views or sub-queries following the approach in [54], leaving 38 query templates. Then, we generate 125 queries for model training and 812 queries for testing based on these templates. In particular, the scale factor of the dataset is set to 10.
- **DSB [8].** The DSB benchmark is an extension of the TPC-DS benchmark with more complex data distributions and several additional query templates. We generate 137 queries based on its 41 query templates for model training and generate 891 queries for evaluation experiments. In particular, the scale factor of the dataset is set to 10.

7.2 Overall Performance

In this section, we verify the practical usability of our proposed Athena by comparing the end-to-end latency of each query processing among these learned optimizer enhancers w.r.t PostgreSQL.

Figure 8 illustrates the performance enhancement and regression of each query on four tested benchmarks with these evaluated learned optimizer enhancers (i.e., Bao, Lero, and Athena) w.r.t the corresponding end-to-end query execution latency in PostgreSQL. We omit cases where PostgreSQL

Table 3. The end-to-end latency (in seconds) of robustness evaluation on four benchmarks

Workload	JOB	STATS	TPC-DS	DSB
PostgreSQL	162	932	76	115
Bao	5751	1189	258	176
Lero	1328	917	52	172
LOGER	183	2057	–	–
Athena	**143**	**723**	**37**	**84**

chooses a similar plan (with a runtime difference of less than 1 second) to the learned optimizer enhancer. The key experimental observations are threefold.

Firstly, Athena obviously exhibits less regressing queries and more improved queries in each tested benchmark among all learned optimizer enhancers as it has fewer red bars and more green bars. In particular, the detailed statistics are as follows:

- Among 983 queries in JOB, there are 310 (31.5%) queries slowed down in Bao, 200 (20.3%) in Lero, and 36 (3.7%) in Athena; while 49 (5.0%) are accelerated in Bao, 66 (6.7%) in Lero and 161 (16.4%) in Athena.
- Among 1406 queries in STATS-CEB, there are 79 (5.6%) queries slowed down in Bao, 49 (3.5%) in Lero, and 26 (1.9%) in Athena; while 37 (2.6%) are accelerated in Bao, 93 (6.6%) in Lero and 161 (11.4%) in Athena.
- Among 812 queries in TPC-DS, there are 56 (6.9%) queries slowed down in Bao, 0 in Lero, and 0 in Athena; while 140 (17.2%) are accelerated in Bao, 10 (1.2%) in Lero and 140 (17.2%) in Athena.
- Among 891 queries in DSB, there are 132 (14.8%) queries slowed down in Bao, 51 (5.7%) in Lero, and 12(1.3%) in Athena; while 159 (17.8%) are accelerated in Bao, 29 (3.3%) in Lero and 164 (18.4%) in Athena.

Secondly, the proportion of regressing queries of Lero and Athena on TPC-DS is significantly smaller than that of the other three benchmarks, i.e., JOB, STATS and DSB. The core reason is the TPC-DS is a widely used synthetic benchmark and PostgreSQL could accurately estimate the cardinalities, which are utilized during the featurization process in both Lero and Athena.

Thirdly, in terms of the worst performance regressions of the tested queries among four benchmarks, there is no doubt that Athena performs much better than Bao and Lero. For example, the maximum slowdown for Athena in STATS-CEB is 85.46s. However, they are 1062.40s and 1199.87s in Lero and Bao, respectively.

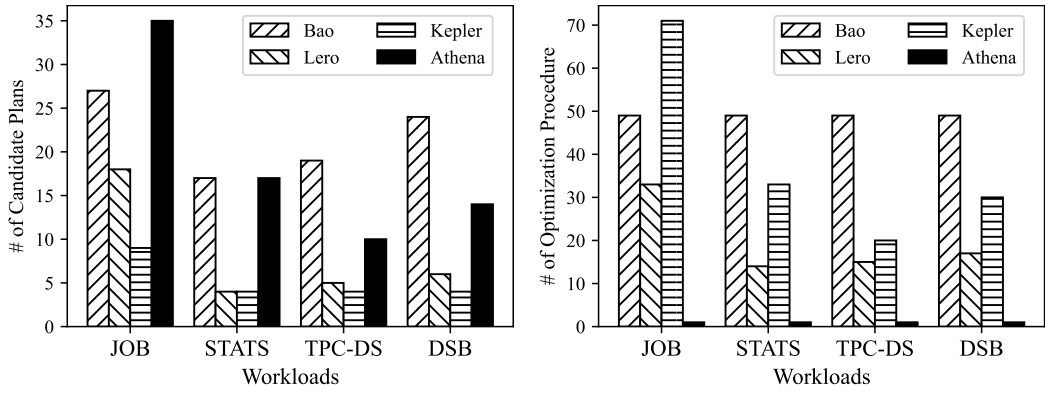
Robustness Evaluation of Athena. Table 2 (in Introduction) illustrates the cumulative end-to-end query latency of all compared systems on four tested benchmarks. Obviously, Athena is the overall winner as it performs the best among all these competitors, i.e., the state-of-the-art learned optimizer enhancers (Bao and Lero) and learned optimizer (LOGER), on every tested benchmark.

In this experiment, we further evaluate the robustness of Athena by splitting the training-testing sets across the query templates in all benchmarks. In particular, the queries of each benchmark are split into training set and testing set w.r.t the query templates, i.e., it guarantees the query templates of queries in training set and testing set are non-overlapped. It differs from the above experiments, where we generate an expanded query set with the query templates in each benchmark, and split these queries into training and testing sets.

We measured the cumulative end-to-end query latency of the queries in the testing set in each benchmark, and reported them in Table 3. It is worth pointing out, in this experiment, the numbers of queries are 54, 60, 56, and 60 in the testing set of JOB, STATS, TPC-DS, and DSB, respectively. However, there are 983, 1406, 812, and 891 queries in the corresponding testing set of these four

Table 4. End-to-end latency (in seconds) of three systems across four benchmarks using variable plan explorers

Plan Explorer	Explorer of Bao				Explorer of Lero				Explorer of Kepler				Explorer of Athena			
Benchmark	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB
Bao	>40611	>14605	2089	2399	1830	>15472	2509	11792	3580	**>10048**	3770	2620	**1109**	>13672	**883**	**1158**
Lero	1851	9830	766	923	1661	**7841**	1106	1649	3595	8932	3792	2617	**1191**	>22855	**688**	**733**
Athena	1911	**3343**	700	905	1624	6659	1135	1672	3576	8405	3774	2617	**954**	4185	**663**	**722**



(a) # of explored plans (b) # of optimization procedures

Fig. 9. Exploration efficiency of four solutions

benchmarks in Table 2. Interestingly, Athena outperforms all other competitors in every tested benchmark. It also is the only one which consistently outperforms vanilla PostgreSQL on all benchmarks. It confirms the robustness of Athena as the query templates of the queries in testing set are unseen before in this experiment. For example, both Bao and Lero perform even worse than the vanilla PostgreSQL with the same setting. The core reason is that Athena explores a wider candidate space with its order-centric explorer in Section 4 and the Tree-Mamba plan comparator proposed in Section 5 captures both local and global information to select a better plan.

7.3 Effectiveness Study

In this section, we evaluate the generality and effectiveness of each component in our proposed learned optimizer enhancer Athena.

7.3.1 Effect of Order-centric Explorer. In this section, we verify the generic and efficiency of order-centric plan explorer in Athena.

Effect of Plan Explorer. We substitute the plan explorers of Bao, Lero, and Athena with those of Bao, Lero, Kepler, and Athena, and keep the other components unchanged, then compare the end-to-end latency of executing all queries in each benchmark. Table 4 shows the experimental results of the three optimizer enhancers across the four benchmarks, where the best-performing explorer is highlighted in bold. The query will be terminated if it cannot return results within 30 minutes, ">" indicates it includes at least one auto-terminated query.

Specifically, the plan explorer of Athena achieves the best performance in 9 out of 12 experiments, while the explorers of Bao, Lero, and Kepler perform the best once, respectively. This is because the order-centric plan explorer in Athena explores a more diverse space in a more efficient manner. Subsequently, we will discuss the exploration efficiency and the quality of the explored plans for Bao, Lero, Kepler, and Athena.

Exploration Efficiency. We compare the number of optimization procedures triggered in PostgreSQL and the number of candidate plans generated by each explorer. Figures 9(a) and (b) present

Table 5. Cumulative execution time (in seconds) of the fastest plan of all queries for four benchmarks

Benchmark	JOB	STATS	TPC-DS	DSB
PostgreSQL	301.88	1613.78	1041.27	198.59
Kepler	294.05	1595.83	1112.40	194.38
Bao	185.19	1293.63	128.23	**95.31**
Lero	164.93	1351.47	261.41	180.55
Athena	**144.76**	**1284.74**	**119.52**	97.83

Table 6. The end-to-end latency (in seconds) of Athena with different models across four benchmarks

Plan Explorer	Explorer of Bao				Explorer of Lero				Explorer of Kepler				Explorer of Athena			
Benchmark	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB
Tree-Mamba	1911	**3343**	**700**	**905**	**1624**	**6659**	1135	1672	**3576**	8405	**3774**	**2617**	**954**	4185	**663**	722
Tree-CNN	1601	3509	729	912	1657	7752	**1107**	1810	3595	**8402**	3792	**2617**	988	7703	665	**710**
Tree-LSTM	**1578**	3427	795	923	1707	6997	1117	**1568**	3588	8411	3779	2622	1026	**3859**	686	714

the number of generated candidate plans and the average number of optimization procedures triggered by each explorer on four benchmarks, respectively. Combining both Figures 9(a) and (b), it is safe to conclude that Athena generates diversified candidate plans with the best efficiency. In particular, Bao, Lero, and Kepler trigger the optimization procedure multiple times, while Athena triggers it only once. This is because the plan explorer in Athena is directly integrated into the conventional query optimizer, whereas other solutions control the plan indirectly, outside of the conventional optimizer. Although Bao explores more plans in TPC-DS and DSB, the quality of its explored plans is suboptimal, as shown in Table 4. This key reason is that the changes to the hints may not change the tree structure and only affect the operators in it.

Quality of the Explored Plans. To evaluate the quality of the plans explored, we compare the cumulative execution times (excluding the exploration time and model inference time) of the fastest plans found by each plan explorer. Specifically, for each query in the training set of every benchmark, we execute all its candidate plans to identify the fastest one in each plan explorer, i.e., Kepler, Bao, Lero, and Athena. Table 5 shows the cumulative execution time of the fastest plans for each query discovered by four plan explorers across four benchmarks. The shortest time for each benchmark is highlighted in bold. Obviously, our explorer outperforms the others in three out of four benchmarks, due to its ability to explore various join orders to increase plan diversity.

7.3.2 Effect of Tree-Mamba Plan Comparator. In this section, we evaluate the effectiveness of the plan comparator from two perspectives: (i) the model used, and (ii) the design choices adopted.

Ablation Study on Models. We substitute the models of Athena and keep the other components unchanged, then compare the end-to-end latency. In particular, we adopt the widely used Tree-CNN implementation in [26, 27, 50, 54] and implement the Tree-LSTM variant proposed in RTOS [40, 53] in Athena. The experimental results are shown in Table 6. Tree-Mamba outperforms the others in 10 out of 16 tested cases, while Tree-CNN in four cases, and Tree-LSTM in three cases. The superiority of Tree-Mamba is provided by effectively capturing both local and global information from the execution plan, as explained in Section 5.

Ablation Study on Design Choices. In this experiment, we conducted an ablation study on two designs proposed in Section 5.3 in Athena: (i) the newly designed Tree-Conv layer, and (ii) the learned average layer. We remove these designs from Athena and compare the end-to-end latency across four benchmarks. The results are shown in Table 7. Specifically, we refer to our proposed Tree-Conv layer as NewConv and the learned average layer as LearnedAve, while the

Table 7. The end-to-end latency (in seconds) of Athena with different design choices across four benchmarks

Plan Explorer	Explorer of Bao				Explorer of Lero				Explorer of Kepler				Explorer of Athena			
Benchmark	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB
NewConv+LearnedAve	1911	**3343**	**700**	**905**	**1624**	**6659**	1135	1672	**3576**	**8405**	**3774**	**2617**	**954**	**4185**	**663**	722
OldConv+LearnedAve	1732	3644	737	912	1689	>10480	**1133**	1998	3577	**8405**	**3774**	**2617**	1037	5231	664	**717**
NewConv+MeanAve	**1680**	3559	743	1009	1748	>9369	1168	**1633**	3577	**8405**	**3774**	**2617**	1257	4925	680	722

Table 8. The end-to-end latency (in seconds) of Athena with different loss functions across four benchmarks

Plan Explorer	Explorer of Bao				Explorer of Lero				Explorer of Kepler				Explorer of Athena			
Benchmark	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB	JOB	STATS	TPC-DS	DSB
WeightedTaiLr	1911	**3343**	**700**	**905**	**1624**	**6659**	**1135**	**1672**	**3576**	**8405**	**3774**	**2617**	**954**	**4185**	**663**	**722**
BCE	**1595**	8236	727	938	1701	>30114	1216	1737	3577	**8405**	**3774**	**2617**	1034	9495	664	731

Table 9. Plan exploration time (in seconds)

Benchmark	JOB	STATS	TPC-DS	DSB
Bao	486	168	26	258
Lero	614	62	15	245
Athena	**132**	**5**	**1**	**42**
PostgreSQL	11	4	1	6

traditional Tree-CNN layer is referred to as OldConv and the mean average described in Section 5.1 is referred to as MeanAve. The results indicate that our design achieves the best performance in 12 out of 16 experiments. This is because the newly designed Tree-Conv layer and the learned average layer are better at distinguishing the type of the node and the number of its children and reduce computational overhead.

7.3.3 Effect of Time-weighted Model Trainer. In this section, we evaluate the effectiveness of our time-weighted model trainer by replacing the loss in Athena with the binary cross entropy (BCE) loss used in Lero [54] and keeping other components not changed. Then, we compare the end-to-end latency across four benchmarks. Table 8 presents the experimental results. In particular, WeightedTaiLr outperforms BCE in 15 out of 16 experiments. The reason is that the time weight emphasizes the most significant good and bad queries to train the model, while the TaiLr loss enhances the stability of the training process. Notably, among all four benchmarks, the STATS dataset exhibits a significantly more complex data distribution. As a result, execution times for different plans of the same query can vary greatly. In this context, the weighted TaiLr loss helps the model avoid selecting disastrous plans, leading to better improvements on the STATS dataset.

7.4 Overhead Study

In this section, we evaluate the overhead of Athena. In particular, we compare the plan exploration time, model inference time, and model training time with the other competitors. Next, we will elaborate on each type of overhead respectively.

Plan Exploration Time. Table 9 shows the plan exploration time for Bao, Lero, Athena, and vanilla PostgreSQL across four benchmarks. Firstly, Athena exhibits significantly shorter exploration time compared to both Bao and Lero across all workloads. This is because our method could enumerate all candidate plans within a single pass of the traditional optimization procedure, as we introduced in Section 4.2. Secondly, the exploration time of Athena is close to PostgreSQL on STATS and TPC-DS but larger than PostgreSQL on JOB and DSB. This is because some queries of JOB and DSB include too many (at least 12) tables, and PostgreSQL will use a genetic algorithm to accelerate the optimization procedure for such queries. Athena closes this optimization and takes advantage of the bottom-up approach of vanilla PostgreSQL for those queries.

Table 10. Model inference time (in seconds)

Benchmark	JOB	STATS	TPC-DS	DSB
Bao	1.35	1.64	0.92	1.46
Lero	**1.28**	**1.38**	**0.84**	**0.95**
Athena	2.57	2.76	1.68	1.85
Athena-LSTM	15.07	11.58	7.78	8.68

Table 11. Model training time (in seconds)

Benchmark	JOB	STATS	TPC-DS	DSB
Bao	**11**	**35**	**5**	**41**
Lero	1134	76	52	133
Athena	5885	224	81	421
Athena-LSTM	11431	519	169	922

Model Inference Time. Table 10 shows the model inference time for Bao, Lero, Athena across four benchmarks. The results show that the inference time of Athena is slower than both Lero and Bao. This is because the tree convolution in Tree-CNN methods can be parallelized, while Tree-Mamba treats the tree as a sequence and cannot be parallelized. Nevertheless, the benefits taken from the Tree-Mamba model outweigh its overhead, as the inference time of Athena only takes 0.30%, 0.07%, 0.25%, and 0.26% of the total time on JOB, STATS, TPC-DS, and DSB, respectively. While the model inference time of Athena is close to that of Bao and Lero, Tree-LSTM is much slower than Bao, Lero, and Tree-Mamba. The reason is that Tree-LSTM heavily depends on matrix multiplication. Thus, more FLOPS are required compared with Tree-Mamba, which does not need matrix multiplication in recurrent steps.

Model Training Time. Table 11 shows the model training time for Bao, Lero, and Athena across four benchmarks. Firstly, the training time of Athena is longer than both Bao and Lero. The reason is that Tree-CNN used in Bao and Lero supports parallel training. In comparison, Athena utilizes Tree-Mamba, which cannot be parallelized. Secondly, the training time of Athena is significantly shorter than Tree-LSTM, especially on JOB, because Tree-LSTM introduces heavy matrix multiplication overhead. Thirdly, Bao performs best in terms of model training time. The reason is that the learning-to-rank training paradigm used in Lero and Athena is slower than the regression-based training paradigm used in Bao.

8 Conclusion

In this work, we proposed Athena, a novel learned optimizer enhancer that advances the current solutions provided by Bao and Lero. Firstly, we proposed an order-centric plan explorer, which offers a comprehensive approach to thoroughly explore the extensive range of potential execution plans. Next, we proposed a Tree-Mamba plan comparator to select the best candidate among the explored execution plans. Finally, we proposed a time-weighted model trainer to bring the precise execution latency into the existing learning-to-rank methods. By conducting extensive experiments, we demonstrated the superiority of Athena compared with the existing learned optimizer enhancers. In the future, we plan to utilize Athena to improve the query processing performance of other DBMSs (e.g., MySQL).

Acknowledgments

We are grateful to the anonymous reviewers and the shepherd for their insightful comments and valuable suggestions. This work was partially supported by National Science Foundation of China (NSFC No. 62422206), Hong Kong Research Grants Council under General Research Fund (project no. 15200023), Theme-based Research Scheme (project no. T43-513/23-N), and a research gift from AlayaDB.AI.

References

[1] 2024. *MySQL*. https://www.mysql.com.

[2] 2025. *Athena Implementation*. https://github.com/DBGroup-SUSTech/Athena

[3] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. 1976. System R: Relational Approach to Database Management. *TODS*. 1, 2, 97–137.

[4] Mehmet Aytimur, Silvan Reiner, Leonard Wörteler, Theodoros Chondrogiannis, and Michael Grossniklaus. 2024. LPLM: A Neural Language Model for Cardinality Estimation of LIKE-Queries. *SIGMOD*. 2, 1, 54:1–54:25.

[5] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *ICML*. 89–96.

[6] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans. *PVLDB*. 16, 7, 1777–1789.

[7] Kyle B. Deeds, Dan Suciu, and Magdalena Balazinska. 2023. SafeBound: A Practical System for Generating Cardinality Bounds. *SIGMOD*. 1, 1, 1–26.

[8] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *PVLDB*. 14, 13 (2021), 3376–3388.

[9] 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. *SIGMOD*. 1, 1, 1–25.

[10] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *PVLDB*. 12, 9, 1044–1057.

[11] Albert Gu and Tri Dao. 2023. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. *arXiv*.

[12] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *PVLDB*. 15, 4, 752–765.

[13] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *SIGMOD*. 1035–1050.

[14] Mike Heddes, Igor Nunes, Tony Givargis, and Alex Nicolau. 2024. Convolution and Cross-Correlation of Count Sketches Enables Fast Cardinality Estimation of Multi-Join Queries. *Proc. ACM Manag. Data* 2, 3, 129.

[15] Dan Hendrycks and Kevin Gimpel. 2023. Gaussian Error Linear Units (GELUs). arXiv:1606.08415

[16] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *PVLDB*. 13, 7, 992–1005.

[17] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *PVLDB*. 13, 12, 3072–3084.

[18] Haozhe Ji, Pei Ke, Zhipeng Hu, Rongsheng Zhang, and Minlie Huang. 2023. Tailoring Language Generation Models under Total Variation Distance. In *ICLR*.

[19] Kyoungmin Kim, Sangoh Lee, Injung Kim, and Wook-Shin Han. 2024. ASM: Harmonizing Autoregressive Model, Sampling, and Multi-dimensional Statistics Merging for Cardinality Estimation. *SIGMOD*. 2, 1, 45:1–45:27.

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

[21] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB*. 9, 3, 204–215.

[22] 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. *PVLDB*. 17, 2, 197–210.

[23] Yingze Li, Hongzhi Wang, and Xianglong Liu. 2024. One Seed, Two Birds: A Unified Learned Structure for Exact and Approximate Counting. *SIGMOD*. 2, 1, 15:1–15:26.

[24] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *PVLDB*. 14, 11, 1950–1963.

[25] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD*. 2530–2542.

[26] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. 1275–1288.

[27] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *PVLDB*. 12, 11, 1705–1718.

[28] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM@SIGMOD*. 3:1–3:4.
[29] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB*. 12, 11, 1733–1746.
[30] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. *SIGMOD*. 1, 4, Article 247, 247:1–247:26 pages.
[31] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*. 1287–1293.
[32] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *PVLDB*. 14, 11, 2019–2032.
[33] 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. *PVLDB*. 16, 6, 1520–1533.
[34] PGTune 2024. *PGTune - calculate configuration for PostgreSQL based on the maximum performance for a given hardware configuration*. https://pgtune.leopard.in.ua
[35] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *PVLDB*. 17, 4, 740–752.
[36] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. 23–34.
[37] Noam Shazeer. 2020. GLU Variants Improve Transformer. *CoRR* abs/2002.05202 (2020).
[38] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *PVLDB*. 19–28.
[39] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD*. 340–355.
[40] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *PVLDB*. 13, 3, 307–319.
[41] Transaction Processing Performance Council(TPC). 2021. *TPC-DS Vesion 2 and Version 3*. http://www.tpc.org/tpcds/
[42] Fang Wang, Xiao Yan, Man Lung Yiu, Shuai Li, Zunyao Mao, and Bo Tang. 2023. Speeding Up End-to-end Query Execution via Learning-based Progressive Cardinality Estimation. *SIGMOD*. 1, 1, 28:1–28:25.
[43] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *PVLDB*. 15, 1, 72–84.
[44] Zilong Wang, Qixiong Zeng, Ning Wang, Haowen Lu, and Yue Zhang. 2023. CEDA: Learned Cardinality Estimation with Domain Adaptation. *PVLDB*. 16, 12, 3934–3937.
[45] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD*. 2009–2022.
[46] Renzhi Wu, Bolin Ding, Xu Chu, Zhewei Wei, Xiening Dai, Tao Guan, and Jingren Zhou. 2021. Learning to be a Statistician: Learned Estimator for Number of Distinct Values. *PVLDB*. 15, 2, 272–284.
[47] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *SIGMOD*. 1, 1, 41:1–41:27.
[48] Ziniu Wu and Amir Shaikhha. 2020. BayesCard: A Unified Bayesian Framework for Cardinality Estimation. *CoRR*. abs/2012.14743.
[49] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. 2020. On layer normalization in the transformer architecture. In *ICML*. 10524–10533.
[50] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. [n. d.]. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD*. 931–944.
[51] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *PVLDB*. 14, 1, 61–73.
[52] 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. *PVLDB*. 13, 3, 279–292.
[53] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*. 1297–1308.
[54] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *PVLDB*. 16, 6, 1466–1479.
[55] 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. *PVLDB*. 14, 9, 1489–1502.

Received October 2024; revised January 2025; accepted February 2025