Functionality-Aware Database Tuning via Multi-Task Learning

Zh Peng Cai[†*], Xuan Zhou[†], Huiqi Hu[†], Rong Zhang[†], Quanqing Xu[§], Chuanhui Yang[§]
[†]*East China Normal University,* [§]*OceanBase, Ant Group*
05903033}@stu.ecnu.edu.cn, {pcai, xzhou, hqhu, rzhang}@dase.ecnu.edu.cn,
{xuquanqing.xqq, rizhao.ych}@oceanbase.com

- knob tuning: process of adjusting configuration parameters to optimize performance of the database
- challengses: modern DBs have hundreds of knobs, tuning one knob might improve performance for one query but degrade others. Not to mention some knobs interact with others, making it hard to predict the effect of changing one without understanding how others work.

***Abstract*—Functionalities of a database system are co-designed and jointly maintain the database performance. Each functionality usually has its own metrics to evaluate its state. Previous knobs tuning methods regard the database system as a black box and aim to automatically find the optimal configurations by collecting and observing the overall performance data (e.g., transaction throughput per second) under various configuration knobs. However, if a functionality is not running in the tuning phase, its knobs *irrelevant* to performance changes can also be tuned by existing tools and potential risks would be introduced. To resolve this problem, we design a database knob tuning framework to support functionality-aware knobs tuning. It uses multi-task learning to take the database overall performance as the objective of main learning task, and each function module as a separate learning task. This framework enhances the tuning results through learning the relationships between different tasks, and avoids adjusting irrelevant knobs by perceiving the status of functionalities. We validate its generalizability on OceanBase and PostgreSQL. Experimental results show that better performances were achieved on the overall performance and the metrics of various functionalities.**

***Index Terms*—knob tuning, database system, multi-task learning, Gaussian process**

I. Introduction

To achieve the best running performance under various workloads and deployment environments, modern database systems allow the users to tune knobs (or parameters) of their functionalities. Finding a set of optimal knob settings requires the DBA to have a deep understanding of the deployed database systems. However, manually tuning these knobs is unscalable, especially for cloud service providers which support a huge number of database instances [1]. Therefore, how to automatically tune database parameters has received much attention [2]–[8].

The workflow of most automatic database tuning systems can be summarized as: First, replay the workload to obtain overall performance (e.g., transaction throughput and latency) or resource occupancy (e.g., CPU and I/O utilization) of the database and judge the quality of the configuration knobs according to the collected performance data. Second, calculate the next set of promising knobs and re-run the workload by applying these knobs to the database. The two steps are repeated until some stop criteria are satisfied (e.g., the allocated tuning time has expired).

In this work, we argue that the change in the overall performance of the tuned database system may mislead the configuration tuning. Modern distributed DBMSes have multiple functionalities such as SQL optimization, load balancing, data compaction, etc. Each functionality has its own knobs and it can be triggered under various conditions. Even under the same knob settings and workloads, the transaction throughput and mean latency may fluctuate as different functionalities are triggered [9]. It will happen that the knobs of a specific functionality are tuned even if this functionality does not work during the phase of automatic parameter tuning. Actually, the adjustment on these knobs of not-running functionality is unrelated to the changes in the overall performance. This may introduce an unknown effect if this functionality starts to work using these *unverified* knobs values. In the following, we present a concrete example for this problem occurred in real-world scenarios.

OceanBase [10], [11] adopted the well-known LSM-based storage engine. Its parameter **major_compact_trigger**[1] of the major compaction functionality is to control how many minor compactions are required to trigger a major compaction. This parameter is to make a trade-off between read amplification and write amplification. Previous tuning tools such as OtterTune [3] could adjust the **major_compact_trigger**'s value to 53000 although the major compaction functionality is not triggered in the tuning process. In production scenarios, the large value of this knob can lead to slow read as large minor compaction files need to be checked for read queries. However, this unreasonable setting may not affect the performance of the tuned DBMS because only few minor compaction operations are conducted during the short iteration. We design and conduct systematic experiments to illustrate the problem of incorrect knobs tuning.

We first define two kinds of knobs to facilitate a better understanding of the experiment:

- `fake knob`: To evaluate whether the parameter auto-tuning tools randomly tune the knobs irrelevant to the

An LSM-based storage engine (Log-Structured Merge-tree) is a type of database storage engine that is designed to optimize write-heavy workloads. It efficiently manages large volumes of data writes while maintaining read performance

In LSM based storage engines like OceanBase, data is written in batches through a process called minor compaction, which merges small, recent data chunks into larger files. Over time, many such files accumulate, leading to slow reads. To fix this, a major compaction is triggered after a set of minor compactions to combine multiple files into one

[1]major_compact_trigger specifies how many minor compactions are required to trigger a major compaction. When the value is 0, minor compactions are disabled.

* Corresponding Author

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00014

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00014

fake knobs are introduced to test whether the auto-tuning tools randomly adjust settings that are irrelevant to database performance

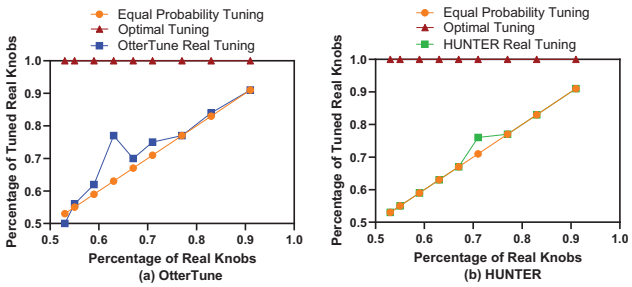

Fig. 1: Fake Knobs Experiments on OtterTune [3] and HUNTER [7]

overall performance changes, we introduce extra configuration knobs to the list of tuned knobs, referred to as fake knobs. The tuning of these fake knobs is just to modify their values which have no effect on the database performance. There is no correlation between database performance and the tuning of these fake knobs. We use such a setup to simulate those parameters that are currently not relevant to the database performance.

- `real knob:` Real knobs are configuration knobs that really exist in the database. The modification of real knobs can change the performance of tuned database systems.

Experiments were conducted on OceanBase Community Edition 3.1.1. For all experiments, we fixed the same 20 real knobs to tune. They are bound to affect on the overall performance. The number of fake knobs in each experiment varied. After each tuning experiment was finished, we calculated how many fake knobs were tuned.

As shown in Figure 1, we use the percentage of real knobs in all knobs (i.e., fake and 20 real knobs in the tuning list) as the horizontal coordinate. The vertical coordinate is the percentage of *tuned* real knobs in all *tuned* knobs. Experimental results are plotted as a line passing filled rectangles. The optimal bound is plotted as a line passing filled triangles, labeled as *Optimal Tuning*. The optimal means only real knobs were tuned in each tuning task. The line passing filled circles (labeled as *Equal Probability Tuning*) demonstrates the case that the real and fake knobs have an equal probability to be tuned. This means fake and real knobs can not be discriminated according to whether they have impact on the DBMS performance. For example, the tuning list includes 30 knobs (i.e., 20 real knobs and 10 fake knobs). After the tuning task is finished, 9 knobs are tuned, where 6 real knobs and 3 fake knobs are tuned. The area above *E*

20 real and 10 fake -> 2:1 ratio -> if 9 are tuned then 6 real and 3 fake

specifies more choice of the real knob being tuned, while the area below *Equal Probability Tuning* specifies more choice of the fake knob being tuned.

Figure 1 (a) presents the results on OtterTune [3]. The curve of OtterTune is far from the *Optimal Tuning* and close to *Equal Probability Tuning* most of the time. This indicates that OtterTune has tuned the fake knobs and it can not find that the changes in fake knobs have nothing to do with the overall performance. Figure 1 (b) shows the results on HUNTER [7]. It can be seen that HUNTER basically overlaps with *Equal Probability Tuning*. It tunes all the knobs in the tuning list, no matter whether they are real or fake knobs.

Although the user would not inject the fake knobs into the tuning list in real applications, it may happen that some important knobs are tuned to invalid or meaningless values. These badly tuned knobs belong to function components that are not in the running state during the tuning task. Meanwhile, these badly tuned knobs have no chance to be fixed as their function components still do not work in the later tuning iterations. The reason can be attributed to short iteration (e.g., 3∼5 minutes) or the functionality not being triggered because the triggering condition is not satisfied.

To resolve the above problems, we introduce OBTune, a functionality-aware database tuning system designed for OceanBase [10], which operates as an offline tuning system. Our intuition is that database functionalities are usually co-designed and jointly maintain their overall performance (see the definition in Section III). *Each functionality has its specific knobs and related metrics to evaluate its health state.* Multiple functionalities can not work well together by only using the overall performance (e.g., transaction throughput and latency) to tune their knobs. Thus, OBTune adopts the concept of multi-task learning [12] to break down the tuning task of a database instance into functionality tuning and overall performance tuning. The knob tuning of each function component is used as the auxiliary task to help the main task of tuning the overall database performance.

To the best of our knowledge, this is the first paper to co-tune the overall performance and various database functionalities. Our contributions are summarized as follows.

- We demonstrate the problem that black box based auto-tuning methods tune knobs unrelated to the changes in the overall database performance. This would introduce potential risks of applying unreasonable parameters to the online database instances.
- We propose a multi-task learning based auto-tuning framework. It allows the DBA to tune the specific database functionality during tuning the overall performance. The knob auto-tuning of a functionality is regarded as an auxiliary learning task, and the main task is to tune the overall performance.
- We implement the functionality-aware auto-tuning method (i.e., OBTune) and evaluate it on the distributed database system OceanBase and the centralized database system PostgreSQL. Experimental results indicate that OBTune can tune the knobs of triggered function components effectively and accurately.

Section VII concludes the paper.

II. RELATED WORK

The challenge of knob tuning in databases has prompted a multitude of solutions. These solutions can be classified into four categories based on the tuning methods employed: rule-based tuning, search-based tuning, Gaussian process-based tuning, and reinforcement learning-based tuning.

Rule-based Tuning. MySQLTuner[2] and PGTune[3] harness database commands to gather pertinent information for tuning, then adjust corresponding knobs according to predefined rules. This approach, however, often restricts tuning to specific subsets of knobs. For example, MySQLTuner and PGTune are limited to tuning only 12 and 14 out of hundreds of MySQL and PostgreSQL knobs, respectively.

Search-based Tuning. BestConfig [2] is a heuristic tuning method designed to tune database knobs. It is executed in three key steps. First, the Latin Hypercube Sampling (LHS) method [13] is employed to sample within the knob space. Next, the surrounding region of the best-performing sample is identified as the new sampling space, where the LHS is conducted again. Finally, if a sample within this new space shows improved performance, the process returns to the second step; if not, it reverts to the first step. What sets BestConfig apart is its highly randomized sampling process, which may result in significant variations in the final outcomes for the same scenario.

Gaussian Process-based Tuning. iTuned [4] pioneered Gaussian process-based modeling the relationship between knobs and database performance. By leveraging the sampling data gathered through the LHS method, it constructs a preliminary tuning model and employs the expected improvement method to balance exploration and exploitation. Conversely, OtterTune [3] builds an initial Gaussian model using historical tuning data, and applies Lasso to identify key knobs. It then utilizes an incremental approach [14] to dynamically increase the number of tuning knobs throughout the process. ResTune [8] also employs a Gaussian process, but aims to minimize system resource utilization while maintaining DBMS performance. Taking into account the influence of various factors such as operating system and Java virtual machine on database performance, CGPTuner [15] employs a Contextual Gaussian Process Bandit Optimization to tune knob of the entire IT stack of the database for performance maximization. In contrast to these offline tuning strategies, ONLINETUNE [1] offers an online approach, using contextual Bayesian optimization for adaptive database tuning in ever-changing cloud environments. The widely utilization of Gaussian processes [16] in database knob tuning is attributed to their theoretical capability to balance exploration and exploitation.

Reinforcement Learning-based Tuning. Unlike OtterTune, which segments the tuning process into various phases and relays the optimal solution from one stage to the next, CDBTune [5] introduces a more unified, end-to-end solution. Utilizing reinforcement learning for database knob tuning, CDBTune employs DDPG [17] as an agent, with the knob value as the action, the database as the environment, the internal state of the database as the state, and changes in database performance as the reward. On the other hand, QTune [6] also incorporates reinforcement learning but with a unique perspective, considering query statements during model training. It allows for multi-level tuning, including query-level, workload-level, and cluster-level adjustments. Typically, trained models struggle to adapt to new tuning scenarios, necessitating a cold start. Addressing this cold start problem, HUNTER [7] proposes a solution that integrates genetic algorithms with reinforcement learning. This method minimizes model training time by independently performing various configurations on multiple replicated database instances.

At present, there are some other studies related to database knob tuning. LlamaTune [18] focuses on enhancing the sampling efficiency of existing optimizers. Studies such as [19] and [20] propose methods for extracting tuning rules from text data. ReIM [21] adopts an empirically-driven white-box method to tune the memory resource allocation in Spark [22]–[24]. [25] utilizes the Plackett-Burman experimental design approach [26] to rank database knobs by importance. [27] concludes, based on relevant experiments, that database knob tuning often requires adjusting only a few knobs. However, it does not provide guidance on how to rapidly identify these crucial knobs in a specific scenario. [28] performs a detailed experimental comparison of relevant tuning tools (e.g., OtterTune and CDBTune) in a real scenario. In a study outlined in [29], the three key aspects of database knob tuning (knob selection, configuration optimization, and knowledge transferring) are experimentally compared, with SHAP [30], SMAC [31], and RGPE [32] identified as the best algorithms for these aspects, respectively.

III. Overview Of OBTune

Domain Knowledge. Shallow domain knowledge such as the knob's value range and how to split the value range for efficient sampling has been used in auto knob tuning [18], [33]. In this work, OBTune is designed to integrate deep domain knowledge. Our motivation is to help DBAs ***optionally*** contribute domain knowledge regarding database functionalities. While OBTune can achieve commendable tuning results without this input, the inclusion of DBA insights enables even better outcomes. Importantly, OBTune uses domain knowledge to selectively tune knobs related to active functionalities, enhancing tuning effectiveness and reducing potential risks. This methodology caters to varying DBA expertise levels, maximizing the use of available knowledge.

Knob Classification. Knobs in OBTune are primarily classified into two categories: functionality knobs and main knobs. Specifically, if the adjustment of knob x directly affects the performance of triggering functionality A, and the change of knob x has no impact on database performance if functionality A is not triggered, knob x is classified to the knobs of functionality A. If multiple knobs simultaneously affect the performance of multiple functionalities, OBTune's strategy is to classify such knobs as main knobs. Furthermore, if a tuning knob does not belong to any specific functionalities, it also is classified to the main knob.

- A functionality knob directly affects the performance of a specific function when it is active. But has not effect when the function is not active
- If a knob affects multiple functions at once, or does not belong to any particular function, it is a main knob

Definition 1. Direct metrics, *that are used to directly reflect the performance of a database functionality. For instance, the direct metric for load balancing functionality in OceanBase*

[2]https://github.com/major/MySQLTuner-perl
[3]https://pgtune.leopard.in.ua/

TABLE I: An example of domain knowledge for the garbage collection functionality in PostgreSQL.

Garbage Collection	Domain Knowledge
Trigger Indicators	Check if the value of "select sum(autovacuum_count) from pg_stat_user_tables" has increased.
Direct Metrics	Find the information resembling "system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.37 s" in the log, where the "0.37" is the metric, and the optimization for this metric is the smaller, the better.
Corresponding Knobs	autovacuum_max_workers, autovacuum_work_mem, autovacuum_vacuum_threshold, autovacuun_naptime, autovacuum_vacuum_scale_factor, autovacuum_vacuum_cost_limit, autovacuum_vacuum_cost_delay, vacuum_cost_page_hit, vacuum_cost_page_miss, vacuum_cost_page_dirty.

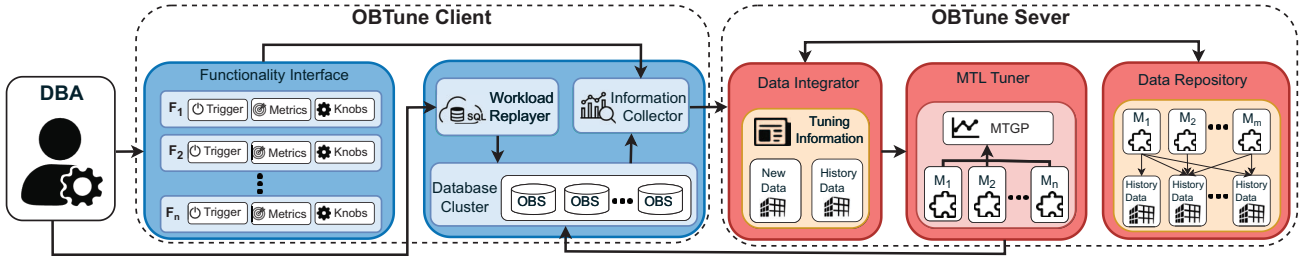

Fig. 2: Architecture of OBTune

is the time used for migrating data partitions from high-load machines to low-load machines.

Definition 2. Indirect metrics, *that are used to represent the overall database performance such as transaction throughput, latency, and total SQL execution time on benchmarks like TPC-H. These metrics denote the feedback on the system's overall end-to-end performance, and they are not bound to any specific database functionality. Thus, these metrics are referred to as "indirect".*

Definition 3. Overall performance, has the same definition as the indirect metrics. The two terms are used interchangeably in this article.

Architecture. Figure 2 presents the architecture of OBTune, which is divided into two parts: the OBTune Client and OBTune Server. The OBTune Client, situated in the user's environment, includes the Information Collector, the Workload Replayer, and the Functionality Interface. Its primary function is to gather database performance metrics in real time, tailored to the DBA's tuning objectives, and to transmit this data to the OBTune Server. The OBTune Server, operating in the tuning environment, consists of the Data Integrator, the Data Repository, and the MTL Tuner—a tuner based on multi-task learning. The Server, upon receiving data from the Client, merges it with relevant historical data from the Data Repository. This integrated data is then analyzed to determine the most effective knob settings, which are sent back to the OBTune Client. The Client applies these settings to the database, re-executes the workload under the new configuration, and initiates another tuning cycle.

Workload Replayer. Upon receiving a tuning signal, the OBTune Client initiates the Workload Replayer to simulate the database instance's activity. This simulation is aligned with the DBAs' specified benchmarks, such as TPC-C or TPC-H, or actual user-provided business workloads. To accurately assess a modern, complex DBMS under genuine business conditions, it's crucial to engage the system with extensive, authentic workloads. This approach enables the activation of a wide array of database function components, ensuring a comprehensive performance evaluation.

Functionality Interface. OBTune features a specialized Functionality Interface to ensure comprehensive integration of domain knowledge about database functionalities. This interface mandates that database administrators input three key details for each functionality: trigger indicators, direct metrics, and its corresponding knobs. The trigger indicators help identify whether a functionality is active. The corresponding knobs are those parameters that are directly associated with the specified functionality. Direct metrics describe which metrics reflect the performance of this functionality and how to retrieve and optimize these metrics. If DBA wants to co-tune the overall performance and some specific functionalities, OBTune requires DBA to input domain knowledge from three aspects. Table I presents an example of required domain knowledge for garbage collection functionality in PostgreSQL.

Information Collector. The Information Collector's primary role is to gather critical data for the ongoing tuning process. During the tuning process, it not only collects essential information relevant to the overall performance, but also gathers information relevant to the domain knowledge provided by the DBA. For functionalities that are triggered, the Information Collector records the name of the functionality, its direct metric values, and its corresponding knobs. Conversely, it does not collect such information for functionalities that are not triggered. This selective data collection approach helps to ensure that the tuning process does not adjust the corresponding knobs of non-triggered functionalities, thus mitigating potential risks caused by altering these knobs.

Data Repository. The primary function of the Data Repository is to securely store data gathered by the Information Collector. This data is organized into different categories based on the triggering functionalities during tuning. Throughout the tuning process, the data repository provides historical data corresponding to the same functionalities triggered in the current iteration to the Data Integrator.

Data Integrator. Upon receiving new data from the Information Collector, the OBTune Server's Data Integrator combines it with historical data provided by the Data Repository. This

integrated dataset serves as the input for the MTL Tuner in the current tuning iteration.

MTL Tuner. The MTL Tuner adjusts knob settings using data provided by the Data Integrator. During this process, it treats functionalities activated by the current workload as auxiliary tuning tasks, while prioritizing the database's overall performance as the primary tuning task. *Notably, the MTL Tuner can be changed to a single-task tuner straightforwardly if the DBA has not input any domain knowledge and no specific database functionalities are co-tuned.*

IV. MTL FRAMEWORK IN OBTUNE

To prevent potential risks stemming from adjusting corresponding knobs of non-triggered functionalities, OBTune is designed to tune knobs according to the triggered functionality. As the database has multiple triggered functionalities, multi-task learning (MTL) can well match our needs by regarding the tuning of each functionality as a learning task and the overall performance tuning as the main task. OBTune uses the multi-task Gaussian Process (MTGP) [34] to learn task relationships.

A. MTL For Tuning

At present, there are two most commonly used multi-task learning methods. One is hard parameter sharing [35]: the same parameters are shared between different tasks. The other is soft parameter sharing [36]: different tasks have different parameters and mine their potential common data characteristics. We use a soft parameter sharing approach to treat each model as a task. A model is trained for each task with its own parameters. Each task can have a cross-task conversation with other tasks. The soft sharing mechanism is very flexible and does not require any assumptions about the dependencies between tasks.

To ensure the effectiveness of learning multiple tasks simultaneously, MTL usually adopts two techniques. (1) Enhancing the sparsity between tasks through canonical regularization, which means increasing the "distance" between tasks so that each task learns more specific and independent knowledge, thereby reducing interference between tasks. (2) Modeling the task relationship to understand the degree of inter-task relevance. These two approaches are implemented to ensure that MTL can effectively share information across multiple tasks. In functionality-aware knob tuning, the sparsity between tasks is guaranteed as each tuning task has its own parameters and metrics. Therefore, our main focus is on how to model the relationships between tasks [37]. We introduce the Multi-task Gaussian Process (MTGP) in Section V to learn the task dependencies. MTGP requires minimal training data and offers flexibility in modeling inter-task relationships.

B. Workflow For OBTune

In Figure 3, we illustrate the scenario that how to map OBTune's tuning process to multi-task learning.

① In addition to the main knobs, users specify functionality A and functionality B and their respective functionality knobs, namely the A knobs and B knobs.

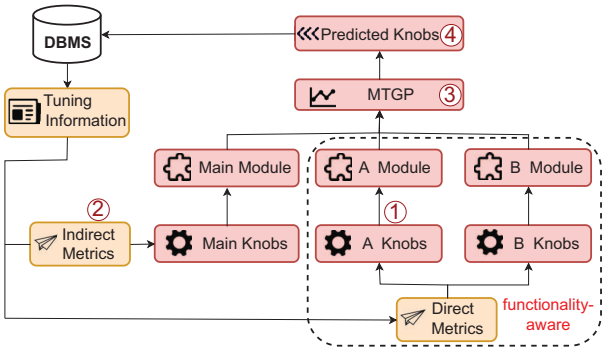

Fig. 3: Multi-Task Learning Framework in OBTune (dashed-line rectangles represent the functionality domain knowledge.)

② The main module in OBTune is created by integrating main knobs with the indirect metrics. Similarly, each functionality module in OBTune is constructed by combining its functionality knobs with its direct metrics. This approach guarantees that each module possesses distinct knobs and tailored optimization objectives, thereby ensuring sparsity between the subsequent tasks.

③ The multi-task Gaussian process in OBTune utilizes data from each module to create a joint probability distribution. In the process of establishing the probability distribution, OBTune follows the basic principle of multi-task learning, and integrates all tuning metrics from both the main task and auxiliary tasks as the objectives of multiple task learning framework. This distribution effectively predicts the mean and variance of each metric at every sampling point, providing a comprehensive overview of the system's behavior under various knob configurations.

④ The selection of the optimal knob configuration is based on this joint probability distribution, using an acquisition function [38]. This acquisition function evaluates the potential performance gains offered by different knob settings. It then identifies the most effective configuration for enhancing system performance.

It's crucial to recognize that OBTune is capable of tuning even in the absence of domain knowledge from the Functionality Interface. In such a scenario without domain knowledge, all tuning knobs are treated as main knobs, and OBTune's tuning process is illustrated in Figure 3 excluding the parts indicated by dashed-line rectangles. Furthermore, the composition of knobs in the main module vary depending whether the Functionality Interface integrates functionality domain knowledge. The main module's knobs include all knobs except those specific to these functionalities. Conversely, if the Functionality Interface lacks any domain knowledge, then all knobs are included in the main module.

Example. For better understanding of the OBTune workflow, we refer to Figure 4 for an illustrative example. Assuming there are six tuning knobs, labeled K_1, K_2, K_3, K_4, K_5 and K_6, along with two indirect metrics, im_1 (tuned for maximal value) and im_2 (tuned for minimal value). The DBA has inputted domain knowledge about functionalities A, B, and C via the Functional Interface. Specifically, for A, its direct metric is dm_1 (tuned for minimal value), and its corresponds

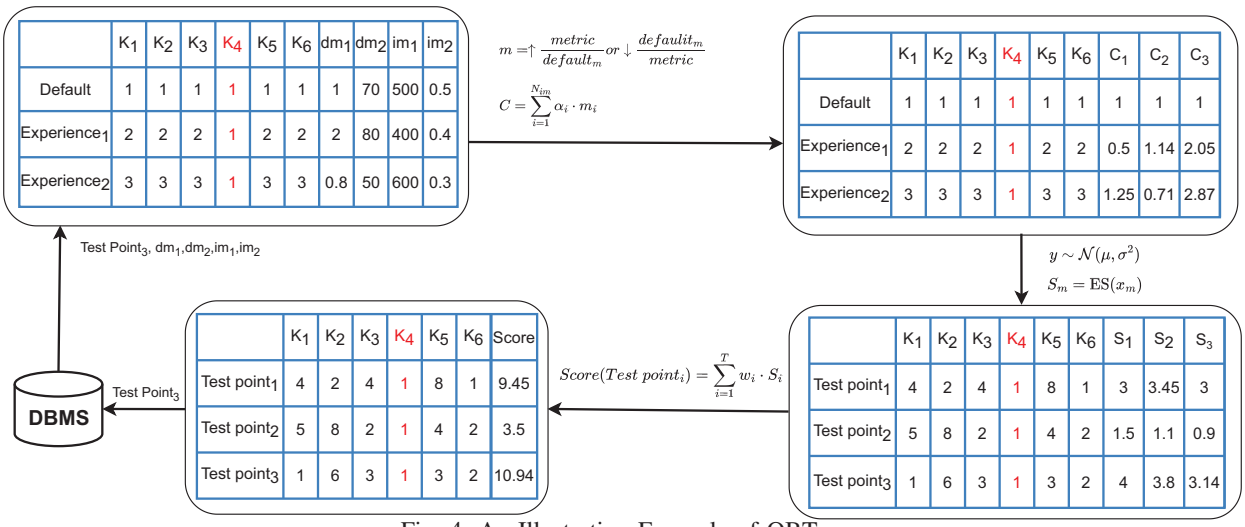

Fig. 4: An Illustrating Example of OBTune

knobs are K_1 and K_2. For B, its direct metric is dm_2 (tuned for maximal value), and its corresponds knob is K_3. For C, its direct metric is dm_3, and its corresponds knob is K_4. Assuming that the tuning workloads will not activate functionality C, therefore, module C does not exist. Regarding module A, the information includes K_1, K_2 and dm_1. For module B, the information comprises K_3 and dm_2. As for the main module, it involves K_5, K_6, im_1 and im_2.

Firstly, we assume that there are three historical experiences for this tuning, as displayed in the top-left table of Figure 4. The initial knob settings, acting as our baseline, are designated as $Default$. The other two historical experiences are labeled as $Experience_1$ and $Experience_2$ respectively.

Secondly, due to the substantial variations in magnitude among different metrics, normalizing each metric within the module is crucial. This normalization ensures that metrics with larger numerical values do not unduly influence the optimization process, thereby guaranteeing a balanced approach. The normalization strategy for each metric is defined as follows:

$$m = \begin{cases} \dfrac{metric}{default_m}, & \text{if the } metric \text{ is preferable larger} \\ \dfrac{default_m}{metric}, & \text{otherwise} \end{cases} \tag{1}$$

In this formula, $metric$ denotes a direct or indirect metric, and $default_m$ is the default value of $metric$. Given that each module may include multiple metrics, in order to specify a simplified optimization direction for each task, we apply the following operation to the multiple metrics within the same module:

$$C = \sum_{i=1}^{N_m} \alpha_i \cdot m_i \tag{2}$$

In this equation, N_m is the number of metrics in this module, and α_i and m_i represent the weight and the normalized value of the i-th direct metric, respectively. In the top-right table of Figure 4, the weight of each metric is set to 1 and the scalar metrics for the main module, A module, and B module are denoted as C_3, C_1, and C_2, respectively.

Thirdly, the multi-task Gaussian process in OBTune employs the data sets (K_1, K_2, C_1), (K_3, C_2), and (K_5, K_6, C_3) to create a joint probability distribution for each module. Each module represents a distinct learning task. This distribution effectively predicts the mean and variance (indicating uncertainty) of each scalar metric at any sample point. Utilizing these predictions, the entropy search acquisition function calculates the entropy reduction values for unexplored points. In the bottom-right table of Figure 4, the predicted entropy reduction values are presented for each module's scalar metric at three test points: $Test\ point_1$, $Test\ point_2$, and $Test\ point_3$.

Fourthly, to compute the total entropy reduction values for each sample point, we use the following formula:

$$Score(Test\ point) = \sum_{i=1}^{T} w_i \cdot S_i \tag{3}$$

In this formula, T represents the total number of modules, while w_i and S_i denote the importance weight and the predicted entropy reduction value of the i-th module, respectively. In the bottom-left table of Figure 4, each module has the same importance weight (i.e., 1), so $Test\ point_3$ achieves the highest score.

Fifthly, we apply the knob configuration of $Test\ point_3$ to the database and re-execute workloads. The tuple $(1, 6, 3, 1, 3, 2, dm_1, dm_2, im_1, im_2)$ is then recorded as $Experience_3$. During the tuning process, it was noted that knob K_4, which is the corresponding knob of inactive functionality C, remained unchanged, thereby effectively mitigates its potential risks incurred by unreasonable knob settings.

V. MTGP For OBTune

The Multi-task Gaussian Process [34] is a type of Multi-task learning within the framework of the Gaussian Process (GP). MTGP inherits the characteristics of MTL that use information between related tasks to help the model obtain better prediction results. Furthermore, MTGP is an extension of the GP while retaining its high-quality features.

A. Dataset and Task Definition

Observed data. Each tuning round may trigger a different number of database functionalities when replaying the workload. Due to the varying combinations of triggered functionalities resulting in different relationships, only the data collected from the current round and historical data from previously triggered functionality combinations that were the same as the current round are used as observed data. First, we define observed input points $\mathbf{X}$,

$$\mathbf{X} = (x_1, ..., x_n) \tag{4}$$

where n is the number of observed data for all identical combinations of modules. x_i is the combination of all triggered modules' knobs, i.e., $\mathbf{x_i} = (knob_{i1}, ...knob_{im})$. Note, m is the total number of the triggered modules' knobs and is dynamically changed in every tuning round according to the different combinations of triggered modules.

The output can be expressed as:

$$\mathbf{Y} = (y_1, ..., y_n) \tag{5}$$

where y_i is the metric of the i-th input x_i and each $y_i = (metric_{i1}, ...metric_{il})$. Like m, l also is the metrics total number of the triggered modules and changes with the different triggered modules in each round.

Metrics of different functionalities may have correlation or conflicts. For example, if more resources are used for optimizing the direct metrics of load balance task it may hurt the indirect metrics (e.g., transaction throughput) when the load balance functionality is running. Knobs turning need to find the balance in these metrics. Knob classification in OBTune could help avoid tuning irrelevant knobs, and thus reduce the possibility of optimizing one functionality at the expense of another. To find a balanced point, all metrics are combined into a single objective for choosing next knob combination via linear weighted scalarization (Equation 3). Advanced method for automatically setting the weight of each metrics in multi-task learning is regarded as future work [39].

Problem Statement. Consider a set of p tunable knobs $K_1, K_2, \ldots, K_p$ and their respective domains $\Theta_1, \Theta_2, \ldots, \Theta_p$, which can be continuous, discrete, or categorical. The configuration space for the knobs is thus defined as the Cartesian product $\Theta = \Theta_1 \times \Theta_2 \times \ldots \times \Theta_p$. Assume there are q metrics to be optimized, denoted as $metric_1, metric_2, \ldots, metric_q$, with at least one being an indirect metric. Given a parameter configuration $k \in \Theta$, we assume each metric $metric_i$ is calculated by the objective function f_i, that is, $metric_i = f_i(k)$. The optimization goal is to find the optimal configuration k^* that maximizes the composite value, formalized as:

$$k^* = \underset{k \in \Theta}{\operatorname{argmax}} \left(\sum_{i=1}^{q} (\lambda_i \cdot norm(f_i(k))) \right) \tag{6}$$

Here, λ_i represents the multiplication of the module's weight w (Equation 3), to which the i-th metric belongs, by the weight α (Equation 2) assigned to this metric within the module, and $norm$ denotes the normalization method (Equation 1).

According to our definition of observed inputs and outputs, it is clear that the observed set is composed of knobs and corresponding metrics of the same module combination in all historical data. The combination of triggered modules' knobs serves as observed input $\mathbf{X}$ and the combination of triggered modules' scalar metrics serves as output $\mathbf{Y}$. Here, we can get the latent function from observed data through:

$$\mathbf{Y} = \mathbf{F}(\mathbf{X}) \tag{7}$$

Kernel. GP is determined by mean and kernel function. Similarly, the key to MTGP is to reconstruct the kernel to identify the correlation between multiple tasks which is also the key to achieving multi-dimensional output. The new kernel in MTGP is:

$$K_{MTGP} = K^f \otimes K^x \tag{8}$$

where K^f is the similarity matrix between tasks, indicating the similarity or correlation between different tasks. K^x is the covariance matrix of all observed data points, containing the observed data of multiple tasks. To ensure that the newly constructed kernel function K_{MTGP} is positive semi-definite, K^f is constructed based on Cholesky decomposition. $\otimes$ denotes the Kronecker product, a matrix operation that operates on two matrices in a specific way to obtain a new matrix. The new kernel function obtained by Kronecker product processing of the task similarity matrix and observed data matrix can jointly model the features of multiple tasks and learn the shared and differential features between them. This method can improve the effectiveness and efficiency of multi-task learning because it can share information and parameters between tasks, making the learning process more stable and reliable [40].

Latent function. Under the assumption of MTGP, all latent functions $f(X)$ of the tasks are assumed to follow a multivariate Gaussian distribution, where $f(X) = (f_1(X), f_2(X), ..., f_m(X))^T$ and m is the number of tasks. For a test input X_*, $f_i(X_*)$ means the predicted output of task i at input variable X_*. For any two tasks i and j, their covariance matrix is K^f_{ij}, which represents the correlation between their latent functions.

We can also assume that the input variable X follows a Gaussian distribution, so the covariance between input variables X and X_* is $k^x(X, X_*)$. Based on these assumptions, the following formula can be derived:

$$< f_i(X) f_j(X_*) >= K^f_{ij} k^x(X, X_*) \tag{9}$$

where K^f is a positive semi-definite matrix that specifies the similarity between tasks, K^f_{ij} is the shared kernel function between the i-th and j-th tasks that describes the similarity between the two tasks, and $k^x(X, X_*)$ is the covariance function between input data X and X_* that represents the similarity between the test point X_* and the observed point X. This formula represents the covariance between $f_i(X)$ and $f_j(X_*)$. It multiplies the shared kernel function with the kernel function between the test and observed points to obtain the covariance matrix K^f_{ij} between $f_i(X)$ and $f_j(X_*)$. In MTGP, the covariance structure between tasks can be shared through

K_{ij}^f, which improves the prediction accuracy of multiple tasks under limited observation data.

Using this formula, the Gaussian distribution of the output $f_i(X_*)$ of task i at input variable X_*, which is the required Gaussian distribution of the latent function, can be obtained:

$$p(f_i(X_*)|f_j(X_*), X, X_*) = \mathcal{N}\left(K_i^T(K_f + \sigma^2 I)^{-1} f_j(X_*), K_{ii} - K_i^T(K_f + \sigma^2 I)^{-1} K_i\right) \tag{10}$$

where K_i is the covariance vector between task i and other tasks, K_{ii} is the variance of task i, K_f is the covariance matrix of all tasks, σ^2 is the noise variance, and I is the identity matrix.

Predicted new data. The MTGP predictions can be interpreted as a linear combination of historical observations. By extending f_i in the above induced formula to all tasks and transforming the formula form, we can get the predicted mean of the new data point x_* for the j-th task:

$$\overline{f}_j(x_*) = (k_j^f \otimes k^x(X, X_*))^T K_{MTGP}^{-1} \mathbf{Y} \tag{11}$$

where k_j^f is the j-th column of K^f, Y is the output. By extending f_j to all tasks, we get a model that can predict metrics for unknown knob combinations based on the observed data.

B. Acquisition Function

OBTune chooses the popular acquisition function called entropy search (ES) to find the next knob combination for evaluation [41]–[43]:

$$\begin{aligned} &\arg\max_{x \in \mathcal{X}} ES_m(x) = \\ &\arg\max_{x \in \mathcal{X}} H(P_m(x^*)) - E_{f(x)}[H(P_m(x^*|x, f(x)))] \end{aligned} \tag{12}$$

where $\mathcal{X}$ is the entire knob search space. $P(x)$ is the distribution of point x. f is a latent function. $H(P_m(x^*))$ is the entropy of the distribution of the global optimal point x^* based on the currently available observations. $E_{f(x)}[H(P_m(x^*|x, f(x)))$ is the entropy of the distribution of x^* after updating the distribution by adding the point $(x, f(x))$ to the existing observations. The one with the highest $ES(x)$ among all knobs test points is then selected as the next test point.

VI. Experimental Results and Analysis

Comparative Tuning Methods. In this section, we conduct a comparative performance analysis of OBTune against published auto-tuning methods. These methods include OtterTune [3], BestConfig [2], Default (i.e., the default database parameter settings), and the more recent HUNTER [7]. It's worth noting that LlamaTune [18] is not designed to introduce a new tuning method, but rather to increase the sample efficiency of existing methods, including SMAC [31], GP-BO [16], and DDPG [17]. This enhancement means that LlamaTune (as an orthogonal work) would improve the sample efficiency of OBTune. As this is beyond the scope of this work, LlamaTune's tuning method was not selected for comparison in our experiment.

Workload. For our experiments, we selected three benchmark tools: TPC-C, TPC-H, and Sysbench, all of which are compatible with OceanBase [10], a system that supports HTAP (Hybrid Transactional/Analytical Processing). Our experiments incorporated four types of workloads: TPC-C, TPC-H, and both read-only (RO) and write-only (WO) workloads from Sysbench. Specifically, the TPC-C workload was used to simulate an OLTP environment, configured with 400 warehouses, amounting to a data size of approximately 25 GB, and operated with 500 terminals. The TPC-H workload, simulating an OLAP environment, utilized a scale factor of 30 with a data size around 30 GB. Sysbench was employed to simulate RO and WO workloads, set up with 16 tables containing 400K records each, totaling a data size of about 17 GB, and executed with 128 threads. During the knobs tuning, each of the four workloads is executed for 300 seconds per iteration; there are a total of 100 iterations.

A. Experiments On OceanBase

Experimental Setup. The experiments are run on OceanBase Community Edition 3.1.1. In the evaluation, we used 3 clusters to simulate the classic deployment of three data centers. Each cluster has two nodes, and each node (referred to as an observer) has 4-core 2.00 GHz CPU, 16 GB memory, and 150 GB disk. The total number of knobs tuned by OBTune is 70, and the weight for each metric and module is set to 1.

Functionality. In the Functionality Interface, we provided domain knowledge for a total of four functionalities in our experiments: load balancing functionality, minor compaction functionality, SQL functionality, and major compaction functionality. Table II shows the corresponding knobs of each functionality, along with the main knobs when all four functionalities are in an active state.

Load Balancing Functionality: This functionality is responsible for routing traffic to low-load machines or migrating replicas from high-load machines to low-load machines. Its direct metric is the time taken to complete the load balancing operation. A shorter balancing time indicates more effective load management, thus contributing to optimal utilization of software and hardware resources.

Minor Compaction Functionality: In the OceanBase database cluster, data is initially written to the MemTable on each server. When the size of the MemTable exceeds a specific threshold, it becomes necessary to transfer the data from the MemTable to an SSTable in order to free up memory. This process is known as minor compaction. The direct metric associated with minor compaction is the time it takes to complete the operation. Faster minor compaction operations not only enhance the performance of the functionality but also allow more resources for other database functionalities, ultimately improving overall system performance.

SQL Functionality: When OceanBase receives an SQL statement, it undergoes five stages of processing: syntax pars-

TABLE II: The knob classification for four active functionalities with provided domain knowledge.

Classification of Knobs	Knobs
Main Knobs	cpu_count, memory_chunk_cache_size, memory_reserved, system_memory, use_large_pages, tableapi_transport_compress_func, log_disk_size, bf_cache_priority, cache_wash_threshold, fuse_row_cache_priority, user_row_cache_priority, trx_2pc_retry_interval, px_workers_per_cpu_quota, enable_perf_event, tablet_ls_cache_priority, bf_cache_miss_count_threshold, user_block_cache_priority, autoinc_cache_refresh_interval, workers_per_cpu_quota, index_block_cache_priority, opt_tab_stat_cache_priority, standby_fetch_log_bandwidth_limit, weak_read_version_refresh_interval, enable_sql_audit, plan_cache_evict_interval, memory_limit_percentage, enable_syslog_recycle, global_write_halt_residual_memory, trace_log_slow_query_watermark, enable_monotonic_weak_read, sys_bkgd_io_low_percentage, sys_bkgd_io_high_percentage, cpu_quota_concurrency
Load Balancing Knobs	log_storage_warning_tolerance_time, server_balance_critical_disk_waterlevel, migrate_concurrency, balancer_tolerance_percentage, server_balance_disk_tolerance_percent, enable_auto_leader_switch, sys_bkgd_net_percentage, server_data_copy_in_concurrency, data_copy_concurrency, auto_leader_switch_interval, server_data_copy_out_concurrency
Minor Compaction Knobs	minor_deferred_gc_time, minor_warm_up_duration_time, minor_merge_concurrency, freeze_trigger_percentage, memstore_limit_percentage, minor_compact_trigger
SQL Knobs	max_px_worker_count, dtl_buffer_size, px_task_size, ob_sql_work_area_percentage, optimizer_capture_sql_plan_baselines, ob_enable_batched_multi_statement, location_cache_refresh_sql_timeout, large_query_threshold, sql_login_thread_count, enable_sql_operator_dump, large_query_worker_percentage, fuse_row_cache_priority
Major Compaction Knobs	merger_completion_percentage, merger_check_interval, micro_block_merge_verify_level, major_compact_trigger, merge_stat_sampling_ratio, merger_check_interval, merge_thread_count, enable_merge_by_turn

ing, semantic analysis, query rewriting, query optimization, and code generation, ultimately resulting in the generation of a physical execution plan. The direct metric used to evaluate this functionality is the time it takes for an SQL statement to be transformed into a physical execution plan. Shorter execution times indicate improved module performance, facilitating more efficient resource allocation within the database.

Major Compaction Functionality: When the number of minor compactions exceeds a certain threshold or during daily low-traffic periods, the system consolidates the baseline SSTable with subsequent incremental SSTables generated during minor compactions into a single SSTable. This process is referred to as Major Compaction. The direct metric for this functionality is the major compaction time. Shorter major compaction times indicate better performance, allowing the database to allocate more resources to other areas and thus enhancing overall system performance.

Mixed workloads for functionality-aware tuning. Previous tuning methods couldn't activate multiple database functionalities due to either simple workloads or short workload replay duration. Therefore, to activate multiple database functionalities for evaluating functionality-aware tuning, we developed MIX_1, a combination of TPC-C and Sysbench, each running for 2400 seconds per iteration. This duration ensured the activation of all four functionality modules. For MIX_2, we modified MIX_1 by setting the knob enable_rebalance to 'False' and constraining the value of major_compact_trigger within the range [60000, 65000]. This prevented the activation of the load balancing functionality and the major compaction functionality. MIX_3 was created based on MIX_1 to randomly activate the load balancing functionality and major compaction functionality. We limit the value of major_compact_trigger to 10 specific values, with half of them activating the major compaction functionality and the rest not. Furthermore, in each iteration, the knob enable_rebalance was randomly set to 'False' or 'True'.

1) No functionality domain knowledge: In the absence of domain knowledge, OBTune treats all knobs as main knobs and tunes these knobs (a total of 70). In this scenario, we conducted comparative experiments between OBTune and other tuning methods, as illustrated in Figure 5. Our results indicate that OBTune's performance is on par with the other methods in both final overall performance and convergence speed. This means OBTune can work similar to previous state-of-the-art methods in the traditional black box based tuning setting.

2) Limited functionality domain knowledge: Given that the Functionality Interface supplied domain knowledge for both the load balancing functionality and the minor compaction functionality, and both functionalities were activated, OBTune tuned a total of three modules. These included the load balancing module with 11 knobs, the minor compaction module with 6 knobs, and the main module. The latter encompassed all remaining knobs, totaling 53, excluding those assigned to the first two modules. In this setup, the indirect metrics assessed were throughput and latency.

The tuning results for OBTune, OtterTune, BestConfig, and HUNTER are depicted in Figure 6 in this scenario. Experimental results demonstrate that, without compromising convergence speed, OBTune's final tuning results outperform those of OtterTune, BestConfig, and Hunter across all metrics. We also observed the correlation or conflicts in tuning multiple metrics by OBTune's MTL framework. In Figure 6(c) and (d), the metrics of balance time and minor compaction time achieved at 31th iterations were slightly perform worse. However, as the metrics of throughput and average latency had better results, OBTune still chose this configuration for 31th iteration. Additionally, due to the neglect of signals from direct metrics in the tuning process by OtterTune, BestConfig, and HUNTER, their final performance on these metrics did not show significant improvement and were even inferior to the performance of Default on certain metrics (e.g., the final results of HUNTER in Figure 6(d)).

OtterTune, HUNTER, and BestConfig originally focused on tuning the database's indirect metrics. They can be regarded as a single knob tuning task. We modified their optimization objectives to the sum of normalized direct and indirect metrics to evaluate whether the single task learning framework can co-tune these metrics together. These variants are referred to as OtterTune+, HUNTER+, and BestConfig+

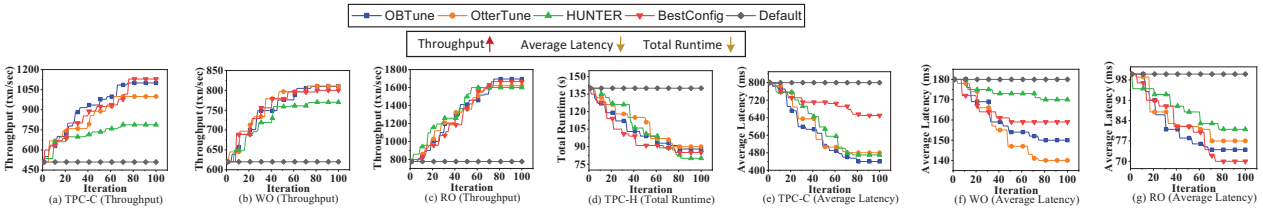

Fig. 5: Tuning results of various tuning methods without domain knowledge of of functionalities (OceanBase).

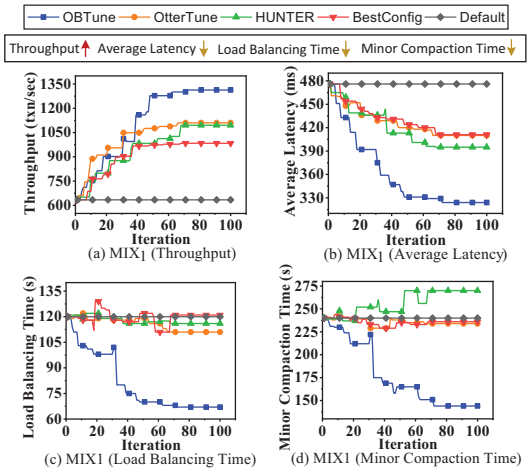

Fig. 6: Results when two functionalities are activated (OceanBase).

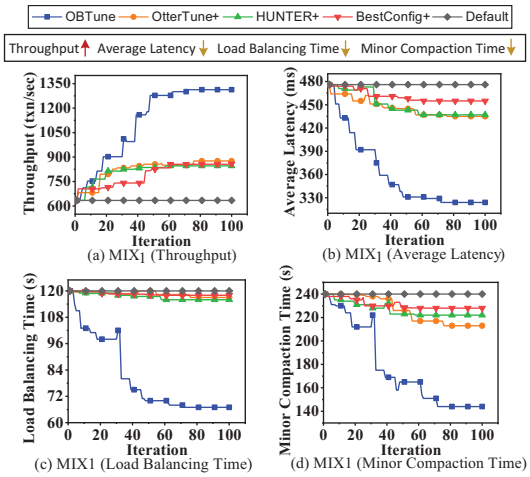

Fig. 7: Results of OBTune and variants of other tuning methods when two functionalities are activated (OceanBase).

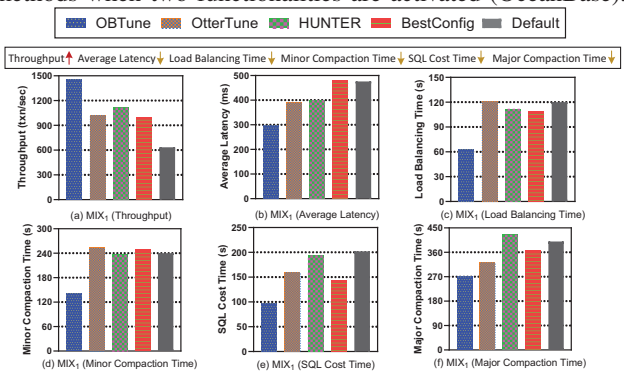

Fig. 8: Results when four functionalities are activated (OceanBase).

respectively. Experimental results are presented in Figure 7. We observed that OBTune still achieved the best results across all metrics. Hence, relying solely on existing tuning methods to optimize multiple metrics of a database poses a challenge in significantly improving the performance of various modules within the database. This challenge arises from the difficulty of existing tuning methods in learning the relationships between different modules. Compared to the Default, OtterTune+, HUNTER+, and BestConfig+ showed improvements in various metrics. However, their overall performances (i.e., indirect metrics) were not as favorable as those of OtterTune, HUNTER, and BestConfig (as illustrated in Figure 6). This is because directly combining these metrics may mislead the knobs tuning direction of these single task learning frameworks. Taking into account all tuned metrics, the tuning results of OtterTune, HUNTER, and BestConfig are slightly superior to their respective variants. Hence, we have only reported the results of OtterTune, HUNTER, and BestConfig for the subsequent OceanBase experiments. For convergence speed, since these methods reach convergence at a similar number of iterations in Figure 6 and Figure 7 (around 70 iterations), subsequent experiments only demonstrate the final tuning results.

3) More domain knowledge: As the Functionality Interface provided domain knowledge for four functionalities, OBTune categorized the tuning knobs into five categories as shown in Table II: load balancing knobs (including 11 knobs), minor compaction knobs (including 6 knobs), SQL knobs (including 12 knobs), major compaction knobs (including 8 knobs), and main knobs (including 33 remaining knobs). In this setup, we assessed throughput and latency as the indirect metrics.

Firstly, we conducted experiments, as shown in Figure 8, where all four functionalities were activated, thus requiring OBTune to tune five modules. The results indicate that OBTune still achieved the best results across all metrics. The results in Figure 6 and Figure 8 were achieved by the same workload, but they had different domain knowledge. OBTune's tuning results on Figure 6, where domain knowledge for two functionalities was provided, were not as favorable as those in Figure 8, which benefited from the provision of domain knowledge for all four functionalities. This suggests that the integration of more extensive domain knowledge through the Functionality Interface can lead to improved results. The reason is that, with more comprehensive domain knowledge, MTGP can more accurately model the interrelationships between the various modules.

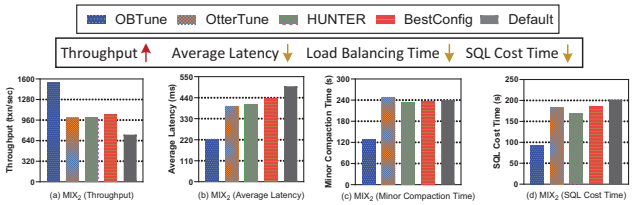

Fig. 9: Results when two functionalities are activated, and two other functionalities are not activated (OceanBase).

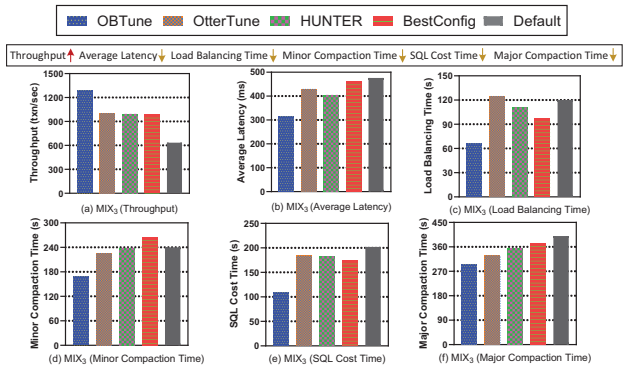

Fig. 10: Results when two functionalities are activated, and two other functionalities are randomly activated (OceanBase).

Secondly, we conducted experiments, as depicted in Figure 9, where the load balancing functionality and the major compaction functionality were not triggered, while the SQL functionality and the minor compaction functionality were active. Throughout the tuning process, it was observed that OBTune maintained the original settings of the 19 knobs associated with the inactive functionalities, while other methods altered these knob values. This difference stems from OBTune's capacity to recognize active functionalities, enabling it to focus solely on tuning the main knobs, minor compaction knobs, and SQL knobs, a total of 51 knobs. Conversely, lacking this capability to identify active functionalities, other methods were required to adjust all 70 knobs. As shown in Figure 9, the tuning results of BestConfig, OtterTune, and HUNTER were better than the database's default values across various metrics. Due to their tuning of more knobs and their inability to learn the relationships between modules, their tuning results on various metrics were inferior to those achieved by OBTune.

Finally, We conducted experiments, as depicted in Figure 10, where the load balancing functionality and the major compaction functionality are triggered randomly. This is to more accurately simulate real-world business scenarios, where the functionality modules triggered in each iteration are not entirely identical. In this experiment, each module had been triggered, requiring OBTune to tune five modules, totaling 70 knobs. The experimental results still demonstrate that OBTune continues to achieve the best optimization results across all metrics compared to other tuning methods.

4) Execution Time Breakdown: Similar to OtterTune and BestConfig, OBTune was also an external tool for database systems. Those tools would not introduce extra complexity into the database system. Database parameter tuning iterations consist of two primary phases: configuration generation and workload execution. According to experimental data from OceanBase, configuration generation times for OBTune, OtterTune, BestConfig, and HUNTER are 9.18 seconds, 6.01 seconds, 0.003 seconds, and 0.04 seconds, respectively. The extended configuration generation time for OBTune is mainly due to its reliance on MTGP to manage complex covariance structures and conduct matrix inversion calculations, which are computationally intensive. Nevertheless, considering that the workload execution takes either 300 seconds or 2400 seconds in each iteration, OBTune's configuration generation time may have a minor impact on the overall tuning process.

5) Summary: In the experiments conducted on the distributed database OceanBase, when domain knowledge about database functionality is not provided by the DBA, tuning results of OBTune are comparable to those of state-of-the-art tuning methods. However, when the DBA provides domain knowledge about functionalities specific to DBMS products, OBTune outperforms OtterTune, HUNTER, and BestConfig, as well as their variants in various aspects.

For database knob tuning, enhancing end-to-end performance is a key objective, but the metrics of other functionalities should also receive significant attention. For instance, the load balancing functionality is crucial in practical applications. Fast load balancing supports the system's flexible expansion and horizontal scaling, enabling it to swiftly adapt to changes and growth in business requirements, thereby enhancing system flexibility. The experimental results from both Figure 6 and Figure 8 consistently demonstrate OBTune's capability to effectively address these issues in real-world scenarios.

B. Experiments On PostgreSQL

To demonstrate the generalizability of this tuning framework, we apply OBTune to tune knobs of PostgreSQL.

Experimental Setup. These experiments are run on PostgreSQL 12 deployed on a server with 8-core 2.10 GHz CPU, 16 GB memory, and 150 GB disk. The total number of knobs tuned by OBTune is 65, and the weight for each metric and module is set to 1.

Functionality. In the Functionality Interface, we provided domain knowledge for a total of two functionalities in our experiments: the garbage collection functionality and the lock functionality [44]. A brief overview of both is provided below.

Garbage Collection (Vacuum) Functionality: This functionality periodically checks for unused space in the database, releasing it for reuse. In PostgreSQL, when a row is deleted or updated, it is not immediately removed from disk. Instead, PostgreSQL marks the row as 'deleted' and designates it as reusable space. However, unused space can accumulate over time, leading to disk space wastage. The tuning goal of this functionality is to minimize garbage collection operation time, thereby allowing PostgreSQL to allocate more resources to other database functionalities.

Lock Functionality: This functionality is a crucial for maintaining data consistency, isolation, and integrity. It enables multiple sessions to access the database concurrently while ensuring the correct execution of data operations. During parallel

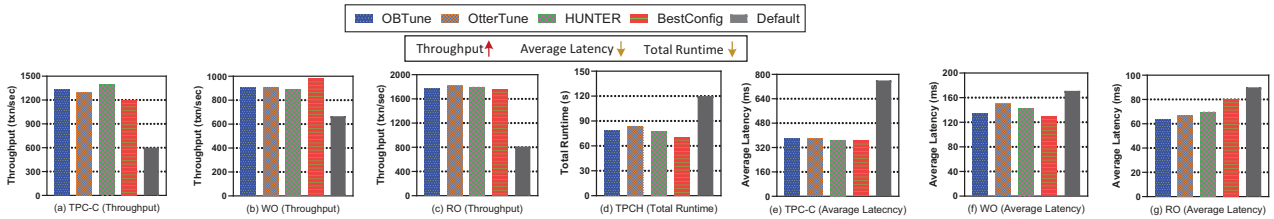

Fig. 11: Tuning results of various tuning methods without domain knowledge of of functionalities on PostgreSQL.

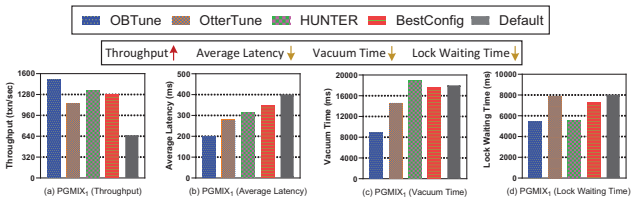

Fig. 12: Results when two functionalities are activated (PostgreSQL).

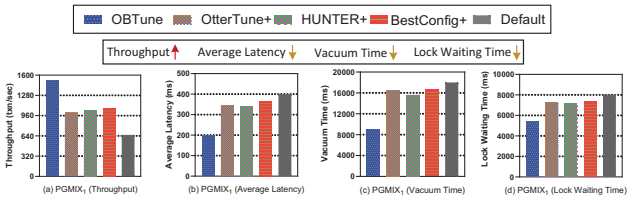

Fig. 13: Results of OBTune and variants of other tuning methods when two functionalities are activated (PostgreSQL).

workload execution, extended lock waiting times can cause delays in other sessions, impacting the database's concurrency performance. Therefore, the tuning goal of this functionality is to minimize lock waiting time.

Mixed Workload for functionality-aware tuning. We created PGMIX_1, a mixed workload combining TPC-C and Sysbench. Each run of PGMIX_1 lasts for 1500 seconds, ensuring the activation of both functionalities. To introduce randomness into the triggering of the garbage collection functionality, we modified PGMIX_1 by randomly toggling the value of autovacuum between 'on' and 'off' in each iteration, creating a workload version named PGMIX_2.

1) No functionality domain knowledge: In the absence of domain knowledge, we conducted the experiment depicted in Figure 11. The results show that OBTune's tuning performance for PostgreSQL remains competitive with other methods. Notably, BestConfig's tuning outcomes exhibit greater variability, a consequence of leveraging multiple randomized strategies throughout its tuning process. Specifically, in the throughput optimization experiments for TPC-C, it was observed that the standard deviation of BestConfig's tuning results, obtained across 5 tuning sessions, was approximately 5.5 times greater than that of other tuning methods. Thus, for all experiments involving PostgreSQL, the tuning results for BestConfig presented here are averaged over 5 tuning sessions.

2) Provision of functionality domain knowledge: In this context, the Functionality Interface provided domain knowledge for both the garbage collection functionality and the lock functionality, with the assessed indirect metrics being throughput and latency.

The experimental results, with both functionalities triggered during tuning, are depicted in Figure 12. The results demonstrate that OBTune outperforms other tuning methods in both direct and indirect metrics. This finding supports the notion that OBTune can achieve superior optimization results across different databases with domain knowledge for functionalities.

We compared the tuning results of OBTune with its variants on PostgreSQL, as shown in Figure 13. The results demonstrate OBTune's superiority over the variants of other tuning methods across different databases. Furthermore, considering all tuned metrics, results of OtterTune, HUNTER, and BestConfig are also slightly superior to their respective variants.

In the experiment illustrated in Figure 14, we randomly trigger the garbage collection functionality. The tuning results of OBTune outperformed the other methods in all metrics. This finding supports that OBTune consistently achieves superior optimization results across different databases when the tuned modules are triggered randomly.

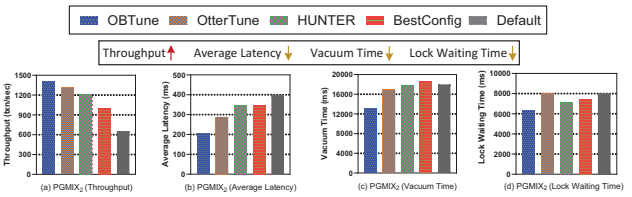

Fig. 14: Results when two functionalities are activated, and two other functionalities are randomly activated (PostgreSQL).

3) Summary: In experiments conducted on PostgreSQL, OBTune also demonstrated outstanding performance. Therefore, the generalizability of OBTune across different databases is validated. Additionally, experimental data from PostgreSQL show that OBTune requires an average of 8.56 seconds for configuration generation.

VII. Conclusion

We propose an auto-tuning system called OBTune, which can accommodate DBAs of varying expertise levels. OBTune utilizes domain knowledge during the tuning process to tune knobs effectively. It also avoids adjustments to corresponding knobs of inactive functionalities, reducing potential risks. Our experiments demonstrate that OBTune identifies more effective knob configurations than current state-of-the-art tuning systems when given domain knowledge for functionalities.

Acknowledgements

We thank the anonymous reviewers for their insightful comments and feedback. This work is supported by grants from the National Natural Science Foundation of China (U22B2020 and 62072179), OceanBase, and the Natural Science Foundation of Shanghai (23ZR1418300). Peng Cai is the corresponding author.

REFERENCES

[1] X. Zhang, H. Wu, Y. Li, J. Tan, F. Li, and B. Cui, "Towards dynamic and safe configuration tuning for cloud databases," in *SIGMOD*, 2022, pp. 631–645.

[2] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Socc*, 2017, pp. 338–350.

[3] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *SIGMOD*, 2017, pp. 1009–1024.

[4] V. Thummala and S. Babu, "ituned: a tool for configuring and visualizing database parameters," in *SIGMOD*, 2010, pp. 1231–1234.

[5] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu *et al.*, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *SIGMOD*, 2019, pp. 415–432.

[6] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2118–2130, 2019.

[7] B. Cai, Y. Liu, C. Zhang, G. Zhang, K. Zhou, L. Liu, C. Li, B. Cheng, J. Yang, and J. Xing, "Hunter: An online cloud database hybrid tuning system for personalized requirements," in *SIGMOD*, 2022, pp. 646–659.

[8] X. Zhang, H. Wu, Z. Chang, S. Jin, J. Tan, F. Li, T. Zhang, and B. Cui, "Restune: Resource oriented tuning boosted by meta-learning for cloud databases," in *SIGMOD*, 2021, pp. 2102–2114.

[9] J. Huang, B. Mozafari, G. Schoenebeck, and T. F. Wenisch, "A top-down approach to achieving performance predictability in database systems," in *SIGMOD*, 2017, pp. 745–758.

[10] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi, H. Yu, B. Liu, Y. Pan, B. Yin, J. Chen, and Q. Xu, "OceanBase: A 707 Million tpmC Distributed Relational Database System," *Proc. VLDB Endow.*, vol. 15, no. 12, pp. 3385–3397, 2022.

[11] Z. Yang, Q. Xu, S. Gao, C. Yang, G. Wang, Y. Zhao, F. Kong, H. Liu, W. Wang, and J. Xiao, "Oceanbase paetica: A hybrid shared-nothing/shared-everything database for supporting single machine and distributed cluster," *Proc. VLDB Endow.*, vol. 16, no. 12, pp. 3728–3740, 2023.

[12] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[13] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 42, no. 1, pp. 55–61, 2000.

[14] E. Danna and L. Perron, "Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs," in *International Conference on Principles and Practice of Constraint Programming*, 2003, pp. 817–821.

[15] S. Cereda, S. Valladares, P. Cremonesi, and S. Doni, "Cgptuner: a contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions," *Proc. VLDB Endo.*, vol. 14, no. 8, pp. 1401–1413, 2021.

[16] B. Ru, A. Alvi, V. Nguyen, M. A. Osborne, and S. Roberts, "Bayesian optimisation over multiple continuous and categorical inputs," in *ICML*, 2020, pp. 8276–8285.

[17] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[18] K. Kanellis, C. Ding, B. Kroth, A. Müller, C. Curino, and S. Venkataraman, "Llamatune: Sample-efficient dbms configuration tuning," *Proc. VLDB Endow.*, vol. 15, no. 11, p. 2953–2965, jul 2022.

[19] I. Trummer, "The case for nlp-enhanced database tuning: Towards tuning tools that "read the manual"," *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1159–1165, 2021.

[20] Immanuel.T, "Db-bert: a database tuning tool that "reads the manual"," in *SIGMOD*, 2022, pp. 190–203.

[21] M. Kunjir and S. Babu, "Black or white? how to develop an autotuner for memory-based analytics," in *SIGMOD*, 2020, pp. 1667–1683.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012, pp. 15–28.

[24] B. Chambers and M. Zaharia, *Spark: The definitive guide: Big data processing made simple*. " O'Reilly Media, Inc.", 2018.

[25] B. K. Debnath, D. J. Lilja, and M. F. Mokbel, "Sard: A statistical approach for ranking database tuning parameters," in *ICDE Workshop*, 2008, pp. 11–18.

[26] R. L. Plackett and J. P. Burman, "The design of optimum multifactorial experiments," *Biometrika*, vol. 33, no. 4, pp. 305–325, 1946.

[27] K. Kanellis, R. Alagappan, and S. Venkataraman, "Too many knobs to tune? towards faster database tuning by pre-selecting important knobs," in *12th USENIX Workshop on Hot Topics in Storage and File Systems*, 2020.

[28] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, and A. Pavlo, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1241–1253, 2021.

[29] X. Zhang, Z. Chang, Y. Li, H. Wu, J. Tan, F. Li, and B. Cui, "Facilitating database tuning with hyper-parameter optimization: A comprehensive experimental evaluation," *Proc. VLDB Endow.*, vol. 15, no. 9, pp. 1808–1821, 2022.

[30] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017.

[31] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *LION*, 2011, pp. 507–523.

[32] M. Feurer, B. Letham, and E. Bakshy, "Scalable meta-learning for bayesian optimization using ranking-weighted gaussian process ensembles," in *ICML Workshop*, vol. 7, 2018.

[33] T. Schmied, D. Didona, A. Döring, T. Parnell, and N. Ioannou, "Towards a general framework for ml-based self-tuning databases," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, 2021, p. 24–30.

[34] E. V. Bonilla, K. Chai, and C. Williams, "Multi-task gaussian process prediction," *Advances in neural information processing systems*, vol. 20, 2007.

[35] J. Baxter, "A bayesian/information theoretic model of learning to learn via multiple task sampling," *Machine learning*, vol. 28, no. 1, pp. 7–39, 1997.

[36] S. Vandenhende, S. Georgoulis, W. Van Gansbeke, M. Proesmans, D. Dai, and L. Van Gool, "Multi-task learning for dense prediction tasks: A survey," *IEEE transactions on pattern analysis and machine intelligence*, 2021.

[37] T. Standley, A. Zamir, D. Chen, L. Guibas, J. Malik, and S. Savarese, "Which tasks should be learned together in multi-task learning?" in *ICML*, 2020, pp. 9120–9132.

[38] J. Wilson, F. Hutter, and M. Deisenroth, "Maximizing acquisition functions for bayesian optimization," *Advances in neural information processing systems*, vol. 31, 2018.

[39] R. Cipolla, Y. Gal, and A. Kendall, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7482–7491.

[40] G.-F. Lu, H. Li, Y. Wang, and G. Tang, "Multi-view subspace clustering with kronecker-basis-representation-based tensor sparsity measure," *Machine Vision and Applications*, vol. 32, pp. 1–12, 2021.

[41] K. Swersky, J. Snoek, and R. P. Adams, "Multi-task bayesian optimization," *Advances in neural information processing systems*, vol. 26, 2013.

[42] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast bayesian optimization of machine learning hyperparameters on large datasets," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 528–536.

[43] D. Hernández-Lobato, J. Hernandez-Lobato, A. Shah, and R. Adams, "Predictive entropy search for multi-objective bayesian optimization," in *ICML*, 2016, pp. 1492–1501.

[44] Z. Yang, C. Qian, X. Teng, F. Kong, F. Han, and Q. Xu, "LCL: A lock chain length-based distributed algorithm for deadlock detection and resolution," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2023, pp. 151–163.