OBELISK: Efficient Offline Query Planning with Bayesian Optimization-Informed Language Model Reasoning

Zhicheng Pan*
East China Normal University
zcpan@stu.ecnu.edu.cn

Wenwen Sun
East China Normal University
wwsun@stu.ecnu.edu.cn

Yuanjia Zhang
PingCAP
zhangyuanjia@pingcap.com

Terence Purcell
PingCAP
terry.purcell@pingcap.com

Yu Dong
PingCAP
yu.dong@pingcap.com

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

Rong Zhang
East China Normal University
rzhang@dase.ecnu.edu.cn

Xuan Zhou
East China Normal University
xzhou@dase.ecnu.edu.cn

Jianliang Xu
Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

ABSTRACT

Query optimization (QO) remains a fundamental challenge in the database community. Despite decades of research, cost-based QO (CQO) is still susceptible to performance regressions due to inherent inaccuracies in cardinality estimation, cost modeling, and plan enumeration. To mitigate the instability, modern databases employ SQL plan management (SPM), which reuses curated plans and bypasses CQO. However, there exists a fundamental issue in SPM: *how can we efficiently identify the optimal plans to manage*? The existing approach falls short due to low generalizability and poor interpretability. Thus, we argue for revisiting this problem from a novel perspective, where we intervene in the sensitivity of CQO through well-designed cost scaling knobs. Nevertheless, this transformation poses three key challenges: (1) efficient search guidance, (2) comprehensive semantic utilization, and (3) cost-effective performance evaluation. To address these challenges, we propose OBELISK, an Offline Bayesian optimization-informed quEry pLannIng framework, with language model reaSoning over cost scaling Knobs. OBELISK is training-free and can efficiently find optimal query plan through a closed-loop process: a timeout-constrained Bayesian optimization technique to identify promising knob subspaces, thereby informing the search; a feedback-aware self-evolving reasoner to recommend knob configurations; and a lightweight evaluator with history-based admission gatekeeper to avoid redundant evaluations. Extensive experiments on well-established benchmarks demonstrate the effectiveness and superiority of our OBELISK.

PVLDB Reference Format:
Zhicheng Pan, Wenwen Sun, Yuanjia Zhang, Terence Purcell, Yu Dong, Chengcheng Yang, Rong Zhang, Xuan Zhou, and Jianliang Xu. OBELISK: Efficient Offline Query Planning with Bayesian Optimization-Informed Language Model Reasoning. PVLDB, 19(7): 1674 - 1687, 2026.
doi:10.14778/3801059.3801077

*Zhicheng Pan is also affiliated with Hong Kong Baptist University.
†Chengcheng Yang is the corresponding author.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801077

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DaSECandyLab/obelisk-offlineqo.

1 INTRODUCTION

Cost-based query optimization (CQO) is the long-standing "brain" of relational database management systems (DBMSs), translating the declarative SQL into the executable plan [15, 47]. The enduring dominance of CQO stems from its heuristic cost model [17, 64]. Generally, the cost model consists of a set of formulas that estimate the cost of each physical query operator, along with rules for aggregating these costs throughout the entire plan. During the plan enumeration process, classic *Selinger-style* optimizers [46] usually employ dynamic programming to explore the search space and select the plan with the lowest estimated cost. This plan is assumed to closely approximate the optimal plan observed at runtime.

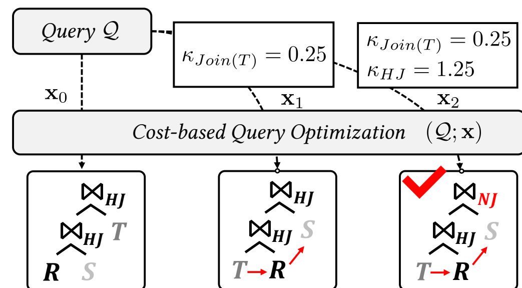


Figure 1: A Motivating Example.

However, CQOs cannot guarantee *optimality* [17, 75] due to cardinality and cost estimation errors. To alleviate this issue, learning-based query optimizers (LQOs) [4, 27, 33, 55, 65, 70] are widely studied, which leverage machine learning (ML) techniques to enhance or replace key components of the optimizer, such as cardinality estimation [19, 25, 45, 61, 66, 77], cost modeling [33, 58, 59, 64, 74], and plan selection [4, 7, 8, 32]. Although these approaches have demonstrated significant performance improvements on benchmark workloads, they face serious challenges in real-world deployments [3, 76]. First, they generally rely on neural network-based methods that lack interpretability. Second, they require extensive training on specific workloads and hardware environments, which

limits their generalizability. Moreover, they aim to reduce average latency, but fail to address tail latency issues [72, 73].

Despite extensive efforts to improve both CQOs and LQOs, they remain vulnerable to performance regression issues [56]: *queries that were once executed efficiently might degrade over time.* Moreover, production workloads are often dominated by a small set of representative queries that are executed repeatedly, which calls for more refined optimization [16, 59]. Recently, BayesQO [50] introduces the concept of offline QO, which uses offline resources to search the optimal plan and record it for online reuse. BayesQO leverages Bayesian optimization (BO) [11, 12, 14] to navigate a deep latent embedding space in search of optimal plans. However, its effectiveness relies heavily on schema-specific query representation model, which introduces two key limitations. First, exploration in the latent space is opaque and difficult to debug. Second, learned representations lack generalizability across workloads and systems. Furthermore, BayesQO remains unaware of the intrinsic flaws of the CQO, providing little valuable feedback for optimizer development.

Fortunately, the core methodology of BayesQO inspires us: why not apply BO within a *white-box* "search space" that carries concrete and practical database semantics? A naive approach is query hint (e.g., *disable_hash_join*), which is widely used in several LQOs [4, 7]. However, the hint is a coarse-grained intervention: it often enforces all-or-nothing constraints, i.e., "*disable_hash_join* = ON" forbids any hash joins, even when the optimal plan requires them locally for specific sub-joins. This characteristic, to some extent, limits ability of hint-based methods to find the optimal plan. Furthermore, the hint-based search space is explicitly discrete and not diverse. For example, Bao [32] leverages only 49 fixed hint sets to explore candidate plans, but fails to find the optimal plan in an offline scenario [50]. Therefore, hint-based methods are only suitable for the online scenario, which requires that plan generation is extremely fast, whereas the offline scenario affords a sufficient budget for finding the optimal plan and demands a more exhaustive plan exploration. Then, we recall the cornerstone of CQO, i.e., cardinality and cost estimation. If we can effectively adjust these estimates, the optimizer is able to find the optimal plan [16–18, 25, 35, 42, 43, 48, 51, 58, 60, 64, 74].

Therefore, we propose to connect the offline QO with a suite of fine-grained *cost scaling knobs* (*C-knob*s). As shown in Fig. 1, such knobs can selectively scale the estimated cost of specific physical operators (e.g., knob $\kappa_{\texttt{HJ}}$) or the cost associated with joining *a base table* (e.g., knob $\kappa_{\texttt{Join(T)}}$), thereby adjusting the optimizer's sensitivity to different cost components. We extend the conventional CQO to be conditioned on a knob configuration $\mathbf{x}$. Formally, we denote this process as $\texttt{CQO}(Q; \mathbf{x})$, where all knobs default to 1. Injecting a non-default configuration $\mathbf{x}$ intervenes in the CQO process and produces a query execution plan denoted by $P(\mathbf{x})$. ① Under the default configuration $\mathbf{x}_0$, the CQO underestimates the cardinality of `Join(R,S)`, and chooses $P(\mathbf{x}_0)$ = `HashJoin(HashJoin(R,S),T)`, which leads to poor runtime. ② By decreasing $\kappa_{\texttt{Join(T)}}$ to `0.25`, we reduce the estimated cost of joining base table `T`, making plans that join `T` earlier more favorable in the cost model. As a result, the optimizer selects a better join order $P(\mathbf{x}_1)$ = `HashJoin(HashJoin(T,R),S)`. ③ Building on this improved join order, we further adjust an operator-level knob by slightly increasing $\kappa_{\texttt{HJ}}$ to `1.25`. This change biases the optimizer away from hash join for the final join, leading to an alternative join method that better matches the intermediate results. Consequently, $P(\mathbf{x}_2)$ = `NestedLoopJoin(HashJoin(T,R),S)` achieves the most efficient execution[1]. While we introduce these *C-knob*s, we need to address the following three challenges:

Efficient Search Guidance (C1). As we treat the offline QO problem as a *knob tuning* problem, a natural question arises: why not directly apply existing knob tuning methods? The distinction lies in the different nature of our *C-knob* and traditional knobs. Specifically, traditional knob tuning methods [5, 13, 49, 52] focus on system parameters (e.g., *buffer_size*). Changes to these parameters often yield a smooth and continuous objective function [14]. This characteristic allows traditional tuners to effectively navigate the search space. In contrast, our *C-knob*s indirectly affect performance by guiding the optimizer's selection of a discrete query plan. This results in a non-smooth, step-wise objective function, as multiple distinct *C-knob* configurations can map to the same plan and yield identical performance [58, 64]. Moreover, poor configurations can trigger extremely long-running plans or even crashes, making feedback costly or unavailable. Consequently, searching these redundant or poor configurations offers no information gain, thus hindering the search efficiency. This nature renders traditional tuners, which presume a continuous response surface, ineffective in our scenario.

Comprehensive Semantic Utilization (C2). A purely statistical approach treats *C-knob*s as abstract dimensions in a vector space, ignoring their underlying physical meaning within the query optimizer. This "blind" search might waste significant time exploring poor configurations [1]. In contrast, an effective strategy should leverage the rich semantics of the knobs. For instance, understanding that increasing the cost of *hash join* is a targeted intervention to encourage the selection of *merge join* or *nested loop join.* Furthermore, traditional statistical methods are inflexible. They struggle to incrementally integrate high-level reasoning behind observed failures. For example, after discovering that some specific knob settings consistently cause plan timeouts for queries involving large relations, this semantic insight cannot be easily encoded to proactively prune entire regions of the search space. An ideal optimization framework should be able to internalize such lessons from a handful of meaningful observations—both successes and failures.

Cost-effective Performance Evaluation (C3). The ultimate measure of a plan's quality is its true execution latency, but invoking this "ground truth" evaluation for every plan is prohibitively expensive. This evaluation bottleneck is compounded by two key factors. First, the mapping from knob configurations to physical plans is many-to-one. That is, different knob settings might cause the optimizer to generate the identical execution plan. Re-executing these duplicate plans is a redundant waste of resources. Second, even an unexplored plan may be *predictably disastrous* if it contains sub-structures (e.g., a specific join operator on a pair of large tables) that have consistently demonstrated poor performance in prior executions. A naive evaluation strategy would blindly execute such a plan, but only rediscovers what historical data already finds.

Contributions. To address these challenges, we propose OBELISK, a novel offline query planning framework that tunes cost scaling knobs to find the optimal plan. Specifically, to tackle **C1**, we design a timeout-constrained statistical surrogate model with dual

[1] In TiDB, `NestedLoopJoin` can leverage indices of the *inner* table to accelerate.

objectives: maximizing performance while avoiding timeouts. As statistical models are constructed from empirical data and thus exhibit higher reliability, we leverage them to guide the search towards promising subspaces of knob configurations during the tuning process. To address **C2**, we combine the language model's comprehensive semantic understanding with the statistical model's guidance in the knob configuration space, and introduce a feedback-aware, self-evolving reasoner. This reasoner utilizes statistically informed contextual example generation and iteratively searches for improved solutions through feedback-driven evolution. To overcome **C3**, we propose a two-stage performance evaluator. The evaluator first employs a lightweight admission controller, built upon historical executions, to rapidly filter out redundant or clearly sub-optimal configurations without execution. Moreover, the evaluator provides real-time feedback to the reasoner and augments observations with historical data, thereby facilitating continuous updates to the statistical surrogate. These key components eventually form a closed loop, enabling efficient offline query planning.

In summary, we make the following contributions:

- We introduce a suite of physical *C-knob*s and logical *C-knob*s for intervening in cost-based query optimization.
- We formulate offline query planning as a *C-knob* tuning problem, without losing the expertise embedded in traditional optimizers.
- We propose a generic optimization framework that integrates the ability of Bayesian optimization to identify promising regions with the context-aware reasoning abilities of LLMs, thereby fully leveraging their synergistic strengths.
- We design a timeout-constrained surrogate model with dual objectives of ensuring high performance while avoiding timeouts.
- We introduce a self-evolving reasoner that leverages statistically informed contextual example generation and searches for improved solutions through feedback-driven evolution.
- We devise a lightweight admission-controlled evaluator that fully exploits historical execution information to avoid redundant evaluations and augment observable performance data.
- Extensive evaluations on widely used benchmarks demonstrate that OBELISK achieves significant improvements in both effectiveness and efficiency.

2 BACKGROUND AND RELATED WORK

2.1 Query Optimization

Learned Query Optimization. Recent works have explored the integration of ML techniques into several components, such as learned cardinality estimators [19, 25, 45, 61, 66, 77], join order optimization [34, 63, 69], cost modeling [17, 18, 35, 48, 58, 64, 74], etc. We refer to them as component-enhanced *learned query optimization* (LQO). In contrast, the end-to-end LQOs [7, 8, 32, 33, 55, 65, 70] offer a holistic framework and can be broadly divided into two categories: *full* LQOs and *steering* LQOs. Specifically, *full* LQOs replace CQO [33, 65] by synthesizing entire query execution plans from scratch, while *steering* LQOs sit on top of traditional CQO [4, 7, 32]. However, these works are all black-box methods that lack stability, which are difficult to deploy in production systems [3, 56, 62].

Offline Query Optimization. To the best of our knowledge, BayesQO [50] is the first work to formalize the problem of offline query optimization. It formats queries as strings inspired by molecular dynamics, which are then encoded into a high-dimensional latent space by a variational auto-encoder (VAE). Next, BO is employed to search for the optimal query plan in this latent space. However, BayesQO suffers from two primary limitations. First, its VAE representation is tightly coupled with the training schema, which limits its transferability. Second, searching within the high-dimensional latent space is inefficient. Consequently, it prevents BayesQO from finding the optimal plan efficiently. Besides, another related work, LimeQO [68], focuses on workload-level optimization by leveraging matrix completion techniques. Since our focus is on query-level optimization, aligning with BayesQO, we do not include the workload-level method LimeQO in comparative analysis.

2.2 Bayesian Optimization

Bayesian optimization (BO) [14] aims to find an optimal solution that maximizes a black-box function $f : \mathcal{X} \rightarrow \mathbb{R}$ over a domain $\mathcal{X}$, i.e., $x^* = \arg\max_{x \in \mathcal{X}} f(x)$, by sequentially selecting query points.

Vanilla BO Loop. BO operates by constructing a probabilistic surrogate model of the objective function [10, 23, 26, 52], which is iteratively refined as new observations are acquired. The general optimization procedure follows these steps: ❶ *Initialization*: build a surrogate model $\tilde{f}_\theta$ to approximate the objective function f. ❷ *Sampling next point*: optimize the acquisition function to select the next point $\mathbf{x}_{\text{next}}$ to evaluate, balancing exploration and exploitation. ❸ *Verification*: evaluate the actual performance value $f(\mathbf{x}_{\text{next}})$. ❹ *Surrogate update*: update the surrogate model from $\tilde{f}_\theta$ to $\tilde{f}'_\theta$ with the new observation $(\mathbf{x}_{\text{next}}, f(\mathbf{x}_{\text{next}}))$ and repeat steps ❷–❹.

2.3 SQL Plan Management

To mitigate performance instability caused by plan regression, database systems usually employ SQL plan management (SPM) [44, 78]. In SPM, the *plan baseline* is a repository of verified high-quality plans, denoted as $\mathcal{P}$ for a given Q. When a query is compiled, the optimizer generates a best-cost plan P'. Then, the SPM checks whether $P' \in \mathcal{P}$. If so, the plan P' is selected and executed directly. If not, to prevent a potential regression, the system is constrained to select the lowest-cost plan within the existing plan baseline.

Unlike Oracle-style SPM which manages multiple baseline plans, TiDB SPM is distinct in two main aspects. First, TiDB employs a SQL plan binding strategy, which preserves only a single, verified optimal plan. This enforcement is achieved by reverse-engineering the plan into a set of hints that are automatically injected during query compilation. Second, TiDB incorporates an offline *plan evolution* process to discover even better plans over time. In this process, newly discovered plans are periodically tested in a "shadow mode", where their runtime performance is measured without affecting the live query. A candidate plan will only be promoted to replace the current binding if its execution time is demonstrably faster. Consequently, our work falls on the plan evolution module, with the goal of accelerating the identification of the optimal plan binding.

3 OFFLINE QUERY PLANNING VIA C-KNOB

3.1 Cost Scaling Knob

It is notoriously difficult to diagnose and resolve the root cause of QO issue due to the complexity of query optimizers [3, 56, 76]. While existing query optimizers expose some cost units (hereafter referred

to as *C-units*), e.g., *cpu_tuple_cost*, they cannot be effectively tuned. First, *C-units* globally alter the cost weights of CPU or memory resources, which cannot offer fine-grained control over individual operator behavior. Consequently, adjusting a single *C-unit* to fix one regression often has unintended system-wide side effects, which might degrade the performance of unrelated queries. Besides, *C-units* offer no direct influence for the join order or operator type. As such, tuning them cannot lead to diverse query plans. Finally, *C-units* are rarely adjusted in practice and require deep expertise to tune, making them unfriendly for non-expert users.

In contrast, our *C-knob* provides fine-grained control to CQO. Specifically, a *C-knob* allows to selectively increase or decrease the estimated cost of specific operators, without altering the underlying CQO logic. Specifically, ❶ users can manually calibrate the sensitivity of the optimizer, e.g., over- or under-estimation of a certain operator. Moreover, the well-tuned knob configuration can be tailored to specific production instances or workloads, which makes the optimizer tunable per instance. ❷ The effects of these knobs reuse the decades of expertise within the CQO logic. This ensures that the tuning process benefits from the optimizer's robustness.

How These Knobs Work? A cost-based query optimizer takes a SQL query Q as input and selects the plan with the minimum estimated cost [47, 58, 64]. With cost-scaling knobs enabled, the plan selection is conditioned on a knob configuration $\mathbf{x}$:

$$P(\mathbf{x}) = \arg \underbrace{\min_{P' \in \mathcal{P}}}_{①} \underbrace{C(P'; \mathbf{x})}_{②}, \tag{1}$$

where $\mathcal{P}$ is the set of candidate plans considered by the optimizer, and $C(\cdot)$ is the cost model. Note, our knobs do not alter the enumeration logic or the plan space $\mathcal{P}$; they intervene only through the objective term by extending $C(P')$ to $C(P'; \mathbf{x})$. For example, given an operator $op_i \in P'$ associated by a knob value $x_{op_i} \in \mathbf{x}$, the optimizer's base operator cost $C(op_i)$ is scaled to $x_{op_i} \cdot C(op_i)$. The overall plan cost $C(P'; \mathbf{x})$ is then obtained by aggregating all scaled operator costs under the optimizer's original cost-composition rules.

This mechanism differs fundamentally from *steering* LQOs (e.g., Bao [32]), which apply hard constraints (e.g., `enable_hash_join = off`) to prohibit entire classes of operators. Such hints directly prune the search space $\mathcal{P}$ (① in Eq. 1) by removing all plans that contain the disabled operator type, regardless of their local optimality. In contrast, our knobs apply continuous scaling to the cost objective (②), thereby biasing—but not forbidding—the optimizer's choices. As a result, the optimizer continues to enumerate plans from the full space $\mathcal{P}$ and might still select a penalized operator in subplans where it remains locally superior to other operators.

Remark. The theoretical plan space of OBELISK is equivalent to the search space $\mathcal{P}$ of the TiDB optimizer. However, the reachability of OBELISK is limited by the internal heuristics of the optimizer, e.g., the join enumerator. Besides, for any knob configuration, the resulting plan is still produced through a full CQO pipeline, which incorporates decades of expertise. Therefore, OBELISK may not *forcefully construct* arbitrary plans outside the optimizer [65].

3.2 Problem Formalization

Given a set of n knobs $\boldsymbol{\kappa} = \{\kappa_1, \kappa_2, \ldots, \kappa_n\}$, where κ_j is defined over a value domain $\Theta_j \subseteq \mathbb{R}$. The configuration space $\mathcal{X}$ can be defined as $\mathcal{X} = \Theta_1 \times \Theta_2 \times \cdots \times \Theta_n$. A configuration $\mathbf{x} = (x_1, \ldots, x_n) \in \mathcal{X}$ affects the optimizer during the plan enumeration [58, 64].

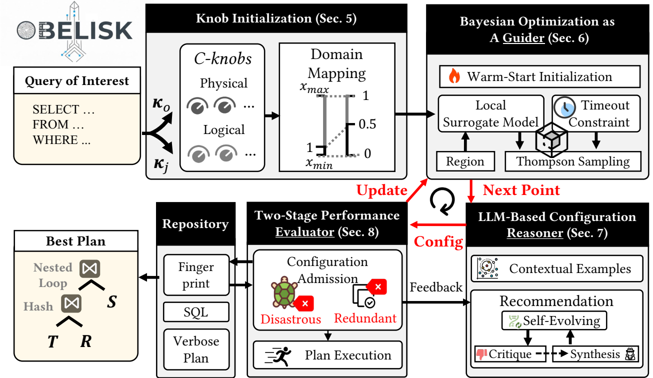


Figure 2: Overview of OBELISK.

For a configuration $\mathbf{x} \in \mathcal{X}$, the optimizer selects a plan $P(\mathbf{x})$ with lowest estimated cost. Formally, $P(\mathbf{x}) = \arg\min_{P' \in \mathcal{P}} C(P'; \mathbf{x})$, where $\mathcal{P}$ denotes equivalent candidate plans. Then, $P(\mathbf{x})$ is executed and returns an observed latency $y \in \mathbb{R}^+$. We omit the symbol of SQL query Q (as Q is fixed when tuning), and model the performance function as $y = f(\mathbf{x})$. Therefore, the problem studied in this paper is to find an optimal configuration $\mathbf{x}^* \in \mathcal{X}$ that minimizes $f(\mathbf{x})$:

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}). \tag{2}$$

4 OVERVIEW OF OBELISK

As shown in Fig. 2, OBELISK includes the following main components: ❶ Knob initialization, ❷ Bayesian optimization as a guider (GUIDER), ❸ LLM-based configuration reasoner (REASONER), and ❹ Two-stage performance evaluator (EVALUATOR). More specifically,

❶ Knob Initialization (§5). Given a specific query, we first identify its relevant knobs based on relevance detection (§5.2). Then, we normalize the feasible range of such knobs to a tunable [0,1] range since their value domains vary significantly (§5.3).

❷ Bayesian Optimization As a Guider (§6). To guide the offline QO process, we build timeout-constrained statistical surrogates with dual objectives: maximizing performance while avoiding timeouts. At each iteration, GUIDER proposes the next point $\mathbf{x}_{\mathrm{BO}}$ to represent the promising region based on current observations, thereby informing the REASONER to recommend good configurations.

❸ LLM-based Configuration Reasoner (§7). To maximize the utilization of rich database semantics embedded within *C-knobs*, we integrate the LLM's in-context learning capabilities with statistically informed signals from GUIDER. First, we leverage proposals from GUIDER to generate diverse and effective contextual examples, which serve as the key demonstration for initial configuration recommendations. Subsequently, upon receiving feedback from EVALUATOR, the REASONER interprets this feedback and refines its reasoning, further improving configuration recommendations in an iterative manner. This process repeats until the final recommended configuration is satisfactorily accepted by EVALUATOR.

❹ Two-Stage Performance Evaluator (§8). It receives the configurations from the REASONER and obtains their corresponding query plans. To avoid redundant evaluations, we perform a two-stage workflow. The first stage is a lightweight configuration admission

gatekeeper, filtering out redundant or predictably disastrous candidates. Otherwise, we proceed to execute the query plan and feed the observed performance back to the GUIDER.

Closed Loop. We repeat the loop ❷–❹ until the termination condition (e.g., number of iterations) is satisfied. Then, the best-performing configuration and plan are put into effect, updating the binding plan in SPM (see §2.3) for subsequent online use.

5 KNOB INITIALIZATION

5.1 Knob Category

In this work, *C-knob*s can be divided into two main categories: ❶ *physical knob* that influences the cost of physical operator type; and ❷ *logical knob* that affects join order enumeration.

Physical Knob ($\boldsymbol{\kappa}_o$). This category adjusts the estimated cost of individual physical operators in the query execution plan. The involved operators[2] span a wide range of types, including: ❶ *Join*, e.g., hash join, merge join, and nested loop join; ❷ *Aggregation*, e.g., hash aggregation and stream aggregation; ❸ *Scan*, e.g., full table scan, range scan, index scan, and columnstore scan; ❹ *Other operators*, e.g., sort, limit, top-N, and table reader.

Logical Knob ($\boldsymbol{\kappa}_j$). This category is designed to influence the cost estimation within the join order enumeration. Specifically, for Q involving i tables, we introduce i corresponding knobs (e.g., $\kappa_{Join(T)}$ in Fig. 1), each associated with one of the participating tables. Then, these knobs act as scaling factors applied to the estimated cardinality of an output table before joining another table. Therefore, we can adjust these factors to optimize the join ordering.

5.2 Knob Relevance

Various *C-knob*s lead to a vast search space $\mathcal{X}$ that suffers from the curse of dimensionality. To mitigate this, we prune the search space by identifying and retaining only the knobs relevant to the given query Q. For example, when Q is a single-table query, all join-related physical knobs (e.g., those corresponding to *hash join* or *merge join*) are intrinsically irrelevant and are thus excluded from the tuning process. Specifically, we identify such knob relevance by instrumenting the CQO process. During the optimizer's enumeration of potential physical operators and join orders, we mark the knobs associated with any considered operator as relevant. Thus, we can identify operator-specific knobs $\boldsymbol{\kappa}_o$ that might have an effect. Moreover, for a SQL query involving i tables, we also introduce i join-specific knobs $\boldsymbol{\kappa}_j$, which influence the join order enumeration. In the end, we tune the knobs $\boldsymbol{\kappa} = \boldsymbol{\kappa}_o \cup \boldsymbol{\kappa}_j$.

Implementation. To enable relevance detection, we instrument the optimizer kernel to intercept the plan enumeration process. When alternative physical operators are enumerated (e.g., hash join vs. merge join), our hook function captures the event and generates the corresponding knob. Finally, we expose this functionality via a dedicated SQL interface, `EXPLAIN format='relevant_knob'`.

5.3 Knob Domain Mapping

The *physical* value domain of a knob $\Theta(\kappa)$ is defined as $[\frac{1}{n}, n]$, where n represents the maximum table size it operates. This range is determined based on the estimated upper and lower bounds of the scaling error. However, the range $[\frac{1}{n}, n]$ spans several orders of magnitude as n might be huge, resulting in *scaling imbalance*.

[2] https://docs.pingcap.com/tidb/stable/system-variables/

Therefore, we apply a logarithmic transformation that maps the physical knob values into the interval $[0, 1]$, allowing us to explore tuning space more effectively. Formally, let x be the physical knob value, and we define the transformation as:

$$\tilde{x} = \frac{\log(x) - \log(x_{\min})}{\log(x_{\max}) - \log(x_{\min})}, \tag{3}$$

where $x_{\min}$ and $x_{\max}$ denote the lower and upper bounds. This maps $x \in [x_{\min}, x_{\max}]$ to $\tilde{x} \in [0, 1]$ in log-space. The reverse mapping recovers the physical value from the normalized space via:

$$x = \exp\left(\tilde{x} \cdot (\log(x_{\max}) - \log(x_{\min})) + \log(x_{\min})\right). \tag{4}$$

Remark. Unless otherwise specified, the knob domain is normalized to $[0, 1]$ during the tuning process. When evaluating, the knob values are de-normalized and applied to the query optimizer.

6 BAYESIAN OPTIMIZATION AS A GUIDER

To minimize the cost of executing extra queries against the database, and given that we do not know how to efficiently explore the space of possible query plans, we require that our *offline planning* process is *guided*. In this context, BO [14] is a promising approach, which optimizes black-box functions that are expensive to evaluate, aiming for *sample efficiency*. Therefore, we employ the BO as a guider (GUIDER). The input of GUIDER is the configuration with execution time, which is denoted as the observation $\{(\mathbf{x}_i, y_i)\}$. The goal of GUIDER is to propose the next point $\mathbf{x}_{\text{BO}}$ to inform the REASONER.

The Necessity of Combining BO and LLM. Relying solely on BO often suffers from slow convergence, while relying solely on LLMs risks hallucinations. In consequence, both approaches are susceptible to local optima. By contrast, we harmonize the best of both worlds: leverage BO to provide quantitative guidance to locate the promising regions, which serves as grounded demonstrations to steer the LLM's reasoning. Such synergy accelerates the search for high-quality solutions and mitigates hallucinations.

6.1 Warm-Start Initialization

To build an initial surrogate model $\tilde{f}_\theta(\mathbf{x})$ that approximates $f(\mathbf{x})$, we begin with a warm-start phase. As exhaustive sampling of high-dimensional search space is infeasible [5, 23], we generate m sample points $\{\mathbf{x}_i\}_{i=1}^m$ using a Sobol sequence [22]. Unlike pseudo-random sampling, this low-discrepancy, quasi-random sequence ensures more uniform coverage of the high-dimensional space, which is critical for fitting a reliable initial Gaussian process (GP) [57].

Initial Observations. Each configuration $\mathbf{x}_i$ is evaluated with a conservative timeout threshold τ, typically set to a multiple of the default configuration's performance, e.g., $2 \cdot f(\mathbf{x}_0)$. This process yields an initial dataset $\mathcal{D}_0 = \{(\mathbf{x}_i, y_i, o_i)\}_{i=1}^m$, where y_i is the observed execution time (or τ if a timeout occurs) and $o_i \in \{0, 1\}$ is a binary indicator denoting whether this evaluation timed out. This dataset is used to initialize the surrogate models as described next.

6.2 Surrogates with Timeout Constraint

The *C-knob* configuration space induces a high-dimensional, discrete, and non-smooth objective function. As such, a single global surrogate model can not capture this configuration-performance relationship. Furthermore, executing a query with a poor configuration can be extremely time-consuming, yielding no informative

performance signal. Therefore, our goal here is thus twofold: find an optimal configuration while efficiently avoiding costly timeouts.

To address these challenges, we propose timeout-constrained Bayesian optimization (TCBO), a method inspired by the vanilla trust region BO technique [11, 12, 50]. TCBO employs a collection of local models to capture performance space's heterogeneity and explicitly models the probability of timeouts as a safety constraint.

Local Objective Modeling. Following Eriksson et al. [11], we maintain a set of L concurrent trust regions (TRs). Each trust region TR_ℓ is centered around its *incumbent* $\mathbf{x}^\star_\ell$, which is the configuration with the best observed performance within that region so far. For each trust region, we train a dedicated local GP surrogate model, $\mathcal{GP}_{\mathrm{obj},\ell}$, using only the performance observations $\{(\mathbf{x}_i, y_i)\}$ that fall within that region. This allows each model $\mathcal{GP}_{\mathrm{obj},\ell}$ to capture local performance variations. Besides, to improve model fitting, we apply a Gaussian copula transformation [41] to the objective values y_i, which magnifies differences near the optima.

Adaptive Trust Region. Each trust region TR_ℓ is a hyper-rectangle whose shape is adaptively scaled to match the learned characteristics of the objective function. The side length in dimension i is given by: $L_{\ell,i} = \lambda_i L_\ell / \left(\prod_{j=1}^{d} \lambda_j\right)^{1/d}$, where d is the dimensionality of $\mathbf{x}$. Here, λ_i is the learned GP lengthscale [11] for dimension i. Finally, we create an *anisotropic* (non-uniform) trust region that is shorter in sensitive dimensions and longer in less sensitive ones.

Probabilistic Timeout Constraint. The cornerstone of TCBO is treating the risk of a timeout as an explicit constraint. To model this constraint probabilistically, we employ a GP classifier $\mathcal{GP}_{\mathrm{timeout}}$. This choice is natural within our BO framework as it provides a posterior distribution over the timeout likelihood, rather than a deterministic label [31]. Specifically, $\mathcal{GP}_{\mathrm{timeout}}$ is trained on all available timeout observations $\{(\mathbf{x}_i, o_i)\}$. It learns the timeout probability $P(o = 1|\mathbf{x})$, which is a continuous value in $[0, 1]$. We then formalize a safety requirement by constraining this probability to be below a small risk threshold $\delta \in (0, 1)$ (e.g., $\delta = 0.05$):

$$c_{\mathrm{timeout}}(\mathbf{x}) = P(o = 1|\mathbf{x}) - \delta \leq 0. \tag{5}$$

While the posterior mean probability can express this constraint, our acquisition function (see §6.3) directly leverages the full posterior from $\mathcal{GP}_{\mathrm{timeout}}$ to capture uncertainty and avoid unpromising regions. Note, for ease of presentation, we omit the notation o_i in other sections, since timeout status is easily determined.

6.3 Acquisition via Thompson Sampling

With the surrogate models $\mathcal{GP}_{\mathrm{obj},\ell}$ and $\mathcal{GP}_{\mathrm{timeout}}$ defined, we use an acquisition function to decide which new configuration to evaluate. We employ Thompson sampling (TS), a posterior sampling strategy for multi-armed bandit problem [32]. By sampling from the posterior, TS makes a decision based on a plausible hypothesis of the performance landscape for each candidate. The generation process of $\{\mathbf{x}^\star_1, \ldots, \mathbf{x}^\star_q\}$ is as follows, repeated for each index $k = 1, \ldots, q$:

Step 1: Draw a Realization from the Posterior. A GP posterior defines a distribution over functions consistent with the observed data [71]. In this step, we draw a single sample from this distribution for each model. This yields a function $\tilde{f}^{(k)}_\ell$ from each $\mathcal{GP}_{\mathrm{obj},\ell}$ and a function $\tilde{c}^{(k)}_{\mathrm{timeout}}$ from $\mathcal{GP}_{\mathrm{timeout}}$. Each sampled function is a realization that is plausible given our current beliefs.

Step 2: Optimize within the Sampled Realization. For each trust region, we find the optimal configuration within the functional landscape defined by the samples drawn in the previous step:

$$\mathbf{x}^\star_{\ell,k} = \arg\min_{\mathbf{x}\in\mathrm{TR}_\ell} \tilde{f}^{(k)}_\ell(\mathbf{x}) \quad \text{s.t.} \quad \tilde{c}^{(k)}_{\mathrm{timeout}}(\mathbf{x}) \leq 0. \tag{6}$$

Step 3: Global Candidate Selection. From the set of local optima found in step 2, we select the globally best one as the k-th candidate:

$$\mathbf{x}^\star_k = \arg\min_{\ell\in\{1,\ldots,L\}} \tilde{f}^{(k)}_\ell(\mathbf{x}^\star_{\ell,k}). \tag{7}$$

Finally, we generate a batch of candidates $\{\mathbf{x}^\star_1, \ldots, \mathbf{x}^\star_q\}$. Then, $\mathbf{x}_{\mathrm{BO}}$ is selected from them with the best predicted performance according to the GP models' posterior mean. Let ℓ_k be the index of the trust region from which $\mathbf{x}^\star_k$ was generated. $\mathbf{x}_{\mathrm{BO}}$ is defined as:

$$\mathbf{x}_{\mathrm{BO}} = \arg\min_{k\in\{1,\ldots,q\}} \mu_{\mathrm{obj},\ell_k}(\mathbf{x}^\star_k), \tag{8}$$

where $\mu_{\mathrm{obj},\ell_k}$ is the posterior mean function of local $\mathcal{GP}_{\mathrm{obj},\ell_k}$.

Trust Region Update. After receiving the results of Evaluator, the TRs themselves also adapt as the optimization progresses. The size of each TR, controlled by its base side length L_ℓ, is adjusted based on the outcome of recent evaluations within it. A "*success*" is registered if a new evaluation improves upon the TR's incumbent $\mathbf{x}^\star_\ell$; otherwise, it is a "*failure*". Specifically, (1) Expansion: After ϱ_{succ} consecutive successes, the TR is expanded: $L_\ell \leftarrow \min\{2L_\ell, L_{\max}\}$. (2) Contraction: After ϱ_{fail} consecutive failures, the TR is shrunk: $L_\ell \leftarrow L_\ell/2$. (3) Restart: If L_ℓ falls below a minimum threshold $L_{\min}$, the TR is discarded and a new one is re-initialized.

7 LLM-BASED CONFIGURATION REASONER

To maximize the utilization of rich database semantics of *C-knob*s, we integrate the LLM's in-context learning capabilities [30] with statistically informed signals from Guider. Specifically, we introduce a self-evolving Reasoner that leverages statistically informed contextual example generation (see §7.1) and search improved solutions through feedback-driven evolution (see §7.2).

7.1 In-Context Example Generation

Directly asking an LLM to suggest knob configurations is unreliable, as the LLM lacks knowledge of the current execution environment. As such, we leverage the in-context learning capability of LLMs by providing them with empirical observations. A naïve approach is to include the full observation set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ [31]. However, it is suboptimal due to the excessive context length and the fact that LLMs inherently lack a rigorous statistical foundation [1].

As shown in Fig. 3, inspired by the maximal marginal relevance (MMR) [6], we propose to select a compact yet informative subset $\mathcal{K}$ of the full observation set $\mathcal{D}$, i.e., $\mathcal{K} \subseteq \mathcal{D}$. To obtain $\mathcal{K}$, we treat $\mathbf{x}_{\mathrm{BO}}$ as the query point and retrieve key observations from $\mathcal{D}$, which adheres to the following two MMR criteria: (i) spatial proximity to the promising region around $\mathbf{x}_{\mathrm{BO}}$ to ensure *relevance*; (ii) *diversity* across performance subspaces, which reduces redundancy.

MMR-based Neighbor Search. We initialize $\mathcal{K} = \emptyset$, and iteratively collect a candidate at a time. Since the query plan $P(\mathbf{x})$ is easily known via EXPLAIN statement, we can simplify the *diversity* measure by enforcing a *distinct* plan. In each iteration, we select

$$\mathbf{x}' = \arg\min_{\mathbf{x}_i\in\mathcal{D}\setminus\mathcal{K}} \|\mathbf{x}_i - \mathbf{x}_{\mathrm{BO}}\|_2 \quad \text{s.t.} \quad \forall \mathbf{x}_j \in \mathcal{K},\ P(\mathbf{x}_i) \neq P(\mathbf{x}_j), \tag{9}$$

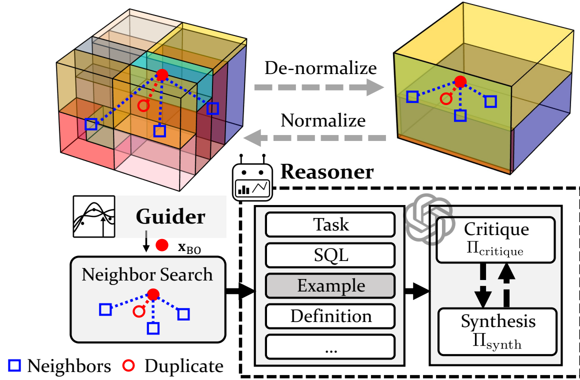


Figure 3: LLM-based Configuration Reasoner.

and update $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\mathbf{x}', y')\}$. This procedure ends when $|\mathcal{K}| = k$.

7.2 Rejection-Aware Configuration Reasoning

To equip the Reasoner with a meta-reasoning capability for self-evolution, we leverage a feedback-driven *Critique-and-Synthesize* loop, inspired by recent advancements in automated prompt optimization [2]. This mechanism is triggered upon critical failure, enabling the Reasoner to analyze its own mistakes and refine its internal reasoning strategy to escape local optima.

Initial Recommendation. The Reasoner initiates the process by generating a batch of candidate configurations $\mathbf{X}$ via an LLM, conditioned on a carefully constructed prompt Π_{rec}:

$$\mathbf{X} \leftarrow \mathrm{LLM}(\Pi_{rec}(\mathcal{K}, Q)). \tag{10}$$

The prompt integrates the task description, knob definition with its mapping, and the in-context examples $\mathcal{K}$ retrieved in §7.1. To ensure the generation of high-quality and valid configurations, we incorporate two key refinements. First, to counteract the LLM's propensity for low-precision outputs (e.g., 0.6 vs. 0.1)—which can translate to vastly different physical values (see Eq. 4) and cause uneven exploration—our prompt explicitly demands high-precision decimals (e.g., 0.115728). Second, to ensure robustness, we enforce any out-of-range or syntactically invalid values to their defaults, preventing failures in the end-to-end workflow. Then, the final set of configurations sent to the Evaluator is the union of the LLM's proposals and the point proposed by the Guider, i.e., $\mathbf{X} \cup \{\mathbf{x}_{BO}\}$.

Critique-and-Synthesize Loop. However, the candidate plan space is notoriously vast, with sparse regions of high quality [29]. Consequently, common configurations might yield performance far inferior to the default plan [8]. Therefore, it creates a high-rejection-rate environment where recommended configurations are frequently discarded by the Evaluator. Thus, when $\mathbf{X}$ is rejected, it signals a fundamental flaw in the current reasoning strategy. Upon receiving such feedback, the Reasoner triggers the following evolutionary procedure rather than resorting to simple heuristics.

❶ **Critique Phase.** The Reasoner first enters a critique phase to perform an error-driven self-reflection [2, 20]. It is prompted to analyze the root causes of the configuration rejection. Specifically, we construct a critique prompt, $\Pi_{critique}$, which instructs the LLM to identify the flawed patterns in its rejected configurations.

($\Pi_{critique}$). *Given the successful examples <$\mathcal{K}$> and that <$\mathbf{X}$> were all rejected, please analyze the shared characteristics of the rejected configurations. Identify potential flawed patterns, such as overly conservative exploration or neglect of knob interactions evident in $\mathcal{K}$. Output a concise analysis of these failures as structured feedback.*

As such, the LLM's response, i.e., a structured feedback text F_c, encapsulates its understanding of this rejection:

$$F_c \leftarrow \mathrm{LLM}(\Pi_{critique}(\mathbf{X}, \mathcal{K}, Q)). \tag{11}$$

❷ **Synthesis Phase.** The feedback F_c is then used to guide the next generation attempt. A new synthesis prompt Π_{synth} is dynamically constructed, incorporating the insights from the critique phase.

(Π_{synth}). *Your previous recommendations failed. Your self-critique is: <F_c>. Based on this feedback, generate a new batch of candidate configurations. Explicitly avoid the identified flawed patterns. Your new suggestions should be more diverse and strategically explore the knob space informed by your analysis.*

To further stimulate exploration and prevent repeated failures, popular prompt techniques [31] are also employed during this phase: the order of the in-context examples $\mathcal{K}$ is shuffled to $\mathcal{K}'$, and the LLM's temperature is increased. This combination of guided refinement allows the Reasoner to effectively escape local optima. Then, a potentially superior batch of candidates $\mathbf{X}'$ is generated:

$$\mathbf{X}' \leftarrow \mathrm{LLM}(\Pi_{synth}(F_c, \mathcal{K}', Q)). \tag{12}$$

Hierarchical Fallback Policy. To ensure the robustness and timely termination of the tuning process, we employ a hierarchical fallback policy. If the Reasoner fails to produce a valid candidate configuration within a predefined budget ξ_{llm}, the Reasoner gracefully degrades to a more traditional exploration strategy. It falls back to Latin hypercube sampling (LHS) [36], a space-filling design that guarantees broad coverage of the search space [13, 49]. This policy ensures that our framework maintains forward progress, leveraging the sophisticated reasoning of LLMs when possible, while relying on a different exploration to handle unrecoverable model failures.

8 TWO-STAGE PERFORMANCE EVALUATOR

Since some plan executions are expensive and may even be infeasible [49, 50, 59, 68], the primary objective of Evaluator is to judiciously avoid unnecessary executions. To this end, we design a two-stage evaluation workflow, as illustrated in Fig. 4. The first stage is a lightweight configuration admission gatekeeper, filtering out *redundant* or *disastrous* candidates. Only configurations that pass this gate proceed to the second stage, where the corresponding plan is executed under a timeout constraint.

8.1 Plan Repository

As illustrated in the lower-right of Fig. 4, we maintain a plan repository $\mathcal{R}$. It is designed to satisfy two requirements: (i) *high fidelity*, i.e., every entry is derived from *actually* executed plans; and (ii) *fine-grained annotation*, i.e., each recorded plan is augmented with low-level execution metrics (e.g., storage-/compute-layer breakdowns). These properties enable Evaluator to reuse reliable execution evidence to strengthen configuration admission (§8.2).

Given a SQL statement Q with its *actually* executed plan P, the repository $\mathcal{R}$ stores a record $\langle \mathcal{F}(P), Q, \widetilde{P} \rangle$, where $\widetilde{P}$ is a *verbose* plan obtained by enriching the operator tree with additional low-level execution metric. Specifically, we store: (1) *Plan fingerprint* $\mathcal{F}(P)$: a canonical identifier widely used in DBMSs for plan lookup. Before

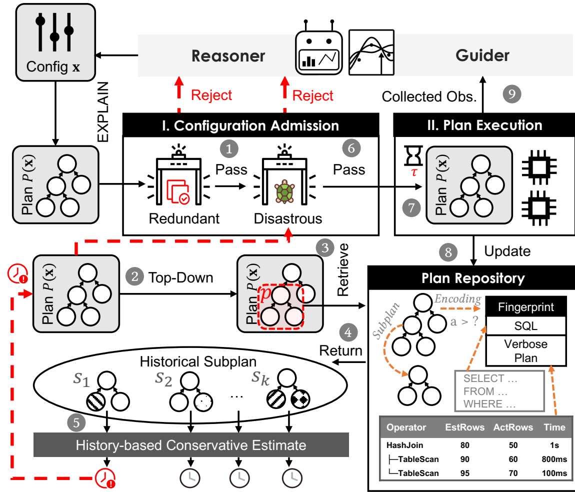


Figure 4: Overview of Evaluator.

computing $\mathcal{F}(P)$, the literals in predicates are normalized, e.g., replacing `a>100` with `a>?`, and then deterministically encode the resulting operator tree; (2) *SQL statement Q:* the originating SQL statement for provenance; (3) *Verbose plan $\widetilde{P}$:* the operator tree in which each operator node is annotated with rich runtime statistics, e.g., estimated/actual processed rows at the storage/compute layer, execution time, and other engine-specific details. Note, we maintain one repository per schema to ensure that stored plans and metrics remain comparable under a fixed physical design.

8.2 Configuration Admission

As shown in Fig. 4, given a configuration $\mathbf{x}$ recommended by the Reasoner for a specific SQL statement Q, we invoke the optimizer (via `EXPLAIN`) to obtain the corresponding plan $P(\mathbf{x})$, abbreviated as P. ❶ We first attempt to eliminate exact duplicates by querying the plan repository $\mathcal{R}$ for a full-plan match under the same Q. If the full plan already exists in $\mathcal{R}$, we reject $\mathbf{x}$ and reuse the recorded latency as a true observation $(\mathbf{x}, y)$ since there is no need to re-execute $P(\mathbf{x})$. Otherwise, we continue to the next step. ❷ As the full-plan cannot be matched, we traverse P from top to bottom. ❸ For each traversed subplan p, we query $\mathcal{R}$ based on the exact match of the fingerprint $\mathcal{F}(p)$. If the match is found, we stop the traversal since p is the largest matched subplan along the traversal. Otherwise, we continue the traversal. ❹ Since the fingerprints have normalized literals, this retrieval does not require identical parameter values and might return multiple matched subplans, denoted as $\mathcal{S}(p) = \{s_1, \ldots, s_k\}$. ❺ Based on the fine-grained statistics of each s_j, we compute a list of conservative latency estimates $\{\mathcal{E}(p, s_j)\}_{j=1}^{k}$ and aggregate them by the maximum: $\widehat{\mathcal{L}}(p) \triangleq \max_{s \in \mathcal{S}(p)} \mathcal{E}(p, s)$. ❻ If $\widehat{\mathcal{L}}(p)$ is no less than the timeout threshold τ, we reject $\mathbf{x}$ as the corresponding plan $P(\mathbf{x})$ is predictably disastrous. Otherwise, $\mathbf{x}$ is admitted to the next stage. Note that we extrapolate $\widehat{\mathcal{L}}(p)$ of the subplan to imply the latency of the full plan P.

History-based Conservative Estimate. Here, we describe how to compute $\mathcal{E}(p, s)$ from a matched historical subplan s. Since the repository $\mathcal{R}$ stores high-fidelity, fine-grained records from past executions, we can reuse them to derive a conservative latency estimate of the candidate plan during admission stage. Note, our goal here is *not* to build an accurate runtime predictor; instead, we seek a lightweight yet effective estimator that provides a safe upper bound for early pruning. As follows, we first introduce the latency decomposition model, and then describe how we infer a conservative latency bound for a given plan (or subplan) by extrapolating from matched historical executions. For simplicity, we use the term *plan* to refer to an operator tree and do not distinguish between a full plan and a subplan unless otherwise stated.

In modern disaggregated architectures [21, 53, 67], the execution latency of a plan $\mathcal{L}(p)$ can be decomposed into $\mathcal{L}(p) = \mathcal{L}_s(p) + \mathcal{L}_c(p)$, where $\mathcal{L}_s(p)$ is a storage-layer (TiKV) component (e.g., data scans), and $\mathcal{L}_c(p)$ is a computation-layer (TiDB) component (e.g., joins). Given a matched historical subplan $s \in \mathcal{S}(p)$, we can estimate the execution latency of p by scaling the observed latency of s according to cardinality differences. From the verbose plan of s, we extract: (1) the storage latency $\mathcal{L}_s(s)$, defined as the maximum wall-clock time among parallel storage tasks, together with the corresponding processed row count $R_s(s)$; and (2) the compute latency $\mathcal{L}_c(s) = \mathcal{L}(s) - \mathcal{L}_s(s)$ with total processed rows $R_c(s)$. Using these statistics, we derive per-row unit latencies:

$$\mu_s(s) = \frac{\mathcal{L}_s(s)}{R_s(s)}, \quad \mu_c(s) = \frac{\mathcal{L}_c(s)}{R_c(s)}. \tag{13}$$

For a candidate plan p, the optimizer provides estimated row counts $\hat{R}_s(p)$ and $\hat{R}_c(p)$. We then extrapolate its execution latency as

$$\mathcal{E}(p, s) = \hat{R}_s(p) \cdot \mu_s(s) + \hat{R}_c(p) \cdot \mu_c(s). \tag{14}$$

Scope and Applicability. Our design assumes the availability of reliable *subplan-level runtime measurements*. Therefore, it can not be directly applied to the *morsel-driven or compiled systems* [24, 28, 39, 40] that generate low-level code for a SQL query that fuses all adjacent non-blocking operators of a query pipeline into a single, tight loop [37]. On one hand, the compilation overhead can distort runtime measurements. On the other hand, due to the operator fusion [24, 54], the lack of traditional operator boundaries makes it difficult to extract individual operator times.

8.3 Plan Execution & Post-Processing

Once a configuration $\mathbf{x}$ passes the admission stage, it proceeds to the execution stage. However, blindly executing every plan is often impractical, as suboptimal plans can exhibit exceedingly long execution times, potentially lasting for days or even weeks. Therefore, we set a pre-defined timeout threshold τ (see ❼). If the execution of $P(\mathbf{x})$ does not complete within τ, we terminate it early. In this case, the true latency is *right-censored*, i.e., the run is truncated at τ, and we only know that $f(\mathbf{x}) \geq \tau$. Following common practice [50, 68], we record the latency of such *time-out* plans as τ.

Note that we use a *fixed* timeout threshold τ (rather than dynamic one in [50]) for two reasons. First, our surrogates train the primary objective model exclusively on uncensored observations, i.e., plans that complete execution within τ. This prevents the censored data from directly biasing the model's performance predictions in the well-performing regions. Second, our dataset of poor-performing plans is not limited to the fixed label τ; it is augmented with estimated latencies from our admission stage, which can exceed τ. Thus, this provides the Guider with a richer profile of the high-latency search space. Moreover, dynamic timeouts based on model uncertainty can be highly unreliable in the early, sparse-data stages

of optimization. In contrast, our fixed-timeout design, together with admission-stage estimates, provides a more robust mechanism.

Repository Update. To maximize the utility of each execution, we decompose every evaluated plan into its constituent subplans, and store them in the plan repository (see ❽). This fine-grained data persistence is applied to both successfully executed plans and timed-out plans. For a timed-out plan, we store the execution data for all subplans that completed before the timeout is triggered. Note, when the leaf operators fail to complete, no subplan is stored.

Observation Update. Throughout both stages, all "tested" configurations yield useful feedback for Guider. We collect *all* such configuration-performance pairs to update the statistical surrogate (❾). Specifically, the collected observations include: (i) successful executions with exact latencies; (ii) timed-out executions recorded as τ; and (iii) rejected configurations, including duplicates with reused true latencies and disastrous ones with conservative estimates.

9 EXPERIMENTAL EVALUATION

9.1 Experimental Setup

Environment. OBELISK is implemented in Python 3.12 and the default DBMS used in our experiments is TiDB V8.5 server with 8 TiKV nodes (each with 8C16G). Unless otherwise specified, the LLM used in our experiments is OpenAI's GPT-4o-mini. Experiments are conducted on a server running Ubuntu 24.04 equipped with two 28-Core Intel Xeon 6330 processors, 256 GB RAM, 960GB SSD, and two NVIDIA GeForce RTX 4090 GPU.

Benchmarks. Experiments are conducted on the following queries:

- **JOB**: The full join order benchmark (JOB) [29], which consists of 113 realistic queries over the IMDB dataset.
- **CEB:** The Cardinality Estimation Benchmark (CEB), introduced by Negi et al. [38], consists of numerous queries across 16 query templates over the IMDB dataset. Following Tao et al. [50], we adopt the same subset of 233 queries used in their evaluation, which are representative instances from 13 templates.
- **TPC-H**: A classic OLAP benchmark with 8 tables and 22 queries, with complexity slightly weaker than TPC-DS. For Q15, we precreate the view to avoid DDL during benchmarking.
- **TPC-DS**: An industry-standard decision-support benchmark with 24 tables and 99 complex queries. We exclude queries that rely on common table expression (CTEs) or ROLLUP, which are not fully supported in TiDB, thus yielding 63 queries.
- **DSB**: Ding et al. [9] introduce DSB, a decision support benchmark adapted from TPC-DS. DSB enhances TPC-DS with more complex data distributions, where tables must be generated according to a partial order, as well as enriched query templates. Following Tao et al. [50], we adopt the same set of 90 queries, consisting of three generated instances from each of 30 templates.

Unless otherwise noted, JOB and CEB use the real-world IMDB dataset [29], whereas the others use synthetically generated data with scale factor SF=50. For DSB, we use the default setting for data generation. In addition, we create indexes on all join keys.

Baselines. To our knowledge, BayesQO [50] is the first work to study this problem, and we employ it as the main baseline for comparison. We train BayesQO's VAE from scratch using the released source code (latent dimension is set to 64), with minor adaptations to support TiDB (e.g., join tree construction); the training for each schema completes within 16 hours (800,000 steps), and the resulting reconstruction accuracy is comparable to that reported in the original paper. In addition, we employ Bao [32] as another baseline. It was proposed as a RL-based method that intervenes by selecting one of the fixed 49 hint sets. Since our experiment is offline query optimization, Bao here refers to enumerating 49 hint sets, and we add a dialect mapping that rewrites PostgreSQL hints to their TiDB equivalents, thus the intended hint behavior is preserved.

Metric. We report *% improvement* as the latency-reduction ratio between the best plan and the default plan. To preclude confounding effects from potential optimizations in the storage engine, we disable TiDB's coprocessor cache by setting *copr-cache-size* to 0.

Settings. For the Guider, we adopt a trust-region BO scheme as described in §6. The initial trust-region side length is set to $L_0 = 0.2$, bounded by $L_{\min} = 0.5^8$ and $L_{\max} = 0.8$. The success and failure tolerances for trust-region adaptation are $\varrho_{\text{succ}} = 3$ and $\varrho_{\text{fail}} = 3$, respectively. For the probabilistic timeout constraint, we set the risk threshold to $\delta = 0.05$. For the Reasoner, the default LLM temperature is 0.7. Unless otherwise specified, the tuning budget consists of 6 warm-start rounds followed by 15 iterations.

9.2 Quality Evaluation

We run OBELISK and baselines on five benchmarks, tuning each query under a fixed budget. Note that BayesQO inevitably incurs Bao's 49 trial runs before its own optimization begins. Therefore, in addition to iteration budget of BayesQO, we also impose a maximum execution time of one hour per query. This choice is based on two considerations: (i) prior results by Tao et al. indicate that most tuning benefits are achieved within approximately one hour (see Figs. 4, 5, and 10 in [50]); and (ii) given the large number of queries in our evaluation, allocating substantially longer time per query would make the overall experimental cost prohibitive. Fig. 5 shows the final per-query improvement distribution, where curves closer to the top right indicate that a larger fraction of queries achieves larger speedups. Table 1 summarizes the corresponding statistics.

First, we observe that OBELISK consistently achieves stronger offline optimization benefits than the baselines, as reflected by curves that are closer to the upper right across all benchmarks. Specifically, OBELISK's average improvement ranges from 38.0% (CEB) to 58.6% (DSB), implying the high effectiveness of OBELISK.

Second, the performance gains of OBELISK are distributed across a broad set of queries rather than being concentrated on a few cases. As shown in Table 1, the P_{75} metric (sorted by improvement in descending order) characterizes the *harder-to-improve* portion of the workload. However, Bao and BayesQO exhibit negligible gains on such bottom 25% of queries, since the traditional optimizer already selects the near-optimal plans, and uncovering further improvement requires more sophisticated exploration. In contrast, OBELISK still achieves nontrivial gains for part of this harder subset. We attribute this to the fine-grained knob design combined with LLM-based reasoning, which helps escape local optima and discover alternative plans. Although BayesQO also provides fine-grained interventions, its tuning process is substantially more time-consuming and remains susceptible to local optima in practice.

Third, we find that offline optimization yields larger benefits on more challenging workloads, and OBELISK is particularly effective in such scenarios. For example, on DSB, OBELISK achieves more

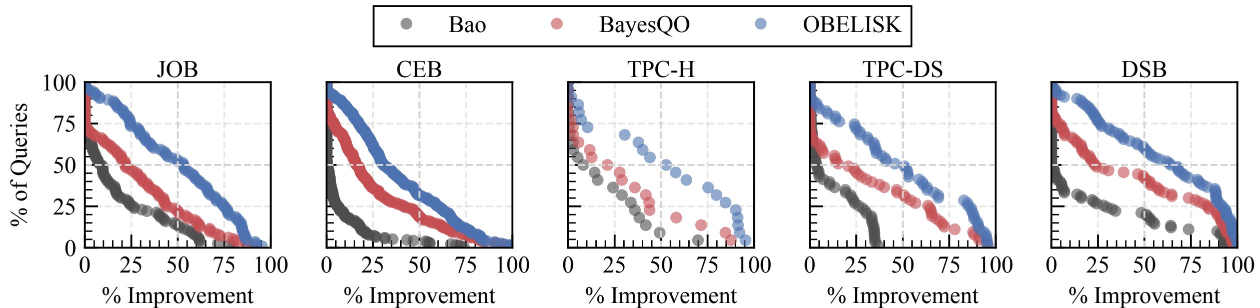


Figure 5: Distribution of Offline Optimization Results. Towards the top right corner is better.

Table 1: Statistics of Offline Optimization Results. Higher is better.

	JOB (N=113)				CEB (N=233)				TPC-H (N=22)				TPC-DS (N=63)				DSB (N=90)			
Method	Mean↑	P_{75} ↑	≥25%↑	≥50%↑	Mean↑	P_{75} ↑	≥25%↑	≥50%↑	Mean↑	P_{75} ↑	≥25%↑	≥50%↑	Mean↑	P_{75} ↑	≥25%↑	≥50%↑	Mean↑	P_{75} ↑	≥25%↑	≥50%↑
Bao	17.2%	0.0%	24.8%	13.3%	6.7%	0.0%	7.3%	3.9%	16.9%	0.3%	31.8%	4.5%	12.0%	0.0%	25.4%	0.0%	18.8%	0.0%	25.6%	20.0%
BayesQO	27.4%	0.0%	46.9%	22.1%	25.3%	4.0%	36.1%	18.5%	26.4%	1.9%	45.5%	18.2%	30.1%	0.0%	44.4%	31.7%	40.1%	3.3%	47.8%	41.1%
OBELISK	**49.1%**	**24.4%**	**72.6%**	**52.2%**	**38.0%**	**17.4%**	**61.4%**	**31.3%**	**48.1%**	**7.9%**	**68.2%**	**50.0%**	**47.7%**	**19.9%**	**71.4%**	**49.2%**	**58.6%**	**26.0%**	**75.6%**	**58.9%**

than 50% performance improvement for 58.9% of the queries. This is because DSB contains queries where the default optimizer is more likely to select suboptimal plans, thereby leaving greater room for improvement. As query complexity increases, the likelihood of optimizer mis-estimation grows, and OBELISK is well positioned to exploit this tuning potential. Finally, we observe that there exists a non-negligible fraction of queries whose performance remains largely unchanged after tuning, which highlights that traditional optimizers remain competitive for many average-case queries. This is because the traditional optimizer is able to select the right plan for some queries due to the decades of expertise. Furthermore, it reveals that offline optimization techniques should serve as a supplement to the query optimizer, covering the cases it cannot handle.

9.3 Case Study

To further compare the offline optimization effectiveness of OBELISK and BayesQO, we select several queries from the benchmark with the highest average improvement (DSB) and the benchmark with the lowest average improvement (CEB). For these queries, we allocate a larger tuning budget to enable a more thorough comparison. As shown in Fig. 6, for `CEB_11b44`, BayesQO gradually improves over time, but its final best plan remains slightly worse than the plan found by OBELISK. For `DSB_Query019_0`, BayesQO can eventually reach the same best plan as OBELISK. However, it spends substantially more tuning time. Importantly, the other two cases reflect the dominant behavior we observe in our extensive experiments: BayesQO converges early to local optima. For `CEB_1a1371`, BayesQO improves several times early on but stabilizes at a suboptimal plan. For `DSB_Query_013_spj_2`, BayesQO explores only a few distinct plans and then settles into a poor local optima. We attribute the *sample-efficiency* advantage of OBELISK to three factors. First, Guider effectively identifies promising regions while proactively avoiding costly timeouts, thereby steering the LLM toward high-quality recommendations. Second, the self-evolution mechanism in Evaluator promotes exploration beyond the local optima that traditional BO often converges to. Third, the admission mechanism in Evaluator rejects redundant and predictably disastrous candidates before execution, reducing wasted iterations and improving the effective use of the tuning budget.

Robustness Evaluation under Data Drift. We model data drift by scaling up the IMDB dataset to IMDB+, where the maximum value of *production_year* increases from 2016 to 2024 and the data size increases from 7.53GB to 19.80GB. Then, we evaluate robustness by comparing a *past plan* and a *future plan*. Specifically, the *past plan* is the well-optimized plan found by OBELISK on the original IMDB snapshot, whereas the *future plan* is the default plan produced by TiDB's optimizer on IMDB+. As shown in Fig. 7, we report the numbers of "positive" and "regression" queries, along with the improvement distributions before and after drift. We draw three observations from these results. (i) The past plans largely retain their advantage under data drift. For CEB, 92.3% of queries still run faster with the past plan than with the future plan. For JOB, this holds for 85.8% of queries. This suggests that OBELISK performs an instance-specific *calibration* that preserves the optimizer's built-in robustness, such as rule-based enumeration and pruning, while correcting problematic cost components. Such calibrations therefore tend to remain effective after moderate drift. (ii) CQO might repeat similar misestimation patterns under drift and thus re-select suboptimal plans, since they do not internalize feedback from prior executions. This observation is also consistent with the motivation behind SQL plan management (SPM) [44, 78]. (iii) The improvement margins on IMDB+ are smaller than those on the IMDB for some queries, suggesting that the past calibration is not necessarily optimal for the future data distribution. This naturally motivates re-invoking offline optimization once drift is detected to further refine plan bindings. Finally, for the minority of queries that regress under drift, the slowdown magnitude is limited, indicating that the plans produced by OBELISK are comparatively safer.

Interpretability Analysis. In our setting, interpretability does not rely on the exact value of a single knob, but on the relative ordering of knob values within a small set of competing alternatives. For example, we consider several typical competition groups of physical

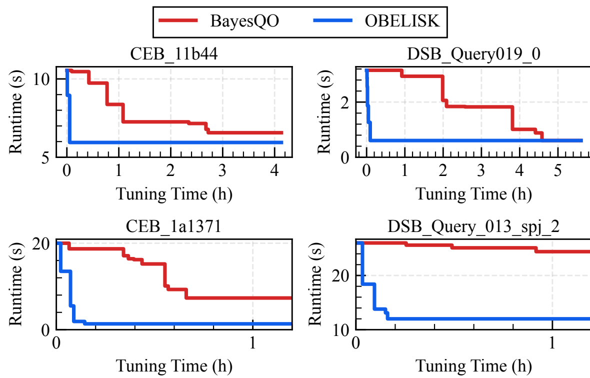


Figure 6: Case Studies with 10-Hour Tuning Budget. For visualization clarity, we omit the late-stage segments with no further improvement in the best-seen latency.

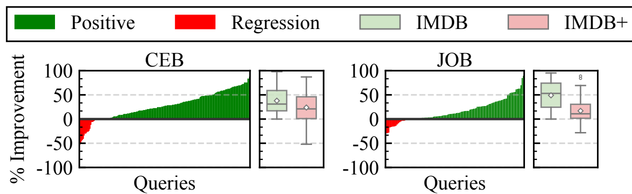


Figure 7: Robustness Evaluation under Data Drift.

knobs: (i) Access path (e.g., `TableFullScan` vs. `IndexScan`), (ii) Join algorithm (e.g., `HashJoin`, `MergeJoin`, and `NestedLoopJoin`), and (iii) Aggregation (e.g., `StreamAgg` vs. `HashAgg`). For join order, the table-level knobs encode preferences over when a base table should be joined; lower relative cost scaling for joining a table indicates a preference for bringing that table into the join earlier.

Then, we conduct a conservative plan–knob alignment experiment. For each query, we collect the default plan $P(\mathbf{x}_0)$ with $\mathbf{x}_0 = \mathbf{1}$, the best-performing plan $P(\mathbf{x}^*)$, and the corresponding knob configuration $\mathbf{x}^*$. With the help of an optimizer expert to compare $P(\mathbf{x}_0)$, $P(\mathbf{x}^*)$, and $\mathbf{x}^*$, we label whether the observed plan differences are consistent with the *preference* implied by the knob ordering within each *competition group*. For physical operator groups (access path, join algorithm, and aggregation), we check whether the most influential knob (i.e., strongest relative ordering) corresponds to a plan that uses the corresponding operator type in places where it competes with alternatives, even if only part of the plan changes. For join order, we check whether the preferred table appears earlier in the join sequence, and report *Join Order @3* based on whether the preferred table is among the first three joined tables, without attributing the change exclusively to any single knob. We report accuracy as the fraction of inspected cases where the expected outcome is observed. As shown in Fig. 8, we observe consistently high alignment accuracy across these four key intervention behaviors. These results support that our approach leverages knob semantics rather than treating knobs as opaque search variables.

9.4 Overhead Analysis

First, we evaluate the time overhead of the tuning process. Note, a fully *fair* efficiency comparison across methods is inherently difficult, because the plan quality explored in each iteration can differ

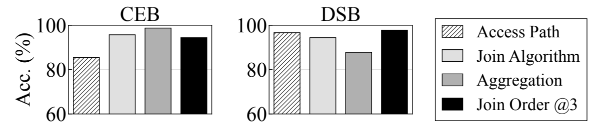


Figure 8: Accuracy of Plan-Knob Alignment for the Most Influential Knobs (%). Higher is better.

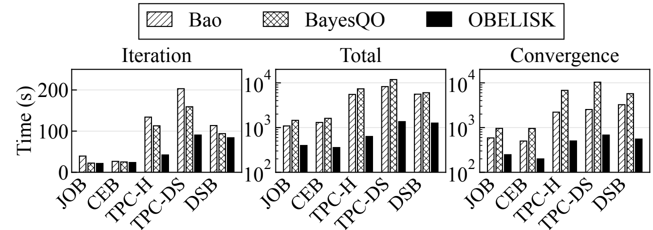
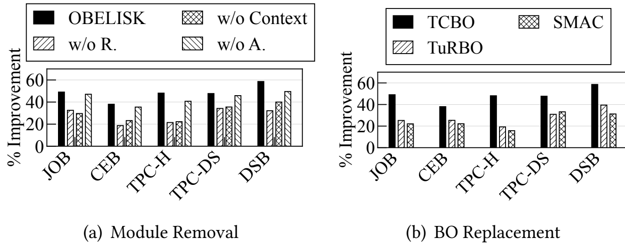


Figure 9: Overhead Analysis.

substantially. When poor plans are executed repeatedly, the overall tuning time inevitably increases, leading to large variability across iterations. Therefore, we report the following three complementary metrics: (i) *Iteration*: the total tuning time divided by the number of executed iterations. This metric roughly reflects how frequently the method executes poor plans, as such trials inflate the average per-iteration time. (ii) *Total*: the total tuning time from start to finish. (iii) *Convergence*: the elapsed time from the start of tuning until the best configuration (i.e., the best value observed during the run) is first reached. We exclude queries with no performance gain after tuning, as convergence is not meaningful in such cases. Unless otherwise noted, we compute each metric per query and then average across queries within each benchmark.

As shown in Fig. 9, in terms of *Iteration*, when query execution is relatively fast, the three methods exhibit little difference. The main reason is that during the offline optimization process, the quality discrepancy of different plans is relatively small, plans that are too poor will also be intercepted by the timeout mechanism. For *Total*, we observe that OBELISK completes tuning substantially faster than the others. The main reasons are twofold. First, the *Total* time is partly influenced by the characteristics of the method. Bao applies different hint sets sequentially, thus its tuning time inevitably accumulates across 49 executions. BayesQO further depends on Bao's exploration data as a warm start to ensure effectiveness. In our evaluation, we explicitly include this phase in BayesQO's end-to-end cost. Second, Bao's coarse-grained hint might produce inferior plans, which prolongs offline tuning since these candidates need to be completely executed. This dependency raises an additional deployment concern. In production, expensive training and warm start overhead might hinder the tuning methods' practicality at scale. Finally, in terms of *Convergence*, we observe that OBELISK often reaches its best-seen configuration early in the tuning process. Thus, users can stop tuning after obtaining a good plan that delivers sufficient gains, even though global optimality cannot be certified. These results also call for offline optimization methods that either (i) effectively avoid catastrophic plans, by preventing them from being explored or by skipping them, or (ii) quickly find a plan that provides sufficient improvement. Satisfying either of these goals can substantially reduce the overall offline tuning overhead.



(a) Module Removal (b) BO Replacement

Figure 10: Ablation Study.

Monetary Overhead. The monetary cost is primarily incurred by the REASONER. On average, an individual query requires approximately 106k–178k prompt tokens and 36k–98k completion tokens, causing a monetary cost of about 0.05 - 0.11 USD.

9.5 Ablation Study

To quantify the contribution of each key component to OBELISK's overall gains, we conduct an ablation study that evaluates GUIDER, REASONER, and EVALUATOR by comparing OBELISK variants and alternative tuners. In this study, we report results using mean improvement across queries within each benchmark. Specifically, *w/o R.* removes the REASONER module and relies solely on TCBO; *w/o Context.* removes the in-context examples provided to REASONER; and *w/o A.* disables the admission mechanism in EVALUATOR. In addition, to assess the effectiveness of the BO surrogate used in OBELISK, we replace TCBO with *TuRBO* [11] and *SMAC* [71], which are widely used hyperparameter optimization baselines based on trust-region BO and random-forest regression, respectively.

As illustrated in Fig. 10 (a), OBELISK consistently delivers the best performance across all benchmarks, demonstrating the effective synergy. First, removing the REASONER causes a substantial degradation, with a 33.7% – 51.8% drop, which indicates that relying on TCBO alone can miss high-quality opportunities. Second, *w/o Context.* uses the LLM to recommend configurations without contextual examples. While it generally outperforms TCBO alone, the improvement is marginal, suggesting that context is crucial for producing promising recommendations. Third, disabling the admission mechanism within EVALUATOR (*w/o A.*) reduces performance, e.g., a 15.5% drop on DSB. This suggests that filtering out infeasible candidates before evaluation can also mitigate noisy observations that can mislead the GUIDER. Moreover, as shown in Fig. 10 (b), TCBO consistently outperforms *TuRBO* and *SMAC* by large margins across all workloads. For example, on CEB, *SMAC* and *TuRBO* incur 41.8% and 33.4% lower improvements than TCBO, respectively. This is because *TuRBO* does not explicitly account for the timeout constraint, which might introduce excessive noisy observations and reduce the effectiveness. Besides, *SMAC* usually requires a larger number of observations to achieve substantial gains, whereas the high-fidelity evaluations are expensive and therefore limited.

9.6 Hyperparameter Sensitivity Study

First, we investigate the effect of warm-start rounds within GUIDER. Since the warm-start phase is typically limited in practice, we vary the number of warm-start rounds among 3, 6, and 9. As shown in Fig. 11, we observe that using 6 warm-start rounds consistently yields the best performance, which motivates our default choice. Increasing the warm-start rounds beyond this brings only marginal influence, e.g., an average 3.1% drop in DSB. In contrast, using too

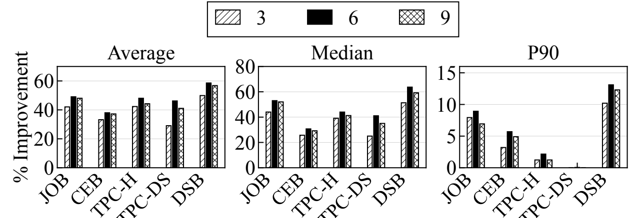


Figure 11: Hyperparameter Sensitivity Study (Warm-Start).

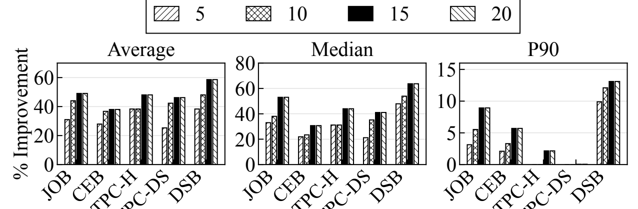


Figure 12: Hyperparameter Sensitivity Study (Budgets).

few warm-start rounds degrades performance more noticeably. For instance, 3 rounds lead to an average 12.6% drop in CEB. Note, even with fewer warm-start rounds, OBELISK still attains strong final performance. This is because warm start mainly affects the initialization of GUIDER. Subsequent improvements are driven by the closed-loop tuning process, where REASONER continues to propose high-quality configurations throughout optimization.

Second, we study the impact of the tuning budget. As shown in Fig. 12, OBELISK typically converges within 15 rounds, and in some cases within only 10 rounds. Note, even with as few as 5 rounds, OBELISK achieves competitive performance, indicating that it can identify good configurations with very limited exploration. For example, on JOB, the median improvement after 5 rounds already reaches exceeds 60% of the final effect. Based on the above experimental results, we choose 15 rounds as our default setting.

Limitations and Future Work. The primary limitation is that our implementation is tailored to TiDB. This specificity arises from our methodology's requirement for deep, white-box integration with the optimizer's cost model. To enable OBELISK, we leverage our long-term experience with the TiDB optimizer to perform custom modifications, exposing *C-knob* investigated in this paper. Consequently, a significant avenue for future work is the extension of the OBELISK to other database systems, which would involve identifying and exposing analogous *C-knob* in their respective optimizers. Nevertheless, we posit that the core principles of OBELISK are broadly applicable. OBELISK's key components—the *C-knob*, the BO GUIDER, and the LLM-based REASONER —are designed to be orthogonal. This modularity ensures that OBELISK can readily benefit from independent advances in different academic fields.

10 CONCLUSION

This paper presents OBELISK, an offline query optimization framework that revisits the problem from the perspective of *C-knob*. OBELISK integrates a timeout-constrained GUIDER, a self-evolving REASONER, and a lightweight EVALUATOR, whose synergy consistently demonstrates superiority across multiple benchmarks.

ACKNOWLEDGMENTS

This work is supported by NSFC (62461146205, 92270202), National Key R&D Program of China (2024YFC3308102).

REFERENCES

[1] Dhruv Agarwal, Manoj Ghuhan Arivazhagan, Rajarshi Das, Sandesh Swamy, Sopan Khosla, and Rashmi Gangadharaiah. 2025. Searching for Optimal Solutions with LLMs via Bayesian Optimization. In *ICLR*.

[2] Eshaan Agarwal, Raghav Magazine, Joykirat Singh, Vivek Dani, Tanuja Ganu, and Akshay Nambi. 2025. PromptWizard: Optimizing Prompts via Task-Aware, Feedback-Driven Self-Evolution. In *ACL*. 19974–20003.

[3] Remmelt Ammerlaan, Gilbert Antonius, Marc Friedman, HM Sajjad Hossain, Alekh Jindal, Peter Orenberg, Hiren Patel, Shi Qiao, Vijay Ramani, Lucas Rosenblatt, et al. 2021. PerfGuard: deploying ML-for-systems without performance regressions, almost! *PVLDB* 14, 13 (2021), 3362–3375.

[4] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. Autosteer: Learned query optimization for any sql database. *PVLDB* 16, 12 (2023), 3515–3527.

[5] Baoqing Cai, Yu Liu, Lin Ma, Pingqi Huang, Bingcheng Lian, Ke Zhou, Jia Yuan, Jie Yang, Xiaofan Cai, and Peijun Wu. 2025. SCompression: Enhancing Database Knob Tuning Efficiency Through Slice-Based OLTP Workload Compression. *PVLDB* 18, 6 (2025), 1865–1878.

[6] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *SIGIR*. 335–336.

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

[8] Xingguang Chen, Rong Zhu, Bolin Ding, Sibo Wang, and Jingren Zhou. 2024. Lero: applying learning-to-rank in query optimizer. *The VLDB Journal* 33, 5 (2024), 1307–1331.

[9] 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.

[10] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *PLVDB* 2 (2009), 1246–1257.

[11] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. 2019. Scalable global optimization via local Bayesian optimization. *NeurIPS* 32 (2019).

[12] David Eriksson and Matthias Poloczek. 2021. Scalable constrained Bayesian optimization. In *AISTATS*. 730–738.

[13] Chongjiong Fan, Zhicheng Pan, Wenwen Sun, Chengcheng Yang, and Wei-Neng Chen. 2024. LATuner: An LLM-Enhanced Database Tuning System Based on Adaptive Surrogate Model. In *ECML-PKDD*. 372–388.

[14] Roman Garnett. 2023. *Bayesian optimization*. Cambridge University Press.

[15] Luca Gretscher and Jens Dittrich. 2025. How to Optimize SQL Queries? A Comparison Between Split, Holistic, and Hybrid Approaches. *PVLDB* 18, 11 (2025), 3910–3922.

[16] 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.

[17] Roman Heinrich, Manisha Luthra, Johannes Wehrstein, Harald Kornmayer, and Carsten Binnig. 2025. How Good are Learned Cost Models, Really? Insights from Query Optimization Tasks. *Proc. ACM Manag. Data* 3, 3, Article 172 (2025), 27 pages.

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

[19] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *PVLDB* 13, 7 (2020), 992–1005.

[20] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *ICLR*.

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *PVLDB* 13, 12 (2020), 3072–3084.

[22] Stephen Joe and Frances Y Kuo. 2008. Constructing Sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2635–2654.

[23] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *PVLDB* 15, 11 (2022), 2953–2965.

[24] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB* 11, 13 (2018), 2209–2222.

[25] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. *CIDR*.

[26] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2025. GPTuner: An LLM-Based Database Tuning System. *ACM SIGMOD Record* 54, 1 (2025), 101–110.

[27] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *PVLDB* 17, 7 (2024), 1565–1577.

[28] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. 743–754.

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

[30] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. Llm for data management. *PVLDB* 17, 12 (2024), 4213–4216.

[31] Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. 2024. Large Language Models to Enhance Bayesian Optimization. In *ICLR*.

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

[33] 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 (2019), 1705–1718.

[34] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM*. Article 3, 4 pages.

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

[36] Michael D McKay. 1992. Latin hypercube sampling as a tool in uncertainty analysis of computer models. In *Proceedings of the 24th conference on Winter simulation*. 557–564.

[37] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. *PVLDB* 11, 1 (2017), 1–13.

[38] 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 (2021), 2019–2032.

[39] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* 4, 9 (2011), 539–550.

[40] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[41] Yoojeong Noh, KK Choi, and Liu Du. 2009. Reliability-based design optimization of problems with correlated input variables using a Gaussian Copula. *Structural and multidisciplinary optimization* 38, 1 (2009), 1–16.

[42] Zhicheng Pan, Yihang Wang, Yingying Zhang, Sean Bin Yang, Yunyao Cheng, Peng Chen, Chenjuan Guo, Qingsong Wen, Xiduo Tian, Yunliang Dou, et al. 2023. Magicscaler: Uncertainty-aware, predictive autoscaling. *PVLDB* 16, 12 (2023), 3808–3821.

[43] Zhicheng Pan, Yuanjia Zhang, Chengcheng Yang, Ahmad Ghazal, Rong Zhang, Huiqi Hu, Xiaoju Wu, Yu Dong, and Xuan Zhou. 2025. Hyper: Hybrid Physical Design Advisor with Multi-Agent Reinforcement Learning. In *ICDE*. 1565–1578.

[44] PingCAP. [n.d.]. *SQL Plan Management*. https://docs.pingcap.com/tidb/stable/sql-plan-management/

[45] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *PVLDB* 17, 4 (2023), 740–752.

[46] Edward Sciore. 2020. Query Optimization. In *Database Design and Implementation: Second Edition*. Springer, 419–454.

[47] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *SIGMOD*. 23–34.

[48] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *SIGMOD*. 99–113.

[49] Wenwen Sun, Zhicheng Pan, Zirui Hu, Yu Liu, Chengcheng Yang, Rong Zhang, and Xuan Zhou. 2025. Rabbit: Retrieval-Augmented Generation Enables Better Automatic Database Knob Tuning. In *ICDE*. 3807–3820.

[50] Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. *Proc. ACM Manag. Data* 3, 3, Article 179 (2025), 29 pages.

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

[52] 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.

[53] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.

[54] Benjamin Wagner, André Kohn, Peter Boncz, and Viktor Leis. 2024. Incremental Fusion: Unifying Compiled and Vectorized Query Execution. In *ICDE*. 462–474.

[55] Ziyun Wei and Immanuel Trummer. 2024. ROME: Robust query optimization via parallel multi-plan execution. *Proc. ACM Manag. Data* 2, 3 (2024), 1–25.

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

[57] Christopher KI Williams and Carl Edward Rasmussen. 2006. *Gaussian processes for machine learning*. Vol. 2. MIT press Cambridge, MA.

[58] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*. 1081–1092.

[59] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query execution time prediction in amazon redshift. In *SIGMOD*. 280–294.

[60] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: a new cardinality estimation framework for join queries. *SIGMOD* 1, 1 (2023), 1–27.

[61] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2021. BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation. arXiv:2012.14743 [cs.DB]

[62] Haibo Xiu, Pankaj K. Agarwal, and Jun Yang. 2024. PARQO: Penalty-Aware Robust Plan Selection in Query Optimization. *PVLDB* 17, 13 (Sept. 2024), 4627–4640.

[63] Zhengtong Yan, Valter Uotila, and Jiaheng Lu. 2023. Join order selection with deep reinforcement learning: fundamentals, techniques, and challenges. *PVLDB* 16, 12 (2023), 3882–3885.

[64] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch?. In *SIGMOD*, Vol. 1. 1–27.

[65] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a query optimizer without expert demonstrations. In *SIGMOD*. 931–944.

[66] 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 (2020), 61–73.

[67] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million tpmC Distributed Relational Database System. *PVLDB* 15, 12 (2022), 3385–3397.

[68] Zixuan Yi, Yao Tian, Zachary G Ives, and Ryan Marcus. 2025. Low Rank Learning for Offline Query Optimization. *Proc. ACM Manag. Data* 3, 3 (2025), 1–26.

[69] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *ICDE*. 1297–1308.

[70] Qihan Zhang, Shaolin Xie, and Ibrahim Sabek. 2025. LIMAO: A Framework for Lifelong Modular Learned Query Optimization. *PVLDB* 18, 11 (2025), 4546–4559.

[71] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating database tuning with hyper-parameter optimization: a comprehensive experimental evaluation. *PVLDB* 15, 9 (2022), 1808–1821.

[72] Yunjia Zhang, Yannis Chronis, Jignesh M Patel, and Theodoros Rekatsinas. 2023. Simple adaptive query processing vs. learned query optimizers: Observations and analysis. *PVLDB* 16, 11 (2023), 2962–2975.

[73] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3, 3, Article 146 (2025), 28 pages.

[74] 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.

[75] Rong Zhu, Lianggui Weng, Bolin Ding, and Jingren Zhou. 2024. Learned query optimizer: what is new and what is next. In *SIGMOD*. 561–569.

[76] Rong Zhu, Lianggui Weng, Wenqing Wei, Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian, Bolong Zheng, and Jingren Zhou. 2024. Pilotscope: Steering databases with machine learning drivers. *PVLDB* 17, 5 (2024), 980–993.

[77] 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 (2021), 1489–1502.

[78] Mohamed Ziauddin, Dinesh Das, Hong Su, Yali Zhu, and Khaled Yagoub. 2008. Optimizer plan change management: improved stability and performance in Oracle 11g. *PVLDB* 1, 2 (2008), 1346–1355.