WAter: A Workload-Adaptive Knob Tuning System

Abstract

Selecting appropriate values for the configurable parameters of Database Management Systems (DBMS) to improve performance is a significant challenge. Recent machine learning (ML)-based tuning systems have shown strong potential, but their practical adoption is often limited by the high tuning cost. This cost arises from two main factors: (1) the system needs to evaluate a large number of configurations to identify a satisfactory one, and (2) for each configuration, the system must execute the entire target workload on the DBMS, which is both time-consuming and resource-intensive. Existing studies have primarily addressed the first factor by improving sample efficiency, that is, by reducing the number of configurations that must be evaluated. However, the second factor, improving *runtime efficiency* by reducing the time required for each evaluation, has received limited attention and remains an underexplored direction.

We develop WAter, a runtime-efficient and workload-adaptive tuning system that finds near-optimal configurations at *a fraction of the tuning cost* compared with state-of-the-art methods. Instead of repeatedly replaying the entire target workload, we divide the tuning process into multiple time slices and evaluate only a small subset of representative queries from the workload in each slice. Different subsets are evaluated across slices, and a runtime profile is used to dynamically identify more representative subsets for evaluation in subsequent slices. At the end of each time slice, the most promising configurations are selected and evaluated on the original workload to measure their actual performance. Technically, we design a query-level metric and propose a novel Greedy Algorithm that continually refines the query subset (e.g., removing uninformative queries and adding promising ones) as the tuning progresses. We then develop a hybrid scoring mechanism, built upon a global surrogate model, to balance exploitation and exploration and to recommend promising configurations for evaluation on the entire workload. Finally, we evaluate WAter across different workloads and compare it with state-of-the-art approaches. WAter identifies the best-performing configurations with up to **73.5%** less tuning time and achieves up to **16.2%** higher performance than the best-performing alternative. We also demonstrate WAter's robustness across different hardware platforms and optimizers, as well as its scalability across database sizes.

1 Introduction

Database management systems (DBMSs) rely on many configuration parameters (i.e., knobs) to control their behavior [35]. Tuning these knobs is crucial for achieving high performance [43]. Conventionally, these knobs are adjusted manually by database administrators (DBAs), involving extensive workload, system, and hardware analysis. However, DBAs encounter substantial difficulties identifying promising configurations for a specific workload due to the high dimensionality of the configuration space, where each knob can have continuous or discrete values (heterogeneity). This challenge becomes even more pronounced in the cloud, where the underlying hardware resources can vary significantly across DBMS instances.

Recent works focus on using Machine Learning (ML) techniques to automate knob tuning to reduce the manual tuning efforts, and have shown promising results [5, 9, 12, 17, 21, 23, 48, 49, 54, 57–59]. These ML-based tuning systems iteratively select a configuration using a tuner, balancing between the exploration of unseen regions and the exploitation of known space. The selected configurations are then evaluated by executing the target workload on the DBMS. Since it is challenging to explore the high-dimensional and heterogeneous search space, many techniques are proposed to explore the space efficiently, such as search space pruning [23, 59] and transfer learning [5, 27, 42, 57].

Although state-of-the-art systems reduce the required iterations to only hundreds to identify ideal configurations, the tuning cost is *still high* because it takes a long time to execute the workload in *each iteration*. For example, in our experiment in Figure 8, it takes 10 minutes to execute the 22 queries in the TPC-H benchmark with a scale factor of 50, leading to about 17 hours of optimization for 100 valid iterations. Figure 1 shows the breakdown of the tuning time of a state-of-the-art method [23] for TPC-H benchmark under different scale factors. Notably, more than 70% of tuning time is spent on executing the target workload on DBMS, and this becomes more pronounced (e.g., more than 97%) as the data size increases or the workload becomes more complex, an observation similar to previous work [43].

Therefore, we argue that *it is important to reduce the workload execution time while keeping the tuning effective*, given that the major tuning costs come from substantial workload execution time, a factor overlooked by prior research. In this paper, we propose a new concept of *runtime efficiency*, which refers to minimizing the workload execution time in each tuning iteration and thus achieving the overall minimum tuning time. This approach is compatible with previous works focusing on decreasing the number of tuning iterations, but goes one step further by trying to reduce the running time of each iteration.

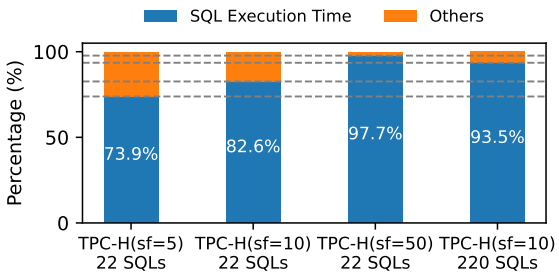


Figure 1: Tuning Time Breakdown (Percentage)

However, achieving runtime efficiency of knob tuning presents several challenges. ***C1. It is non-trivial to reduce the workload execution time.*** There are two possible ways to cut down the workload execution time: one is to decrease the volume of the target database and the other is to reduce the number of queries in the workload (workload compression). Decreasing the data volume by sampling a subset of data can severely affect the performance.

Because knob tuning is highly sensitive to the underlying data, reducing the data size is likely to change the performance bottleneck and thus mislead the tuning process. In contrast, workload compression [2, 6–8, 16, 39] presents a promising approach. It aims to identify a substitute query subset that approximates the runtime behavior of the DBMS under the full workload, without significantly degrading the performance of workload-driven tasks like index tuning [7].

Unfortunately, ***C2. It is challenging to obtain a compressed workload that is truly representative for the specific task of knob tuning.*** While there has been substantial work on workload compression [2, 7, 8, 16, 39, 53], these techniques are not effective in our context because they were originally designed for index tuning [2, 7, 39, 52, 53], which is a fundamentally different problem that focuses on selecting table columns for index construction. These methods often rely on query-level features such as shared "indexable columns", which do not translate to the knob tuning task. Index-agnostic approaches, including random sampling and GSUM [8], only use generic workload information and therefore perform sub-optimally for specialized tasks (e.g., index tuning or knob tuning), as confirmed by our experimental results in Section 8.2. Selecting a representative subset of queries as the compressed workload for knob tuning remains an open challenge.

C3: Knob tuning introduces new challenges when applying workload compression, as good performance on a subset of the workload does not necessarily imply good performance on the full workload. When tuning one subset, we only evaluate the configurations on this subset. However, a good configuration for this subset does not necessarily perform well on the original workload, and may even lead to performance degradation. Moreover, even if the tuning is guided by advanced optimization algorithms [14, 40], there is no guarantee that the configurations recommended later are better than the previous ones (not monotonic). Therefore it is infeasible to simply evaluate configurations from later iterations on the original workload. We need a sophisticated mechanism to identify well-performing configurations for the entire workload without verifying every proposed option, as doing so is exhausting and would negate the benefits of workload compression. Challenges remain regarding how to determine whether a configuration is worth evaluating, how to trade off between subset tuning and configuration verification, whether the subset should be dynamically updated and if so, how to achieve that.

Our Approach. To address these challenges, we develop WAter, a runtime-efficient and workload-adaptive tuning system, and it identifies near-optimal configurations at a fraction of the tuning time compared to state-of-the-art methods. The key observation of WAter is that aforementioned limitations of existing approaches (***C1*** and ***C2***) are rooted in the intractable difficulty to find a perfect subset in one try. Differently, WAter starts with an imperfect subset and continually refine it based on runtime profile on the fly (Section 5.3). Instead of replaying the whole workload or a fixed subset repeatedly, we divide the tuning process into many time slices and evaluate different subsets at different time slices (Section 5.1). To continually refine the subset as the tuning proceeds, we carefully design a runtime metric to measure the representativity of a subset to its original workload (Section 5.2), and propose a novel greedy algorithm based on this metric (Section 5.3). Moreover, to mitigate the overhead of switching between tuning different subsets, we develop a history reuse mechanism for efficient subset tuning (Section 6). Regarding ***C3***, we design heuristic-based rules to prune unpromising configurations (e.g., configurations perform significantly worse than the default configuration are discarded). After pruning, we propose a hybrid scoring mechanism to score and rank configurations, only verifying the most promising configurations on the original workload. The scoring mechanism is based on a global surrogate model, predicting the performance as well as the uncertainty of the prediction for configurations to balance between exploration and exploitation (Section 7). Finally, we conduct extensive experiments to evaluate WAter's effectiveness, robustness and scalability.

Experimental Overview. Our extensive experiments demonstrate WAter's decisive advantages over state-of-the-art tuners across multiple OLAP benchmarks. On average, WAter finds optimal configurations 4.2× faster while discovering superior solutions that yield up to 16.2% better final performance, with time-to-optimal speedups reaching a remarkable 12.9× on complex workloads. This state-of-the-art performance is proven to be robust across different hardware, optimizers, and larger database scales where WAter's runtime efficiency provides the greatest benefit. A detailed ablation study confirms the criticality of each of our core components, while a cost analysis reveals that WAter's efficiency stems from drastically reducing the dominant cost of workload evaluation time.

Contributions. Our contributions are as follows. (1) We develop WAter, a *runtime-efficient* knob tuning system that identifies near-optimal configurations at a fraction of the tuning time compared to state-of-the-art methods. (2) We introduce a new paradigm that applies workload compression to enhance the knob tuning process and identify the associated technical challenges. (3) We develop a set of techniques to address these challenges, including: a time-slicing design that partitions the tuning process into multiple time intervals and evaluates different query subsets across them (Section 4); an adaptive mechanism that incrementally refines the query subset using a greedy algorithm to make it increasingly representative of the complete workload based on runtime feedback (Section 5); a history reuse mechanism that minimizes the overhead of switching between query subsets (Section 6); and a hybrid scoring algorithm that selects only the most promising configurations for validation (Section 7). The source code for WAter is available at: https://anonymous.4open.science/r/WAter-1BDD.

2 Background and Related Work

2.1 Database Knob Tuning

Database Tuning Problem. We formulate database knob tuning as an optimization problem. Given a *target workload* $\mathbf{W}$ and the *configuration space* $\mathbf{\Theta}$, the performance metric is given by an *objective function* $f_{\mathbf{W}} : \mathbf{\Theta} \rightarrow \mathbb{R}$, that projects each configuration to a value of the performance metric (e.g., latency or throughput). Database knob tuning aims to find a configuration $\theta^* \in \mathbf{\Theta}$, where

$$\boldsymbol{\theta}^* = \arg\max_{\theta \in \mathbf{\Theta}} f_{\mathbf{W}}(\theta) \tag{1}$$

Finding an optimal database configuration is challenging due to the vast configuration space. Such difficulty goes beyond the

capability of even the best human experts, so database community turns to ML-based automatic tuning methods.

ML-based Knob Tuning. Recently, ML-based approaches have demonstrated promising results, achieving better performance than human DBAs as well as static rule-based tuning tools [41, 46]. Moreover, ML-based approaches are automatic and can adapt well to a variety of workloads and hardware configurations. Figure 2 presents the paradigm of the ML-based knob tuning framework which mainly contains *(i)* a *tuner* that suggests a configuration over a given search space to improve the pre-defined performance metrics, and *(ii)* a *DBMS instance* that runs the workload under the proposed configuration to obtain the performance metric. The knowledge base $\mathcal{D} = \{\theta_i, f_{\mathbf{W}}(\theta_i)\}$ is an optional component which records all previously evaluated configurations, and updates every time a new evaluation is conducted. These systems can be broadly classified into two main categories based on the techniques used in the *tuner*: Bayesian Optimization (BO)-based [40] and Reinforcement Learning (RL)-based [14].

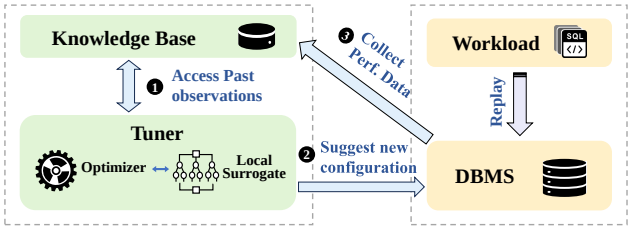


Figure 2: Overview of Knob Tuning Paradigm

• **RL-based.** RL-based methods explore the configuration space in a trial-and-error manner. The agent (e.g., a neural network) iteratively tries new configurations and learns from the rewards (e.g., performance improvement or degradation) obtained from the environment (e.g., DBMS). Deep Deterministic Policy Gradient (DDPG) [28] is the most popular RL algorithm adopted in knob tuning [12, 27, 54], as DDPG can work over a continuous space.

• **BO-based.** BO-based methods [5, 9, 23, 24, 42, 57, 58] model the tuning as a black-box optimization problem. BO consists of two main components: (1) *surrogate model* is an ML model to approximate the objective function $f_{\mathbf{W}}$, given a set of observations $\{\theta_i, f(\theta_i)\}$. It provides both a prediction and the confidence of the prediction for an unseen configuration. (2) *acquisition function* uses the surrogate model's outputs to choose which candidate point to evaluate next, balancing between exploitation and exploration.

• **Tuning Frameworks.** Frameworks such as MLOS [20] do not directly improve tuning efficiency, but serve to bridge the gap among benchmarking, experimentation, and optimization. In contrast, other frameworks like OtterTune [42] and LlamaTune [17] are designed to enhance tuning efficiency. These frameworks focus on sample efficiency and are orthogonal to our work, which focuses on runtime efficiency. They can co-exist with WATER to further enhance the performance of existing optimizers.

Performance Comparison. According to [55], RL-based methods require more iterations to work well due to the complexity of the neural networks used. The majority of previous works use BO-based methods, and [55] concluded that the best performing optimizer was Sequential Model-based Algorithm Configuration (SMAC [30]), since it is efficient in modeling the heterogeneous search space. With the recent advent of Large Language Model (LLM), GPTUNER [23] uses LLM to read manuals and constructs structured knowledge to guide the BO-based tuning process. *We regard GPTUNER and SMAC as the current state-of-the-art methods with and without text as inputs. We integrate WATER with these methods.*

2.2 Workload Compression

Workload compression is first studied in [7]. Given a workload **W**, it aims to find a SQL subset **W'** (**W'** has fewer queries and each query comes from **W**), such that the workload execution cost is reduced (fewer queries to execute for knob tuning, or fewer columns to consider for index tuning), and the tuning performance does not degrade too much at the same time. However, the performance degradation is inevitable in practice. Existing works essentially trade performance for runtime efficiency [2, 8, 16, 39]. The primary aim of our work differs significantly from previous works. Instead of trading performance for runtime efficiency, our approach can achieve superior performance compared to tuning the original workload within the same time budget, as reduced iteration costs allow for more thorough exploration of the configuration space. A detailed formulation of the problem and its underlying intuition can be found in Section 5.1.

There are both generic and indexing-aware workload compression techniques in the literature. GSUM [8] is a recent generic workload compression system that maximizes the coverage of features (e.g., columns contained) of the workload while ensuring that the compressed workload remains representative (i.e., having similar distribution to that of the entire workload). For indexing-aware compression, ISUM [39] selects queries greedily based on their potential to reduce the costs and the similarity between queries, and the two metrics are computed using indexing-specific featurization. The most recent work, WRED [2], rewrites each query in the original workload to eliminate columns and table expressions that are unlikely to benefit from indexes. These methods compress the workload in a single step, lacking further refinement. More importantly, they require manual feature engineering of queries, and some even require indexing-specific features, making these methods not applicable to knob tuning. In contrast, WATER focuses on knob tuning that seamlessly integrates workload compression through the entire tuning process, continuously refining the subset. Additionally, WATER does not rely on any form of featurization; instead, it selects queries based on runtime statistics, allowing it to handle any executable query. In comparison, methods like WRED are unable to handle 19 out of 99 queries from TPC-DS that its parser cannot process.

A recent concurrent work, SCompression [3], also addresses the high cost of workload execution and targets Online Transaction Processing (OLTP) workloads. It uses time-slices as the compression unit to preserve the inherent concurrency and temporal relationships of the original workload, and performs a one-time, static compression to generate a fixed workload that is used for the entire tuning process. In contrast, our approach in WATER is fundamentally different in several key aspects. First, WATER is designed specifically for OLAP workloads, where it operates by selecting a representative subset of individual queries from the entire workload. Our experiments in Section 8.2 show that compressing OLAP

workloads for knob tuning is a non-trivial task. Existing OLAP compression techniques, such as GSUM [8] and random sampling, yield poor performance in this setting. More importantly, unlike SCompression's static approach, WAter introduces a dynamic and adaptive compression strategy that continually refines its selected query subset throughout the tuning process based on an evolving runtime profile. This adaptability prevents the tuning process from being misled by a fixed suboptimal subset, ensuring more robust optimization results. We summarize the main differences between them in Table 1.

Table 1: Main Difference between WAter and SCompression

	Target Workload	Comp. Unit	Strategy
WAter	OLAP	Query	Dynamic
SCompression	OLTP	Time slice	Static

Some works on training data collection also involve sampling a SQL subset from the original workload. However, they focus on different application scenarios. From the perspective of model training, these works either aim to minimize the cost to obtain a labeled training dataset [29, 31] or select the most valuable training data (queries) [51] for a learned database component (e.g., learned cost estimators) effectively. Moreover, in contrast to knob tuning, their target workload is typically a streaming query workload produced in the online scenario, rather than a fixed set of queries.

3 Motivation

In this section, we discuss the motivations behind the design and implementation of WAter as well as how this paper is structured.

M1: The search space for knob tuning is extremely large yet underexplored. The search space of knob tuning is extremely large due to: (1) *the large number of knobs that require tuning*, and (2) *the wide value range for each knob*. For example, PostgreSQL v14.9 has 346 knobs, and some most frequently tuned knobs like `shared_buffers` range from 0.125 MB to 8192 GB, and `random_page_cost` can be set to any real value between 0 and 1.79769×10^{308}. Moreover, some methods [5, 58] even add contextual information (e.g., workload feature) into the space which could further expand it. In the literature, it is commonly assumed that the number of evaluations required to find an optimum is proportional to the size of the search space [50]. However, existing ML-based tuning methods only conduct hundreds to at most thousands of samplings and evaluations [5, 17, 23, 38, 57, 59], which is very sparse in such a colossal search space. The exploration of the search space is insufficient, and we need to explore it more thoroughly to identify better configurations.

M2: Under-exploration stems from high workload execution time, workload compression presents a promising method to reduce the costs. As discussed in Section 2.1, evaluating a configuration requires executing the target workload, with each workload execution taking minutes or more. Such high costs greatly limit the number of configurations to try. A naive approach to mitigate **M1** involves sampling a small subset of queries from the original workload. By executing fewer queries, we decrease the workload execution time, allowing for exploration of a larger portion of the search space within a given time budget. We conduct an experimental study for this idea by randomly sampling 3 subsets of 26 queries from TPC-DS's 88 queries, using GPTuner [23] as the optimizer for its efficiency. Whenever a proposed configuration outperforms the default configuration on the subset, this configuration is immediately evaluated on the original workload to obtain real performance. We also use GPTuner to tune the original workload directly as a comparison. Figure 3 shows the latency of the best configuration found (y axis) as a function of optimization time (x axis). It is worth noting that *tuning a subset can make the tuner produce well-performing configurations with much less time.* The reason is that reduced execution time enables more configuration evaluations, improving exploration and increasing the likelihood of finding optimal solutions.

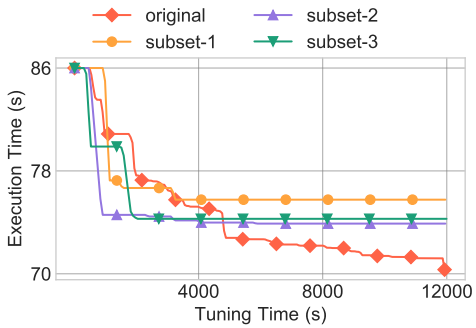


Figure 3: Tuning Subsets VS Tuning the Original Workload

M3: Identifying a representative subset is important but very challenging. From Figure 3, we find that different subsets can lead to different optimization results, and a bad subset can make the optimization stuck in local optima and fail to find better configurations even after a long tuning time. An interpretation could be that, tuning a subset essentially involves optimizing an alternative objective function that approximates the real objective function of the entire workload, and the similarity between the objective functions of query subsets and the objective function of the original workload differs greatly for different subsets. A more representative subset can result in faster and more thorough optimizations, while a bad subset could even mislead the process. Selecting a good subset is critical for the end-to-end tuning performance, but unfortunately, we do not even have a method to quantify the representativity of a subset to its original workload in the context of knob tuning.

M4: It is nearly impossible to find a perfect subset in a single attempt, but we can continually refine the subset based on the evolving runtime profile. Knob tuning is such a complex problem which involves almost all aspects of a DBMS, including resource management, background process management, query optimization and execution, and so on [60]. Therefore, it is almost impossible to identify a perfect subset at the beginning in a single attempt, just based on the workload information. Given the iterative nature of knob tuning, runtime statistics are accumulated incrementally throughout the tuning process. So it is reasonable to select a good but not perfect subset as the starting point, and then we continually refine this subset based on the evolving runtime profile.

Outline. To alleviate the under-exploration issue caused by costly workload execution (**M1**), we propose to just tune a subset and find

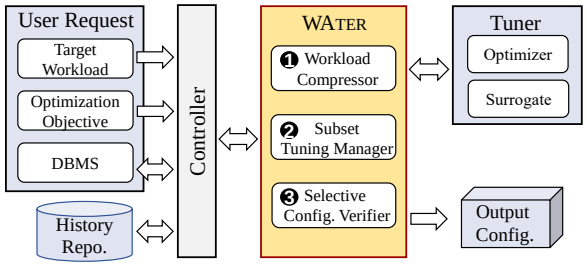


Figure 4: Overview of the Components in the WAter System

this approach promising (**M2**). Although we find that identifying a representative subset is crucial for effective tuning, this process is challenging due to the absence of a metric to quantify the representativity of a subset to its original workload (**M3**). Moreover, given the complexity of knob tuning, it is too difficult to identify a representative subset in a single attempt (**M4**). Therefore, we make the following technical contributions. To handle **M3**, we propose (1) a *representativity* metric based on runtime profile in Section 5.2, and (2) use a greedy algorithm to compress the workload in Section 5.3. Based on **M4**, we develop (3) a workload-adaptive knob tuning framework that periodically updates the subset in Section 4. Moreover, it (4) reuses runtime statistics for efficient subset tuning in Section 6, and (5) prunes, scores and ranks the proposed configurations for verification in Section 7.

4 System Overview

WAter is a workload-adaptive knob tuning system that speeds up the tuning process by reducing workload execution time using workload runtime profile. The high-level idea is that instead of repeatedly executing the entire complex workload, we split the tuning process into a series of short time slices and evaluate only a small subset of SQL queries in each. A time slice is a tuning cycle, where WAter selects a representative SQL subset (Section 5), tunes the subset to obtain configurations (Section 6) and finally evaluates promising configurations over the original workload (Section 7). Different subsets are selected in different time slices, and we continuously refine the subset based on evolving runtime profile.

Architecture. Figure 4 presents an overview of the architecture of WAter. On the client side, the user provides the target workload, optimization objective (e.g., throughput or latency) and the DBMS to tune. The *controller* deploys new configurations on DBMS, executes a set of queries, and collects performance metrics. WAter interacts with the *controller* to request query execution under specified configurations, gather the resulting execution data, and store it in the *history repository*. WAter contains three modules corresponding to the three steps in a time slice. First, the *workload compressor* uses the runtime profile to select a representative subset of queries from the target workload. Second, the *subset tuning manager* designates this SQL subset as the target workload for the current time slice and reuses existing tuning history to bootstrap the *tuner*'s *surrogate*, thereby enabling efficient subset tuning. Third, the *selective configuration verifier* prunes, ranks, and selects configurations proposed when tuning the aforementioned subset. We then verify the most promising configurations on the original workload to measure their actual performance.

Workflow. Figure 5 shows the tuning workflow. Instead of repeatedly replaying the entire workload or a fixed set of SQL queries, WAter divides the tuning process into multiple time slices, each evaluating a small subset of queries. The tuning consists of a sequence of time slices, with each slice comprising three steps: ❶ **Workload Compression:** Given an input workload, WAter uses a greedy algorithm driven by runtime statistics to compress the workload, aiming to maximize a custom *representativity* metric (Section 5). Since there is no runtime profile at the beginning, WAter uses existing methods like GSUM or random sampling to initialize the subset. ❷ **Subset Tuning:** WAter reuses its tuning history to initialize the local surrogate model for the current subset, thereby enabling efficient subset tuning that yields a series of configurations (Section 6). ❸ **Configuration Verification:** WAter uses heuristic rules and a hybrid scoring mechanism to identify the most promising configurations proposed in step ❷, which it then evaluates on the entire workload to determine their actual performance (Section 7).

5 Workload Compression

In this section, we redefine the workload compression problem for knob tuning (Section 5.1), introduce a runtime metric to measure the representativity of a SQL subset (Section 5.2), and present a greedy algorithm that optimizes this metric for runtime-adaptive workload compression (Section 5.3).

5.1 Problem Formulation

We first formulate the conventional workload compression problem and then redefine it within the context of knob tuning.

First, we formally define what is an original workload, a compressed workload and the corresponding compression ratio.

Definition 5.1 (Original Workload). Original Workload is a multi-set $\mathbf{W} = \{q_1, \ldots, q_n\}$ consisting of n SQL queries. Users' goal is to minimize the latency when executing this workload.

Definition 5.2 (Compressed Workload). Compressed Workload $\mathbf{W}'$ is a subset of $\mathbf{W}$: $\mathbf{W}' \subseteq \mathbf{W}$. Formally, $\mathbf{W}' = \{q_1, \ldots, q_m\}$ where $q_i \in \mathbf{W}$ and $m \leq n$.

Next, since workload compression task should be constrained by a given budget B, we define a cost of each query as follows:

Definition 5.3 (Query Cost). Each SQL query q_i is associated with a non-negative cost $c(q_i)$, where $c(q_i)$ is a function that quantifies the cost a query introduces to the tuner A in completing the tuning task. This cost could, for example, represent the number of indexable columns considered for index tuning, or the query execution time for knob tuning.

Definition 5.4 (Compression Ratio). Compression Ratio, $\eta = 1 - \frac{c(W')}{c(W)}$, is the fraction of workload that has been pruned.

Let $C(\mathbf{W})$ be the execution time of workload $\mathbf{W}$ under the default configuration, and $C_{K(\mathbf{W}',A)}(\mathbf{W})$ be its execution time under the configuration $K(\mathbf{W}', A)$ recommended by tuner A for subset $\mathbf{W}'$.

The conventional workload compression problem is defined as follows: given a compression budget $B \geq 0$, construct a compressed workload $\mathbf{W}' \subseteq \mathbf{W}$ such that [2, 7, 39]:

- $\sum_{\mathbf{q} \in \mathbf{W}'} c(\mathbf{q}) \leq B$, i.e., the cost of the compressed workload is less than the budget.

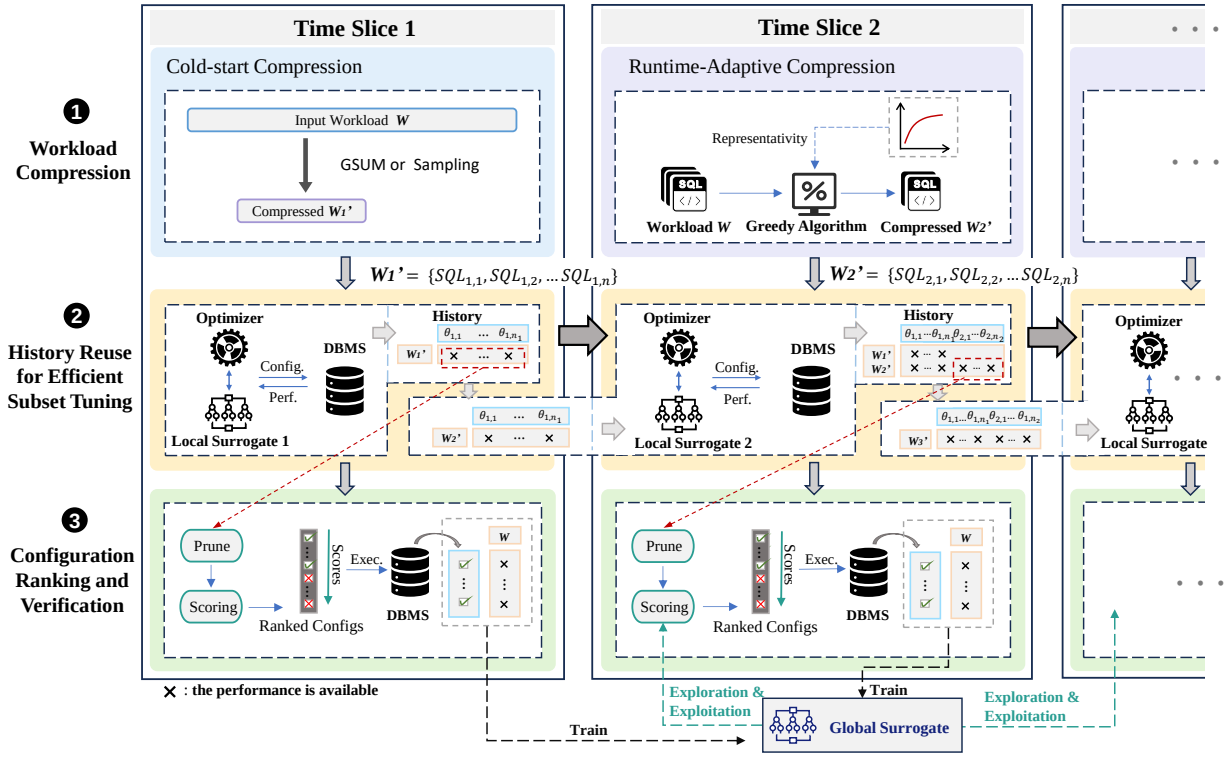


Figure 5: WATER Tuning Workflow

- $\mathbf{W}' = \arg\max_{\mathbf{W}' \subseteq \mathbf{W}} C(\mathbf{W}) - C_{K(\mathbf{W}', A)}(\mathbf{W})$, i.e., the reduction in the execution time of $\mathbf{W}$ is maximized when using the configuration $K(\mathbf{W}', A)$.

Existing methods on workload compression reduce tuning costs by identifying a subset of SQL queries to tune, which takes less time to execute and can serve as the representative of the original workload. These approaches prioritize a reduction in tuning time (i.e., runtime efficiency) at the expense of the resulting configuration's performance [2, 8, 16, 39].

While our method produces a configuration that achieves better performance than tuning on the full workload given the same tuning time. Given a time budget t, we *redefine* the workload compression problem in the context of knob tuning as follows:

$$\textbf{maximize} \quad C_{K(\mathbf{W}, A, t)}(\mathbf{W}) - C_{K(\mathbf{W}', A, t)}(\mathbf{W})$$

$$\textbf{subject to} \quad \textstyle\sum_{\mathbf{q} \in \mathbf{W}'} c(\mathbf{q}) \leq B, \quad \mathbf{W}' \subseteq \mathbf{W},$$

where $K(\mathbf{W}, A, t)$ is the configuration recommended by tuner A for $\mathbf{W}$ within time budget t, and $c(\mathbf{q})$ is the execution time of query $\mathbf{q}$ under the default configuration. Following prior work [2, 7, 8, 39], workload compression must be highly efficient, avoiding expensive operations like query execution or computing complex statistics.

There is an inherent trade-off between the quality of feedback and the associated evaluation cost in each tuning iteration. Traditional methods execute the entire workload to obtain feedback, producing high-quality results but incurring substantial overhead. In contrast, our method evaluates only a subset of the workload to reduce costs. Although this sacrifices some accuracy in each feedback iteration, it enables a greater number of tuning iterations within the same time constraints. As discussed in **M1** and **M2**, given the extremely large search space, the number of tuning iterations is insufficient to explore such a large space, and this is the main bottleneck of knob tuning. To address this challenge, we select a representative subset for tuning, trading off per-iteration feedback quality for an increased overall number of iterations, and finally achieving better performance than tuning the original workload under the same time budget. Moreover, we propose methods to mitigate the impact of reduced per-iteration quality as much as possible, which are discussed next.

Although $\mathbf{W}'$ allows faster convergence by evaluating more configurations due to reduced workload execution time, $K(\mathbf{W}, A, t)$ will eventually outperform $K(\mathbf{W}', A, t)$ with a sufficiently large tuning time budget t. This happens because $\mathbf{W}'$ is just an approximation of $\mathbf{W}$, and the bias between them will eventually lead to the optimization stagnating in later stages (see Figure 3). To address this, instead of maintaining a fixed compression ratio η, we: (1) refine the SQL subset without changing η, (2) once the subset's capacity is reached, decrease η to include more queries. This strategy balances the efficiency of subset tuning with the thoroughness of full workload tuning as the process continues.

5.2 Representative Subset

In this section, we introduce a *representativity* metric to measure how closely a selected subset's behavior aligns with the original workload in the context of knob tuning.

What is a representative subset? A representative subset is a small collection of queries whose performance accurately mirrors that of the full workload across different system settings. The goal is to preserve relative performance: if configuration A is faster than configuration B on the subset, it must also be faster on the full workload. This alignment is essential because system tuning relies on knowing whether one configuration is better than another, not on their absolute execution times. A truly representative subset ensures optimization decisions are based on this reliable ranking. This is why many statistical and other methods [2, 6–8, 39] are inadequate—they fail to maintain this critical performance relationship across configurations.

Representativity Metric Definition. Before introducing representativity, we need to maintain a run history defined as follows.

Definition 5.5 (Run History). Run history H is a two-dimensional table recording each query's execution time across all evaluated configurations. Specifically, $H[\mathbf{q}, \theta]$ represents the execution time of $\mathbf{q}$ under configuration θ.

Example 1. Table 2 illustrates an example of run history. The execution time of the workload $q_1, q_2, \ldots, q_n$ under configuration θ, denoted by $H[q_1, q_2, \ldots, q_n, \theta]$, is the sum of the individual query execution times: $H[q_1, \theta] + H[q_2, \theta] + \cdots + H[q_n, \theta]$.

The run history is updated each time a query is executed. Using this run history, we calculate *representativity* based on concordant performance pairs [38, 59]. For two configurations, θ_1 and θ_2, and two workloads, $\mathbf{W}$ and $\mathbf{W'}$, a performance pair is *concordant* if the ranking of $(H[\mathbf{W}, \theta_1], H[\mathbf{W}, \theta_2])$ matches that of $(H[\mathbf{W'}, \theta_1], H[\mathbf{W'}, \theta_2])$. Here, $H[\mathbf{W}, \theta_1]$ denotes the execution time of workload $\mathbf{W}$ under configuration θ_1.

Definition 5.6 (Representativity). Representativity of a compressed workload $\mathbf{W}'$ to its original workload $\mathbf{W}$ can be computed as:

$$R(\mathbf{W}', \mathbf{W}) = \frac{2}{|H| \times (|H| - 1)} \sum_{j=1}^{|H|} \sum_{k=j+1}^{|H|} \big(\mathbf{1}(H[\theta_j, \mathbf{W}] \le H[\theta_k, \mathbf{W}]) \oplus \mathbf{1}(H[\theta_j, \mathbf{W}'] \le H[\theta_k, \mathbf{W}'])\big). \tag{2}$$

where $|H|$ is the number of configurations in H, and $\oplus$ is the exclusive-nor operator. Essentially, *representativity* is the ratio of concordant performance pairs between the two workload in the history H.

Example 2. Assume we have obtained the execution time of $\mathbf{W}$ as (4, 5, 7) and $\mathbf{W'}$ as (3, 2, 6) over three configurations θ_1, θ_2, and θ_3, respectively. Then $R(\mathbf{W}', \mathbf{W})$ is computed as follows:

1. Pair the configurations in all possible combinations. We get (θ_1, θ_2), (θ_1, θ_3) and (θ_2, θ_3).

2. Judge the consistency of the performances of the two workloads on each configuration pair. We get $\mathbf{1}(4 \le 5) \odot \mathbf{1}(3 \le 2) = 0, \mathbf{1}(4 \le 7) \odot \mathbf{1}(3 \le 6) = 1$ and $\mathbf{1}(5 \le 7) \odot \mathbf{1}(2 \le 6) = 1$.

3. Compute $R(\mathbf{W}', \mathbf{W}) = \frac{2 \times (0+1+1)}{3 \times 2} = \frac{2}{3}$.

Representativity $R(\mathbf{W'}, \mathbf{W})$ ranges from $[0, 1]$. A higher $R(\mathbf{W'}, \mathbf{W})$ indicates that $\mathbf{W'}$ performs more similar to $\mathbf{W}$ across different configurations, and thus $\mathbf{W'}$ is more representative. In practice, $R(\mathbf{W'}, \mathbf{W})$ typically falls within $(0.5, 1]$, since random performances yields $R(\mathbf{W'}, \mathbf{W}) = 0.5$. When $R(\mathbf{W'}, \mathbf{W}) = 1$, two workloads are equivalent for knob tuning.

5.3 Runtime-Adaptive Compression

In this section, we demonstrate how to derive a representative subset of SQL queries from the evolving runtime profile by employing a greedy algorithm that optimizes the *representativity* metric. Subsequently, we introduce the adaptive compression strategy.

Greedy Algorithm-based SQL Subset Selection. We formalize the compression problem as follows:

$$\textbf{maximize} \quad R(\mathbf{W}', \mathbf{W})$$

$$\textbf{subject to} \quad \textstyle\sum_{\mathbf{q} \in \mathbf{W}'} c(\mathbf{q}) \le B, \quad \mathbf{W}' \subseteq \mathbf{W}$$

Optimizing the set function in this formulation is NP-hard [34]. However, we need to calculate an effective compressed workload with low overheads, otherwise we would lose the very purpose of workload compression in the first place [7]. Therefore, we develop a greedy algorithm (Algorithm 1) that trades-off accuracy to optimize *representativity* efficiently. Instead of enumerating all possible query combinations and finding the one which maximizes *representativity*, we loop over queries in $\mathbf{W}$ time after time, and each time we add one query that maximizes the normalized marginal gain $\Delta(\mathbf{q}|\mathbf{W'}_{i-1})$ to the current compressed workload $\mathbf{W'}_i$. The marginal gain is defined as

$$\Delta(\mathbf{q}|\mathbf{W}') = \frac{R(\mathbf{W}' \cup \{\mathbf{q}\}, \mathbf{W}) - R(\mathbf{W}', \mathbf{W})}{c(\mathbf{q})}.$$

In other words, the algorithm greedily chooses the query with the best gain per unit of cost [8]. A $1 - \frac{1}{e}$ approximation guarantee [34] is achieved by it when optimizing objectives holding two attributes: (1) *monotonicity* which means adding more samples cannot decrease the function value, and (2) *submodularity* which means the marginal gain of adding a new element decreases as the set grows. It is apparent that *representativity* holds these two attributes.

More importantly, in the context of knob tuning, adding a new query to the current set incurs the cost of executing this query for the missing configurations (detailed in Section 6). Since different queries have different costs, we need to consider this factor in our greedy algorithm. We quantify the additional costs of a query as follows:

Definition 5.7 (Lacked History). Given a run history H that records each query's performance across all proposed configurations (see Table 2), $\#lacked\_history(\mathbf{q})$ is the number of configurations in H for which performance on $\mathbf{q}$ is missing.

We define the final marginal gain (before normalization) to simultaneously maximize representativity and minimize additional costs as follows:

$$\Delta(\mathbf{q}|\mathbf{W}') = \frac{R(\mathbf{W}' \cup \{\mathbf{q}\}, \mathbf{W}) - R(\mathbf{W}', \mathbf{W})}{c(\mathbf{q})} - \beta \times \#lacked\_history(\mathbf{q}).$$

where β serves is a hyperparameter that balances overhead and the marginal gain of adding a new query. As β increases, queries with fewer missing performances in existing configurations are more likely to be selected.

The algorithm starts with an empty set (line 1) and initializes the marginal gain of each query (line 2). At the i-th iteration of the main

Algorithm 1: Greedy SQL Subset Selection

Input: Target Workload $\mathbf{W}$; Compression Ratio η;
Output: Compressed Workload $\mathbf{W}'$.

1 $\mathbf{W'} \leftarrow \emptyset$
2 $\forall \mathbf{q} \in \mathbf{W} : \Delta(\mathbf{q}) \leftarrow R(\{\mathbf{q}\}, \mathbf{W})$
3 **while** $W \neq \emptyset$ **do**
4 $\quad \Delta^* \leftarrow -\infty$
5 $\quad M_1 = \max_{\mathbf{q} \in \mathbf{W}} \frac{R(\mathbf{W}' \cup \{\mathbf{q}\}, \mathbf{W}) - R(\mathbf{W}', \mathbf{W})}{c(\mathbf{q})}$
6 $\quad m_1 = \min_{\mathbf{q} \in \mathbf{W}} \frac{R(\mathcal{S} \cup \{\mathbf{q}\}, \mathbf{W}) - R(\mathbf{W}', \mathbf{W})}{c(\mathbf{q})}$
7 $\quad M_2 = \max_{\mathbf{q} \in \mathbf{W}} \#lacked\_history(\mathbf{q})$
8 $\quad m_2 = \min_{\mathbf{q} \in \mathbf{W}} \#lacked\_history(\mathbf{q})$
9 $\quad$ **for** q *in* W **do**
10 $\quad\quad$ **if** $\Delta(q) > \Delta^*$ **then**
11 $\quad\quad\quad \Delta(\mathbf{q}) \leftarrow \frac{R(\mathbf{W}' \cup \{\mathbf{q}\}, \mathbf{W}) - R(\mathbf{W}', \mathbf{W}) - c(\mathbf{q}) m_1}{c(\mathbf{q})(M_1 - m_1)} - \beta \frac{\#lacked\_history(\mathbf{q}) - m_2}{M_2 - m_2}$
12 $\quad\quad$ **end**
13 $\quad\quad$ **if** $\Delta(q) > \Delta^*$ **then**
14 $\quad\quad\quad \Delta^* \leftarrow \Delta(\mathbf{q})$
15 $\quad\quad\quad \mathbf{q}^* \leftarrow \mathbf{q}$
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 $\quad$ **if** $c(\mathbf{W}') + c(q^*) \leq \eta \times c(\mathbf{W})$ **then**
19 $\quad\quad \mathbf{W}' \leftarrow \mathbf{W}' \cup \{\mathbf{q}^*\}$
20 $\quad$ **end**
21 $\quad \mathbf{W} \leftarrow \mathbf{W} \backslash \{\mathbf{q}^*\}$
22 **end**
23 **return** $\mathbf{W}'$;

loop (line 3), it first computes the maximum and minimum values of the marginal gain (line 5, 6) and $\#lacked\_history(\mathbf{q})$ (line 7, 8) a query from $\mathbf{W}$ could have. These values are later used to normalize the marginal gain and the penalty in line 11. Then we loop over queries in $\mathbf{W}$ to find the query with the highest marginal gain (line 9-16). Note that we employ the CELF (Cost-Effective Lazy Forward selection) algorithm [26] which utilizes the monotonicity and submodularity attributes of R to minimize function evaluations (lines 10). Assume we are in the i-th iteration of the **while** loop. In line 10, if $\Delta(\mathbf{q})$, which is actually $\Delta(\mathbf{q}|\mathbf{W'}_j)(j < i)$, is not more than the current best marginal gain Δ^*, then the real $\Delta(\mathbf{q}|\mathbf{W'}_i)$ should be less than Δ^* due to the submodularity of R (i.e., $\Delta(\mathbf{q}|\mathbf{W'}_{i-1}) > \Delta(\mathbf{q}|\mathbf{W'}_i)$ holds). In this case, we are exempt from calculating $\Delta(\mathbf{q}|\mathbf{W'}_i)$ in line 11. In line 11, we normalize both the marginal gain and the lacked history penalty to a common scale (0 to 1) using min-max scaling. This ensures that the hyperparameter β provides a balanced trade-off between the two competing objectives. Finally, the query that maximizes the marginal gain is added to **W'**, provided it does not exceed the compression ratio η (line 18, 19).

Runtime Analysis. The algorithm's runtime depends on the number of queries, n, and the number of configurations, m. In the worst-case scenario, the main **while** loop (line 3) iterates n times, computing the marginal gain (line 11) for each query in every iteration. This nested process leads to a time complexity of $O(n^2)$. The computation of R involves a nested loop of m configurations, adding a complexity of $O(m^2)$ to that step. Consequently, the total runtime complexity for compressing a workload is $O(n^2 m^2)$. Despite the polynomial complexity, the method is highly practical for its intended use. In the knob-tuning literature, both n and m are typically on the order of hundreds [5, 9, 10, 12, 17, 23, 24, 27, 42, 54, 57, 58], making the computation feasible. Furthermore, as we show in the cost analysis in Section 8.5, the overhead from WAter is negligible compared to the overall cost of the tuning process itself. Finally, handling streaming workloads or workloads with a huge number of queries is considered outside the scope of this work and constitutes a separate research direction [51].

Adaptive Compression Strategy. Workload compression occurs at the beginning of each time slice, leveraging an evolving runtime profile to continuously refine the subset. As more data becomes available, the subset becomes increasingly representative. Periodically updating the subset also prevents the optimization process from getting trapped in local optima, which can happen with a fixed subset. The compression ratio η is dynamic and decreases to increase the subset size when optimization fails to find a better configuration within a time slice. This indicates that the subset may not be sufficiently representative, reaching its representativity limit. Although reducing η increases overhead, it enhances subset representativity and enables more effective optimization.

6 History Reuse for Efficient Tuning

After workload compression, we get a newly selected SQL subset which is then frequently evaluated to guide the optimization. In this section, we first introduce the challenge when tuning different subsets in different time slices, then we discuss how we address this challenge to achieve efficient subset tuning.

Challenge. Effective knob tuning depends on a well-trained surrogate model that accurately predicts a workload's performance across various configurations. Existing methods maintain a single surrogate because they focus on a fixed workload [17, 23, 55]. In contrast, we tune different SQL subsets over time slices, necessitating a new surrogate for each subset since one surrogate cannot model multiple workloads. Bootstrapping a surrogate from scratch is costly, requiring numerous workload executions to gather training data. Although some transfer learning techniques enhance efficiency, they demand collecting thousands to tens of thousands of observations in advance [27, 42, 57], which is time-consuming and not universally applicable across different systems, hardware, and workloads. Moreover, transferred observations may not fit new subsets well, potentially misleading the optimization process.

History Reuse for Surrogate Bootstrapping. We leverage execution statistics for queries in the selected subset **W'** from previous time slices, recorded in the tuning history H, to bootstrap the surrogate without expensive workload executions. There are two scenarios: *S1 (Complete History):* If every query in **W'** has been executed for all configurations in H, we sum their execution times per configuration to determine the total execution time for **W'** on those configurations. This data is then used to bootstrap the surrogate (Example 3). *S2 (Incomplete History):* If some queries in **W'** lack execution times for certain configurations, we execute these

missing queries for those configurations (Example 4) and then aggregate as in S1. Although this incurs some costs, it is significantly cheaper than bootstrapping the surrogate from scratch. To account for these costs, we incorporate a penalty term in the marginal gain computation (Algorithm 1, line 11) as discussed in Section 5.3.

This method allows us to bootstrap the surrogate on the fly without costly initial workload executions and ensures the data accurately reflects the subset's performance. As optimization progresses, accumulating data enhances the surrogate's accuracy for subsequent time slices.

Subset Tuning. After bootstrapping the *surrogate*, we use the tuner to optimize the subset. In each iteration, a proposed configuration is evaluated on the subset to update its best performance, and the resulting performance metrics are sent to the tuner to guide subsequent optimizations.

Table 2: A Toy Example of History

	θ_1	θ_2	θ_3
$\mathbf{q}_1$	$H[\mathbf{q}_1, \theta_1]$	$H[\mathbf{q}_1, \theta_2]$	$H[\mathbf{q}_1, \theta_3]$
$\mathbf{q}_2$	$H[\mathbf{q}_2, \theta_1]$		$H[\mathbf{q}_2, \theta_3]$
$\mathbf{q}_3$	$H[\mathbf{q}_3, \theta_1]$		
$\mathbf{q}_4$	$H[\mathbf{q}_4, \theta_1]$	$H[\mathbf{q}_4, \theta_2]$	$H[\mathbf{q}_4, \theta_3]$

Example 3. Assume $\mathbf{W}' = \{\mathbf{q}_1, \mathbf{q}_4\}$ for the subsequent time slice and we have run history illustrated in Table 2. The surrogate for the next time slice should be bootstrapped with the data: $\{(\theta_1, H[\{\mathbf{q}_1, \mathbf{q}_4\}, \theta_1]), (\theta_2, H[\{\mathbf{q}_1, \mathbf{q}_4\}, \theta_2]), (\theta_3, H[\{\mathbf{q}_1, \mathbf{q}_4\}, \theta_3])\}$.

Example 4. Assume $\mathbf{W}' = \{\mathbf{q}_2, \mathbf{q}_3\}$ for the subsequent time slice and we have run history illustrated in Table 2 which lacks $H[\mathbf{q}_2, \theta_2], H[\mathbf{q}_3, \theta_2], H[\mathbf{q}_3, \theta_3]$. We need to deploy θ_2 and run $\mathbf{q}_2, \mathbf{q}_3$, and deploy θ_3 and run $\mathbf{q}_3$.

7 Configuration Pruning, Ranking, Verification

After multiple subset tuning iterations, we identified and evaluated several configurations on the selected subset. However, our ultimate goal is to find configurations that perform well on the entire workload and report the real performance. To avoid exhaustive verification, we focus only on promising configurations by applying heuristic rules to eliminate unpromising ones, ranking the remaining options with a hybrid scoring mechanism, and selecting top configurations (e.g., 30%) for verification on the entire workload.

Motivation. When selecting configurations for the entire workload, we face the exploration–exploitation dilemma [1]. *Exploitation* chooses the best configuration based on current knowledge, including subset performance and model predictions. However, this can lead to suboptimal configurations, as a configuration that performs well on a subset may perform poorly on the full workload. Additionally, prediction models may be biased toward familiar configurations while underestimating unfamiliar ones. *Exploration*, on the other hand, involves testing some unfamiliar configurations to discover unexpectedly high performers. Balancing exploration and exploitation is essential for achieving a global optimum. To address this, we propose a *hybrid scoring mechanism* that effectively balances both strategies by scoring candidate configurations and selecting the top-ranked ones for further verification.

Global Surrogate. We maintain a global surrogate model, $\mathcal{RF}$, for scoring, trained on historical $(\theta, H[\mathbf{W}, \theta])$ pairs. It predicts performance and uncertainty estimates. We use a random forest regressor for its superior performance in knob tuning [55] and ability to quantify uncertainty [15].

Definition 7.1 (Uncertainty). Given an unlabeled configuration θ and a Random Forest $\mathcal{RF} = \{rf_1, \ldots, rf_n\}$ with n estimators, the uncertainty $\Psi(\theta, \mathcal{RF})$ is the variance of their predictions for θ.

Exploitation. We approximate a candidate configuration θ's performance by combining its subset execution time $cost(\mathbf{W}')$ with the global surrogate $\mathcal{RF}$'s prediction. The *predicted performance* (lower is better), $\hat{f}_{\mathbf{W}}(\theta)$, is formulated as follows:

$$\hat{f}_{\mathbf{W}}(\theta) = -[(1 - \frac{|\mathbf{W}'|}{|\mathbf{W}|})\mathcal{RF}(\theta) + cost(\mathbf{W}')]. \tag{3}$$

Exploration. Inspired by active learning [4, 32, 37], we prioritize configurations that differ significantly from already labeled instances (i.e., $H[\mathbf{W}, \theta]$ is available) or those where the surrogate has low confidence in the prediction. We first introduce the definition of SetSimilarity and Uncertainty.

Definition 7.2 (SetSimilarity). Given a labeled configuration set $\mathcal{D}$ and an unlabeled configuration θ, SetSimilarity $\Phi(\theta, \mathcal{D}) = \max_{d \in \mathcal{D}} \phi(\theta, d)$, where ϕ is the similarity function.

We use the Gower distance $D(x, y)$, which measures the distance between two data points with mixed types of variables (numerical and categorical) [11], to define the similarity function ϕ:

$$\phi(x, y) = \frac{1}{1 + D(x, y)},$$

where

$$D(x, y) = \frac{1}{n} \sum_{i=1}^{n} d_i(x, y),$$

and for numerical variables:

$$d_i(x, y) = \frac{|x_i - y_i|}{\max(x_i) - \min(x_i)},$$

and for categorical variables:

$$d_i(x, y) = \begin{cases} 0, & \text{if } x_i = y_i \\ 1, & \text{if } x_i \neq y_i \end{cases}.$$

We use $1 - \Phi(\theta, \mathcal{D})$ to give high scores to instances that do not share much similarity with already labeled documents (diversity).

We use the verification ratio α, the proportion of labeled to proposed configurations, to balance diversity and uncertainty prioritization. The *exploration potential* of θ is defined as:

$$g(\theta) = \alpha(1 - \Phi(\theta, \mathcal{D})) + (1 - \alpha)\Psi(\theta, \mathcal{RF}) \tag{4}$$

Hybrid Scoring Mechanism. In each time slice, we randomly choose to either *exploit* ($\hat{f}_{\mathbf{W}}(\theta)$) or *explore* ($g(\theta)$) a configuration θ. We select *exploitation* with probability $1 - \eta$ (subset volume), reflecting our current knowledge of θ. The more we know about a configuration, the more likely we are to exploit it:

$$S(\theta) = \begin{cases} \hat{f}_{\mathbf{W}}(\theta), & \text{with probability } 1 - \eta \\ g(\theta), & \text{with probability } \eta \end{cases} \tag{5}$$

Configuration Pruning and Selection. In each time slice, when $\hat{f}_{\mathbf{W}}$ is selected, configurations worse than the default are pruned.

When g is selected, configurations performing 1.2 times worse than the default are discarded. The remaining configurations are scored by the corresponding function, and the top-scoring ones are selected for verification. In the first time slice, due to the lack of labeled data, we simply discard configurations worse than the default and rank the remaining ones based on their performance on the compressed workload.

Verification. To verify the selected configurations on the original workload $\mathbf{W}$, we deploy them to the database and execute only the remaining subset $\mathbf{W} - \mathbf{W}'$, since $\mathbf{W}'$ was already evaluated during tuning. The execution results update the global surrogate, and WAter outputs the best-performing configuration.

8 Experiments

8.1 Experimental Setup

Workloads. We focus exclusively on OLAP workloads, as OLTP workloads are typically evaluated over fixed intervals, making workload compression inapplicable. Our experiments utilize three well-known database benchmarks: TPC-DS, JOB [25], and TPC-H. Since TPC-DS is unsuitable for knob tuning [55], we exclude templates with execution times significantly longer than others, following [13, 18, 19]. For TPC-H, we use two variants: TPC-H and TPC-H×10, the latter of which includes 10 instances generated with different random seeds per template. Table 3 summarizes the used workloads.

Table 3: Summary of Workloads

Workload	Queries	Templates	Tables	Columns
TPC-DS* (sf=1)	88	88	24	237
JOB (5.2GB)	113	113	21	38
TPC-H (sf=10)	22	22	8	55
TPC-H×10 (sf=10)	220	22	8	55

*: template 1, 4, 6, 11, 14, 23, 24, 39, 74, 81, and 95 are removed.

Hardware. All experiments are conducted on (C1) a virtual machine with 32 vCPU and 60GB of RAM on a private server with an AMD EPYC 9654 96-Core Processor, or (C2) Alibaba Cloud Platform with an ecs.e-c1m4.xlarge instance with 4 vCPU and 16 GB of RAM.

Adopted Tuners. WAter is a generic optimization framework that enhances the tuning efficiency of existing tuners. We integrate it with **SMAC** [30], which recent evaluations [55] show outperforms eight state-of-the-art DBMS tuners, and with **GPTuner** [23], which leverages domain knowledge for knob tuning. We utilize the open-source GPTuner code, updating its knowledge based on hardware, and implement SMAC using the SMAC3 [30] library.

Baselines. We compare WAter with the following baselines: **1. Original.** Utilizes the vanilla tuner (SMAC or GPTuner) to optimize the entire workload, highlighting WAter's advantages. **2. GSUM [8].** A state-of-the-art workload compression method that maximizes both *coverage* and *representativity* as described in Section 2.2. **3. Random.** Selects SQLs uniformly at random. Both **GSUM** and **Random** are static pre-processing techniques applied initially to obtain a subset for tuning. If a configuration outperforms the default on this subset, it is immediately evaluated on the entire workload. The compression ratios for **GSUM** and **Random** are set to be the same as WAter's initial compression ratio by default.

WAter Implementation. We implement WAter in Python3 on top of the two tuners. The global surrogate uses scikit-learn's RandomForestRegressor [36] with default parameters. The compression ratio η starts at 0.75 and decreases by 0.1 if no better configuration is found within a time slice. In Algorithm 1, β is set to 0.1. In each time slice, 20 valid configurations are proposed during subset tuning, with 25% (verification ratio α) evaluated on the entire workload. For the initial time slice without runtime history, we employ Latin Hypercube Sampling (LHS) [33], a space-filling sampling strategy, to generate ten samples for surrogate initialization, following previous works [9, 17, 23, 55]. We use GSUM to select the initial subset.

Tuning Settings. We conduct experiments with PostgreSQL v14.9, tuning 57 knobs from GPTuner's open-source repository [22]. For each method, we perform three tuning sessions and report the average best performance (over the entire workload) with a solid line and [5%, 95%] confidence interval shaded in the same color [17]. Following [48, 49, 56], we use total workload execution time as the performance metric. Each method undergoes at least 100 tuning iterations, with the first 10 generated randomly using LHS [33], following previous works [9, 23, 42, 55]. For failed or long-running configurations (those causing DBMS crashes or taking more than twice the execution time of the default), we assigned twice the default performance to prevent scaling issues [43].

Evaluation Metrics. Following LlamaTune [17], we use two metrics to evaluate WAter: ***final performance improvement*** (i.e., execution time reduction) and relative ***time-to-optimal speedup***, which reports the earliest iteration at which WAter has found a better-performing configuration compared to the baseline optimal, as well as the relative speedup.

8.2 Performance Comparison

End-to-end Comparison. Figure 6 compares WAter (integrated with GPTuner) against baselines. The initial gap in the red line reflects cold-start compression and subset tuning times during the first time slice. Compared to **Original**, WAter achieves the best performance identified by GPTuner **4.2**× faster across all four workloads on average. Specifically, WAter delivers time-to-optimal speedups of **2.5**× for TPC-DS, **2.1**× for JOB, and **11.0**× for TPC-H×10, thanks to improved runtime efficiency. In terms of final results, WAter reduces execution time by an average of **39.1**% compared to the default and is **6.4**% faster than GPTuner's best. WAter's advantage over GPTuner on the TPC-H benchmark is initially modest, a result of two key factors. First, with only 22 queries, any small subset struggles to capture the diverse performance characteristics of the full workload. Second, the workload's shorter execution time lessens the overall impact of runtime efficiency gains. However, WAter's dynamic strategy proves crucial in the long run. By continuously refining its query subset, it avoids the performance plateaus that cause static methods like GSUM and Random to stagnate, ultimately allowing it to find superior configurations in later stages.

While **GSUM** and **Random** find interesting configurations early on, their optimization stagnates, ultimately failing to outperform GPTuner on average. Despite being a generic compression framework, **GSUM** does not always surpass random sampling in knob tuning, as its features are not specifically designed for this task and may lead to suboptimal compression. **Random** typically produces

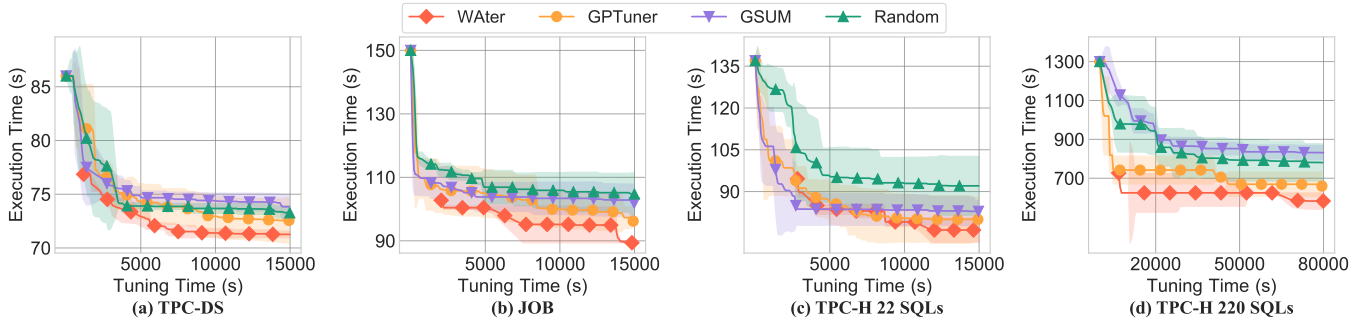


Figure 6: Performance on different benchmarks (bottom-left is better)

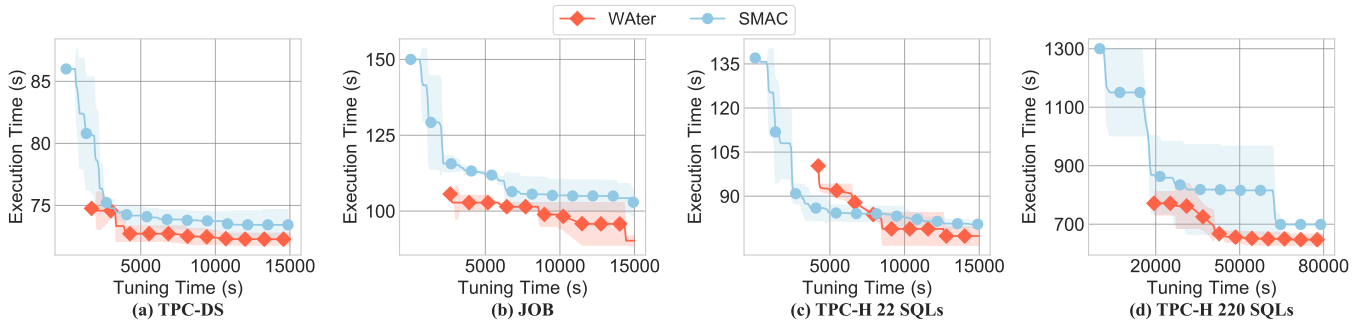


Figure 7: Performance on different benchmarks (SMAC-based) (bottom-left is better)

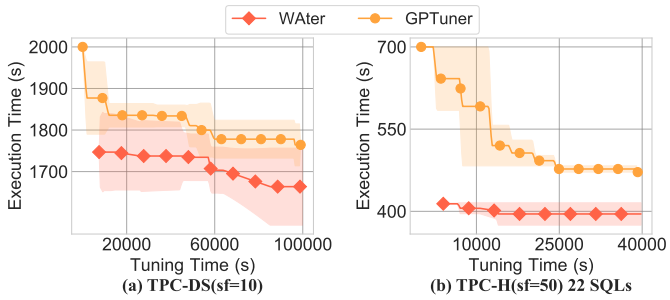


Figure 8: Performance under different scale factors

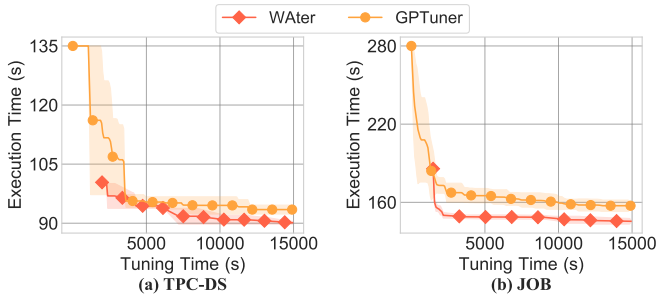


Figure 9: Performance on different machine

wider shadows in the figure, particularly for workloads with fewer SQLs (e.g., TPC-H), indicating greater instability. To ensure fairness, we also compare **GSUM** under modified compression ratios with WAteron TPC-DS. As shown in Figure 10, **GSUM** underperforms WAter on all compression ratios of 0.3, 0.5, and 0.7.

8.3 Robustness Study

Different Optimizer. To demonstrate WAter's versatility with different optimizers, we replace the optimizer with SMAC. As shown in Figure 7, WAter outperforms vanilla SMAC across all four workloads, achieving a **37.5**% mean reduction in execution time compared to the default and **6.6**% less time than SMAC's best configuration. Additionally, WAter provides a **3.1**× time-to-optimal speedup on average. We achieve speedups of **3.8**× and **5.3**× on TPC-DS and JOB, reducing execution time by **15.9**% and **40.0**%, respectively. While TPC-H remains a challenge for WAter, it initially lags but ultimately outperforms the vanilla optimizer as the subset evolves.

Different Data Size. We study WAter's scalability across different database sizes by varying the scale factor of TPC-H from 10 to 50 and TPC-DS from 1 to 10. As shown in Figure 8, compared to GPTuner, WAter finds better configurations in much less time. For both of the workloads, WAter finds better configurations than the optima of GPTuner at the very beginning, achieving time-to-optimal speedups of **12.9**× and **9.8**× for TPC-DS and TPC-H. In the end, WAter achieves execution times which are **16.8**% and **43.5**% less than default and **5.7**% and **16.2**% less than GPTuner on TPC-DS and TPC-H respectively. This demonstrates that as the cost of a single evaluation increases, the benefit of WAter's runtime efficiency becomes overwhelmingly significant, allowing it to explore many more configurations in the same time budget.

Different Hardware. We switch from hardware C1 to C2, which has significantly fewer CPU cores and less RAM. This change makes optimization more challenging because reduced resources increase the complexity of modeling the relationship between configurations

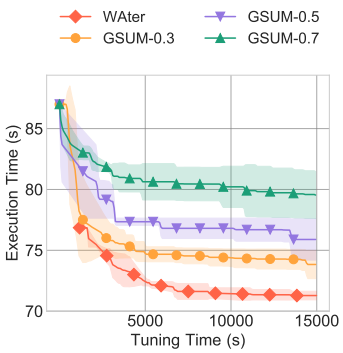
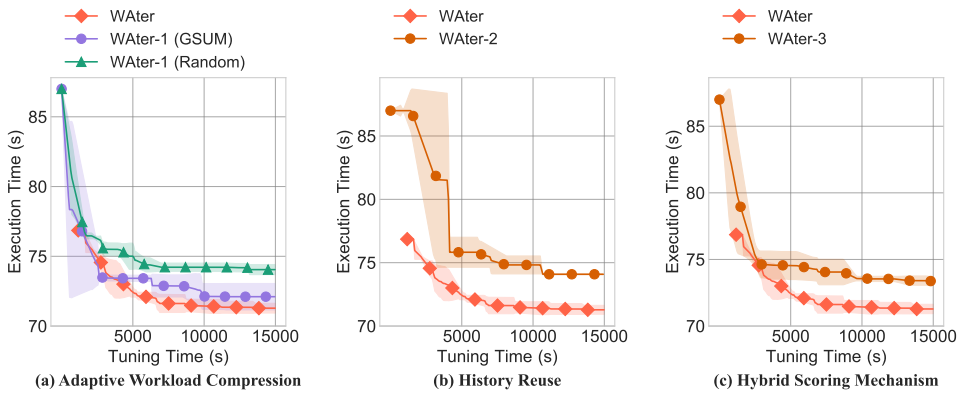


Figure 10: GSUM with Different Compression Ratio



(a) Adaptive Workload Compression (b) History Reuse (c) Hybrid Scoring Mechanism

Figure 11: Ablation study of WAter on TPC-DS (bottom-left is better)

and DBMS performance, revealing more bottlenecks and shrinking the feasible region [23]. We exclude TPC-H experiments as they frequently cause system crashes on C2. Figure 9 shows that compared to GPTuner, WAter finds better configurations in fewer iterations for both workloads. For JOB, WAter identifies a configuration superior to GPTuner's best on the first attempt (**7.9**× speedup) and ultimately achieves a workload execution time **7.6**% less than GPTuner's best. For TPC-DS, WAter achieves a **1.9**× time-to-optimal speedup and reduces execution time by **3.6**%.

8.4 Ablation Study

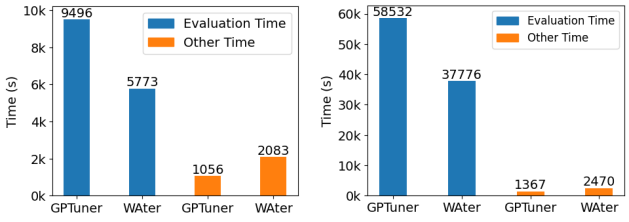
Effect of Adaptive Workload Compression. We evaluate our adaptive workload compression framework and the algorithm from Section 5 by keeping the subset fixed across all time slices. Using the same subsets as **GSUM** and **Random**, denoted "WAter-1 (GSUM)" and "WAter-1 (Random)", Figure 11(a) shows that WAter outperforms both, achieving speedups of **2.0**× and **4.3**×, and reducing execution time by **1.1**% and **3.6**%, respectively.

Effect of History Reuse. To assess *History Reuse for Efficient Subset Tuning* (Section 6), we use LHS [33] to randomly sample and evaluate configurations to bootstrap the surrogate in each time slice, which is referred to as "WAter-2". As shown in Figure 11(b), WAter-2 stagnates early in optimization, while WAter achieves an additional **3.8**% reduction in execution time and a **3.5**× speedup. The result is due to WAter-2's initialization overhead and undertrained surrogates from limited observations.

Effect of Hybrid Scoring Mechanism. To demonstrate the hybrid scoring mechanism's effectiveness (Section 7), we replace it with a scoring method based solely on subset performance, denoted "WAter-3". Figure 11(c) shows that WAter achieves a **4.1**× speedup and reduces execution time by **3.1**% compared to WAter-3. This is because WAter-3 cannot reliably identify configurations that perform well across the entire workload, since configurations that perform well on the subset do not necessarily also perform well across the entire workload.

8.5 Cost Analysis

We divide the tuning time into two parts: (1) Evaluation Time, the duration spent executing queries, and (2) Other Time, covering tuner's overhead, algorithmic overhead and so on. Figure 12 shows the time spent in both categories during 100 tuning iterations for TPC-H (sf=10) and TPC-H (sf=50) using WAter and GPTuner. WAter reduces the overall tuning time by **25.5**% and **32.8**% for TPC-H (sf=10) and TPC-H (sf=50), respectively, compared to GPTuner, primarily due to a reduction in "Evaluation Time". Although WAter incurs more "Other Time" due to additional overhead (e.g., model training, more configuration deployments), the large decrease in "Evaluation Time" more than offsets this. WAter's advantage is particularly significant for workloads with a large "Evaluation Time," as it can substantially reduce this component. In contrast, "Other Time" remains unaffected by workload size and stays constant. This explains why WAter performs better on workloads with larger scale factors (Figure 8). In real-world production environments, evaluation times of OLAP workloads are typically much longer than those presented in our experiments [44, 45, 47], where WAter demonstrates even greater potential.



(a) TPC-H (sf=10) (b) TPC-H (sf=50)

Figure 12: Cost Analysis

9 Conclusion

This paper presents WAter, a runtime-efficient and workload-adaptive knob tuning system. To reduce evaluation costs, WAter divides the tuning process into time slices and evaluates small, representative query subsets instead of the full workload. Experiments show that WAter substantially reduces tuning time and finds superior configurations compared to state-of-the-art methods.

References

[1] Oded Berger-Tal, Jonathan Nathan, Ehud Meron, and David Saltz. 2014. The exploration-exploitation dilemma: a multidisciplinary framework. *PloS one* 9, 4 (2014), e95693.

[2] Matteo Brucato, Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2024. Wred: Workload Reduction for Scalable Index Tuning. *Proc. ACM Manag. Data* 2, 1, Article 50 (mar 2024), 26 pages. doi:10.1145/3639305

[3] 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. *Proceedings of the VLDB Endowment* 18, 6 (2025), 1865–1878.

[4] Thiago N.C. Cardoso, Rodrigo M. Silva, Sérgio Canuto, Mirella M. Moro, and Marcos A. Gonçalves. 2017. Ranked batch-mode active learning. *Information Sciences* 379 (2017), 313–337. doi:10.1016/j.ins.2016.10.037

[5] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: a contextual gaussian process bandit approach for the automatic tuning of IT configurations under varying workload conditions. *Proc. VLDB Endow.* 14, 8 (apr 2021), 1401–1413. doi:10.14778/3457390.3457404

[6] Surajit Chaudhuri, Prasanna Ganesan, and Vivek Narasayya. 2003. Primitives for workload summarization and implications for SQL. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 730–741.

[7] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. 2002. Compressing SQL workloads. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 488–499. doi:10.1145/564691.564747

[8] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey Naughton, and Stratis Viglas. 2020. Comprehensive and efficient workload compression. *Proc. VLDB Endow.* 14, 3 (nov 2020), 418–430. doi:10.14778/3430915.3430931

[9] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[10] Victor Giannakouris and Immanuel Trummer. 2024. Demonstrating -Tune: Exploiting Large Language Models for Workload-Adaptive Database System Tuning. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 508–511. doi:10.1145/3626246.3654751

[11] John C Gower. 1971. A general coefficient of similarity and some of its properties. *Biometrics* (1971), 857–871.

[12] Yaniv Gur, Dongsheng Yang, Frederik Stalschus, and Berthold Reinwald. 2021. Adaptive Multi-Model Reinforcement Learning for Online Database Tuning.. In *EDBT*. 439–444.

[13] Stefan Halfpap. 2023. Hybrid Index Selection Using Integer Linear Programming Based on Cached Cost Estimates of Heuristic Approaches. In *Proceedings of the 1st Workshop on Simplicity in Management of Data* (Bellevue, WA, USA) *(SiMoD '23)*. Association for Computing Machinery, New York, NY, USA, Article 5, 4 pages. doi:10.1145/3596225.3596227

[14] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[15] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization* (Rome, Italy) *(LION'05)*. Springer-Verlag, Berlin, Heidelberg, 507–523. doi:10.1007/978-3-642-25566-3_40

[16] Shrainik Jain, Bill Howe, Jiaqi Yan, and Thierry Cruanes. 2018. Query2Vec: An Evaluation of NLP Techniques for Generalized Workload Analytics. arXiv:1801.05613 [cs.DB] https://arxiv.org/abs/1801.05613

[17] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. arXiv:2203.05128 [cs.DB] https://arxiv.org/abs/2203.05128

[18] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proc. VLDB Endow.* 13, 12 (July 2020), 2382–2395. doi:10.14778/3407790.3407832

[19] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. 2022. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning.. In *EDBT*, Vol. 2. 155–2.

[20] Brian Kroth, Sergiy Matusevych, Rana Alotaibi, Yiwen Zhu, Anja Gruenheid, and Yuanyuan Tian. 2024. MLOS in Action: Bridging the Gap Between Experimentation and Auto-Tuning in the Cloud. *Proc. VLDB Endow.* 17, 12 (Nov. 2024), 4269–4272. doi:10.14778/3685800.3685852

[21] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1667–1683. doi:10.1145/3318464.3380591

[22] Jiale Lao. 2024. *GPTuner code.* Retrieved October 1, 2024 from https://github.com/SolidLao/GPTuner/blob/main/knowledge_collection/postgres/target_knobs.txt

[23] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (may 2024), 1939–1952. doi:10.14778/3659437.3659449

[24] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Yuanchun Zhou, Mingjie Tang, and Jianguo Wang. 2024. A Demonstration of GPTuner: A GPT-Based Manual-Reading Database Tuning System. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 504–507. doi:10.1145/3626246.3654739

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. doi:10.14778/2850583.2850594

[26] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Jose, California, USA) *(KDD '07)*. Association for Computing Machinery, New York, NY, USA, 420–429. doi:10.1145/1281192.1281239

[27] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: a query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2118–2130. doi:10.14778/3352063.3352129

[28] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[29] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. 2024. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 3680–3693. doi:10.14778/3681954.3682030

[30] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research* 23, 54 (2022), 1–9. http://jmlr.org/papers/v23/21-0888.html

[31] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active Learning for ML Enhanced Database Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 175–191. doi:10.1145/3318464.3389768

[32] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active Learning for ML Enhanced Database Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 175–191. doi:10.1145/3318464.3389768

[33] 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* (Arlington, Virginia, USA) *(WSC '92)*. Association for Computing Machinery, New York, NY, USA, 557–564. doi:10.1145/167293.167637

[34] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. 1978. An analysis of approximations for maximizing submodular set functions–I. *Math. Program.* 14, 1 (Dec. 1978), 265–294. doi:10.1007/BF01588971

[35] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[37] Burr Settles. 2009. Active learning literature survey. (2009).

[38] Yu Shen, Xinyuyang Ren, Yupeng Lu, Huaijun Jiang, Huanyong Xu, Di Peng, Yang Li, Wentao Zhang, and Bin Cui. 2023. Rover: An Online Spark SQL Tuning Service via Generalized Transfer Learning. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 4800–4812. doi:10.1145/3580305.3599953

[39] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 660–673. doi:10.1145/3514221.3526152

[40] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* 25 (2012).

[41] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. 2004. Using probabilistic reasoning to automate software tuning. *SIGMETRICS Perform. Eval. Rev.* 32, 1 (June 2004), 404–405. doi:10.1145/1012888.1005739

[42] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. doi:10.1145/3035918.3064029

[43] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proc. VLDB Endow.* 14, 7 (mar 2021), 1241–1253. doi:10.14778/3450980.3450992

[44] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 3694–3706. doi:10.14778/3681954.3682031

[45] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1413–1425. doi:10.14778/3583140.3583156

[46] Oleksii Vasyliev. [n. d.]. *PGTune.* Retrieved October 1, 2024 from https://pgtune.leopard.in.ua

[47] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 449–462.

[48] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: universal database optimization using reinforcement learning. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3402–3414. doi:10.14778/3484224.3484236

[49] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: universal database optimization using reinforcement learning. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3402–3414. doi:10.14778/3484224.3484236

[50] D.H. Wolpert and W.G. Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (1997), 67–82. doi:10.1109/4235.585893

[51] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proc. ACM Manag. Data* 2, 1, Article 38 (March 2024), 27 pages. doi:10.1145/3639293

[52] Yang Wu, Xuanhe Zhou, Yong Zhang, and Guoliang Li. 2024. Automatic index tuning: A survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 12 (2024), 7657–7676.

[53] Tao Yu, Zhaonian Zou, Weihua Sun, and Yu Yan. 2024. Refactoring Index Tuning Process with Benefit Estimation. *Proc. VLDB Endow.* 17, 7 (may 2024), 1528–1541. doi:10.14778/3654621.3654622

[54] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 415–432. doi:10.1145/3299869.3300085

[55] 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. *Proc. VLDB Endow.* 15, 9 (may 2022), 1808–1821. doi:10.14778/3538598.3538604

[56] Xinyi Zhang, Zhuo Chang, Hong Wu, Yang Li, Jia Chen, Jian Tan, Feifei Li, and Bin Cui. 2023. A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning. *Proc. ACM Manag. Data* 1, 2, Article 186 (June 2023), 26 pages. doi:10.1145/3589331

[57] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2102–2114. doi:10.1145/3448016.3457291

[58] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 631–645. doi:10.1145/3514221.3526176

[59] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. 2023. An Efficient Transfer Learning Based Configuration Adviser for Database Tuning. *Proc. VLDB Endow.* 17, 3 (nov 2023), 539–552. doi:10.14778/3632093.3632114

[60] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2023. Automatic Database Knob Tuning: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (2023), 12470–12490. doi:10.1109/TKDE.2023.3266893