WAter: A Workload-Adaptive Knob Tuning System

Yibo Wang
Sichuan University
Chengdu, China
wangyibo.cs@gmail.com

Jiale Lao
Cornell University
Ithaca, New York, USA
jiale@cs.cornell.edu

Chen Zhang
Sichuan University
Chengdu, China
zhangbihan999@gmail.com

Cehua Yang
Sichuan University
Chengdu, China
yangcehua@stu.scu.edu.cn

Yuanchun Zhou
Computer Network Information
Center, Chinese Academy of Sciences
Beijing, China
zyc@cnic.cn

Jianguo Wang
Purdue University
West Lafayette, Indiana, USA
csjgwang@purdue.edu

Mingjie Tang
Sichuan University
Chengdu, China
tangrock@gmail.com

Abstract

Selecting appropriate values for the configurable parameters of Database Management Systems (DBMS) to improve performance is an important problem in the database community. Recently, machine learning (ML)-based automatic tuning systems have shown promising results. These systems require hundreds to thousands of iterations to explore the search space, and finally identify an ideal configuration. However, with each iteration taking minutes or more to execute the target workload on the DBMS, such high tuning cost presents a core challenge for practical relevance. Existing works focus on reducing the number of required iterations (i.e., sample efficiency), but unfortunately overlook the potential to reduce the huge workload execution time (i.e., runtime efficiency).

In this work, we develop WAter, a runtime-efficient workload-adaptive tuning system that finds near-optimal configurations at *a fraction of the tuning cost* compared to state-of-the-art methods. Instead of replaying the whole workload repeatedly, we divide the tuning process into many time slices, and evaluate only a small subset of queries from the workload within each slice. Different subsets of queries are evaluated in different time slices, and we use a runtime profile to identify more representative subsets to evaluate in the following time slices on the fly. At the end of each time slice, we select the most promising configurations to be replayed on the original workload to obtain the real performance improvement. Technically, since there is no runtime profile at the beginning, we employ Large Language Models (LLM) to identify a representative subset of queries using the database domain knowledge. Next, we design a query-level metric and propose a novel Greedy Algorithm to continually refine above query subset (e.g., remove useless or add promising queries) as the tuning proceeds. Then, we develop a hybrid scoring mechanism, which is based on a global surrogate and aims to balance between exploitation and exploration, to recommend promising configurations to be evaluated on the entire workload. Finally, we evaluate WAter across different workloads and compare it with state-of-the-art approaches. It identifies the best performing configurations with up to **73.5**% less tuning time, achieving up to **16.2**% better performance than the **best-performing** alternative. We also show WAter's robustness across different hardware and optimizers, and scalability across different database sizes. We open-source our system implementation at https://anonymous.4open.science/r/WAter.

1 Introduction

Database Management Systems (DBMSs) often expose hundreds of configuration parameters (i.e., knobs) to control their system behaviors [34], and tuning these knobs is of critical importance to achieve high system performance [44]. Conventionally, these knobs are adjusted manually by database administrators (DBAs), involving extensive workload and hardware analysis. However, DBAs encounter substantial difficulties in identifying promising configurations tailored to a specific SQL workload, considering the high dimensionality of the configuration space, with each knob having continuous or discrete values. This challenge becomes even more pronounced in the cloud, where the underlying physical configurations can differ greatly between different 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 [3, 8, 10, 15, 16, 18, 46, 47, 52, 54–56]. 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 further evaluated by executing the target workload on DBMS (Figure 1). Since it is challenging to explore the high-dimensional and heterogeneous search space, numerous techniques are proposed to explore the space efficiently, such as search space pruning [18, 56] and transfer learning [3, 22, 43, 54].

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, 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 2 shows the breakdown of the tuning time of one state-of-the-art method [18] for TPC-H benchmark under different scale factors. Notably, more than 70% of tuning time is spent on executing the

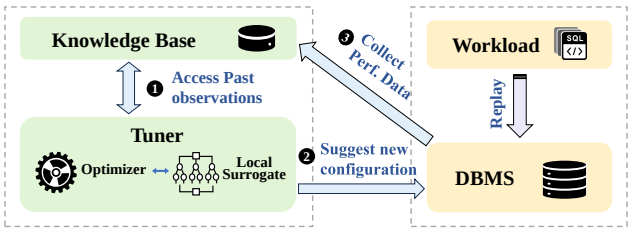


Figure 1: Overview of Knob Tuning Paradigm

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 [44].

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 to 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.

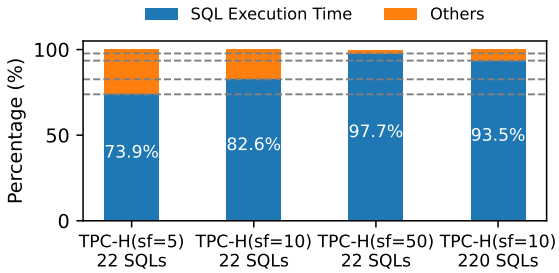


Figure 2: Tuning Time Breakdown (Percentage)

However, achieving runtime efficiency of knob tuning presents several challenges. ***C1. It is non-trivial to reduce the workload evaluation time.*** There are two possible ways to cut down the workload evaluation time: one is to decrease the volume of the target database and the other is to reduce the number of queries in the workload. Decreasing the data volume by sampling a subset of data can severely affect the performance. This is because knob tuning is very sensitive to the underlying data, reducing the data size is very likely to change the performance bottleneck and mislead the optimization. In contrast, workload compression [1, 4–6, 14, 38] presents a promising approach. It aims to identify a substitute query subset to approximate the real runtime behavior of DBMS without sacrificing the performance of workload-driven applications (e.g., index tuning) significantly [5].

Unfortunately, ***C2. It is challenging to select a representative subset of SQL, especially in the context of knob tuning.*** Although there are substantial works on workload compression [1, 5, 6, 14, 38, 51], none of them are effective for knob tuning. Most compression techniques [1, 5, 38, 51] are specially designed for index tuning and are not transferable to knob tuning. These approaches design representative metric functions and the similarity between queries based on indexing-specific query features (e.g., indexable columns), which is not usable in knob tuning. Other indexing-agnostic approaches like random sampling and GSUM [6] neglect some key factors to distinguish queries (e.g., selectivity of a predicate). More importantly, the representativity of a subset to its original workload depends on the downstream task (e.g., index selection or knob tuning), such generic compression techniques cannot capture this, and may lead to sub-optimal results.

Even if we can identify a representative SQL subset for knob tuning, ***C3: There is an obvious gap between subset tuning and whole-set tuning.*** When tuning one subset, we only evaluate the configurations on this subset. While 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 [11, 39], 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 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 insight 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, and our solution is to continually refine an imperfect subset based on runtime profile on the fly. 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. 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, and propose a novel greedy algorithm based on this metric. Since there is no runtime profile at the beginning, we employ Large Language Models (LLM) to identify a representative subset as a good starting point using database domain knowledge. Moreover, to mitigate the overhead of switching between tuning different subsets, we develop a history reuse mechanism for efficient subset tuning. 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. Finally, we conduct extensive experiments to evaluate WATER's effectiveness, robustness and scalability.

In summary, we make the following contributions:

- We introduce *runtime efficiency*, to the best of our knowledge, we are the first to expedite knob tuning by reducing workload evaluation time rather than the number of evaluations.
- We develop WATER, a *runtime-efficient* knob tuning system that reduces workload evaluation costs by only selecting small SQL subsets to evaluate at different time slices.
- We propose a novel greedy algorithm based on a carefully designed runtime *representativity* metric to continually refine the subset as the tuning proceeds.
- We develop a new history reuse mechanism to achieve efficient subset tuning, mitigating the overheads of switching between tuning different subsets.
- We propose a hybrid scoring method to prune, score and rank configurations, evaluating only the most promising configurations to achieve minimum overheads.
- We conduct experiments to evaluate WATER's effectiveness over state-of-the-art methods, robustness across different benchmarks, hardwares and optimizers, and scalability across different database sizes.

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 solution for this problem in the context of database is challenging because there are hundreds of knobs to tune in DBMS. 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 [40, 45]. Moreover, ML-based approaches are automatic and can adapt well to a variety of workloads and hardware configurations. Figure 1 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 trying to improve the pre-defined performance metrics, and *(ii)* a *DBMS instance* that runs the workload under the proposed configuration to get 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 [39] and Reinforcement Learning (RL)-based [11].

• **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) [23] is the most popular RL algorithm adopted in knob tuning [10, 22, 52], because DDPG can work over a continuous space instead of setting a knob from a finite set of predefined values.

• **BO-based.** BO-based methods [3, 8, 18, 19, 43, 54, 55] 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. *Surrogate model* is updated each time a new configuration is evaluated. (2) *acquisition function* uses the surrogate model's outputs to choose which candidate point to evaluate next, balancing between exploitation and exploration.

Performance Comparison. According to [53], 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 [53] concluded that the best performing optimizer was Sequential Model-based Algorithm Configuration (SMAC [26]), since it is efficient in modeling the heterogeneous search space. With the recent advent of Large Language Model (LLM), GPTUNER [18] 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 [5]. 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 evaluation 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 time efficiency [1, 6, 14, 38]. The primary aim of our work differs significantly from previous works. Instead of trading performance for time efficiency, we achieve better performance even when compared to tuning the original workload. This is possible since existing tuning methods suffer from the under-exploration of huge configuration space within a given time budget, and WATER tackles this by using workload compression to explore and evaluate much more promising configurations. A more detailed problem formulation is provided in Section 5.1.

There are both generic and indexing-aware workload compression techniques in the literature. GSUM [6] 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 [38] 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 [1], 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 is the first study focusing

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.

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 [24, 28] or select the most valuable training data (queries) [49] 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: Search space of knob tuning is extremely large yet under explored. 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 [3, 55] 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 [48]. However, existing ML-based tuning methods only conduct hundreds to at most thousands of samplings and evaluations [3, 15, 18, 37, 54, 56], 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, each configuration requires executing the target workload to get the performance as feedback, 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 [18] 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 less workload execution time means the tuner can try out more configurations in the same time budget, allowing the tuner to gather more observations and gain a more comprehensive understanding of the objective function. As more configurations are explored, the likelihood of discovering optimal configurations also increases.

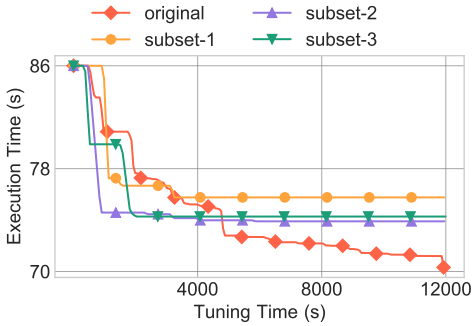


Figure 3: Tuning Subsets VS Tuning the Original Workload

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 one try, 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 [59]. So it is almost impossible to identify a perfect subset at the beginning in one try, just based on the workload information. While considering the iterative nature of knob tuning, the runtime statistics are incrementally accumulated during 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 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 one try (**M4**). Therefore, we make the following technical contributions. To handle **M3**, we propose (1) a representativity metric based on runtime profile with a greedy algorithm to compress the workload in Section 5.2, and (2) an LLM-based algorithm for cold-start scenario in Section 5.3. Based on

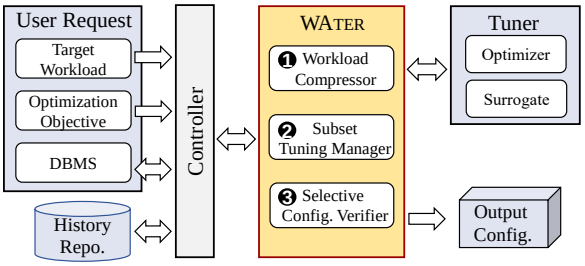


Figure 4: Overview of the Components in the WATER System

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 executing the complex original workload over and over again, we split the tuning process into many small time slices, and evaluate only a small subset of SQL queries in each time slice. A time slice is a tuning cycle, where WATER selects a representative SQL subset (Section 5), tunes the subset to get configurations (Section 6) and finally evaluates promising configurations over the original workload (Section 7). Different subsets are selected in different time slices, and the selection is based on evolving runtime profile to refine the subset continually.

Architecture. Figure 4 presents an overview of WATER's architecture. In the client side, 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 submit requests to execute queries under given configurations, collect execution information and store it in the *history repository*. WATER contains three modules corresponding to the three steps in a time slice. Firstly, the *workload compressor* uses runtime profile to select a SQL subset from the user-provided target workload, aiming to identify a subset that is representative to its original workload. Secondly, this SQL subset is regarded as the target workload in the current time slice and tuned by *Subset Tuning Manager*, reusing the tuning history to bootstrap *tuner's surrogate* to achieve efficient subset tuning. Thirdly, *Selective Configuration Verifier* prunes, ranks and selects configurations proposed when tuning the aforementioned subset. And we only verify the most promising configurations on the original workload to obtain the real performance improvement, and update the tuning history stored in History Repository for the verified configurations.

Workflow. The tuning workflow is illustrated in Figure 5. Instead of replaying the entire workload or a fixed SQL subset repeatedly, WATER divides the tuning process into many time slices, and evaluates only a small subset of queries in the workload in one time slice. The whole tuning process is made of a sequence of time slices and each time slice involves three steps. ❶ Given an input workload, WATER either uses LLM to select a representative SQL subset (only for time slice 1 since there is no runtime profile at the beginning), or greedily compresses the workload based on runtime statistics to maximize the *representativity* defined in Section 5. ❷ WATER organizes and reuses the tuning history when tuning other subsets, to bootstrap the local surrogate model of the tuner assigned for the current subset to tune, achieving efficient subset tuning (Section 6). Since we do not assume any runtime history in the first time slice, we employ Latin Hypercube Sampling (LHS) [32], a space-filling sampling strategy, to generate ten samples for initializing the surrogate, following previous works [8, 15, 18, 53]. ❸ WATER uses heuristic-based rules to prune unpromising configurations, then ranks and chooses a proportion (e.g., 30%) of configurations proposed in step ❷ of the current time slice to be verified on the entire workload (Section 7). In the first time slice, the configurations are ranked based on their performances on the subset. In the subsequent time slices, WATER maintains a global surrogate to provide the *predicted performance* and the *uncertainty* of this prediction for configurations. The outputs of the global surrogate together with the performance of configurations on the subset are then used for a hybrid scoring mechanism, balancing between exploring high-uncertainty configurations and exploiting well-performing configurations. Note that it would be exhausting to verify all proposed configurations when tuning subsets, so only Top-k configurations are verified on the original workload, and the global surrogate is continually refined based on the verified configurations.

5 Workload Compression

In this section, we first redefine the workload compression problem formally in the context of knob tuning (Section 5.1). After that, we design a runtime-calculated metric to quantify the representativity of a SQL subset to its original workload, and then propose a novel greedy algorithm based on this metric to achieve run-time workload compression (Section 5.2). Since we do not assume any runtime profile at the beginning, we propose using LLM to select a representative subset as the starting point using database domain knowledge (Section 5.3).

5.1 Problem Formulation

We formulate the conventional workload compression problem and then redefine the workload compression problem in 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 maximize the throughput or minimize the latency when executing this workload on their DBMS.

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 $m \leq n$.

Definition 5.3 (Compression Ratio). Compression Ratio, $\eta = 1 - \frac{c(W')}{c(W)}$, is the fraction of workload that has been pruned.

Next, since workload compression task is always constraint to a given budget B, we define a cost for each query as follows:

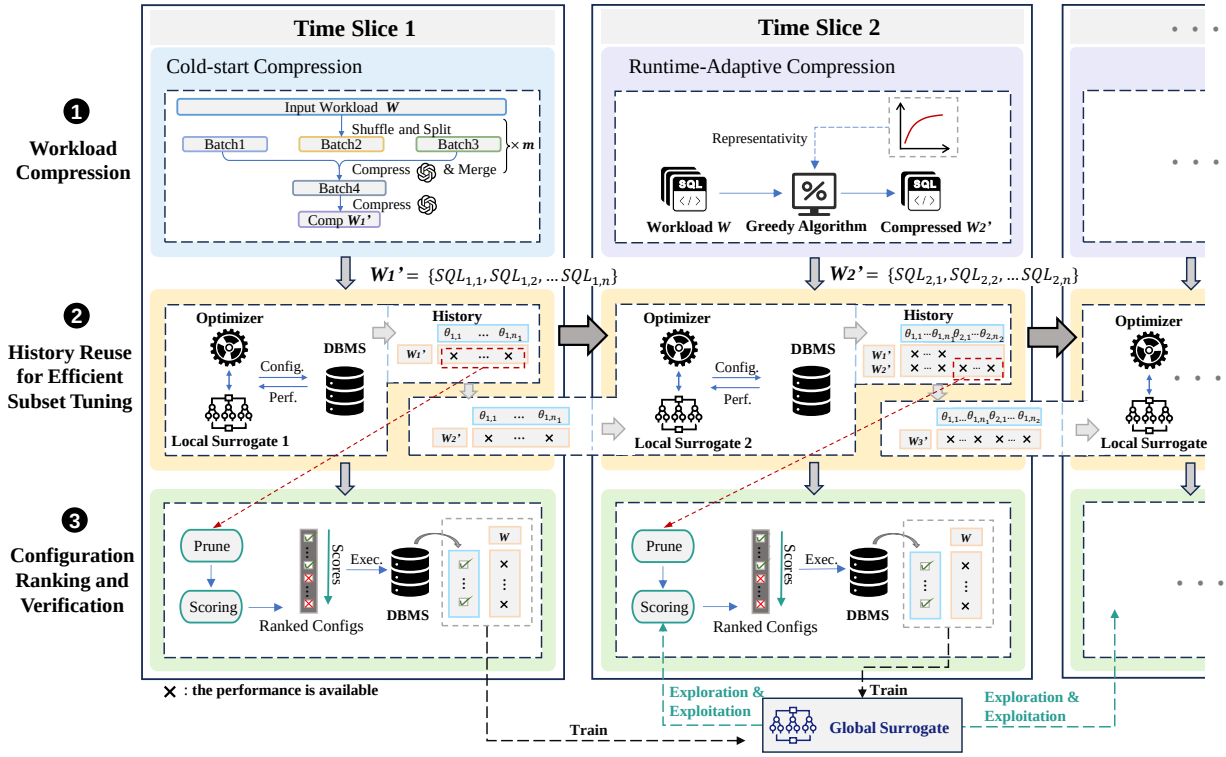


Figure 5: WATER Tuning Workflow

Definition 5.4 (Query Cost). Each SQL query q_i is associated with a non-negative cost $c(q_i)$, where $c(q_i)$ is a function to quantify the cost a query introduces to the tuner A to finish the tuning task (e.g., indexable column number for index tuning, and query execution time for knob tuning).

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

Now we can define the conventional workload compression problem formally. Given a compression budget $B \geq 0$, the goal of workload compression is to construct a compressed workload $\mathbf{W}' \subseteq \mathbf{W}$ such that [1, 5, 38]:

- $\sum_{\mathbf{q}\in\mathbf{W}'} c(\mathbf{q}) \leq B$, i.e., the cost of the compressed workload is less than the budget.
- $\mathbf{W}' = \arg\max_{\mathbf{W}'\subseteq\mathbf{W}} C(\mathbf{W}) - C_{K(\mathbf{W}',A)}(\mathbf{W})$, i.e., the decrease of the execution cost of $\mathbf{W}$ is maximized when using the configuration $K(\mathbf{W}', A)$.

As shown above, conventional workload compression aims to find a SQL subset such that tuner could finish the tuning task faster, while ensuring the tuning performance does not degrade too much at the same time. Existing works essentially trade performance for time efficiency because the performance degradation is inevitable in practice [1, 6, 14, 38]. In contrast, the primary aim of our work differs significantly from previous works. Instead of trading performance for time efficiency, we aim to achieve even better performance compared with tuning the original workload, given a time budget t. This is counter-intuitive but possible since existing tuning methods suffer from the under-exploration of huge search space within a limited time budget, and WATER tackles this by using workload compression to explore and evaluate more promising configurations within given budget. Therefore, we redefine the workload compression problem in the context of knob tuning as follows:

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

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

where $K(\mathbf{W}, A, t)$ is the configuration produced by tuner A when tuning $\mathbf{W}$ within the time budget t, and $c(q_i)$ is the i-th query's execution time under the default configuration. Like prior works, we also want workload compression to be extremely fast. Specifically, it should avoid costly operations such as query execution or computing complex statistics, otherwise we would lose the very purpose of compressing a workload in the first place [1, 5].

Note that with a sufficiently large time budget t, the performance of $K(\mathbf{W}, A, t)$ will eventually surpass that of $K(\mathbf{W}', A, t)$, if $\mathbf{W}'$ is of a fixed size. This is because $\mathbf{W}'$ is just an approximation of $\mathbf{W}$, which could lead to fast convergence initially since we can evaluate more configurations due to the reduced workload execution time, but would ultimately lead to a sub-optimal solution due to the bias between $\mathbf{W}'$ and $\mathbf{W}$, which is shown in Figure 3. To deal with this problem, instead of assuming a fixed compression ratio to work well in all scenarios, we (1) refine a SQL subset without changing the ratio, (2) after reaching the capability of a ratio, we decrease the ratio to contain more queries in a subset. Consequently, we

achieve both the efficiency of subset tuning, and the thoroughness of whole-set tuning as the tuning proceeds.

5.2 Runtime-Adaptive Compression

To obtain a representative SQL subset, we propose a runtime-calculated metric to quantify the representativity of a SQL subset to its original workload, and select queries by optimizing this metric with a greedy algorithm.

Representativity Metric Definition. We want to select a SQL subset that is representative to its original workload. That is, when varying the DBMS configurations, the performance change when executing the SQL subset should be similar to that of executing the original workload. Before calculating the 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 the performance (e.g., execution time) of each query under all evaluated configurations. $H[\mathbf{q}, \theta]$ represents the performance of $\mathbf{q}$ when executed under the configuration θ.

Example 1. An example of run history is provided in Table 1. The workload performance $H[\{q_1, q_2, \ldots, q_n\}, \theta]$ is equivalent to the sum of $H[q_1, \theta] + H[q_2, \theta] + \cdots + H[q_n, \theta]$, representing the metric when executing the workload $\{q_1, q_2, \ldots, q_n\}$ under the configuration θ.

Now, we can calculate *representativity* based on concordant performance pairs [37, 56]. Given two configurations θ_1 and θ_2 and two workloads **W** and **W'**, the performance pair is *concordant* if the sort order of $(H[\mathbf{W}, \theta_1], H[\mathbf{W}, \theta_2])$ and $(H[\mathbf{W}', \theta_1], H[\mathbf{W}', \theta_2])$ agrees, where $H[\mathbf{W}, \theta_1]$ is the performance of θ_1 on **W**.

Definition 5.6 (Representativity). Representativity of a compressed workload $\mathbf{W}'$ to its original workload **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. We may abbreviate $R(\mathbf{W}', \mathbf{W})$ as $R(\mathbf{W}')$ or R in this paper.

Example 2. Assume we have obtained the execution time of **W** as (4, 5, 7) and **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 $R(\mathbf{W}', \mathbf{W}) = \frac{2 \times (0+1+1)}{3 \times 2} = \frac{2}{3}$.

The rationale behind *representativity* is that, when tuning two workloads across different configurations, tuner focuses on relative values of the performance rather than the absolute values. The metric ranges from $[0, 1]$, with higher values indicating that the compressed workload better represents the input workload. In practice, R typically falls within $(0.5, 1]$, because random performances would lead to $R = 0.5$. When $R = 1$, two workloads are considered equivalent for knob tuning.

We formalize the compression problem as follows:

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

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

Greedy Algorithm-based SQL Subset Selection. Optimizing above set function is NP-hard [33]. 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. 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 **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

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

In other words, the algorithm greedily chooses the query with the best gain per unit of cost [6]. A $1 - \frac{1}{e}$ approximation guarantee [33] 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, the add of a new query to the current set comes with the cost of executing this query in the missing configurations. Since different queries have different costs, we need to consider this factor in our greedy algorithm. We quantify the costs of a query as follows:

Definition 5.7 (Lacked History). Given a run history H which records each query's performance under all evaluated configurations as shown in Table 1, $\#lacked\_history(\mathbf{q})$ is the number of configurations that are recorded in H but their performance on $\mathbf{q}$ is lacked.

Now we can define the final marginal gain (before normalization) as follows, trying to maximize the representativity and minimize the costs at the same time:

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

where β serves as a hyper-parameter that balances the trade-off between the overhead and the marginal gain of adding a new query. With β getting larger, we are more likely to select queries with less missing performances in the existing configurations.

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 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 **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 **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 [21] 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. Finally, the query that maximizes the marginal gain is added to **W'**, provided it does not exceed the compression ratio η (line 18, 19).

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** $c(\mathbf{W'}) + c(\boldsymbol{q}^*) \leq \eta \times c(\mathbf{W})$ **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** $\mathbf{W'}$;

5.3 LLM-based Cold-start Compression

In the first time slice, since we do not have any runtime statistics for workload compression, we rely on database domain knowledge to select a representative and diverse query subset as the starting point. We resort to LLM to solve this problem, because LLM has a great knowledge of database, and the database community has experienced a surge in the adoption of LLMs to solve various problems of DBMS, including data profiling [42, 57], code generation [12, 41] and table-based question answering [50, 58]. In the context of workload compression, a naive approach is to feed the entire workload to LLM and ask it to select a representative subset directly. However, there exist two problems: (1) Most workload texts are very long in size, exceeding the context length limit of LLM. (2) It is hard for LLM to do long-context understanding and reasoning tasks, leading to a severe performance degradation [25, 27].

To address the above problems, we propose an LLM-based batch-prompt algorithm which reduces the context size by dividing the workload into manageable batches. The algorithm is illustrated in the "cold-start compression" part of Figure 5. It takes the entire workload as input, then shuffles and splits it into several batches. For each batch $\mathbf{B}$ of queries, an LLM instance is invoked to select a representative and intra-diverse subset $\mathbf{B'}$. If we repeat the process for m times and we want to achieve an end-to-end compression ratio of η, then $\mathbf{B'}$ is subject to $|\mathbf{B'}| = \sqrt[m]{1-\eta}|\mathbf{B}|$ (i.e., the compression ratio for each batch is $\sqrt[m]{1-\eta}$). The resulting subsets of each batch are then merged to a workload smaller than the original workload. The number of queries decreases gradually and ultimately yield a compressed workload $\mathbf{W'}$ with a compression ratio of η.

We utilize SQL texts for query representation, since alternatives like execution plans [7, 31] or embeddings [30, 60] tend to vary under different configurations and are often semantically ambiguous and token-heavy. The prompt we feed to LLM is:

Prompt: `Given the SQL text of a workload, select {num} SQLs that exhibit significant diversity and are representative of the overall workload. Return the names of the selected SQLs in JSON format.`

```
WORKLOAD: {SQL1: select ...; SQL2: select ...; }
ANS FORMAT: { "selected_sqls": [] }
```

There may be rounding issue in each selection since a query is the minimum division unit while $\sqrt[m]{1-\eta}|\mathbf{B}|$ is not likely to be an integer. This rounding problem can be exacerbated through multiple compression layers, causing the end-to-end compression ratio to diverge from η. To address this, we make **B'** subject to $|\mathbf{B'}| = ceil(\sqrt[m]{1-\eta}|\mathbf{B}|)$ each time, while for the last iteration, we make the resulting **B'** of every batch B satisfy $|\mathbf{B'}| = floor(\frac{\eta|\mathbf{W}|}{b}|\mathbf{B}|)$, where b is the number of queries in all batches.

6 History Reuse for Efficient Subset Tuning

After workload compression, we get a newly-selected SQL subset which is then executed to obtain performance metric to guide the optimization. In this section, we first introduce the challenge when tuning many subsets, then we discuss how we cope with the challenge to achieve efficient subset tuning.

Challenge. In knob tuning, effective optimization heavily relies on a well-trained surrogate model that approximates the target workload's performance across different configurations within the search space. Existing works only maintain one surrogate, because they only tune a fixed workload [15, 18, 53]. In contrast, we tune different SQL subsets across different time slices. Since one surrogate could only model the performance of one workload, we need to shift to a new surrogate when switching to tune a newly-selected subset. However, bootstrapping a surrogate (i.e., use many samples to make surrogate get a general understanding of the configuration space) from scratch is very costly, because we need to execute the workload many times to get the training data. Although some transfer learning techniques have been proposed to improve the efficiency, they require collecting thousands to tens thousands of observations in advance [22, 43, 54], which is very time-consuming and is not universal (to different systems, hardware, workloads and

so on). Moreover, the transferred observations may not fit the new subset well, which could mislead the optimization.

History Reuse for Surrogate Bootstrapping. In previous time slices, we have obtained extensive execution statistics for queries from the newly-selected subset **W'** across various configurations. These statistics are recorded in the tuning history H. We aim to directly use these statistics to bootstrap the surrogate. *Case 1:* All queries from **W'** have been executed over all configurations in the history. In this case, we add these queries' execution time on the same configuration to get **W'** execution time on these configurations (Example 3). The obtained data is then used to bootstrap the surrogate for **W'**. *Case 2:* The performance data for some queries from **W'** is unavailable for some configurations. We convert this case to *Case 1* by executing every query q_i over every configuration θ_j wherever $H[q_i, \theta_j]$ is unavailable in the history (Example 4). Note although this method reduces query execution costs significantly compared to the naive approach where we bootstrap new surrogate model with randomly sampled configurations, the execution of queries under missing configurations is inevitable and this means costs. So we incorporate a penalty term in the marginal gain computation (line 11 of Algorithm 1) as discussed in Section 5.2.

In this way, we bootstrap the surrogate on the fly without costly workload execution to generate initialization data, while ensuring the data aligns precisely with the performance of the subset. Moreover, as optimization proceeds, the data continues to accumulate, resulting in better-bootstrapped surrogates for the later time slices.

Subset Tuning. After the *surrogate* has been bootstrapped, we invoke the tuner to optimize the subset. In each iteration, the subset is evaluated over a proposed configuration which tries to update the subset's best performance and the performance is sent to the tuner to guide the following optimization.

Table 1: 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 1. 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 1 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 one round of subset tuning, we have obtained a set of configurations and their performance on the subset. Our ultimate goal is to find configurations that perform well on the original workload, so we need to further verify these configurations. However, it is exhausting to verify all configurations and each verification is expensive because we need to execute the complex original workload for one verification. So in this section, we prune unpromising configurations based on heuristic-based rules, and then ranks and chooses a proportion (e.g., 30%) of configurations to be verified on the entire workload, using a hybrid scoring mechanism.

In the first time slice, since no runtime information is available, we simply discard configurations that perform worse than the default configuration on the current subset. Next, we rank the remaining configurations based on their performance on the compressed workload. Finally, the top-performing configurations are evaluated on the original workload.

Hybrid Scoring Mechanism. In the subsequent time slices, we propose a hybrid scoring mechanism to rank and select the most promising configurations. We use exploration-exploitation dilemma: when choosing a configuration, we should balance between exploration and exploitation, because relying solely on either can lead to sub-optima. For exploitation, we design a metric named *Predicated Performance*. We train a model and use the model's prediction together with the configuration performance on previous subsets to estimate a configuration's utility. As for exploration, we design another metric named *Exploration Potential*. We select configurations that diverge from evaluated configurations and for which the model has little confidence in the prediction.

• **Predicted Performance.** We maintain a global surrogate which is trained on all $(\theta, H[\mathbf{W}, \theta])$ pairs where $H[\mathbf{W}, \theta]$ is readily available in the history. The global surrogate provides two outputs for a configuration: the *predicted performance* and the *uncertainty* of the prediction. We choose random forest as the surrogate model not only because it performs best in knob configuration, but also allows us to quantify the uncertainty of a prediction [13]. The predicted performance of θ is composed of: (1) the scaled prediction generated by the global surrogate $\mathcal{RF}$, and (2) its performance $cost(\mathbf{W}')$ on the current SQL subset $\mathbf{W}'$ obtained during subset tuning:

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

• **Exploration Potential.** We also want to try some configurations that we have little knowledge about. Inspired by active learning [2, 29, 36], 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 7.1 (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) [9], 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:

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

Definition 7.2 (Uncertainty). Given an unlabeled configuration θ and a Random Forest with n estimators $\mathcal{RF} = \{rf_1, rf_2, \ldots, rf_n\}$, Uncertainty $\Psi(\theta, \mathcal{RF})$ is the variance of the estimators' predictions for θ.

We use the verification ratio of configurations α, which is also the ratio between the labeled instances and the total proposed instances, to prioritize instances between diversity and uncertainty. The exploration potential of θ is defined as:

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

Configuration Selection. In each time slice, we choose between $\hat{f}_{\mathbf{W}}$ and g as the scoring function to rank the configurations proposed during subset tuning. When selecting $\hat{f}_{\mathbf{W}}$, configurations worse than the default are pruned. For g, those performing 1.2 times worse than the default are discarded. We then select the top-scoring configurations from the remaining ones for verification.

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

Verification. To verify the selected configurations' performance on the original workload $\mathbf{W}$, we deploy them on the database and then execute the workload. Note that since the subset $\mathbf{W}'$ has already been evaluated across these configurations during the subset tuning, we only need to execute $\mathbf{W} - \mathbf{W}'$ for verification. The obtained observations are used to update the global surrogate, and the configuration with the best performance is identified as WAtER output.

8 Experiments

8.1 Experimental Setup

Workloads. We run our experiments on three well-known database benchmarks: TPC-H, TPC-DS and JOB [20]. For TPC-DS, we exclude templates 1, 4, 6, 11, 14, 23, 24, 39, 74, 81, and 95 because the queries in these templates have execution times that are orders of magnitude longer than those in other templates. For TPC-H, we offer two variants: TPC-H and TPC-H×10 which includes 10 instances generated with different random seeds per template. Table 2 summarizes the workloads we use in our experiments.

Table 2: 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

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. Since WAtER is a generic optimization framework, we integrate it with SMAC [26] and GPTuner [18]. The most recent experimental evaluation paper [53] found that SMAC performs best among the eight state-of-the-art tuners for DBMS tuning . And GPTuner is the most recent work that leverages domain knowledge to enhance the knob tuning process. We use the open-sourced code of GPTuner and update its knowledge based on the hardware. We implement SMAC with SMAC3 [26] library.
Baselines. We compare WAtER with the following baselines: **1. Original.** It directly uses the vanilla tuner (i.e., SMAC and GPTuner) to tune the entire workload. We compare WAtER with the corresponding vanilla tuner to show its superiority. **2. GSUM [6].** It is the most recent generic workload compression technique that maximizes both *coverage* and *representativity* as discussed in Section 2.2. **3. Random.** It uniformly picks random samples from the workload. Since **GSUM** and **Random** are static pre-processing techniques, we execute them at the beginning and then tune the obtained subset. Whenever a configuration outperforms the default on the subset, we immediately evaluate it over the entire workload. The compression ratio of **GSUM** and **Random** is set to be the same as the initial compression ratio of WAtER.
WAtER Implementation. We implement WAtER in Python3 on top of the two tuners. The global surrogate is implemented using the RandomForestRegressor from the scikit-learn library [35] with default hyperparameters. The compression ratio η is initially set to be 0.75 and decreases by 0.1 each time a better configuration cannot be found within a time slice. When computing the marginal gain of selecting a query, β is set to be 0.1. In each time slice, 20 legal configurations are proposed during subset tuning, and 25% (i.e., the verification ratio α) of them are evaluated on the entire workload.
Tuning Settings. We run experiments with PostgreSQL v14.9, and tune the same 57 knobs provided by GPTuner's open-source code repository [17]. We conduct three tuning sessions for each method, reporting the average of the best performances (over the entire workload) with solid line and [5%, 95%] confidence interval with shadow of the same color [15]. We take the execution time of the entire workload as the performance metric. The tuning time is sufficient for all methods to complete at least 100 tuning iterations [18]. Following iTuned [8], OtterTune [43] and GPTuner [18], the configurations of the first 10 iterations for all the methods are generated randomly using LHS [32]. As for failed configurations (i.e., those causing DBMS crash or fail to start) and long-running configurations (i.e., those lead to more than twice of the execution time), we set the results to be twice of the performance on default setting for **Original** to avoid scaling problem [44].
Evaluation Metrics. Following LlamaTune [15], we use two metrics to evaluate the performance of WAtER. One is the final performance improvement (i.e., execution time reduction). The other is the relative time-to-optimal speedup, which reports how faster (the earliest iteration at which) WAtER has found a better-performing configuration compared to the baseline optimal.

8.2 Performance Comparison

End-to-end Comparison. We build WAtER over GPTuner and compare its performance with baselines. The results for all four workloads are illustrated in Figure 6, the initial gap of the red line stems from the cold-start compression time and the subset tuning time of the first time slice.

When comparing with **Original**, we find that WAtER reaches the best performance identified by GPTuner 4.2× faster across the

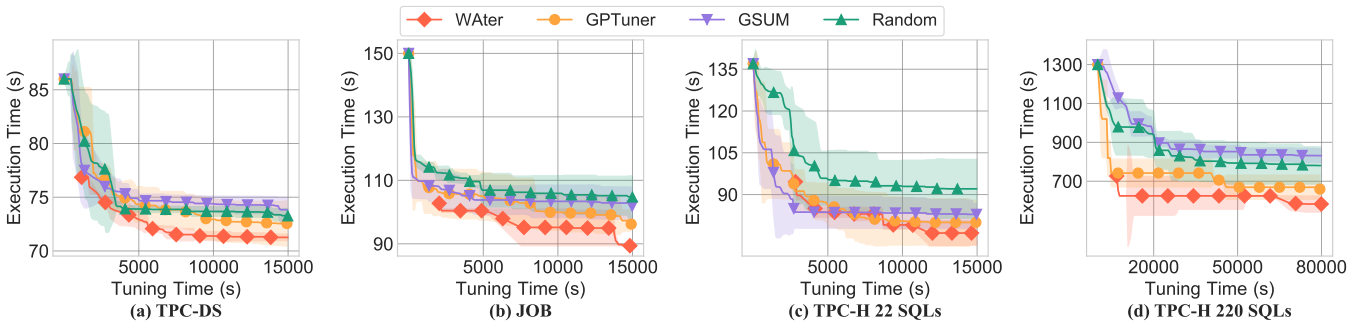


Figure 6: Performance on different benchmarks (bottom-left is better)

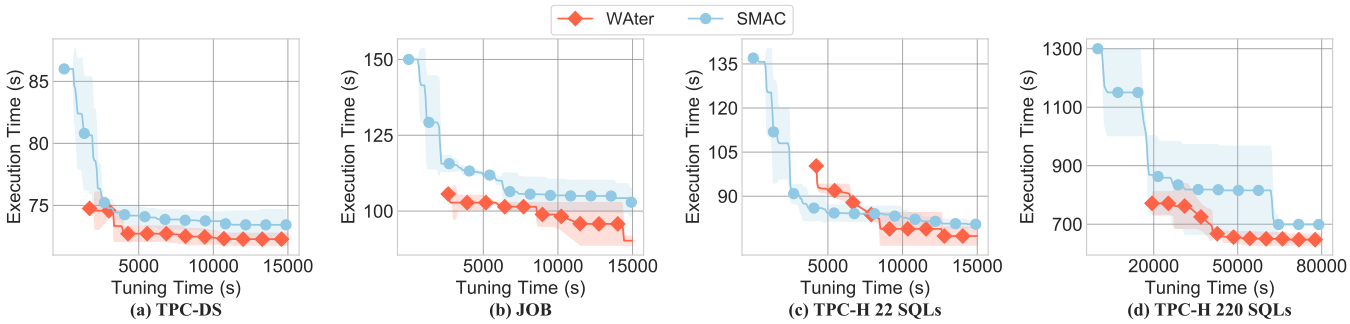


Figure 7: Performance on different benchmarks (SMAC-based) (bottom-left is better)

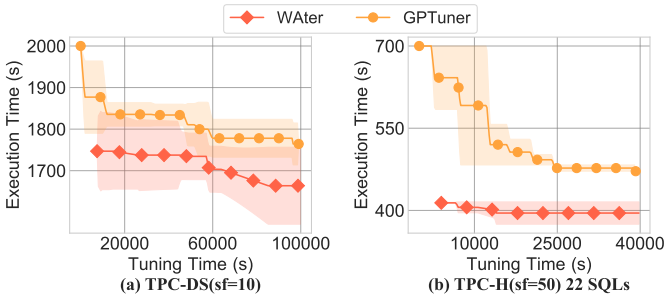


Figure 8: Performance under different scale factors

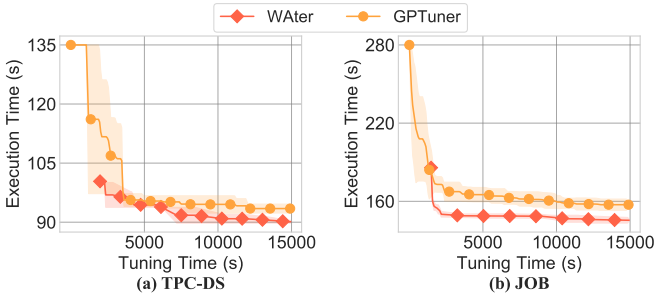


Figure 9: Performance on different machine

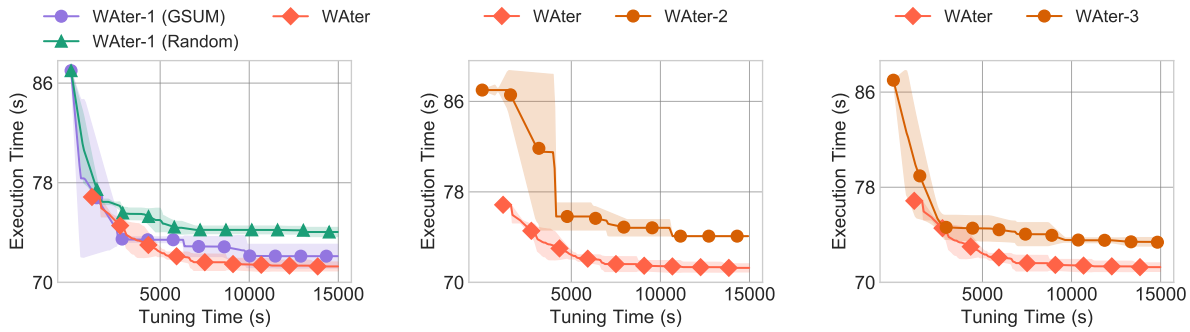
four workloads on average. Specifically, for TPC-DS, JOB and TPC-H×10 workloads, WAter yields time-to-optimal speedups of **2.5**×, **2.1**× and **11.0**× respectively, which is credited to WAter's improved runtime efficiency. As for the final results, WAter reduces the workload execution time by an average of **39.1**% compared to the default, and the achieved execution time is **6.4**% less than GPTuner's best.

We notice that WAter shows minor advantages over GPTuner on TPC-H, the reason could be twofold: (1) TPC-H only has 22 SQLs, it is too difficult to figure out a representative subset from such few queries, and (2) the execution time of TPC-H is relatively short, which may diminish the impact of runtime efficiency advantages. However, WAter surpasses GPTuner in the later stages, likely due to the continuous subset refinement (otherwise, the optimization would stagnate like **GSUM** and **Random**).

As for **GSUM** and **Random**, although they can find interesting configurations at the early stage, their optimization stagnates later, and finally they fail to find better configurations than GPTuner (on average). Moreover, although **GSUM** is proposed as a generic workload compression framework, it does not necessarily outperform random sampling in the context of knob tuning, because it only takes some features of each query into consideration, which may not reflect the critical bottlenecks to improve the performance using knob tuning and those features may mislead the query selection process. Also, **Random** generally leads to wider shadows in the figure, especially for workloads with fewer SQLs (e.g., TPC-H), which reflects the instability of random sampling.

8.3 Robustness Study

Different Optimizer. To showcase the applicability of WAter to a different optimizer, we change the underlying optimizer of WAter to SMAC . As shown in Figure 7, we get similar results to that on GPTuner. WAter again outperforms the vanilla SMAC on all four workloads. It achieves a mean reduction of the execution time by **37.5**% over the default (and **6.6**% less execution time than SMAC's best performing configurations) and a time-to-optimal speedup



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

Figure 10: Ablation study of WAter on TPC-DS (bottom-left is better)

of 3.1× on average. In particular, we have 3.8× and 5.3× time-to-optimal speedup on TPC-DS and JOB, and 15.9% and 40.0% less execution time on TPC-DS and JOB. Also, we find that TPC-H still remains a challenge for WAter. However, although WAter fails to find better configurations in the early stage, it again surpasses the vanilla optimizer later on, with the evolving subset.

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 shows our superiority on expensive-to-evaluate workloads over vanilla tuner. A detailed analysis is provided in Section 8.5.

Different Hardware. We change the hardware from C1 to C2 which possesses much fewer CPU cores and less RAM. In this setting, the optimization should be more difficult because the complexity of modeling the relationship between configurations and DBMS performance increases with reduced hardware resources, since more performance bottlenecks are revealed and the feasible region shrinks [18]. We exclude the experiments on TPC-H because it frequently lead to system crash on C2. The results are shown in Figure 9, compared with GPTuner, WAter finds better configurations in much fewer iterations on both of the workloads. For JOB, WAter identifies a configuration superior to GPTuner's best on the first attempt (7.9× speedup) and finally achieves a workload execution time 7.6% less than GPTuner's best. For TPC-DS, we have 1.9× time-to-optimal speedup and 3.6% less execution time.

8.4 Ablation Study

Effect of Adaptive Workload Compression. To verify the effectiveness of our adaptive workload compression framework and the workload compression algorithm in Section 5, we make the subset invariable across all time slices. We adapt the same subsets used by **GSUM** and **Random** correspondingly, denoted as "WAter-1 (GSUM)" and "WAter-1 (Random)". As shown in Figure 10(a), WAter significantly outperforms WAter-1 (GSUM) and WAter-1 (Random) in both time-to-optimal speedup (2.0× and 4.3×) and execution time reduction (1.1% and 3.6% less execution time than WAter-1 (GSUM) and WAter-1 (Random) respectively).

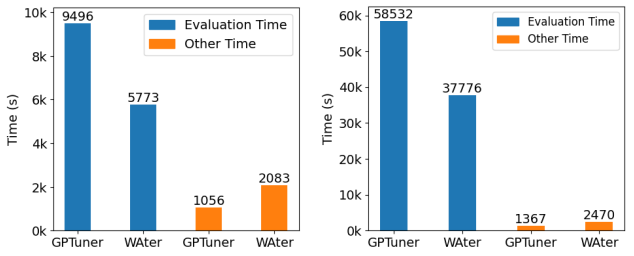
Effect of History Reuse. To prove the effectiveness of history reuse for efficient subset tuning in Section 6, we use LHS [32] to randomly sample configurations and evaluate them to bootstrap the surrogate in each time slice, which is denoted as "WAter-2". As shown in Figure 10(b), WAter-2 stagnates the optimization at an early stage, and WAter finally reduces workload execution time by an additional 3.8% compared to WAter-2 and achieves a 3.5× speedup. This is due to WAter-2's additional overhead of initialization, and the surrogates are significantly undertrained with only a limited number of observations.

Effect of Hybrid Scoring Mechanism. To demonstrate the effectiveness of the hybrid scoring mechanism in Section 7, we replace the scoring mechanism to just use the selected subset's performance, which is denoted as "WAter-3". As shown in Figure 10(c), WAter significantly outperforms WAter-3 in both time-to-optimal speedup (4.1×) and execution time reduction (3.1% less than WAter-3). This is because although WAter-3 can propose configurations as good as WAterduring subset tuning, it fails to identify effective configurations for evaluation on 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, representing the total duration spent executing queries during the tuning process, and (2) Other Time, covering all remaining time not accounted for in Evaluation Time. This mainly includes time for data collection, generating the next configuration, deploying the configuration, and executing any other WAter's algorithms.

Figure 11 demonstrates the time spent in the two categories throughout the tuning sessions for TPC-H (sf=10) and TPC-H (sf=50) using WAter and GPTuner. For both of the methods, we perform 100 iterations. WAter reduces the overall tuning time by 25.5% and 32.8% on TPC-H (sf=10) and TPC-H (sf=50) to try 100 configurations compared with GPTuner, primarily due to the reduced "Evaluation Time". We find that while WAter incurs more "Other Time" than GPTuner due to the added overhead (e.g., model training and more configuration deployments), the significant reduction in 'Evaluation Time' more than compensates for this increase. Moreover, this advantage becomes even more pronounced



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

Figure 11: Cost Analysis

when tuning larger workloads. As we can see, when switching from TPC-H (sf=10) to TPC-H (sf=50), WAter reduces the absolute value of "Evaluation Time" much more, while "Other Time" is much less sensitive to the volume of the target workload. So, WAter is better suited for tuning large workloads. For smaller workloads, WAter may actually degrade performance, as the reduction in evaluation time may not compensate for the overhead introduced and the potential performance loss from tuning with a subset.

9 Conclusion

We introduce WAter, a workload-adaptive knob tuning system. Instead of replaying the entire workload repeatedly, WAter divides the tuning process into many time slices and tune different SQL subsets based on runtime profile to reduce tuning costs. For the proposed configurations, WAter selects the most promising configurations to evaluate their performance on the original workload. Extensive experiments demonstrate that WAter performs better than state-of-the-art approaches and is robust across different hardware, optimizers and data sizes.

References

[1] 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. https://doi.org/10.1145/3639305

[2] 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. https://doi.org/10.1016/j.ins.2016.10.037

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

[4] 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.

[5] 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. https://doi.org/10.1145/564691.564747

[6] 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. https://doi.org/10.14778/3430915.3430931

[7] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1241–1258. https://doi.org/10.1145/3299869.3324957

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

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

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

[11] 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.

[12] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. *arXiv preprint arXiv:2406.08426* (2024).

[13] 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. https://doi.org/10.1007/978-3-642-25566-3_40

[14] 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

[15] 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

[16] 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. https://doi.org/10.1145/3318464.3380591

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

[18] 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. https://doi.org/10.14778/3659437.3659449

[19] 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. https://doi.org/10.1145/3626246.3654739

[20] 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. https://doi.org/10.14778/2850583.2850594

[21] 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. https://doi.org/10.1145/1281192.1281239

[22] 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. https://doi.org/10.14778/3352063.3352129

[23] 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).

[24] 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. https://doi.org/10.14778/3681954.3682030

[25] Jianzhe Lin, Maurice Diesendruck, Liang Du, and Robin Abraham. 2024. BatchPrompt: Accomplish more with less. arXiv:2309.00384 [cs.CL] https://arxiv.org/abs/2309.00384

[26] 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

[27] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. arXiv:2307.03172 [cs.CL] https://arxiv.org/abs/2307.03172

[28] 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. https://doi.org/10.1145/3318464.3389768

[29] 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. https://doi.org/10.1145/3318464.3389768

[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.

[31] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Houston, TX, USA) *(aiDM'18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3211954.3211957

[32] 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. https://doi.org/10.1145/167293.167637

[33] 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. https://doi.org/10.1007/BF01588971

[34] 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.

[35] 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.

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

[37] 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. https://doi.org/10.1145/3580305.3599953

[38] 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. https://doi.org/10.1145/3514221.3526152

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

[40] 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. https://doi.org/10.1145/1012888.1005739

[41] Immanuel Trummer. 2022. CodexDB: synthesizing code for query processing from natural language instructions using GPT-3 codex. *Proc. VLDB Endow.* 15, 11 (July 2022), 2921–2928. https://doi.org/10.14778/3551793.3551841

[42] Immanuel Trummer. 2023. Can Large Language Models Predict Data Correlations from Column Names? *Proceedings of the VLDB Endowment* 16, 13 (2023), 4310–4323.

[43] 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. https://doi.org/10.1145/3035918.3064029

[44] 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. https://doi.org/10.14778/3450980.3450992

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

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

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

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

[49] 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. https://doi.org/10.1145/3639293

[50] Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 174–184.

[51] 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. https://doi.org/10.14778/3654621.3654622

[52] 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. https://doi.org/10.1145/3299869.3300085

[53] 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. https://doi.org/10.14778/3538598.3538604

[54] 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. https://doi.org/10.1145/3448016.3457291

[55] 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. https://doi.org/10.1145/3514221.3526176

[56] 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. https://doi.org/10.14778/3632093.3632114

[57] Yunjia Zhang, Avrilia Floratou, Joyce Cahoon, Subru Krishnan, Andreas C Müller, Dalitso Banda, Fotis Psallidas, and Jignesh M Patel. 2023. Schema matching using pre-trained language models. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1558–1571.

[58] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M Patel. 2023. ReAcTable: Enhancing ReAct for Table Question Answering. *arXiv preprint arXiv:2310.00815* (2023).

[59] 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. https://doi.org/10.1109/TKDE.2023.3266893

[60] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a tree transformer model for query plan representation. *Proc. VLDB Endow.* 15, 8 (April 2022), 1658–1670. https://doi.org/10.14778/3529337.3529349