AQETuner: Reliable Query-level Configuration Tuning for Analytical Query Engines

Lixiang Chen
East China Normal University
& ByteDance
lxchen@stu.ecnu.edu.cn

Yuxing Han*
ByteDance
hanyuxing@bytedance.com

Yu Chen
East China Normal University
& ByteDance
yuchen198@stu.ecnu.edu.cn

Xing Chen
ByteDance
chenxing.xc@bytedance.com

Chengcheng Yang*
East China Normal University
ccyang@dase.ecnu.edu.cn

Weining Qian
East China Normal University
wnqian@dase.ecnu.edu.cn

ABSTRACT

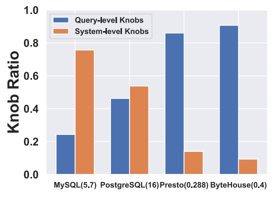
Modern analytical query engines (AQEs) are essential for large-scale data analysis and processing. These systems usually provide numerous query-level tunable knobs that significantly affect individual query performance. While several studies have explored automatic DBMS configuration tuning, they have several limitations to handle query-level tuning. Firstly, they fail to capture how knobs influence query plans, which directly affect query performance. Secondly, they overlook query failures during the tuning processing, resulting in low tuning efficiency. Thirdly, they struggle with cold-start problems for new queries, leading to prolonged tuning time. To address these challenges, we propose AQETuner, a novel Bayesian Optimization-based system tailored for *reliable* query-level knob tuning in AQEs. AQETuner first applies the attention mechanisms to jointly encode the knobs and plan query, effectively identifying the impact of knobs on plan nodes. Then, AQETuner employs a dual-task Neural Process to predict both query performance and failures, leveraging their interactions to guide the tuning process. Furthermore, AQETuner utilizes Particle Swarm Optimization to efficiently generate high-quality samples in parallel during the initial tuning stage for the new queries. Experimental results show that AQETuner significantly outperforms existing methods, reducing query latency by up to 23.7% and query failures by up to 51.2%.

PVLDB Reference Format:
Lixiang Chen, Yuxing Han, Yu Chen, Xing Chen, Chengcheng Yang, and Weining Qian. AQETuner: Reliable Query-level Configuration Tuning for Analytical Query Engines. PVLDB, 18(8): 2709 - 2721, 2025.
doi:10.14778/3742728.3742759

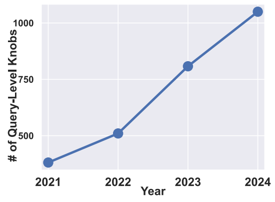
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/chenlx0/aqetuner.

1 INTRODUCTION

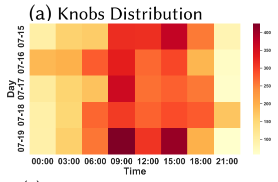
Modern analytical query engines (AQEs), including Snowflake [4], Presto [41], and ByteHouse [13], are integral to business intelligence and decision-making across industries. For example, ByteHouse handles 200 million queries every day, processing multiple exabytes of data to meet the analytical demands of platforms such as TikTok. The performance of AQEs relies heavily on numerous tunable configurations (or knobs). However, knob tuning is an NP-hard problem [40] and requires a deep understanding of query optimization. Even experienced DBAs often struggle to fully grasp the impact of configurations on every individual query execution.



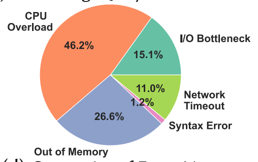
(a) Knobs Distribution



(b) Increasing Query-Level Knobs



(c) Heatmap of Query Failures



(d) Categories of Error Messages

Figure 1: Motivating Examples

To reduce the manual tuning efforts of DBAs, recent approaches have sought to automate the knob tuning via Machine Learning (ML) techniques, including Bayesian Optimization (BO) [3, 5, 44, 51, 52] and Deep Reinforcement Learning (DRL) [2, 10, 25, 49]. BO offers a theoretically-sound method for exploring the configuration space with improvement guarantees, while DRL utilizes Deep Deterministic Policy Gradient (DDPG) [26], a model-free and actor-critic algorithm that operates on continuous action (i.e., configuration) spaces. Most of these approaches target specific workloads and optimize the overall throughput and latency of transactional databases.

However, these approaches have limitations for tuning AQEs. On the one hand, AQEs usually have different knob distributions compared to transactional databases. The database knobs can be broadly classified into two main categories: system-level knobs, which control the overall system behavior (e.g., buffer pool size and log file size [44]), and query-level knobs, which optimize individual query execution (e.g., runtime operator parallelism [30] and execution strategy option [22]). It is often observed that AQEs expose a higher proportion of query-level knobs. Fig. 1a shows the ratio of knob categories in different transactional databases and AQEs,

*Corresponding authors.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742759

with ByteHouse and Presto having more query-level knobs than PostgreSQL and MySQL. Moreover, Fig. 1b highlights the rapid increase in query-level knobs in ByteHouse as the system evolves. This reflects the need for AQEs to offer more fine-grained tuning knobs [1, 36, 39] to handle complex analytical queries that involve aggregations, joins, and window functions on large datasets [4, 41].

On the other hand, analytical engines frequently encounter query failures due to the complexity of query processing and the substantial data volumes involved. Fig. 1c shows the heatmap of query failures throughout one workday in a ByteHouse cluster, while Fig. 1d categorizes the corresponding error messages. Most failures are attributed to suboptimal query-level knob configurations, such as inadequate CPU/memory/storage resource allocations, improper parallelism settings, and poor execution strategies.

Given these tuning requirements, tuning AQEs should meet two criteria, which existing methods struggle to achieve: 1) Query-Level Tuning: queries must be optimized using query-level knobs. 2) Reliability: methods should ensure that query failure rates do not increase during the tuning process. Specifically, the key challenges of tuning the analytical engines can be summarized as follows:

Jointly Encoding the Knobs and Query Plans (C1). Joint representation learning for knobs and queries should capture the impact of knobs on the query execution to enhance downstream tuning tasks. However, encoding them in a shared space poses a complex challenge. Since query plans are closely related to the query execution, we propose to encode the query plan rather than the plain query text. To this end, two critical issues must be addressed: First, how to effectively preserve the hierarchical structure of the tree-structured query plans. Second, how to model the correlation between the query-level knobs and plan nodes, considering knobs impact different nodes in different ways.

Dual-task Knob Tuning (C2). When performing query-level tuning, both query performance and reliability should be considered, as improper knob settings would increase query failures and reduce tuning efficiency. Modeling these two targets is challenging because query performance and reliability are inherently interdependent. For instance, a knob that improves query performance might increase resource consumption, potentially leading to query failures. As a result, modeling both tasks separately is ineffective, as it overlooks important correlations. Moreover, integrating encodings from representation learning further complicates the task.

Cold-start Problem (C3). ML-based tuning systems often face the cold-start issue when encountering new queries without prior knowledge [2]. Collecting high-quality observation data in the early tuning stage is crucial to accelerate the subsequent tuning process. In the absence of such data, the system might experience suboptimal exploration of promising regions, leading to prolonged tuning times. Moreover, the ability to quickly collect such data is essential to maintain overall tuning efficiency.

Our Approach. To address the above challenges, we propose AQETuner, a novel system for reliable query-level tuning in AQEs. AQETuner comprises three main components: knob-plan encoder (addressing C1), dual-task predictor (addressing C2), and warm starter (addressing C3). Specifically, the knob-plan encoder first utilizes the self-attention mechanism [45] to independently encode the query plan and knobs, capturing each of their internal correlations. For tree-structured query plans, we propose a hierarchical spectral position encoding to preserve the global structure information between plan nodes. Then, the encoder utilizes the cross-attention mechanism [11] to jointly synergize the encodings of query plans and knobs into a shared space, which further captures each knob's influence on the plan nodes. To improve computational efficiency, the cross-attention scores are only computed between knobs and plan nodes that are correlated.

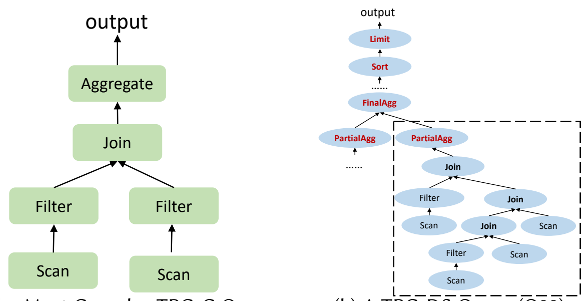
The dual-task predictor leverages Neural Process (NP) [8, 9, 18] to predict both query performance and reliability with the joint knob-plan encodings. The main characteristic of NP is that it can combine the strengths of neural networks with stochastic processes and allow the efficient fitting of observed data. By using NPs, we can naturally leverage the joint encodings within the neural network's embedding space. However, the vanilla NP lacks the capacity to capture the task correlations and struggles to express the complexity of both tasks. To enhance model expressiveness, we introduce two intra-task latent variables to handle information related to the prediction tasks of performance and reliability, respectively. Moreover, cross-attention is employed to enable the model to selectively focus on the observation data that are most relevant to each task. To further exploit the inherent correlations between tasks, we introduce a cross-task latent variable to capture the shared information across both tasks. Subsequently, this variable is used to complement the two intra-task variables with a gating mechanism [12], ensuring that it effectively contributes to the final predictions.

The warm starter employs Particle Swarm Optimization (PSO) [37] to efficiently generate high-quality initial observation data. PSO uses multiple search agents, each selecting samples within the configuration space iteratively. These agents are guided by both the global optimum across all agents and the local optimum of each individual agent. The search process is further accelerated by leveraging the AQEs' parallel execution capability, allowing each agent to search in parallel. The contributions can be summarized as:

- We propose AQETuner, a novel ML-based tuning system for reliable query-level tuning. To our knowledge, this is the first approach that focuses on query-level tuning for analytical query engines.
- We solve the challenge of jointly encoding knobs and query plans with various attention techniques.
- We address prediction for performance and reliability with a dual-task Neural Process, leveraging the correlation between the tasks to improve accuracy.
- We resolve the cold-start issue by applying the Particle Swarm Optimization to generate high-quality samples in the initial tuning phase, ensuring a diverse and efficient exploration of the search space.
- We conduct extensive experiments on both synthetic and real-world workloads. Experimental results show that AQETuner improves the query latency by up to 23.7% and reduces the query failure rate by up to 51.2% compared to state-of-the-art methods.

2 PRELIMINARY

In this section, we introduce the basic concepts in AQEs, including the query plan and the query-level knob, and provide a brief overview of BO-based tuning for database systems. We then clarify related terminology and formulate the query-level tuning problem.



(a) The Most Complex TPC-C Query (b) A TPC-DS Query (Q99)

Figure 2: Comparison of Query Plans

2.1 Query Plan

A query plan in a database is a structured representation of how a query will be executed, usually depicted as a tree-structured Directed Acyclic Graph (DAG). In this DAG, each node represents an *operator* that performs tasks like table scans, record filtering, joins, or aggregations, while the edges indicate the data flow between these operators. To effectively model an analytical query, it is preferable to model the query plan directly rather than the query text since the plan captures the runtime behavior of the engine.

Analytical queries are generally more complex than transactional queries, along with their query plans. This is because they often involve numerous complex operations such as joins across several tables, subqueries, data groupings, and advanced functions such as window functions. In contrast, transactional queries usually consist of simpler CRUD (Create, Read, Update, Delete) operations, which primarily focus on row-level locking and transactional integrity.

Figure 2 compares the query plans for the most complex transactional query from the TPC-C benchmark [42] and a typical analytical query from TPC-DS benchmark [43]. The first plan is generated by PostgreSQL, while the second is produced by ByteHouse. The key differences are: 1) The *longest path* in the DAG-structured plan of the analytical query is significantly longer than that of the transactional query. 2) The number and variety of nodes (i.e., operators) in the analytical query plan are much greater. For example, the plan of TPC-DS query Q14 in ByteHouse has nearly 150 plan nodes, significantly more complex than TPC-C queries.

2.2 Query-Level Knobs

AQEs offer various query-level tuning knobs that enable users to control and optimize the runtime operator of each individual query. These knobs are vital for enhancing query performance, especially in distributed environments where efficient data movement across multiple computing instances is critical. They enable fine-grained tuning, such as resource allocation and runtime settings for specific queries, without changing the overall system configuration. Common query-level knobs include memory allocation, which sets the memory available to execute an operator; the degree of parallelism, which determines the number of parallel processes or threads used to execute specific operators; and execution strategy options, such as deciding whether to shuffle data before or after the aggregation operator to minimize data transfers [27].

These knobs can be either categorical/discrete (e.g., turning an execution option on or off) or numerical/continuous (e.g., setting the maximum memory for a specific operator). Unlike discrete knobs, continuous knobs can take any value within a defined range, which significantly expands the search space and makes finding the optimal settings more challenging. This complexity is further increased by the interdependencies between query-level knobs, where the optimal value of one knob might shift with small changes on another one. For example, the increased parallelism of operators would also lead to more memory contentions among concurrent processes and impact the optimal setting of memory allocation.

2.3 Related BO-based Knob Tuning

The existing Bayesian Optimization (BO) based knob tuning for DBMS [3, 5, 44, 51, 52] starts with an initial random set of configurations. Then, the DBMS is executed under these initial configurations, and performance metrics (e.g., throughput, latency) are observed. Based on these observation data, a Gaussian Process is built as the surrogate model to estimate the relationship between the configuration and the performance metrics. An acquisition function selects the next configuration to evaluate, balancing the exploration of new configurations with the exploitation of known promising ones. This iterative process involves repeatedly evaluating the DBMS with new configuration points, updating the surrogate model, and refining the search for the optimal knobs. It continues until a stopping criterion is met, such as reaching a maximum number of iterations. The configuration that yields the best-observed performance is then chosen for deployment in the production environment.

These methods treat knob tuning as a “black-box” optimization problem [21] and rely on limited features (e.g., keyword statistics) extracted from the query text [51, 52]. However, they ignore the query plans that specify how queries are executed within the query engine. We argue that incorporating the query plan could provide valuable query execution information, which further significantly helps to enhance the tuning performance.

2.4 Problem Definition

Query Representation Space. Given an analytical query q, the representation learning derives a vector representation $\omega_q \in \mathbb{R}^d$ based on extracted relevant features. The learned representation can be utilized for the downstream tuning problem. We denote the set of all possible vector representations as $\Omega = \{\omega_q\}$.
Knob Search Space. For each query-level knob k, whether discrete or continuous, we normalize its domain Θ to a continuous space $[0, 1]$ using min-max uniform scaling. Given a set of query-level knobs $K = \{k_1, k_2, ..., k_n\}$, where each knob k_i has a domain Θ_i, the overall knob search space is represented as $\Theta = \Theta_1 \times \cdots \times \Theta_n$. A specific point in this space represents a possible combination of knob values.
Reliability. The reliability of an analytical query q under a given configuration $\theta_q \in \Theta$ is determined by its execution status (success or failure) within an execution environment E. If the query executes successfully (status 0), the configuration is considered reliable; otherwise, with an execution status of 1, the configuration is considered unreliable. The reliability function can be defined as $g : \Omega \times \Theta \rightarrow \{0, 1\}$.
Query-Level Performance. We define the query-level performance function as $f : \Omega \times \Theta \rightarrow \mathbb{R}_{>=0}$. In this paper, we focus on the end-to-end query-level performance metric: the execution time.

Reliable Query-level Tuning. The objective of reliable query-level tuning for a given query q is to find the optimal configuration $\theta_q^* \in \Theta$ within the knob search space for an analytical query with correct SQL syntax, aiming to minimize the performance metric while avoiding execution failures. We formalize this as an optimization problem:

$$\begin{aligned} &\arg\min_{\theta} f(\omega_q, \theta_q), \\ &\text{s.t. } g(\omega_q, \theta_q) = 0 \end{aligned} \tag{1}$$

3 SYSTEM OVERVIEW

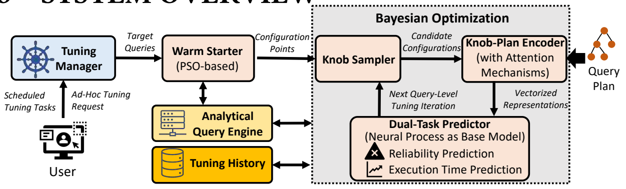


Figure 3: Overall Architecture of AQETuner

Components. Figure 3 shows the overall architecture of AQETuner, which tunes query-level knobs for the analytical engine using a Bayesian Optimization (BO) framework. AQETuner has several critical components to facilitate its knob tuning: *tuning manager*, which receives target queries from the user and manages the tuning process; *knob sampler*, which generates candidate configurations at the start of the BO loop; *knob-plan encoder*, which jointly encodes these configurations and query plans into vectorized representations generated by neural networks with attention mechanisms (discussed in Section 4); *dual-task predictor*, which takes the representations as input to predict query reliability $\tilde{g}$ and execution time $\tilde{f}$ and support query-level knob tuning (discussed in Section 5); *warm starter*, which collects high-quality observation data using Particle Swarm Optimization (PSO) method and establishes a coarse correlation between the knobs and plan nodes (discussed in Section 6).

Based on dual-task predictions, AQETuner employs the Expected Improvement with Constraints (EIC) [7] as the acquisition function. To integrate the reliability constraint, we incorporate the probability of successful execution into expected improvements over the best-observed performance. Specifically, given f^* as the best performance observed so far, the acquisition function is defined as:

$$\alpha_{EIC}(f, g) = \mathbb{E}[max(0, f^* - \tilde{f})] \times p(\tilde{g} = 0). \tag{2}$$

Workflow. Algorithm 1 outlines the main procedures of AQETuner. Given the target queries and knob search space, the tuning process is constrained by a duration $\mathcal{T}$. It starts by initializing the tuning history with the warm starter and constructing an initial surrogate model $\mathcal{M}$ (Lines 3~4). Initial configurations are evaluated on the engine E to derive a coarse correlation C between knobs and plan nodes (Line 5). For each query, AQETuner samples candidate knobs $\mathcal{KS}$ and retrieves its query plan (Lines 6~8). The knob-plan encoder then jointly encodes the query plan and knobs into vectorized encodings $\mathcal{X}$ based on the correlation C (Line 9). The dual-task predictor forecasts both reliability and performance and selects the most promising encodings $x^{\mathcal{B}}$ with the acquisition function (Lines 10~11). The selected knobs $k^{\mathcal{B}}$ of $x^{\mathcal{B}}$ are then evaluated on the analytical engine to collect actual performance metrics (Lines 11~12), and the results are added to the tuning history (Line 13). After the

Algorithm 1: AQETuner Workflow

```
Input: Target queries $Q$; Knob search space $\Gamma$; Analytical query
       engine instance $E$; Maximum tuning duration $\mathcal{T}$
Output: Best configuration explored for each query
1  Initialize tuning history $\mathcal{H} \leftarrow \emptyset$ ;
2  while $\mathcal{T}$ is not reached do
3      $\mathcal{H} \leftarrow$ generateWarmStartSamples($Q$) ;
4      Consturct a surrogate model $\mathcal{M}$ with $\mathcal{H}$;
5      $C \leftarrow$ identifyKnobAndPlanCorrelation($\mathcal{H}$, $E$);
6      for each query $q \in Q$ do
7          $\mathcal{KS} \leftarrow$ generateCandidateKnobSamples($\Gamma$);
8          Retrieve the query plan $p^E$ of $q$ from $E$;
9          $\mathcal{X} \leftarrow$ encodeKnobsAndPlan($\mathcal{KS}$, $p^E$, $C$);
10         $\mathcal{B} \leftarrow$ predictReliabilityAndPerformance($\mathcal{X}$, $\mathcal{M}$);
11         $x^{\mathcal{B}} \leftarrow$ argmax $\alpha_{EIC}(\mathcal{B})$ ;
12         $y^{\mathcal{B}} \leftarrow$ evaluate corresponding knobs $k^{\mathcal{B}}$ of $x^{\mathcal{B}}$ on $E$ to
             obtain true performance of $q$;
13         $\mathcal{H} \leftarrow \mathcal{H} \cup (k^{\mathcal{B}}, y^{\mathcal{B}})$ ;
14     end
15     update $\mathcal{M}$ with $\mathcal{H}$;
16 end
17 return $k^{\mathcal{B}}$ with best $y^{\mathcal{B}}$ for each query in $\mathcal{H}$ ;
```

tuning iterations of queries are finished, the surrogate model is updated accordingly (Line 15). The process iterates until $\mathcal{T}$ is reached, after which the best knob configurations are returned (Line 17).

System Integration. To enable seamless integration with existing analytical query engines, AQETuner is designed as an external, non-intrusive service that runs parallel with the running engines. This design avoids disruption to regular query processing and supports broad compatibility with engines like Presto and ByteHouse. The integration process follows three key phases: (1) *Query Plan Extraction*, (2) *Knob Adjustment*, and (3) *Performance Feedback Collection*. In the *Query Plan Extraction* phase, AQETuner interacts with the engine's SQL interface to retrieve the query plan using EXPLAIN statements without executing the query. The plan is then processed by AQETuner 's knob-plan encoder, which jointly encodes the plan with the tunable knobs. In the *Knob Adjustment* phase, AQETuner generates configuration recommendations, which are applied via external interfaces, such as Presto's SET SESSION command or ByteHouse's query-level SETTINGS syntax. Finally, during the *Performance Feedback Collection* phase, AQETuner gathers execution feedback (e.g., execution time, execution success/failure) through the client APIs or system logs. This feedback is fed into the BO loop for iterative refinement of the tuning process.

4 KNOB-PLAN ENCODER

In this section, we first briefly introduce the attention mechanism and then elaborate on the architecture of the knob-plan encoder.

4.1 Basics: Attention Mechanisms

In recent years, the attention mechanism has become the cornerstone of modern neural networks for sequence modeling [6, 35]. This mechanism allows the model to assign different levels of importance to different parts of the input sequence, enabling it to focus dynamically on the most relevant parts when dealing with prediction tasks. For each token embedding $x \in \mathbb{R}^d$ of the input sequence $X \in \mathbb{R}^{n*d}$, where n is the sequence length and d is the embedding dimension, three vectors Query (Q), Key (K), and Value (V) are constructed by projecting x into three distinct spaces with the learned weight matrices $W_Q \in \mathbb{R}^{d*d_k}$, $W_K \in \mathbb{R}^{d*d_k}$, and $W_V \in \mathbb{R}^{d*d_v}$:

$$Q = x \cdot W_Q, K = x \cdot W_K, V = x \cdot W_V, \tag{3}$$

where d_k denotes the dimension of Q and K, and d_v denotes the dimension of V. The dot-product attention is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{4}$$

The output is an encoded representation of the input sequence. The unnormalized score between any two tokens i and j before applying softmax can be expressed in a matrix form: $A_{i,j} = \frac{Q_i * K_j^T}{\sqrt{d_k}}$.

There are two variants of attention mechanisms, which are *self-attention* [45] and *cross-attention* [11]. Self-attention enables the model to capture relationships between tokens in a sequence, allowing each token to derive a context-aware representation by considering the entire sequence. Cross-attention, on the other hand, allows information from one sequence to influence the representation of tokens in another. It could generate an output for a sequence that is informed by a different input sequence.

In addition, *positional encodings* are usually added to the input embeddings, allowing the model to recognize the positions of tokens within a sequence. Specifically, the Transformer [45] combines the positional encodings with the attention mechanism. This, along with residual connections [46] and a position-wise Feed-Forward Network (FFN), constitutes the *attention block*. This structure allows the Transformer to learn and handle complex dependencies in large-scale sequences effectively.

4.2 Plan Representation

We now describe the architecture of the knob-plan encoder, shown in Fig. 4, which consists of three attention blocks. Specifically, two self-attention blocks are used to separately encode the query plan and candidate configurations, generating vector representations of plan nodes and knobs. Both outputs are mapped to a uniform dimensionality using Multilayer Perceptrons (MLPs). Then, the cross-attention block above them jointly encodes the knobs and query plan based on their correlation, using outputs from the two self-attention blocks as inputs. Finally, its output is consolidated into a single vector representation through the average pooling.

4.2.1 Node Featurization. A query plan node for analytical queries includes the following key elements as its features: the plan operator type, involved tables and columns, predicates, join conditions, aggregation functions, and cardinality/cost estimations. To encode these elements, we use the one-hot encoding for the plan operator type, tables, columns, join conditions, and aggregation functions, due to their relatively static and finite domains under a specific analytical query engine and workload context. Predicates are represented as triplets $\langle column, comparison\_operator, value \rangle$, where the comparison operators (e.g., $>$, $<$, $=$) define the conditions. Each element of the triplet is individually encoded and concatenated

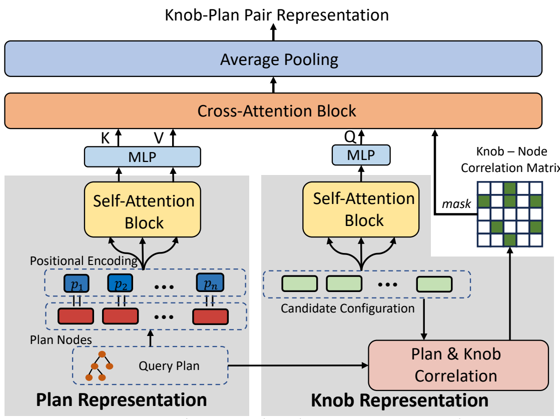


Figure 4: The Knob-plan Joint Encoder.

to form the full predicate encoding. Continuous variables, such as the values in predicates and cardinality/cost estimations, are normalized to the range $[0, 1]$. To handle string-like queries, we employ a trie-based encoding scheme [47], where column strings are organized in a prefix tree structure. In this structure, each node represents a prefix, and leaf nodes are assigned unique IDs along with associated ranges that correspond to matching strings. This structure enables the transformation of string-like queries into range queries for efficient encoding.

All feature encodings are concatenated into a single vector to create the node encoding n_i for each plan node i. Note that if a node does not contain specific information, such as predicates or join conditions, zeros are padded to guarantee consistent dimensionality across all node representations.

4.2.2 Hierarchical Spectral Positional Encoding. Given the encoding of each node, we employ the self-attention to encode the entire query plan. To reduce the computation cost, we only compute the attention scores between the connected nodes [32]. However, preserving information of the plan's structure and the relationships between nodes is challenging. The standard attention mechanism, designed for sequential data, cannot be directly applied to tree-structured plans. To this end, we introduce a Hierarchical Spectral Positional Encoding (HSPE) scheme to incorporate positional information of plan nodes. This scheme combines two components: hierarchical and spectral encoding. The hierarchical encoding employs Breadth-First Search (BFS) to capture the level-based structure, while the spectral encoding uses Laplacian eigenvectors to capture the tree's global properties, such as connectivity patterns between large subtrees or major branches. This positional encoding scheme effectively combines local neighborhood information with long-range dependencies across the entire graph.

Specifically, for each node i, we construct its HSPE as $\mathbf{p}_i = [\mathbf{p}_i^{\text{BFS}} || \mathbf{p}_i^{\text{Lap}}]$. The hierarchical encoding, $\mathbf{p}_i^{\text{BFS}}$, represents the BFS distance from the root to node i, capturing its level information in the tree. For the spectral encoding, we first compute the Laplacian matrix L of the plan tree, which captures the global structure by reflecting the connectivity between nodes, both directly and indirectly. Next, we compute the eigenvectors and eigenvalues of L, where each eigenvector represents a specific connectivity pattern across all nodes, and the corresponding eigenvalue indicates the frequency of these patterns. We select the k eigenvectors associated with the smallest non-zero eigenvalues $v_1, v_2, ..., v_k$, which capture the most significant global structure in the plan tree. The spectral encoding for each node i is then represented as $\mathbf{p}_i^{\text{Lap}} = (v_{1i}, v_{2i}, ..., v_{ki})$, where v_{ji} denotes the value of j-th eigenvector at the i-th node. Note, constructing the Laplacian matrix for a tree and computing its eigenvalues and eigenvectors is efficient, as it scales linearly with the number of plan nodes. Finally, we concatenate each node's HSPE with its original encoding as $n'_i = n_i || \mathbf{p}_i$ and use it as the input of the self-attention block. After applying the self-attention mechanism, we obtain a sequence of node embeddings, the length equal to the number of nodes in the query plan.

4.3 Knob Representation

The self-attention mechanism is also employed to encode the candidate knobs, which helps capture the inter-dependencies between knobs. To construct a vector representation for each configuration as the input of the attention block, one straightforward choice is the one-hot encoding. However, this method could not preserve the specific knob value defined in the continuous spaces.

Instead, we employ a variant of one-hot encoding called *weighted one-hot encoding*, where the value 1 is replaced with the actual configuration value. For instance, consider a set of candidate knobs k_1, k_2, and k_3 with normalized values 0.3, 0.6, and 0.9. Then, we can encode these values into representation vectors as $[0.3, 0, 0]$, $[0, 0.6, 0]$ and $[0, 0, 0.9]$. In cases where the number of knobs to be tuned is large, resulting in especially high-dimensional knob embeddings, an autoencoder [16] can be employed to reduce the high-dimensional embeddings to a more manageable and lower-dimensional space. Note, positional encodings are not applied in encoding knobs since there is no inherent ordering among them. Similar to the plan representation, the output of the self-attention block is a sequence of knob embeddings, with the length equal to the number of knobs.

4.4 Synergizing Plan and Knob

To capture the influence of the knob on the query plan, the knob-plan encoder employs the cross-attention block to synergize the knob and query plan, generating a joint encoding for the downstream predictor. To apply the cross-attention, we treat the sequence of plan embedding as the source sequence and the sequence of knob embedding as the target sequence. The cross-attention A^c_{ij} score represents the correlation between the configurable knob k_i and query plan node n_j. There are two reasons for introducing this kind of attention. First, the goal of AQETuner's encoder is to generate sufficiently informative representations of the knobs, which enables a more accurate prediction of how different knob configurations affect the query performance. Second, the sequence of plan embeddings is usually longer, as the number of nodes in an analytical query plan often exceeds the number of knobs involved in the BO-based tuning process. With this mechanism, it allows the shorter sequence to be interpreted in the context of the longer sequence with richer information [11].

However, it is unnecessary to compute the attention score between every knob-plan pair, since each knob typically impacts only a limited number of query nodes. For example, adjusting a knob that controls filter strategy will affect only the execution time of nodes containing filter-related operators. To this end, we introduce a *correlation matrix* M that illustrates whether the specific configurations affect the execution runtime of each node. This matrix is further used to mask attention scores, ensuring only relevant interactions are considered. Specifically, M_{ij} is set to 0 if knob k_j has no impact on the node n_i, and 1 otherwise. Given the cross-attention score matrix A^c, its masked formula is computed as:

$$A^{c'}_{ij} = M_{ij} \cdot A^c_{ij}. \tag{5}$$

To construct the correlation matrix, one could manually gather the knowledge from DBAs and determine which tuned knobs would affect a given plan node. However, this approach is labor-intensive and inefficient. To address this issue, we propose an automated SHAP-based method (see Section 6) to automatically identify the impact of knobs on the plan nodes by the warm starter.

5 DUAL-TASK PREDICTOR

We propose a dual-task predictor that predicts both the reliability and performance of analytical queries, using the joint representation from the knob-plan encoder as input. In this section, we first give a conceptual overview of the Neural Process, which is the base model of our predictor. Then, we explore the modeling strategy for dual-task prediction. Lastly, we present the structure of our predictor.

5.1 Neural Process Overview

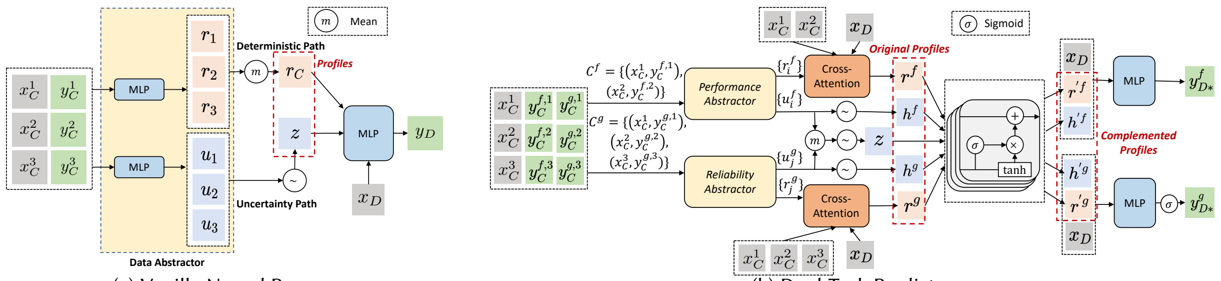
Neural Processes (NPs) are a class of probabilistic models that blend the expressive power of neural networks with the probabilistic reasoning of stochastic processes (e.g., Gaussian Processes). The goal of an NP is to learn from a set of input-output pairs and subsequently predict the output for unseen input points. The core structure of a vanilla NP (as shown in Fig. 5a) revolves around two parallel paths. Formally, given a set C of context pairs (x_C, y_C), NP defines a family of conditional distribution $p(y_D|x_D, C)$, where (x_D, y_D) is a target pair in the set D. To model the conditional distribution, NP utilizes a *data abstractor* to independently encode context pairs using MLPs for two data paths:

- **Deterministic Path.** This path encodes the input-output pairs from the context set into a global representation r_C, enabling the model to capture context-specific information. By leveraging this representation, the model can identify patterns directly from the observed data, enhancing predictive accuracy.
- **Uncertainty Path.** This path incorporates a global latent variable z, shared across all context pairs, to encapsulate information not directly observable from the context set. z is commonly modeled via a variational distribution [17], which learns an approximate posterior distribution conditioned on the context set.

We refer to both r_C from the deterministic path and z from the uncertainty path as *profiles* of the context pairs. The objective of NP is to maximize the likelihood of the target outputs y_D, conditioned on the context set C and the corresponding target inputs x_D from D. Formally, the likelihood of y_D is defined as:

$$p(y_D \mid x_D, C) = \int p(y_D \mid x_D, r_C, z) q(z \mid C) dz. \tag{6}$$

where q represents a variational distribution from which the latent variables z are inferred based on the context C. For a given target x_D, the prediction of y_D is generated using an MLP that takes r_C and z



(a) Vanilla Neural Process (b) Dual-Task Predictor

Figure 5: Model Architecture.

as inputs. The parameters of an NP can be learned by maximizing the following Evidence Lower BOund (ELBO) [9, 19]:

$$\begin{aligned}&\log p\left(y_D \mid x_D, C\right) \geq \\ &\mathbb{E}_{q(z|D)}\left[\log p\left(y_D \mid x_D, r_C, z\right)\right] \\ &-\mathrm{KL}\left(q(z \mid D) \| q(z \mid C)\right),\end{aligned} \tag{7}$$

where the Kullback-Leibler (KL) term acts as a regularizer, ensuring that the distribution of the context data remains close to that of the target data.

5.2 Joint Modeling of Performance & Reliability

We model the task of the dual-task prediction with joint output as: $x \rightarrow y^f \bigcup y^g$, where x is the joint encoding generated by the knob-plan encoder, y^f and y^g represent the two prediction objectives: performance and reliability. Drawing inspiration from the Neural Process (NP), we introduce two kinds of latent variables for dual-task prediction: cross-task and intra-task.

The cross-task latent variable, denoted as z, captures global uncertainty that is shared between query performance and reliability prediction tasks. It allows the model to recognize patterns that affect both tasks simultaneously. For example, increased operator parallelism influences both the query execution time and reliability. By capturing these shared factors, z serves as a bridge for sharing information across both prediction tasks, enabling them to learn from each other and improve predictive accuracy.

Intra-task latent variables, denoted as h^f for query performance and h^g for reliability, are introduced to capture task-specific uncertainty unique to each prediction task. Unlike the cross-task variable z, which models shared global factors, h^f and h^g captures the unique features of each task. For example, h^f focus on the factors that affect performance, such as data distribution patterns and hardware configurations, while h^g focuses on the factors related to reliability, such as memory thresholds and CPU usage constraints that may trigger query failures. This separation allows the model to better understand and predict the unique behaviors of each task.

Formally, given an observation context set C containing performance and reliability context pairs (x_C, y_C^f) and (x_C, y_C^g), and a target set D of pairs (x_D, y_D^f) and (x_D, y_D^g), the prediction likelihood over two tasks can be expressed as:

$$\begin{aligned}&p(y_D^f, y_D^g \mid x_D, C) = \\ &\textstyle\prod_{t\in\{f,g\}} \int\int p(y_D^t \mid h^t, x_C^t) p(h^t \mid z) p(z \mid C) dh^t dz,\end{aligned} \tag{8}$$

5.3 Predictor Structure

Figure 5b presents the architecture of our dual-task predictor. The model employs two data abstractors, each with deterministic and uncertainty paths, to independently encode the performance and reliability context data C^f and C^g. To enhance the deterministic paths, the cross-attention mechanism is used in each task, enabling the model to focus on context data relevant to the specific target. For the uncertainty path, intra-task variables h^f and h^g help capture task-specific uncertainty, while the cross-task variable z is introduced to capture the shared information between the two tasks. In addition, since query performance and reliability are inherently correlated, we utilize a gating mechanism to complement the profiles in the paths of the same type. This approach allows information sharing between the two tasks, which could further help improve both of their prediction accuracy. These complemented profiles are subsequently fed into final MLPs to generate the prediction outputs.

Next, we provide more technical details of the predictor structure, using an observation dataset comprising three instances and a target dataset with one instance (x_D, y_D^f, y_D^g) as the example. We assume that the last instance in the observation dataset is a failed query with $y_C^{g,3} = 1$. Note, in practice, there usually exist multiple instances in the target dataset.

Two Independent Data Abstractors. We first remove the performance data from the observation instances that contain query failures, as such data does not provide meaningful information. After this preprocessing step, we split the example observation data into two sets: $C^f = \{(x_C^1, y_C^{f,1}), (x_C^2, y_C^{f,2})\}$ for performance and $C^g = \{(x_C^1, y_C^{g,1}), (x_C^2, y_C^{g,2}), (x_C^3, y_C^{g,3})\}$ for reliability. Then, we utilize two *data abstractors* to encode the observation data of performance and reliability, respectively. Similar to the vanilla NP in Fig. 5a, each data abstractor consists of two MLPs. That is, for each context pair $(x_C^i, y_C^{f,i})$ in C^f and $(x_C^j, y_C^{g,j})$ in C^g, we would obtain their representations r_i^f, u_i^f and r_j^g, u_j^g through the *data abstractors*, which are subsequently used to process the deterministic and uncertainty paths for each prediction task.

Cross-attention Aggregator. For the deterministic paths of both prediction tasks, we replace the mean aggregator used in the vanilla NP with the cross-attention mechanism. Unlike the mean aggregator, the cross-attention allows each target data item x_D to selectively focus on those context data items that are most relevant to predicting its performance and reliability output y_D^f and y_D^g. We do not use cross-attention in the uncertainty path because this path aims to capture the overall uncertainty that affects all the target predictions collectively rather than each prediction individually [18].

Complemented Profiles. Profiles from the same type of path in the two tasks could provide complementary information, which further helps improve the accuracy of each task's prediction. To achieve this, we utilize a gating mechanism to facilitate information sharing across the paths. While the context representations r^f and r^g are generated from the two deterministic paths independently, they often contain useful information for each other. The gating mechanism allows r^f to "borrow" useful features from r^g while filtering out irrelevant parts. For example, if r^g identifies high memory consumption, this information may also be useful for predicting slow query execution. By controlling how much of r^g is incorporated into r^f, the model avoids overfitting to noise while still benefiting from useful shared features. This information-sharing strategy is not limited to context representations (r^f, r^g) but is also applied to latent variables (h^f, h^g, and z), ensuring only relevant information flows between them across all model paths.

Specifically, we define four profile pairs: (r^f, r^g), (r^g, r^f), (h^f, z) and (h^g, z). To enable dynamic information sharing between each pair, the gating mechanism adjusts the influence of the second profile on the first as follows:

$$p_o' = p_o + \tanh(W_1 p_c + b_1) \times \sigma(W_2 p_c + b_2), \tag{9}$$

where σ denotes the sigmoid function, W_1, W_2 are weight matrices, and b_1 and b_2 are bias vectors. We refer to p_o' as the complemented p_o. Then, after applying the gating mechanism on the four profile pairs mentioned above, we obtained complemented representations r'^f, r'^g, and complemented latent variables h'^f, h'^g.

Prediction Generator. Finally, two generators are used to process the four complemented profiles, producing prediction outputs for both tasks. Specifically, the performance prediction is handled as a regression task, while the reliability prediction is treated as a binary classification task. Each task's complemented profiles, along with the target data item x_D, are fed into two separate MLPs. Note, the reliability prediction task uses an additional sigmoid activation function to map the MLP's output into the range $[0, 1]$.

5.4 Model Training

We then explain how to end-to-end train our AQETuner model, including the knob-plan encoder and dual-task predictor. Specifically, the dataset is randomly divided into an observation set and a target set, each comprising 50% of the data. The model takes the observation set C and x_D from the target set as input, while y_D from the target set serves as the ground truth. As directly optimizing the likelihood in Eq.8 is computationally prohibitive, we approximate it using the variational lower bound [38]:

$$\begin{aligned}
&\log p_\theta\left(y_D^f, y_D^g \mid x_D, C\right) \\
&\geq \mathbb{E}_{q(z|D)}\Bigg[\sum_{t\in\{f,g\}}\Big[\mathbb{E}_{q(h^t|z,D^t)}\left[\log p_\theta(y_D^t \mid x_D, h^t, r^t)\right] \\
&\qquad - \mathrm{KL}\left(q\left(h^t \mid z, D^t\right) \| q\left(h^t \mid z, C^t\right)\right)\Big]\Bigg] \\
&\qquad - \mathrm{KL}\left(q(z \mid D) \| q(z \mid C)\right).
\end{aligned} \tag{10}$$

This equation involves two levels of expectations: first over the intra-task variables h^t, and then over the cross-task variable z. The KL divergence terms act as regularizers, ensuring that the learned distributions for both intra-task variables and the cross-task variable remain close to their observation distributions, respectively.

5.5 Generalization To New Datasets

Real-world analytical query engines handle diverse datasets with varying schemas and query workloads, making it crucial for tuning systems to generalize to unseen datasets. AQETuner achieves this generalization in two ways. First, the knob-plan encoder encodes the query plans with the self-attention mechanism and positional encoding HSPE. Since query plans reflect SQL operations rather than specific data content, they naturally generalize to new workloads. Besides, the encoder employs the cross-attention mechanism to establish correlations between query plans and knobs, enabling AQETuner to generalize to unseen tuning cases by learning the impact of knobs on query execution. Second, the dual-task prediction is built on NPs to predict both query performance and reliability. In essence, NPs learn a distribution over possible functions based on observed data rather than a fixed input-output mapping [9]. This approach allows the predictor to adapt to unseen datasets by selecting function mappings that best adapt to new observations, enhancing AQETuner's generalization capability.

6 WARM STARTER

In this section, we introduce the warm starter, which generates high-quality initial observation data for BO-based tuning and pre-calculates the correlation between knobs and query plans.

6.1 Generating Initial Samples

At the initial stage of BO-based tuning for each target query, high-quality observation data is critical as it provides the first insights for AQETuner's tuning objectives. To generate high-quality observation data, we propose sampling points in the knob space Θ that should meet two key criteria: *diversity* across the search space and *efficiency* to explore potentially good sample regions. The diversity criterion ensures that the initial samples cover the search space as much as possible, enabling the surrogate model to capture the overall shape of the objective function. The efficiency criterion involves quickly identifying regions with potentially good performance, typically around previously observed promising samples.

However, existing sampling methods could not meet the two criteria simultaneously. Specifically, most methods [44, 51] rely on random sampling to collect initial points. Although random sampling meets the diversity criteria, it does not efficiently explore promising regions. Hunter [2] tries to address this issue by using a Genetic Algorithm to target promising regions during sampling. However, its sequential sampling process requires evaluating one point before the next can be sampled, limiting efficiency and reducing the number of samples explored within a given time period.

To address this issue, AQETuner leverages Particle Swarm Optimization (PSO) [37] for initial sample collection. PSO is an optimization method that iteratively tries to improve candidate samples based on a given quality measure. It has the strength of fast convergence without requiring prior knowledge. In PSO, a group of candidate sample points, known as *particles*, move through the search space by adjusting their positions based on their velocities. The velocity determines both the direction and speed of the particle's movement. Each particle acts as an independent agent, which maintains its best-known position and could also see the

Algorithm 2: PSO-based Sampling.

```
Input: A target query $q$; Number of required samples $M$; Number
    of particles $P$;
Output: Configuration Samples $\mathcal{S}$.
1  for each particle $i = 1$ to $P$ do
2      randomly sample a position $s_i$ and a velocity $v_i$;
3  end
4  while $|\mathcal{S}| < M$ do
5      for each particle $i = 1$ to $P$ do
6          $(f_i, g_i)$ = EvaluteOnEngine($q$, $s_i$);
7          $\mathcal{S} \leftarrow \mathcal{S} \cup (s_i, f_i, g_i)$ ;
8          if $g_i = 0$ then
9              Update particle specific best position $LB_i$ if $f_i$ is the
                 best performance in the particle $i$;
10             Update global best position $GB$ if $f_i$ is the best
                 performance so far;
11             $v_i = \mu \cdot v_i + c_1 \cdot r_1 \cdot (LB_i - x_i) + c_2 \cdot r_2 \cdot (GB - x_i)$ ;
12             $s_i = s_i + v_i$ ;
13         end
14         else
15             randomly resample $s_i$ and $v_i$ ;
16         end
17     end
18 end
19 return $\mathcal{S}$ ;
```

best-known position among all particles. The movements of particles are influenced by this information, guiding them to promising regions. In this way, the particles could produce promising samples during the optimization process. Since each particle is evaluated independently, PSO is particularly well-suited for parallelization.

PSO could perfectly meet the two key criteria in our problem. The initial positions of particles are generated randomly to ensure *diversity*. For the *efficiency* of sampling, PSO excels in parallel search, allowing particles to explore promising regions independently. Once a particle completes its evaluation, it immediately updates its next sampling position based on global and local optima, without waiting for others. This ensures fast and efficient exploration throughout the search process.

The procedure of PSO-based sampling for each target query is outlined in Algorithm 2. The procedure has only one hyperparameter, which is the number of particles. We start by randomly initializing the position of the configuration point and velocity for each particle (Lines 1~3). Next, we evaluate each configuration sample with the analytical engine to obtain the query's performance and reliability (Lines 5~7). If the sample is reliable, which means that the query executes successfully under the sampled configuration, then we update the best position for this single particle and the global best position for all the particles (Lines 9~10). Here, the particle-specific best position refers to the optimal point found by an individual particle, while the global best is the optimal point found by all particles. Next, we adjust the position and velocity for the particle's next iteration (Lines 11~12). Note that the position update equations balance exploration and exploitation from particle-specific and global best positions in the search space. Here, c_1 and c_2 are local and global acceleration coefficients, which are used to determine the degree of influence that different kinds of best positions would have on the next move. Besides, μ determines how much of the previous velocity is retained, while r_1 and r_2 are random values between $[0, 1]$ to add variability into the search process. Finally, if the sample is unreliable, we resample the position and velocity to continue the process (Line 15).

6.2 Identifying Knob and Plan Node Correlation

Based on the high-quality samples collected by the warm starter, we propose to identify a coarse correlation between knobs and plan nodes by calculating each knob's contribution to the execution time of different plan nodes (e.g., filter, join). This is essential for building the correlation matrix used by the cross-attention mechanism within the knob-plan encoder (see Section 4.4). To this end, we evaluate the target queries with the sampled configurations using the "explain analyze" statement, which provides the execution time t of each plan node. For each node type nt, we organize the evaluated results as a triplet (nt, s, t). Here, s is a vector with each element s_i representing a normalized knob value.

Next, we propose to apply the SHAP [28] method to identify the importance of each knob by computing its contribution to the prediction of node's execution time. This is because SHAP has been proven to be robust in model explanations [50]. In our scenario, each knob value s_i is treated as a feature to predict the node's execution time. Then, we utilize SHAP to calculate the knob's contribution values for each node. If the contribution value is non-zero, it indicates that the knob has influences on the execution behavior of the specific node type, and they are considered as correlated. Otherwise, they are not correlated.

6.3 Computational Complexity of AQETuner

The computational complexity of AQETuner is divided into three main components. For the knob-plan encoder, given a query with N nodes in its query plan and $|K|$ tunable knobs, the complexity is $O(|K|^2 + N \cdot |K|)$. This arises from the self-attention mechanism applied to the knobs and the cross-attention mechanism between the query plan nodes and knobs. For the dual-task predictor, the primary computational cost comes from calculating the cross-attention scores between the observation and target datasets. The resulting complexity is $O(|C| \cdot |D|)$, where $|C|$ and $|D|$ denote the sizes of the observation and target datasets, respectively. For the PSO-based warm starter, the complexity is dominated by the particle update step. Assuming each evaluation on the analytical engine takes T_{eval} time and the failure probability is λ, the total complexity is $O(\frac{M \cdot T_{eval}}{1-\lambda})$, where M is the number of required samples.

7 EXPERIMENTAL EVALUATION

Outline. In this section, we first describe the experimental setup, followed by an evaluation of AQETuner's tuning efficiency across two analytical engines through an end-to-end tuning process. Lastly, we analyze the impact of our proposed three key components.

Baselines. We evaluate AQETuner against a range of representative ML-based tuning systems, including:

ResTune [51]: A BO-based system-level tuning method that considers the resource constraints during the tuning process. To reduce query failures, we add constraints on CPU utilization and memory usage based on the DBA experience. We used Latin Hypercube Sampling (LHS) [44] to collect initial tuning samples.

Hunter [2]: An RL-based system-level tuning method that applies the Genetic Algorithm for initial exploration to handle the cold-start issue. Besides, it utilizes DDPG as its neural network architecture.

QTune^Q [25]: An RL-based tuning method that optimizes knob configurations at different-grained levels. We choose its query-level version and use random sampling to collect initial tuning data.

Bao [29]: A query-level hint selection method that converts the query plans into single feature vectors with tree convolutional neural network [34] and pooling technique. To support knob tuning, we concatenated the candidate configurations with feature vectors to make predictions and used LHS to generate initial tuning samples.

QueryFormer [53]: A Transformer-based model for query plan representation that supports query-level optimization. Similarly to Bao, we enabled QueryFormer to support knob tuning by concatenating plans' feature vectors with candidate configurations and used LHS to collect initial tuning samples.

Workloads. We evaluate AQETuner using two synthetic benchmarks and two real-world workloads: 1) TPC-DS: An OLAP benchmark consisting of 99 queries, used to assess decision support system performance; 2) TPC-H: Another OLAP benchmark with 22 queries, simulating a business-oriented and ad-hoc query workload for decision support system evaluation; 3) STATS-Analytical: Derived from the STATS dataset [14], originally designed for cardinality estimation with complex join schemas. We extend this workload with additional queries to meet practical analytical needs, such as calculating average post scores or the number of comments per post by year; 4) JOB-Extend: Based on IMDB [23], originally used to test query optimizer performance. We extend the JOB workloads [24] to include more analytical tasks. The data size for TPC-DS and TPC-H is set to 500 GB. Since the original IMDB and STATS datasets are relatively small, we uniformly scale them to 100 GB.

Environment. The experiments involve two analytical query engines, ByteHouse and Presto, each deployed in the pseudo-distributed mode with three instances. Both engines run on a server with an Intel(R) Xeon(R) Platinum 8336C CPU @ 2.30GHz and 1TB of available memory. Based on DBA expertise, 25 query-level knobs are selected for tuning in ByteHouse and 20 in Presto. These knobs include the control of memory allocation, degree of parallelism, execution options for query operators in the plan, etc.

Settings. We implement AQETuner using Python 3.8 and PyTorch 1.13. To ensure a fair comparison, all experiments were conducted without any prior tuning data, and the tuning process for each workload, consisting of multiple queries, was constrained to a maximum duration of six hours. The execution time of failed queries is set as 100 seconds to avoid demonstrating incorrect "good" query performance in the experimental results.

Once sufficient tuning data is collected, the query performance and reliability datasets are randomly split into training and test sets in a 7:3 ratio to train the knob-plan encoder. The tuning dataset is constructed by pairing query plans with their corresponding knob configurations, which are used as inputs for prediction. Each knob-plan pair in the encoder is represented with 32 dimensions, and we select 10 eigenvectors ($k = 10$) for the hierarchical spectral encoding of the plan. For the warm starter, the number of particles is set to 3, with coefficients μ, c_1, and c_2 set to 0.5, 2, and 2, respectively.

Metrics. For query performance, we analyze the average and 95th percentile (P95) of query latencies for a given workload. This analysis captures the effects of query-level tuning on overall processing and tail latency, which are critical for meeting the Service Level Objectives (SLOs) of database services [48]. For reliability, we track the number of query failures throughout the tuning process.

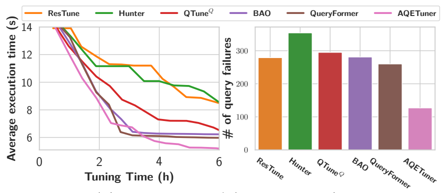
7.1 Efficiency Comparison

ByteHouse. Figure 6a-6f report the optimal average execution time achieved during the tuning process and the total number of query failures observed for different methods across various workloads in ByteHouse. Overall, AQETuner outperforms all the baseline methods across all four workloads. Notably, AQETuner achieves up to a 37.5% improvement in optimal execution time compared to system-level methods and up to a 28.3% improvement over three query-level competitors. Regarding query failures, AQETuner reduces the number by up to 51.2% compared to the next-best method. The details are as follows:

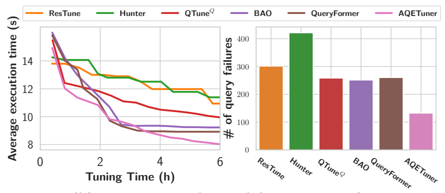
1) System-level tuning methods show similar but generally worse performance than query-level methods. Query-level tuning excels by collecting more observations and using query plans instead of SQL text for modeling. In system-level tuning, the average execution time of all queries is considered a single observation per BO cycle, resulting in a limited number of observations throughout the tuning process. In contrast, query-level tuning treats the execution time of each individual query as an observation, enabling more data collection in one cycle. Furthermore, query-level tuning methods utilize query plans, which capture more detailed execution information than system-level approaches. This enhances tuning efficiency and improves the chances of finding optimal configurations. As a result, AQETuner achieves optimal configurations that demonstrate up to 35.5%, 29.7%, 37.5%, and 24.1% lower average latency than the system-level methods under the JOB-Extend, STATS-Extend, TPC-DS, and TPC-H workloads, respectively.

2) Among the query-level methods, AQETuner surpasses the other three ones (i.e., QTune^Q, Bao, and QueryFormer) for two main reasons. First, these three methods either fail to capture the structure of the query plan or ignore the impact of knobs on plan node execution, limiting their tuning ability. Second, they neglect the cold-start issue or rely on random sampling, resulting in suboptimal exploration during the initial phase. In contrast, AQETuner leverages high-quality initial data from the PSO-based sampling method to accelerate exploration in subsequent BO-based tuning. As a result, AQETuner achieves optimal configurations that demonstrate up to 20.4%, 19.4%, 28.3%, and 10.2% lower average latency than other query-level methods under the JOB-Extend, STATS-Extend, TPC-DS, and TPC-H workloads, respectively.

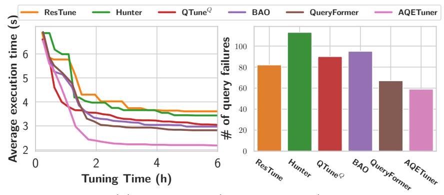
3) To further validate AQETuner's tuning efficiency in ByteHouse, we evaluate the optimal 95th percentile of query latency (P95) on the JOB-Extend workload during the tuning process, as shown in Fig. 6f. The query-level tuning methods consistently outperform the system-level methods. AQETuner has 40% lower P95 latency compared to the best system-level method. This is because system-level tuning methods do not target individual query performance as tuning objectives and cannot effectively optimize slow queries. Notably, AQETuner can reduce P95 latency up to 15.9% and 12.3% compared to Bao and QueryFormer, respectively.



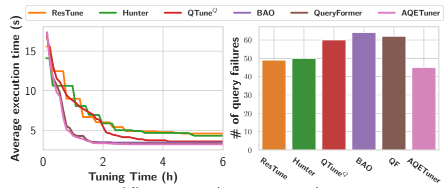
(a) JOB-Extend (ByteHouse)



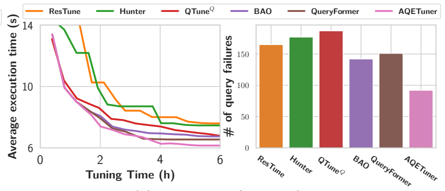
(b) STATS-Analytical (ByteHouse)



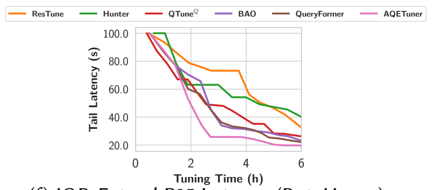
(c) TPC-DS (ByteHouse)



(d) TPC-H (ByteHouse)



(e) TPC-DS (Presto)



(f) JOB-Extend P95 Latency (ByteHouse)

Figure 6: Optimal execution time and the total number of query failures during tuning processes.

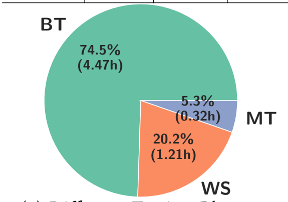
4) For the number of query failures, AQETuner achieves the lowest count across all cases, but its advantage varies by workload. Compared to the next-best method, AQETuner reduces query failures by 51.2% and 47.5% on the JOB-Extend and STATS-Analytical workloads, respectively. On TPC-DS and TPC-H, AQETuner's advantage is smaller, reducing failed executions by only 15.5% and 8%, respectively. This is because queries in TPC-DS and TPC-H are less likely to experience execution failures than those in JOB-Extend and STATS-Analytical, as they involve fewer multi-table joins, and thus demand less memory and computation resources.

Presto. To demonstrate the generality of AQETuner, we also evaluate it on another widely used analytical engine, Presto (Version 0.288). Since Presto only supports cloud storage module, we store the dataset on the HDFS for subsequent querying. We compare the tuning efficiency of different methods on Presto under TPC-DS workloads, which have relatively complex query plans. Note, we fail to run the JOB-Extend and STATS-Analytical workloads on Presto due to the relatively poor performance of cloud storage. For the TPC-H dataset, we observed results similar to those of ByteHouse. The evaluation results are shown in Fig. 6e. For query performance, the average execution time for all methods shows a similar trend, decreasing as tuning progresses. The performance ranking is exactly the same as that observed in ByteHouse: AQETuner > QueryFormer > Bao > QTune^Q > ResTune ≈ Hunter. Specifically, the optimal configurations found by AQETuner result in 20.1%, 17.6%, 9.4%, 8.8% and 6.1% lower average latency compared to ResTune, Hunter, and QTune^Q, Bao and QueryFormer, respectively. Regarding query failures, AQETuner also achieves the lowest number, reducing failed executions by 44.3%, 48.1%, 50.8%, 42.2% and 46.7% compared to ResTune, Hunter, QTune^Q, Bao and QueryFormer.

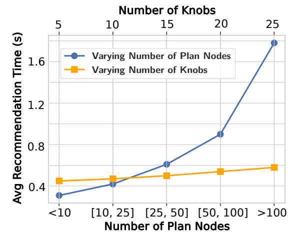
7.2 Study of AQETuner's Tuning Cost

To evaluate the tuning cost of AQETuner, we conducted two experiments on ByteHouse with the TPC-DS workload. First, we divided AQETuner into three phases: warm start, model training (for the knob-plan encoder and dual-task predictor), and BO-based tuning. Over a 6-hour query-level tuning session, we measured the memory usage and computational time for each phase. As shown in Fig. 7a, the BO-based tuning is the most time-intensive, consuming 74.5% of the total time and 3.2 GB of memory due to its extensive Bayesian Optimization exploration. Model training, while requiring 20.2% of the time, exhibits the highest memory usage at 9.4 GB to support the encoder and predictor training. In comparison, the warm start phase requires only 5.3% of the time and 88.3 MB of memory, demonstrating its efficiency in initializing the tuning process.

Phase	Warm Start(WS)	Model Training(MT)	BO-based Tuning(BT)
Memory Usage	88.3 Mb	9.4 Gb	3.2 Gb



(a) Different Tuning Phases



(b) Effect of Plan Nodes and Knobs

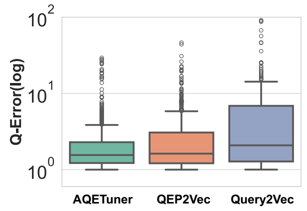
Figure 7: Tuning Cost of AQETuner.

Next, we analyzed the impact of the number of plan nodes and tunable knob on the average recommendation time in the BO-based tuning phase, as shown in Fig. 7b. The results reveal that the recommendation time increases significantly with the number of plan nodes, particularly when exceeding 100 nodes, highlighting a notable scalability challenge for complex query plans. Besides, the relatively limited number of tunable query-level knobs has a comparatively smaller impact on the recommendation time.

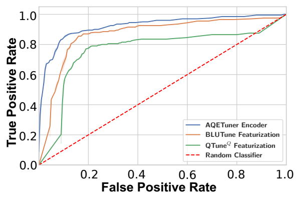
7.3 Evaluation of Knob-Plan Encoder

AQETuner's knob-plan encoder is the first to leverage the correlation between knobs and query plans. To validate its effectiveness, we compare it with two representative encoders: the Query2Vec encoder for query encoding [25] and the QEP2Vec encoder for plan encoding [15]. The Query2Vec converts SQL into vectors by encoding key elements (e.g., query type, involved tables, and operations). The QEP2Vec treats query plans as documents and sub-plans as words, using word2vec [31] to generate embeddings.

To evaluate the effectiveness of different encoders, we apply them to two downstream prediction tasks: query performance and reliability. Each encoder is followed by the same MLP layer to produce the final prediction. We concatenate the query/plan encodings from Query2Vec and QEP2Vec with AQETuner's self-attention-based knob encoding, forming the feature vectors for knob-plan pairs. The evaluation datasets are obtained from the tuning observations of Section 7.1, which contained the true prediction targets. For predict evaluation, we use the Q-Error metric [33], which measures



(a) Query Performance Accuracy



(b) ROC Curve of Query Reliability

Figure 8: Impact Evaluation of Knob-plan Encoders.

Table 1: Evaluation of Surrogate Models.

Surrogate Model	Avg. Infer Time (s)	# of Observations	RMSE
GP	2.29	2827	3.14
CGP	2.10	2842	3.49
DTP	**0.58**	**3831**	**2.96**

the prediction error as: Q-Error(t) = $\max(\frac{actual(t)}{predicted(t)}, \frac{predicted(t)}{actual(t)})$, where t is the execution time. For reliability evaluation, we use the ROC curve, a common tool for assessing classification models.

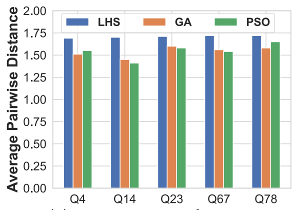
Figure 8a shows the Q-Error distributions for performance predictions. The AQETuner Encoder achieves the lowest median error and a more compact interquartile range (IQR), indicating more consistent and accurate performance predictions. As shown in Fig. 8b, the red dashed line indicates the baseline performance of a random classifier, with all three methods performing substantially better. The curve of AQETuner Encoder is closest to the upper left corner, which indicates that it has the highest prediction accuracy.

In both prediction tasks, Query2Vec has the worst performance as it only extracts keywords and cost information from the query plan without considering its structure. QEP2Vec improves it by incorporating plan structures using fixed templates, such as predefined multi-way joins. However, it still struggles to capture the complex patterns present in analytical queries. In contrast, AQETuner's encoder utilizes fined-grained featurizations and attention mechanisms to capture the structural information of the query plans and knob-plan correlations, both of which are critical to the two downstream prediction tasks. The experimental results show that AQETuner's encoder is more effective than other methods.

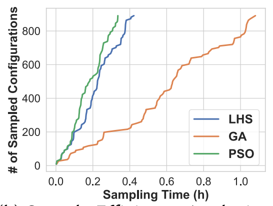
7.4 Effect of Dual-Task Predictor

To evaluate the effectiveness of the dual-task predictor (DTP) in AQETuner, we compare it with two alternative models: Gaussian Process (GP) and Contextual Gaussian Process (CGP) [20]. GP uses an RBF kernel to determine the similarity between two points in the input space, while CGP incorporates contextual information (i.e., query plans) with a composite kernel. Since GP and CGP can only handle a single prediction objective, we set their objective to query performance. For a fair comparison, we use the cross-attention-based knob-plan encoding from AQETuner's encoder as input to GP. For CGP, which can capture the relationship between plans and knobs through its composite kernel, we use only the self-attention-based plan encoding as its input.

Over a fixed 6-hour tuning, we evaluate different surrogate models based on their average inference time, the number of collected observations, and the RMSE of prediction for execution time on the JOB-Extend workload. As shown in Table 1, we observe that DTP infers faster than GP and CGP. This is because GP and CGP need to compute the covariance matrix with cubic complexity regarding the number of observations. Besides, during the fixed duration,



(a) Diversity Analysis.



(b) Sample Efficiency Analysis.

Figure 9: Evaluation of Different Warm Starters.

DTP could collect more observations than GP and CGP due to its lightweight computation and less knob recommendation time. Regarding RMSE, DTP yields the lowest error compared to GP and CGP, since it effectively leverages observation data on query reliability and exploits its correlation with query performance.

7.5 Assessment of Warm Starter

To demonstrate the effectiveness of the PSO-based warm-starter, we compare it with two methods: Latin Hypercube Sampling (LHS) and Genetic Algorithm (GA). LHS is commonly used in BO-based tuning methods to generate initial samples [44, 51, 52], while GA is employed by Hunter [2] to address the cold-start issue. The comparison focuses on two criteria: *diversity* and *efficiency*. For *diversity*, we calculate the average pairwise Euclidean distance between sampling points in the knob space of each method. We selected TPC-DS queries Q4, Q14, Q23, Q67, and Q78 for evaluation due to their complex query structures. As shown in Fig. 9a, we observe that LHS achieves the largest distance, indicating the highest diversity. This is because LHS is a near-random sampling method that ensures the knob space is fully explored. GA- and PSO-based methods show smaller average distances between their sample points as they tend to focus on potentially good regions. However, the distance gap between these methods is not significant.

To measure the sample *efficiency*, we compare how the number of samples increases over time for different methods, as shown in Fig. 9b. All the methods terminate after finding 900 sample points. We observe that PSO requires 20.6% less time than LHS and 68.9% less time than GA to gather sufficient samples. While LHS performs well in maintaining sample diversity, it does not consider the tuning objective, leading to a longer execution time for its sample points than those from PSO, resulting in lower sampling efficiency. PSO is also much faster than GA because it allows parallel sampling, whereas GA requires the analytical engine to evaluate each point before the next one can be sampled.

8 CONCLUSION

This paper presents AQETuner, a BO-based tuning approach for reliable query-level tuning in analytical query engines. It utilizes a knob-plan encoder to capture the correlation between tunable knobs and query plans and a dual-task model to predict query performance and reliability. Besides, it employs a PSO-based warm-starter to address the cold-start issue. AQETuner achieves the best results for various benchmarks compared to the SOTA methods.

ACKNOWLEDGMENTS

This work is supported by NSFC (62202171, 62461146205), Fundamental Research Funds for the Central Universities, and ByteDance Academic Project (PJ20240202900021).

REFERENCES

[1] ByConity. 2024. Complex Query Tuning for ByteHouse. https://byconity.github.io/docs/query-optimization/complex-query-optimization#how-to-enable-query-level-knobs. [Accessed 2024/06/28].

[2] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: an online cloud database hybrid tuning system for personalized requirements. In *SIGMOD*. 646–659.

[3] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: a contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *PVLDB* 14, 8 (2021), 1401–1413.

[4] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *SIGMOD*. 215–226.

[5] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *PVLDB* 2, 1 (2009), 1246–1257.

[6] Andrea Galassi, Marco Lippi, and Paolo Torroni. 2020. Attention in natural language processing. *IEEE transactions on neural networks and learning systems* 32, 10 (2020), 4291–4308.

[7] Jacob R Gardner, Matt J Kusner, Zhixiang Eddie Xu, Kilian Q Weinberger, and John P Cunningham. 2014. Bayesian optimization with inequality constraints.. In *ICML*, Vol. 2014. 937–945.

[8] Marta Garnelo, Dan Rosenbaum, Christopher Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo Rezende, and SM Ali Eslami. 2018. Conditional Neural Processes. In *ICML*. PMLR, 1704–1713.

[9] Marta Garnelo, Jonathan Schwarz, Dan Rosenbaum, Fabio Viola, Danilo J Rezende, SM Eslami, and Yee Whye Teh. 2018. Neural Processes. *arXiv preprint arXiv:1807.01622* (2018).

[10] Jia-Ke Ge, Yan-Feng Chai, and Yun-Peng Chai. 2021. WATuning: a workload-aware tuning system with attention-based deep reinforcement learning. *JCST* 36, 4 (2021), 741–761.

[11] Mozhdeh Gheini, Xiang Ren, and Jonathan May. 2021. Cross-Attention is All You Need: Adapting Pretrained Transformers for Machine Translation. In *EMNLP*. 1754–1765.

[12] Albert Gu, Caglar Gulcehre, Thomas Paine, Matt Hoffman, and Razvan Pascanu. 2020. Improving the gating mechanism of recurrent neural networks. In *International conference on machine learning*. PMLR, 3800–3809.

[13] Yuxing Han, Haoyu Wang, Lixiang Chen, Yifeng Dong, Xing Chen, Benquan Yu, Chengcheng Yang, and Weining Qian. 2024. ByteCard: Enhancing ByteDance's Data Warehouse with Learned Cardinality Estimation. In *SIGMOD*. 41–54.

[14] 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. 2022. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *PVLDB* 15, 4 (2022), 752–765.

[15] Connor Henderson, Spencer Bryson, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. 2022. BLUTune: Query-informed multi-stage IBM db2 tuning via ML. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 3162–3171.

[16] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *science* 313, 5786 (2006), 504–507.

[17] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *Journal of Machine Learning Research* (2013).

[18] Hyunjik Kim, Andriy Mnih, Jonathan Schwarz, Marta Garnelo, S. M. Ali Eslami, Dan Rosenbaum, Oriol Vinyals, and Yee Whye Teh. 2019. Attentive Neural Processes. In *ICLR*.

[19] Diederik P Kingma. 2014. Auto-encoding variational bayes. *ICLR* (2014).

[20] Andreas Krause and Cheng Ong. 2011. Contextual gaussian process bandit optimization. *Advances in neural information processing systems* 24 (2011).

[21] Mayuresh Kunjir and Shivnath Babu. 2020. Black or white? how to develop an autotuner for memory-based analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1667–1683.

[22] Nikolay Laptev, Wenbo Tao, Caner Komurlu, Jason Xu, Deke Sun, Thomas CH Lux, and Luo Mi. 2022. Smarter Warehouse. In *2022 IEEE 38th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 1–8.

[23] 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.

[24] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27 (2018), 643–668.

[25] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A query-aware database tuning system with deep reinforcement learning. *PVLDB* 12, 12 (2019), 2118–2130.

[26] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. http://arxiv.org/abs/1509.02971

[27] Chang Liu, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Zhenyu Guo, and Thomas Moscibroda. 2014. Automating distributed partial aggregation. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–12.

[28] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

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

[30] Manish Mehta and David J DeWitt. 1995. Managing intra-operator parallelism in parallel database systems. In *VLDB*, Vol. 95. 382–394.

[31] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[32] 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.

[33] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[34] 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 AAAI conference on artificial intelligence*, Vol. 30.

[35] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. 2021. A review on the attention mechanism of deep learning. *Neurocomputing* 452 (2021), 48–62.

[36] Presto. 2024. Presto Properties Reference. https://prestodb.io/docs/current/admin/properties.html. [Accessed 2024/06/28].

[37] Tareq M Shami, Ayman A El-Saleh, Mohammed Alswaitti, Qasem Al-Tashi, Mhd Amen Summakieh, and Seyedali Mirjalili. 2022. Particle swarm optimization: A comprehensive survey. *Ieee Access* 10 (2022), 10031–10061.

[38] Jiayi Shen, Xiantong Zhen, Qi Wang, and Marcel Worring. 2024. Episodic multi-task learning with heterogeneous neural processes. *Advances in Neural Information Processing Systems* 36 (2024).

[39] Snowflake. 2024. Parameters, Snowflake Documentation. https://docs.snowflake.com/en/sql-reference/parameters. [Accessed 2024/06/28].

[40] David G Sullivan, Margo I Seltzer, and Avi Pfeffer. 2004. Using probabilistic reasoning to automate software tuning. *SIGMETRICS* 32, 1 (2004), 404–405.

[41] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, et al. 2023. Presto: A Decade of SQL Analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[42] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark C - Standard Specification, Revision 5.11. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf. [Accessed 2024/06/28].

[43] Transaction Processing Performance Council (TPC). 2021. TPC BENCHMARK™ DS Standard Specification Version 3.2.0. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf. [Accessed 2024/06/28].

[44] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*. 1009–1024.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[46] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang. 2017. Residual attention network for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3156–3164.

[47] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A normalizing flow based cardinality estimator. *VLDB* 15, 1 (2021), 72–84.

[48] Hao Xu and Juan A Colmenares. 2024. Bouncer: Admission Control with Response Time Objectives for Low-latency Online Data Systems. In *Companion of the 2024 International Conference on Management of Data*. 400–413.

[49] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*. 415–432.

[50] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2021. Facilitating database tuning with hyper-parameter optimization: a comprehensive experimental evaluation. *arXiv preprint arXiv:2110.12654* (2021).

[51] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *SIGMOD*. 2102–2114.

[52] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards dynamic and safe configuration tuning for cloud databases. In *SIGMOD*. 631–645.

[53] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A tree transformer model for query plan representation. *PVLDB* 15, 8 (2022), 1658–1670.