HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements

Baoqing Cai*
Huazhong University of Science and Technology
Wuhan, China
bqcai@hust.edu.cn

Yu Liu*
Huazhong University of Science and Technology
Wuhan, China
liu_yu@hust.edu.cn

Ce Zhang
ETH Zürich
Zürich, Switzerland
ce.zhang@inf.ethz.ch

Guangyu Zhang
Huazhong University of Science and Technology
Wuhan, China
zhangguangyu@hust.edu.cn

Ke Zhou†
Huazhong University of Science and Technology
Wuhan, China
zhke@hust.edu.cn

Li Liu
Huazhong University of Science and Technology
Wuhan, China
lillian_hust@hust.edu.cn

Chunhua Li
Huazhong University of Science and Technology
Wuhan, China
li.chunhua@hust.edu.cn

Bin Cheng
Tencent Inc.
Shenzhen, China
bencheng@tencent.com

Jie Yang
Tencent Inc.
Shenzhen, China
edgeyang@tencent.com

Jiashu Xing
Tencent Inc.
Shenzhen, China
flacroxing@tencent.com

ABSTRACT

Recently, using machine learning for performance tuning of cloud database (CDB) service has shown great potentials. However, facing personalized requirements such as various restrictions for tuning with very different workloads, pre-trained models may mismatch or recommend suboptimal configurations given a new workload. On the other hand, if the system tunes configurations in an online fashion, the system will suffer from the cold start problem, resulting in long tuning time and performance fluctuation. To accommodate these problems, we propose an online CDB tuning system called HUNTER. The key feature of HUNTER is a hybrid architecture, which uses samples generated by Genetic Algorithm to warm-start the finer grained exploration of deep reinforcement learning. Meanwhile, we employ Principal Component Analysis, Random Forest, and Fast Exploration Strategy to reduce the search space and the update time of the learning model. In addition, we further propose a clone and parallelization scheme to stress-test workloads on multiple cloned CDB instances (CDBs), resulting in faster and safer configuration exploration. Extensive trials on CDB with public and real-world workloads demonstrate that, given the same time budget and resources, HUNTER improves performance and considerably decreases recommendation time compared to state-of-the-art tuning systems, with accelerations of up to 2.8× and 22.8× utilizing 1 and 20 cloned CDBs, respectively.

*Baoqing Cai and Yu Liu contribute equally to this paper.
†Corresponding author.

CCS CONCEPTS

• **Information systems → Autonomous database administration**; • **Computing methodologies → Machine learning**.

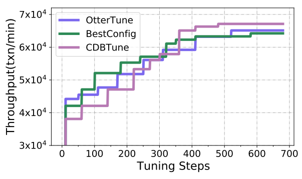
KEYWORDS

Configuration Tuning; Cloud Database; Machine Learning; Parallelization

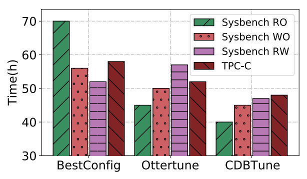
ACM Reference Format:
Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517882

1 INTRODUCTION

The recent development of machine learning techniques promises great potential in automating many aspects of system designs that was previously handled or coded manually. One such example that is of particular interest to the data management community is database tuning. While it is well-known that the performance of a database system can be quite sensitive to particular configurations of hyper-parameters [8, 32]. However, because of the complexity of the database and tuning [33], such a task was done manually by database administrator (DBA) for decades. Recently, many researchers have studied the potential of applying machine learning or metaheuristic to enable automatic tuning. Prominent examples include iTuned [10], BestConfig [48], OtterTune [37, 38], CDBTune [46], QTune [21], ResTune [47].



(a) Tuning Steps for TPC-C



(b) Tuning Time for different workloads

Figure 1: The online tuning steps and time for the optimal throughput.

Table 1: Time breakdown for tuning in each step.

Steps	Workload Execution	Metrics Collection	Model Update	Knobs Deployment	Knobs Recommendation
Time	142.7 s	0.2ms	71ms	21.3s	2.57ms

Current Landscape. This exciting collection of research mostly models the database tuning problem as a black-box optimization problem — given $\mathcal{X}$ the search space of all possible configurations and U the performance function that maps each configuration to its utility: $x \mapsto U(x)$, the goal is to find a well-performing configuration $\hat{x}$ while minimizing the number of times that $U(-)$ is evaluated. A key building block in automatic tuning is to construct a search strategy. Existing work mainly falls into two lines: search-based methods and learning-based methods. The search-based methods use heuristics strategies to explore an ideal configuration in the search space built by candidate values of knobs. For example, BestConfig [48] manually constructs a search strategy using a set of heuristics. To enhance the tuning performance in a large search space where knobs can take continuous values or the number of knobs is large, learning-based methods improve the exploration by learning the historical behavior and experience samples. For example, iTune [10] and OtterTune [37] model the search space as a Gaussian process and apply machine learning classifiers and sorting to shrink the scope of the search space, while using Bayesian optimization to construct a search strategy automatically. Another family of methods uses the reinforcement learning model to explore the search space. For instance, CDBTune [46] and QTune [21] employ Deep Determine Policy Gradient (DDPG) to model the process of exploration and acquire an end-to-end strategy. Since the reinforcement learning model can consider the correlation between decisions (*i.e.*, configurations), these methods have achieved state-of-the-art results. Meta-learning has recently been added into the tuning issue as an optimization approach for reinforcement learning [12]. ResTune [47] employs meta-learning to learn and figure out the initial configurations for various workloads, resulting in increased tuning ability and efficiency when dealing with new workloads, as well as lower resource utilization rate.

Struggles in Real-world Deployment. This paper is inspired by our recent experience in deploying these state-of-the-art methods in a real-world, online cloud database (CDB) environment and from our discussions with DBAs of this large CDB provider. Despite the promising performance obtained by existing methods, we do observe a set of potential improvements that are necessary for better deployments in the real-world. Many of these root in the request of *personalization* from the user, and the goal of this paper is to revisit many design decisions to accommodate this requirement. We summarize these as follows.

(1) *Availability.* Most users expect the database systems to be online during the tuning process. Meanwhile, users wish to avoid restarting the database, albeit it is necessary because some knobs only take effect after the restart. Despite compromising by tuning only dynamic knobs that do not need reboots, users still find the performance fluctuations in this process unsatisfactory.
(2) *Diversity and Challenging Adaptability.* We observe the level of diversity in user requirements that exceeds our expectations. Different users often require different subspaces for their tuning tasks and often provide very different workloads. As a result, we find that transfer knowledge from a set of previously seen datasets (like what systems such as learning-based methods are assuming) to a new tuning task to be challenging [38].
(3) *Cold Start.* As a consequence of the previous challenge, we find that existing systems often fail to warm-start the tuning process. As shown in Figure 1(a), the tuning steps of state-of-the-art methods for optimal parameters on TPC-C are 475 at least. Considering the tuning time in each step shown in Table 1, this is disastrous for the user experience. Figure 1(b) lists the tuning time arriving at their "optimal" performance for the state-of-the-art methods, which demonstrates our point.
(4) *Parallelization and Optimizations.* As a consequence of the cold start problem, most existing methods require non-trivial amount of time for workload execution (Table 1). As shown in Figure 1(b), the tuning time of state-of-the-art methods for optimal parameters is 40 hours at least. What makes it worse is that the sequential nature of many reinforcement learning methods makes parallelization a non-trivial task.

Our Contributions. Inspired by these challenges, we propose an online hybrid automatic tuning system for CDB in this paper, called HUNTER. A prominent feature of HUNTER roots in its design: it incorporates two learning models consisting of (1) a *meta-heuristic algorithm* model and (2) a *deep reinforcement learning* (DRL) model. Instead of relying on a pre-trained model to warm-start DRL (such as DDPG), HUNTER employs a genetic algorithm(GA) to generate samples for DRL and to reduce search space via Principal Component Analysis (PCA) and Random Forest (RF). We further propose a Fast Exploration Strategy (FES) to optimize the search strategy of DDPG, resulting in acceleration of recommendation.

In addition, we optimize the system via simultaneously execute different configurations on multiple database instances with the same workload. Note that although the parallelization scheme has been mentioned in previous great works [4, 24], in this paper, we detail its design and achieve the acceleration by cloning instances of CDB. To accommodate the availability challenge, we conduct explorations mainly on cloned CDB instances (CDBs) and only deploy the optimal configurations after the tuning process. Extensive experimental results show that, compared with the state-of-the-art methods, given the same time budget and resources, HUNTER improves performance and saves the recommendation time up to 2.8× and 22.8× under 1 and 20 cloned CDBs respectively.

We summarize our contributions as follows:

(1) we propose an online tuning system to improve the performance as well as efficiency and practicability on CDB.
(2) We tackle the availability problem by tuning and testing configurations on cloned CDBs.
(3) We tackle the adaptability problem by using online tuning with efficiency.
(4) We tackle the cold start problem with a hybrid architecture, in which we use the samples generated by GA to warm-up DDPG.
(5) Based on the cloned CDBs, we design a parallel scheme for DRL, making DDPG practical for the on-the-fly scene with the fast velocity of exploration.
(6) Experimental results demonstrate HUNTER can greatly reduce tuning time for reliable performance compared with state-of-the-art counterparts.

Overview. This remainder of this paper is organized as follows. §2 introduce the overview architecture and workflow of HUNTER with design rationale and idea. Then, we specify the core modules of our tuning system in §3. In §4, we propose three optimization schemes for above designs. Following, we discuss the future work of HUNTER in §5. In §6, we present our experimental evaluation. Finally, we review the related works in §7.

2 SYSTEM OVERVIEW

In this section, we will demonstrate the architecture and workflow of HUNTER. The design rationale follows closely.

2.1 Overview

Architecture. HUNTER is an online tuning service for CDB and its overview is shown in Figure 2. For the users at the client-side, our system at the cloud-side consists of two parts. First, the **Controller** interacts with the target DBMS, tuning system, and users. It is comprised of *Actors* and CDBs, where *Actors* are designed to collect runtime information from the DBMS, clone the user's instances on CDBs, and manage the parallel execution on cloned CDBs. Note that in the classic architecture (*e.g.*, CDBTune), the Controller only interacts with a single CDB instance. While the Controller in HUNTER manages a collection of Actors, each of which manages a collection of CDBs cloned from the user's instance. Second, the **Hybrid Tuning System** consists of a Shared Pool of sampled configurations, interacting with three components — (a) *Sample Factory*, which generates samples according to target workloads and the user's requirements, via GA and *Rules*. *Rules* are restrictions defined by users or DBAs, including which knobs are fixed and the allowed adjusted range of the rest of the knobs. For example, *innodb_adaptive_hash_index = OFF. thread_handling = pool-of-threads* if *connection > 100.* Obviously, under different rules, a pre-trained model is hard to recommend satisfactory configurations, because the path to the optimal value may be blocked. (b) *Search Space Optimizer*, which takes as input the samples generated by the *Sample Factory* and conduct compression and dimension reduction over both metrics and search space; and (c) *Recommender*, which uses reinforcement learning to further explore the search space, warm-started using the *Shared Pool* of samples.

Samples in the *Shared Pool* are represented as $\{(S_i, A_i, P_i)\}$, where S_i is the metrics, A_i and P_i represent one configuration and its performance. Configurations correspond to knobs with specific values and the performance is measured as throughput and latency. The metrics encode the state of the CDB, such as *lock_deadlocks, buffer_pool_bytes_dirty, buffer_pool_pages_free, etc.* In HUNTER, we select 63 metrics, following the same setting as CDBTune. More details about the Hybrid Tuning System are described in §3.

Workflow. A request from a user consists of a CDB instance, workload information characterized as a set of queries, a personalized requirements (*i.e.*, *Rules*), a time budget, and the maximal degree of parallelization. Given this request, the Controller allocates a set of *Actor*s, each of which first clones the user's instance. Each *Actor* can also use the input workload information to stress-test all cloned CDBs that it allocated.

In the first phase, each *Actor* generates random configurations and stress-tests them on cloned CDBs. Each stress testing generates a sample, (S, A, P), which is put into the *Shared Pool. Sample Factory* takes as input these randomly generated samples and uses GA to generate new configurations. The Controller then stress-tests these newly-generated configurations and produces new samples in the *Shared Pool.* This process is repeated until the number of samples reaches a pre-defined threshold or the performance dose not improve for an extended period of time.

In the second phase, the *Search Space Optimizer* collects statistics using all samples in the Shared Pool, and conducts compression and sifting over metrics and knobs.

In the third phase, *Recommender* acts over samples with reduced dimension and generates new configurations. Different from the *Sample Factory*, *Recommender* uses DDPG instead of GA. One key design decision in HUNTER is to warm-start the *Recommender* using all samples in the *Shared Pool* (see §3 for details). This allows us to take advantage of the massively parallelizable nature of GA to warm-start the powerful, finer grained, exploration process enabled by DRL. Similar to the first phase, these newly-generated configurations are tested by the Controller until the total wall clock time reaches a total time budget. The Controller then deploys configurations with the best performance on the user's instance.

Example. For each new tuning request, *Actor* uses the API provided by the CDB provider to create new CDBs from the resource pool according to the number of user requests, and copies backup of user's instance to these created idle instances to complete the cloning process. If the user does not request to tune using conventional benchmarks, such as Sysbench/TPCC, *Workload Generator* will build the workload by collecting the queries from user's instance in a time window set by the user. We avoid using real-time workload for knobs tuning because the real-time workload is inherently unstable, making knob performance feedback less reliable. Note that when the user requests standard benchmarks with previous *Rules*, HUNTER starts the processes from the *Search Space Optimizer* module with previous samples.

Then, *Actor* registers these CDBs and receives configurations to deploy knobs using the API. During the period of knobs deployment, *Actor* only starts a new workload execution after the former setting has played effect in that some knobs only take effect after restarting the database. If an instance fails to boot because of awful configurations, *Actor* will skip current workload execution and

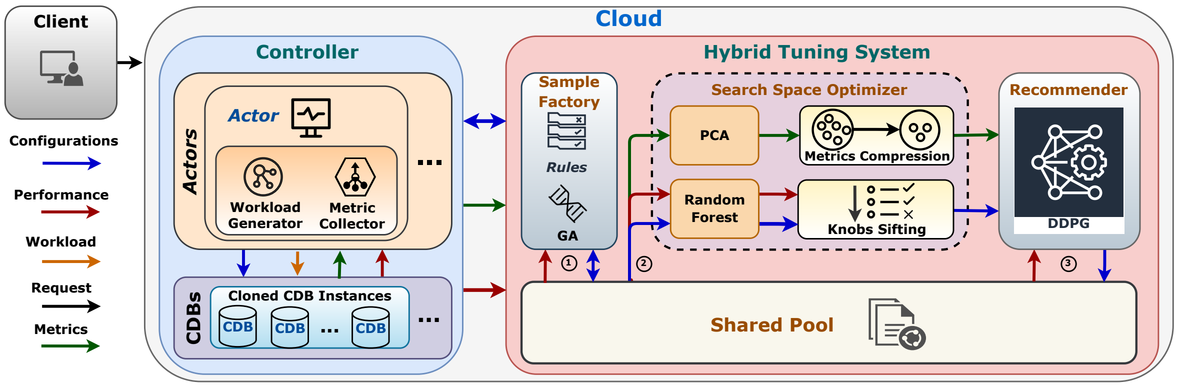


Figure 2: Overview Architecture and Workflow of HUNTER.

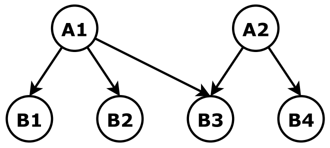


Figure 3: Example of Transactions Dependency Graph

give these configurations a poor performance (specifically, we set its throughput to -1000 and latency to infinity). Following, *Actor* concurrently executes the workload on each registered CDB. For stability and performance, we enable the CDB's warm-up function, which allows the buffer pool to be saved to disks when the CDB shuts down and reloaded from disks when the CDB restarts.

After that, *Metric Collector* starts to collect metrics during workload execution. Note that if the user selected the the real-world workload as the workload, the execution style would be the replay. Although the replay by the arrival order of SQL statements is a simple and reliable scheme, it is hard to get high throughput because of the low concurrency.

To tackle this problem, we propose a replay scheme based on the transactions dependency graph. This graph is built by the conflicts between transactions and the sequence in which they are done, with a transaction being executed only if it has no parent node. The graph's final structure is a directed acyclic graph (DAG). Figure 3 shows, for example, six transactions and their transactions dependency graph. *A1* and *A2* can be run simultaneously at the beginning. *B1* and *B2* can be conducted concurrently after *A1*, whereas *B3* can only be executed after *A1* and *A2*.

In addition, we use the point-in-time recovery tool provided by CDB to ensure the stability of each round of real-world workload replay by keeping the same start point.

2.2 Design Rationale

Availability and Cloned CDBs. To tackle the availability challenge, we avoid executing workload on the user's instance in each tuning step. Considering that tuning relies on the results of workload execution, it seems that the only solution is to implement the execution on a cloned CDB instance rather than the one that the user is using.

In HUNTER, we deploy a configuration on the user's instance only after this configuration is verified on cloned CDBs. The user experience thus can be improved by eliminating fluctuation and frequent restarting caused by semi-finished configurations. Furthermore, cloning the user's instance inspires us to design a parallelization scheme that can simultaneously stress-test these cloned CDBs with different configurations. Despite extra resource usage, parallelization can significantly reduce the inevitable runtime of workload execution.

These considerations inspire the design of *Actors*. Each *Actor* includes a *Workload Generator* module and a *Metric Collector* module, which manages the cloned CDBs and parallelization. *Actor* is given the API to clone the user's instance onto idle CDBs according to the replication factor set by the user. In our cloud environment, each CDB is always deployed as a pair of instances: a *primary instance*, which directly serves the user requests, and a *secondary instance*, which acts as a backup. The goal of the secondary instance is to ensure the security of the database without affecting the user's experience. As a by-product, this also makes it easier for an *Actor* to access the backup and create cloned CDBs. In addition, *Actor* has the permission for workload execution on cloned CDBs, whereas the workload is copied by *Workload Generator.* If workload execution succeed, *Actor* would gather the metrics and performance via *Metric Collector.* Similar to the data analysis process conducted by DBAs, our *Actor* uses the command "show status" (for MySQL) and "select * from pg_stat_archiver, pg_stat_bgwriter, pg_stat_database, *etc.*;" (for PostgreSQL) to gather information.

Hybrid Design. Facing the diversity of personalized requirements such as disabled knobs and adjustable range of knobs on different workloads, we find that transferring knowledge from historical measurements over a different workload or a different search space to be incredibly challenging. As a result, one key design decision that we made is to conduct online explorations for each user's workload. Unfortunately, this poses the cold start challenge since our exploration for each user's workload needs to start from scratch. This challenge inspired our hybrid design.

The first phase of the hybrid design uses GA to quickly explore the search space, at potentially coarser granularity compared with DRL. GA is coarser since it uses less information, but it can concentrate more on developing samples with high short-term gains.

As shown in Figure 4(a), the throughput of GA beyond 6.28×10^4 txn/min after 15 hours, which is 0.99×10^4 txn/min higher than that of BestConfig at this time. Similar convergence velocity is reflected on latency shown in Figure 4(b).

On the other hand, the best performance that can be achieved by GA is lower than other methods, especially for CDBTune employing DDPG, a powerful DRL algorithm. Thus, in the second phase of our hybrid design, we switch to an improved version of DDPG and *warm-start* it using samples collected via GA in the first phase. In our architecture, we design the *Sample Factory* module for GA to generate early samples and design the *Recommender* module for DDPG to achieve the final recommendation.

To further accelerate training and recommendation, we are inspired by OtterTune and try to reduce the complexity of the search space. The search space consists of the state space and the action space, whose complexities are decided by the dimension of metrics and the number and adjustable range of knobs respectively. In HUNTER, we use PCA to compress the metrics space and use RF to rank knobs by their influence on the performance and only select key knobs. To this end, we design the *Search Space Optimizer* module to further speed up tuning.

3 HYBRID TUNING SYSTEM

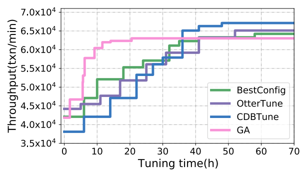
In this section, we will specify the core modules of Hybrid Tuning System including *Sample Factory, Search Space Optimizer*, and *Recommender.*

3.1 Sample Factory

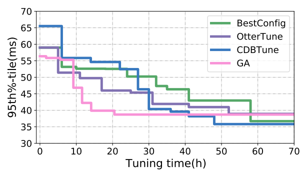
As the first phase of the tuning system, *Sample Factory* includes the *Rules* and GA, where *Rules* record the user's personalized requirements for knobs and GA generates high-quality samples based on *Rules* to overcome the cold start problem.

The cold start problem is sensitive to the quality of samples. The existing methods employ Latin HyperCube Sampling (*e.g.*, BestConfig, OtterTune) or Random Sampling (*e.g.*, CDBTune) to select samples by the try-and-error style, where the set of samples is expected to approximate a path that directs to the optimal configurations. However, if the superior solution not in the current samples, the capture of optimal configurations is a tricky task, no matter what sampling method is used. Therefore, the generation of high-quality samples is prime.

GA earns our trust because of its performance that shown in Figure 4. In fact, GA is a self-organizing, adaptive artificial intelligence technique for solving problems by simulating the processes and mechanisms of biological evolution, based on Darwin's idea of evolution [40]. It draws on a number of phenomena in evolutionary biology [14], such as evolution, mutation, natural selection, hybridization, *etc.* Because this algorithm does not require any prior knowledge, it is competent to tune the database configurations with few samples. Based on this, we set GA to perceive the workload first and generate samples into the *Shared Pool.* To show the quality of samples generated by GA, given within 300 steps for tuning on TPC-C, we rank the samples by their performance in terms of throughput compared with that of 3 methods. As shown in Figure 5, for the samples generated by different methods, GA has 32.75% and 39.75% samples whose throughput is lower within 10% and 10%-20% respectively than the best one. These ratios are much higher than those of other methods in the same segment. GA makes full use of the similarity between high-performance parameters to learn quickly through crossover and mutation. Although it is difficult to find the optimal configuration at more steps, it can get superior samples in the preliminary stage.



(a) MySQL(TPC-C)



(b) MySQL(TPC-C)

Figure 4: Performance Change with Increasing of Tuning Time.

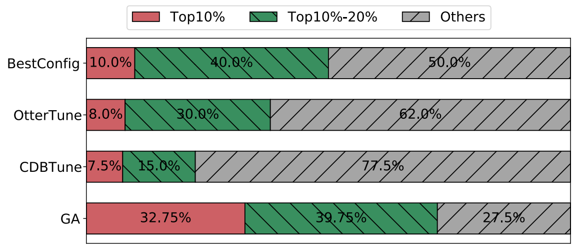


Figure 5: The distribution of performance in terms of throughput for samples generated by BestConfig, OtterTune, CDBTune and GA on TPC-C within 300 steps.

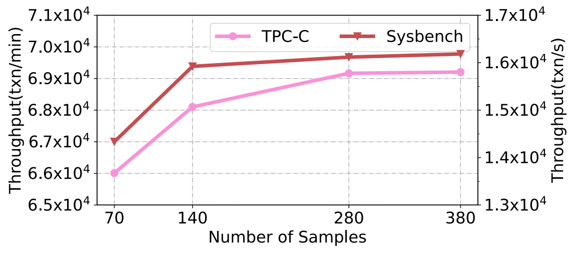


Figure 6: The changes of the best performance with the different number of samples generated by GA – The left and right Y axis represent the throughput of TPC-C and Sysbench respectively**.**

To obtain a reasonable number of generated samples, we perform 10 hours DRL tuning with 65 knobs using different number of samples acquired by GA. The changes of the best performance are shown in Figure 6. We see that, for TPC-C and Sysbench, performance improves as the number of samples increases, but reaches a plateau at 140. Although there is a minor increase after 140, the benefits from the increment will be outweighed by the cost of obtaining the samples. Consequently, we use GA to generate 140 samples in this paper.

Based on the principles of GA, we model the process of knobs tuning. Let $K = \{k^1, k^2, ..., k^m\}$ denotes m knobs to be adjusted. $K_i = \{k_i^1, k_i^2, ..., k_i^m\}$ represents the i-th configuration for CDB. We assess K_i by CDB's performance. Our goal is to find a K_i that makes performance superior.

Following the terminology in GA, K_i is called an *individual* and multiple *individuals* constitute a *population*. GA will calculate the fitness of each *individual* by a fitness function and generate new *individual* by the selection, crossover, and mutation strategies. This process will iterate until a maximum number of iterations or a satisfactory fitness level. We describe this process in detail below.

Initialization. We set a *population* $POP = \{K_1, K_2, ..., K_n\}$ consisting of n initial samples (*i.e.*, configurations) generated by a random strategy.

Fitness Function. To evaluate the fitness of an *individual*, we incorporate the throughput and latency to the function. Formally, the fitness function is defined as:

$$f(K_i) = \alpha \frac{T_{cur} - T_{def}}{T_{def}} + (1 - \alpha) \frac{L_{def} - L_{cur}}{L_{def}} \tag{1}$$

where T_{cur} and T_{def} are the throughput generated by the current and default configuration respectively, L_{cur} and L_{def} are the latency generated by the current and default configuration respectively, $\alpha \in [0, 1]$ is the hyper-parameter to control the attention between throughput and latency. The larger the α is, the more the algorithm will prefer high throughput. Otherwise, it will prefer low latency. In practice, we give α as a user-adjustable parameter in *Rules*. In this paper, we set α=0.5 because we pursue the gains from both throughput and latency.

Selection Strategy. GA selects *individuals* according to the principle of survival of the fitness. K_i with higher fitness will have a higher probability p_i of being selected. Formally, the selection probability p_i is defined as:

$$p_i = \frac{f(K_i)}{\sum_{j=1}^{n} f(K_j)} \tag{2}$$

Crossover Strategy. To generate a new *individual*, GA selects K_i and K_j by the above selection strategy and hybridizes them. Let K_i^a denote a subset of K_i with cardinality a, where $K_i^a = \{k_i^1, k_i^2, ..., k_i^a\}$, $K_i^{m-a} = \{k_i^{a+1}, k_i^{a+2}, ..., k_i^m\}$ and $a \in (0, m) \cap \mathbb{Z}$. Then, the new *individual* $\tilde{K}_k = K_i^a \cup K_j^{m-a}$.

Mutation Strategy. To widely explore the search space, GA intends to change each element of $\tilde{K}_k$ by the probability β and generates a new *individual* $\hat{K}_k$ from $\tilde{K}_k$.

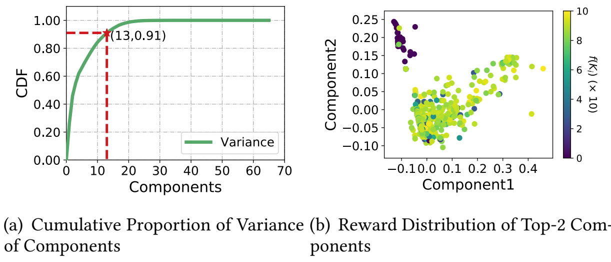
Based on this, we describe the entire GA in Algorithm 1, where K_{BEST} has the highest p in the current POP. With the help of GA, we can gather more samples with high performance.

3.2 Search Space Optimizer

Besides the sample's quality, the data dimensionality is another factor influencing DRL efficiency because samples with high dimensionality may cause big search space for decisions. To reduce this search space, we refer to OtterTune that compresses metrics and sifts knobs. In the implementation, we adopt PCA and RF to complete the state compression and knobs sifting respectively. The details of these methods are below.

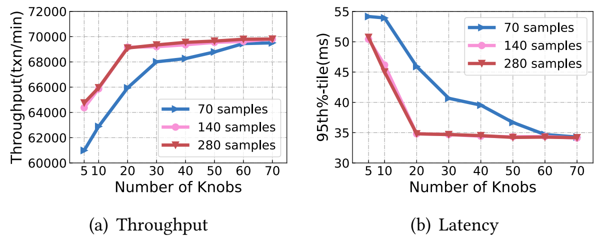
Algorithm 1: Genetic Algorithm

```
1  Execute Initialization to generate $POP$;
2  while Number of samples is not enough do
3      Execute Selection Strategy and save $K_{BEST}$ into $POP_i$;
4      for $k = 1$ to $n$ do
5          Execute Selection Strategy ;
6              Execute Fitness Function ;
7          Execute Crossover Strategy ;
8          Execute Mutation Strategy ;
9          Save new individuals into $POP_j$;
10     end
11     $POP = POP_i + POP_j$;
12 end
```



(a) Cumulative Proportion of Variance of Components (b) Reward Distribution of Top-2 Components

Figure 7: Components selection and effect of PCA on TPC-C.



(a) Throughput (b) Latency

Figure 8: Performance changes with varying tuning knobs on TPC-C.

3.2.1 Metrics Compression.

PCA is a machine learning method that is commonly used to reduce the dimension of data. It transforms a large set of variables into a smaller set that retains most of the information [31, 41]. The training for PCA is very fast, which is why we prefer it.

In our case, PCA transforms the data $X \in \mathbb{R}^{u \times l}$ into the low-dimensional data $Z \in \mathbb{R}^{v \times l}$, where X is the original metrics, u is the dimension of the metric, l is the number of metrics, v is the dimension of the compressed metric and $v < u$. Let $P \in \mathbb{R}^{v \times u}$ denote a transformation matrix. Formally, the transformation can be defined below.

$$Z = PX, \tag{3}$$

where each new variable called component in Z is a linear combination of the original variables in X. P consists of v eigenvectors of $S = XX^T$, where these eigenvectors correspond to the top-v eigenvalues. Therefore, different components are orthogonal to each other. It is instrumental in identifying in which direction the data value has the greatest influence, resulting in a beneficial adjustment. To reduce the inaccuracy caused by empirical judgment, we determine v based on the CDF that is the cumulative distribution function of variance. As shown in Figure 7, we demonstrate the components selection and effect of PCA on TPC-C. In Figure 7(a), we can see that CDF reaches 91% when the number of components is 13, where these components are arranged in descending order of the corresponding variance. Generally, when the CDF beyond 90%, the components are considered to cover the representative data [15]. Thus, we set $v = 13$ for tuning on TPC-C. Furthermore, we depict the reward in Equation 1, *i.e.*, $f(K_i)$ by top-2 components in Figure 7(b), where $\alpha = 0.5$ and components are regularized. It is easy to find that the samples owning Component 1 and Component 2 can be easily distinguished by different rewards, resulting in the reduction of learning time of DRL.

3.2.2 Knobs Sifting. Generally, we choose the number of knobs based on two principles: 1. DBA's advice. 2. empirical results. In practice, we select the RF algorithm to capture the empirical results. RF is a supervised machine learning method to deal with classification and regression problems. It can be deemed a voting algorithm consisting of multiple decision trees, used to calculate the importance of features [20, 28].

In our case, we select the classification and regression tree (CART) to construct the decision tree. Our RF algorithm is composed of 200 CARTs and adopts the majority voting principle. Each CART represents a classifier to judge the importance of knobs. Note that the process of CART construction is also its process of training, where the input of features is a subset of configurations and the labels are their corresponding performance. Formally, let $K \in \mathbb{R}^{m \times n}$ denote n original configurations. The subset of configurations is $C \in \mathbb{R}^{g \times n}$, where g-dimensional features are randomly sampled from m-dimensional features in K and $g < m$. The C of each CART is individual and different. In fact, it means exploring the importance of each knob in different combinations of knobs. Based on C and its performance, the CART builds tree construct from root node by Gini impurity that the best feature (*i.e.*, knob) splits from the node to generate new leaf nodes.

Based on the built CART, we can count the value reduction of Gini impurity for each knob in C. The average value of reduction of 200 CARTs can indicate the importance of each knob. Finally, we sift knobs based on these average values, *i.e.*, importance order.

In the implementation, we rank 70 knobs with $n = 70, 140, 280$ on TPC-C, where knobs are selected by the senior DBA. The performance changes with varying tuning knobs are shown in Figure 8, where knobs are arranged in descending order of importance. It is easy to find that the improvement of throughput and the reduction of latency will slow down when the number of knobs equals 20 in most cases. It means that tuning top-20 knobs will bring similar profits compared with tuning all knobs. Thus, we can rely on RF to select the important knobs and use them only for tuning. In addition, we find that the performance is almost the same when $n = 140$ and $n = 280$. Meanwhile, their performance is better than that of $n = 70$. This phenomenon is consistent with the principle that more samples bring better performance in the machine learning model. It also indicates that we can select a scale of samples to guarantee the effect of knobs sifting. In this paper, we select the top-20 knobs based on at least 140 samples generated by *Sample Factory*.

Compared with the LASSO algorithm, RF can instinctively capture the effects between knobs by its hierarchical structure rather than manually adding polynomial combinations to test correlations. It results in RF being able to generate an accurate importance score for each knob instead of setting the score of most knobs to 0. The accurate importance scores allow us to confidently use more knobs facing various rules predefined by users.

3.3 Recommender

Although the DDPG algorithm does not perform well in the early tuning, it has the best upper bound of performance shown in Figure 4, we select DDPG to learn and recommend the optimal configuration based on the dimension-reduced metrics and selected knobs.

DDPG is an actor-critic, model-free algorithm based on the deterministic policy gradient [22]. In the family of DRL methods, the deterministic policy is a compromise strategy that makes every decision based on the previous decision. Although this strategy may miss the best solution, it can ensure efficiency in practical application. To model our case by the DDPG algorithm, we abstract and introduce the six factors as follow:

Agent: *Agent* is the DDPG model, which decides the *Policy* by *Reward* and *State* from *Environment* and recommends *Action*. It emphasizes how to tune the action according to the environment in order to get the maximum reward.

Environment: *Environment* is the tuning environment, *i.e.*, CDB. In the implementation, we use Controller to manage multiple CDBs, making the algorithm decoupled from the environment.

State: *State s* is the representation of CDB gathered by the current observation of the environment, *i.e.*, the metrics including number of flush statements executed, number of threads activated, *etc.* In the implementation, they are gathered by the metric collector in the Controller. Before inputted to the *Agent*, they will be compressed by PCA.

Action: *Action a* is the configuration of CDB that contains values of top-k selected knobs.

Reward: *Reward r* describes the gap between the performance generated by the current *Action* and the defaulted performance. In the implementation, the reward function is calculated in the same way as the fitness function in Equation 1.

Policy: *Policy* $\mu(s)$ is the parameters of *Agent*, which is used to map s into a. It will be changed by *Reward* and make *Reward* as large as possible. In the implementation, the *Policy* is the parameters of a neural network.

For the actor-critic structure of DDPG, we implement two neural networks consisting of DDPG-Actor and DDPG-Critic. DDPG-Actor learns a policy function $a = \mu(s|\theta^{\mu})$, where μ maps s to a with weights θ^{μ}. Based on the recommended a, DDPG-Critic learns a function $Q(s, a|\theta^{Q})$ to evaluate the μ with *Reward*. Note that DDPG employs an experience replay mechanism to get samples from *Shared Pool* and uses these samples to learn μ and Q.

In fact, although the superior samples can help DDPG acquire better performance, it is hard to reduce the recommendation time without modification, because DDPG, as a DRL algorithm, is difficult to converge quickly [2, 16]. Therefore, with a large number of samples representing sub-optimal solutions, we replace the exploration strategy of DDPG to our strategy named Fast Exploration Strategy (FES). In the implementation, we change the selection of *Action A* as follow:

$$A = \begin{cases} A_c, \\ A_{best}, \end{cases} \tag{4}$$

where A_c denotes *Action* generated in current step, A_{best} denotes *Action* that brings the best performance plus with a random value. We define $P(A_c)$ and $P(A_{best})$ as the probability of selection of A_c and A_{best} respectively. t denotes the number of tuning steps. They abide by the following rules:

$$P(A_c) + P(A_{best}) = 1, \tag{5}$$

$$\lim_{t \to \infty} P(A_c) = 1, \tag{6}$$

$$\frac{\mathrm{d}P(A_c)}{\mathrm{d}t} > 0, \tag{7}$$

When $t = 0, P(A_c) = 0.3$. This improvement enables the model to select configurations with the best performance in the preliminary stage of tuning, and forces the model to explore based on relatively better configurations.

4 SUPPLEMENT OPTIMIZATION SCHEMES

To further improve user experience and tuning efficiency, we optimize the model reuse based on the above designs.

Online Model Reuse. Although HUNTER completes tuning for target workloads by online style, we still provide the optimization scheme of model reuse mechanism with historical data. We set up a matching module in the *Space Optimization* module. After knobs and states of the given workload are optimized, the matching module judges whether there is a historical workload with the same key knobs and dimension of the compressed state. If it exists, HUNTER will load its parameters of *Recommender* to the current *Recommender* and complete the tuning by fine-tuning style.

Model Reuse. As a commonplace, the user usually selects or changes the instance types according to economic status, where the instance types comprise varying combinations of CPU, memory, storage, and networking capacity. This behavior will transfer the user's resources to other CDBs, but not change the content of instance in the Controller and the parameters in the Hybrid Tuning System. If a user requests to tune configurations on the instance which has been changed in types, HUNTER will directly run the *Search Space Optimizer* and *Recommender* modules with their previous parameters, skipping *Sample Factory*.

5 DISCUSSIONS

HUNTER aims to find a superior configuration without any prior knowledge in a short time. We also try to deal with other problems in the tuning system. In this section, we discuss the HUNTER's room for improvement in specific scenes and future work.

Workload Drift and Historical Data Reuse. Workload drift is a challenging problem. However, is it necessary to tune knobs whenever the workload drift occurs? Considering the payment of tuning, the user may only seek a tuning if the performance does not satisfy his or her needs. Thus, instead of keeping tuning knobs during the life of instance, we simplify this issue to whether the tuner can learn from historical data to recommend knobs with high performance in a short time before the workload drift occurs again.

In fact, even if the drift occurs, models learned from historical data could be reliable and responsive in our situation. As shown in the Figure 10(b), we design a workload drift scene. At 48h, we switch the Production workload collected from 9:00 am to the counterpart collected from 9:00 pm and observe the performance changes of different algorithms facing this drift. The results show that the learning-based approaches can retrieve superior configurations in less time than the search-based approaches.

Sensitive Queries. Tuning may improve most queries but regress some sensitive ones. A simple extension could be to consider tail latency, *e.g.*, focusing on optimizing tail-99% latency instead of tail-95% latency. We are optimistic that our framework can be extended to these metrics by changing the objectives. We hope that our paper will serve as a catalyst for future research in this area.

In addition, tuning knobs is a multi-objective optimization problem rather than a single-objective optimization problem, considering both the latency of sensitive queries and the overall performance. It is beyond the scope of this work but is exciting future work.

Warm-up Database. The automatic database tuning depends on credible performance. The hit ratio of buffer pool plays an important role in this reliability. Generally, the large the data set is, the longer the warm-up time will be. Facing a large dataset, it's necessary to keep the warm-up time within an acceptable range.

In the implementation, enabling the CDB warm-up function, the longest warm-up time in our experiments is about 5 seconds for Sysbench. With about 3 minutes for workload execution, the ratio is 1:36. We scale up Sysbench by a factor of 10, the warm-up time increases to about 35 seconds and the ratio becomes 1:5. It is still an acceptable ratio for users. Nevertheless, it is necessary to study warm-up solutions for larger datasets.

Warm-up DRL Model and Representative workload. How to make DRL models converge faster without losing the optimal solution is a difficult task. We have tried two traditional approaches, either replacing the DRL model or optimizing the fetch of samples. However, while the boost in accuracy that a sampling method such as HER [1] can offer does not improve the pace of convergence, DDPG is the rational DRL model for balancing exploration efficiency and exploration accuracy. Therefore, we choose data augmentation technology to improve the quality of samples and expect the DRL model to accelerate converge in the improved environment. Based on this motivation, GA is proposed to generate competent samples and used to warm-up DRL. Frankly, we're still lucky to find an effective scheme to warm-up the DRL model, as the reasons for GA's ability to collect high-quality samples remain obscure.

Finding representative workloads is significant but challenging work. However, this issue can only be discussed in the future for defining the similarity between them is still an open problem.

Tuning Instance Size. In principle our current tuning mechanism allows us to alter the instance size as a parameter; however, in practice, we found that it is often more efficient and friendly to end-users to tune it based on current resource usages. This is due to the fact that the cost of data migration associated with changing the instance size is larger than the cost of the tuning knobs for CDB providers. In addition, it is a challenge to get users to actively take the risks associated with size changes.

6 EXPERIMENTAL EVALUATION

In this section, we compare HUNTER with state-of-the-art methods on cloud databases, where the methods include BestConfig [48] (search-based method), OtterTune [37] (pipeline method using Gaussian Process), CDBTune [46] (end-to-end method with DDPG), QTune [21] (DS-DDPG), and ResTune [47] (meta-learning). HUNTER's Hybrid Tuning System is implemented using TensorFlow and Scikit-Learn. The Controller is implemented using CDB's API.

For standard benchmarks, our evaluation is implemented on two kinds of databases, *i.e.*, MySQL (v5.7) and PostgreSQL (v12.4), where the MySQL instance with 8 cores and 32GB RAM and the PostgreSQL instance with 8 cores and 16GB RAM. For real-world workloads, we run on MySQL (v5.7) instances with 4 cores and 16GB RAM. *Actors* and Hybrid Tuning System are deployed on servers with 48 cores CPU, 128GB RAM, and 200GB disk.

In the experiments, according to the settings of CDBTune in offline training, we initialize 65 knobs and 63 metrics. HUNTER will select key knobs from them and compress the above metrics in the online process. For the sake of fairness, the initialization parameters of *Recommender* are random and the *Shared Pool* is empty for each workload.

We first describe the workloads that we used. Then, we evaluate HUNTER and compare it with BestConfig [48], OtterTune [37], CDBTune [46], QTune [21], and ResTune [47]. After that, ablation experiments are conducted to confirm the effectiveness of modules used in HUNTER and the availability of different warm-up methods. Next, we carry out experiments to verify the benefits of the parallelization scheme for different tuning systems. Note that HUNTER and HUNTER-* represent our proposed system using 1 or * cloned CDBs respectively. Finally, we verify the recommendation time (*i.e.*, the tuning time when the optimal configuration is obtained) and performance via the different model reuse schemes and instance types varying, respectively.

Our experiments are conducted on 5 different workloads: Sysbench read-only (RO), Sysbench write-only (WO), Sysbench read-write (RW), TPC-C, and real-world workload (*i.e.*, Production). The details of all workloads are shown in Table 2.

Sysbench is a popular open source scriptable multi-threaded benchmark tool based on LuaJIT. We used three of the commonly used workloads in Sysbench – Sysbench RO, Sysbench WO, and Sysbench RW. Under the Sysbench workloads, we have 8 tables and each table contains 8 million records (about 8GB of data in total) with 512 threads connected.

Table 2: Details of Workloads

Name	Sysbench			TPC-C	Production
	RO	RW	WO		
Size(GB)	8	8	8	8.97	256
#Thread	512	512	512	32	/
R/W Ratio	1:0	1:1	0:1	19:10	20:29

TPC-C is the current industry standard for evaluating the performance of OLTP systems, which involves a mix of five concurrent transactions of different types and complexity either executed online or queued for deferred execution. Under the TPC-C workload, we have 50 data warehouses (about 8.97GB of data) and 32 clients.

Production is a real-world workload about business of education. We take the queries gathered at 9:00 am as a workload and replay it concurrently to evaluate its performance in terms of time cost. This is a read-write workload with 222 tables and around 250GB of data in total.

6.1 Comparisons with State-of-the-Arts Methods

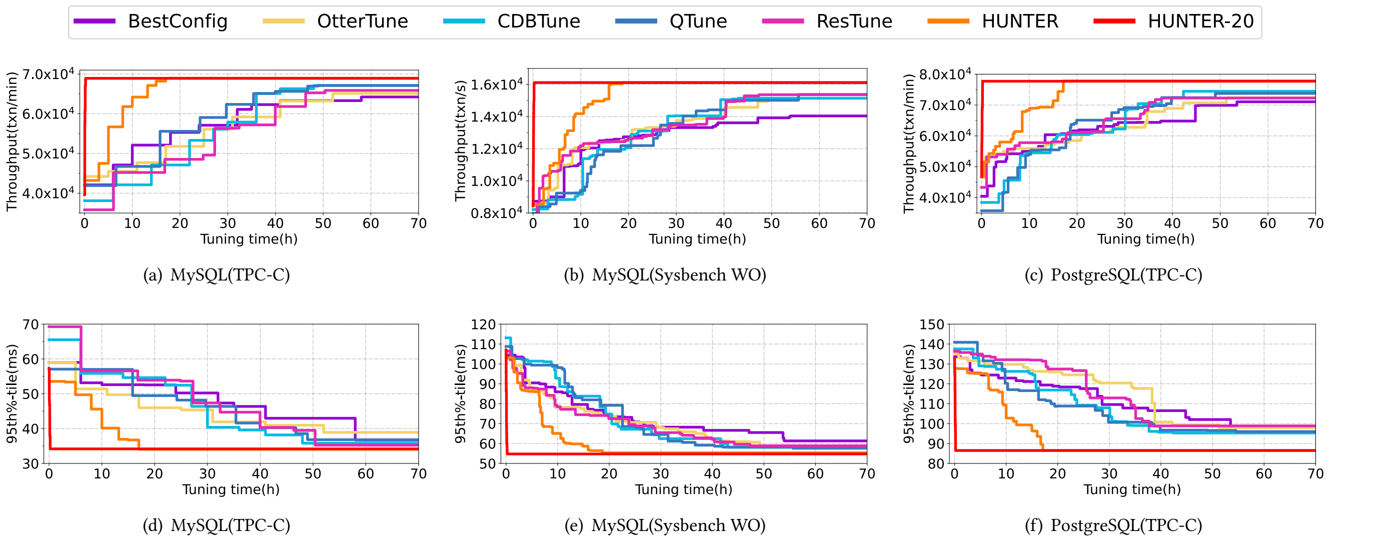
To compare the performance of state-of-the-art methods, all methods start without any prior knowledge. Each method has about 70 hours to explore and recommend configurations.

Figure 9(a) and Figure 9(d) show the best throughput and the best latency curves obtained by each method on MySQL testing TPC-C. As shown in Figure 9(a), HUNTER-20 has the fastest velocity to approach the optimal throughput in 2.1 hours, which is 22.8 times faster than CDBTune. In addition, For the similar optimal throughput, HUNTER spends 17 hours to achieve tuning with one instance, which is 2.8 times faster than that of CDBTune. Note that in the 70 hours, the optimal configurations recommended by other methods do not exceed the peak of HUNTER-20 or HUNTER. Figure 9(d) shows the same dominant of HUNTER-20 in tail latency. Its optimal latency is reduced to 34ms at 2.1 hours. Meanwhile, HUNTER can reach the similar latency in 17 hours.

Figure 9(b) and Figure 9(e) show the performance executing Sysbench WO on MySQL, where the results are similar to the above figures. HUNTER-20 obtains the best configuration for throughput and latency in 2.3 hours, which is 18.7 times faster than that of CDBTune. HUNTER reaches similar performance in 23 hours, which is 1.9 times faster than that of CDBTune.

On the PostgreSQL database, the results for testing TPC-C are shown in Figure 9(c) and Figure 9(c). HUNTER-20 obtain the best throughput and latency when the tuning time is at 1.9 hours, which is 22.1 times faster than that of CDBTune. In addition, HUNTER reaches similar performance in 19 hours, which is 2.5 times faster than that of CDBTune.

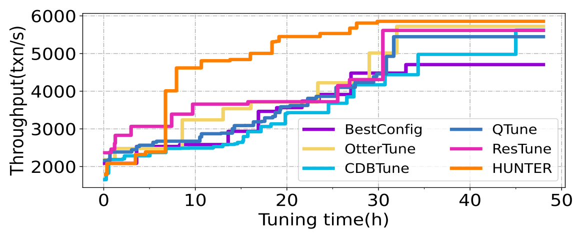
As shown in Figure 9, HUNTER is able to obtain optimal performance in less time by 55% - 65% compared with the state-of-the-art performance tuning systems. Especially with 20 cloned CDBs, the HUNTER-20 is able to achieve the optimal performance in 2.5 hours, which greatly reduces the recommendation time by 94% - 95%, resulting in the tuning for target workloads in an acceptable time.



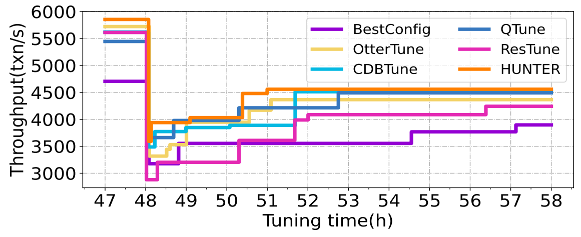
(a) MySQL(TPC-C) (b) MySQL(Sysbench WO) (c) PostgreSQL(TPC-C)

(d) MySQL(TPC-C) (e) MySQL(Sysbench WO) (f) PostgreSQL(TPC-C)

Figure 9: Performance Evaluation with Increasing of Tuning Time – Comparisons of state-of-the-art performance tuning systems on CDBs (*i.e.*, MySQL and PostgreSQL) with workloads (*i.e.*, TPC-C and Sysbench WO).



(a) Throughput on real-world workload, *i.e.*, Production collected at 9:00 am.



(b) Throughput on drifted workload, *i.e.*, Production collected at 9:00 pm.

Figure 10: The changes of throughput with real-world workload drift – Comparisons of state-of-the-art tuning systems with 1 cloned instance.

Figure 10 shows the curves of the best throughput change on a real-world workload, *i.e.*, Production collected from 9:00 am. Obviously, at the 8-hour mark, HUNTER seizes the lead and holds it to the 48-hour mark. At 48-hour mark, the workload drift occurs, where the drifted Production is collected from 9:00 pm. As shown in Figure 10(b), all throughput metrics plummet to below 3700 (txn/s). However, the learned-based approaches all bounce back quickly, with HUNTER finding the best configuration quickly.

Table 3: Ablation Study on MySQL with TPC-C – An ablation comparison uses different modules of HUNTER.

Module					Performance		Recommendation
DDPG	GA	PCA	RF	FES	T (txn/min)	L (ms)	Time (h)
✓					67012	35.8	48
✓	✓				69418	34.8	37
✓	✓	✓			68346	35.4	22
✓	✓		✓		68862	34.7	32
✓	✓			✓	69950	35.4	27
✓	✓	✓	✓	✓	68942	34.0	17

6.2 Ablation Studies

To further explain how HUNTER produces the best performance, we compare the effect of each module of our design on different workloads. We conduct our experiments under different combinations of DDPG, GA, PCA, RF, and FES. The DDPG module is equivalent to the CDBTune system when used as a core module on its own, while HUNTER uses all these modules. We use T and L to represent the throughput and 95%-tail latency respectively. All performance in tables is the optimal value achieved in 72 hours with one cloned CDB instance. The recommendation time in tables denotes the earliest time to obtain the optimal values for T and L.

Table 3 shows the ablation experiments' results about stress-testing on MySQL with TPC-C. The performance of combinations consisting of DDPG and any module is better than that of DDPG. This shows that the design of HUNTER with modules is effective. The best performance comes from the combination of DDPG, GA, and FES. Note that the usage of GA and FES improves performance and tuning velocity compared to CDBTune, where the recommendation time reduces 43.8%. In addition, the fastest velocity occurs with the combination of all modules. HUNTER leads to faster exploration by search space compression, albeit loss of performance.

Table 4: Ablation Study on MySQL with Sysbench (RW) – An ablation comparison uses different modules of HUNTER.

Module					Performance		Recommendation
DDPG	GA	PCA	RF	FES	T (txn/s)	L (ms)	Time (h)
✓					4230	118.3	47
✓	✓				4680	109.3	38
✓	✓	✓			4592	110.2	32
✓	✓		✓		4601	110.1	27
✓	✓			✓	4783	107.6	33
✓	✓	✓	✓	✓	4703	108.1	21

Table 5: Ablation study on PostgreSQL with TPC-C – An ablation comparison uses different modules of HUNTER.

Module					Performance		Recommendation
DDPG	GA	PCA	RF	FES	T (txn/min)	L (ms)	Time (h)
✓					74456	95.7	43
✓	✓				77212	87.7	32
✓	✓	✓			76201	88.5	24
✓	✓		✓		76892	89.2	23
✓	✓			✓	78456	85.7	27
✓	✓	✓	✓	✓	77816	86.5	19

Table 6: Ablation study with TPC-C – An ablation comparison by different DRL warm-up modules.

Database	Module			Performance		Recommendation
	DDPG	HER	GA+	T (txn/min)	L (ms)	Time (h)
MySQL	✓		✓	68942	34.0	17
	✓	✓		67351	36.0	39
PostgreSQL	✓		✓	77816	86.5	19
	✓	✓		74532	95.3	31

Because the throughput reduces 1.5% and the latency increase 1.6% respectively while the recommendation time greatly reduces 40%, we think the design of HUNTER makes a rational trade-off.

For another workload, *i.e.*, Sysbench RW, the results are shown in Table 4. These results show the same tendency that the combination of DDPG, GA, and FES achieves the best performance and the combination of HUNTER is more rational with the least recommendation time. The same tendency also reflects in Table 5 which shows the results on PostgreSQL with TPC-C.

Based on these results, we conclude that GA and FES make contributions to performance and recommendation time. Meanwhile, PCA and RF can further reduce the recommendation time. Note that the introduction of PCA alone lowers the performance, which means the metrics are related to the performance in DRL.

In addition, on MySQL and PostgreSQL with TPC-C, we compare the performance and recommendation time generated by warm-up modules, *i.e.*, HER [1] and GA+, where GA+ represents GA + PCA + RF + FES. As shown in Table. 6, GA+ is able to recommend better performance in faster time, regardless of the database. Therefore, we assert that the use of modules such as GA+ to warm-up DRL is a viable option.

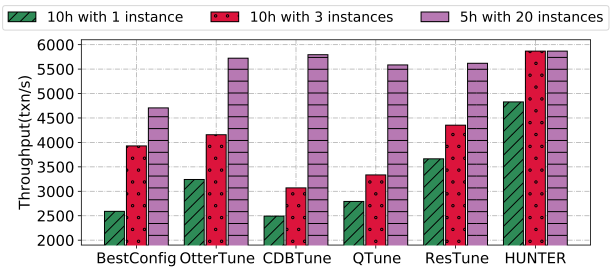


Figure 11: Throughput with different cost – Comparisons of state-of-the-art tuning systems with varying cost (*i.e.*, number of cloned instances and tuning time) on workload Production.

6.3 Effect of Parallelization

In this section, we explore the relationship between the number of cloned CDBs and benefits, *i.e.*, throughput and recommendation time, to provide a reference for the number of instances in parallelization scheme.

To reveal the benefit from parallelization of state-of-the-art methods, we verify the performance of methods when the number of instances changes. We conduct experiments on Production, where the conditions of experiments are 1 instance 10 hours, 3 instances 10 hours, and 20 instances 5 hours. The results are shown in Figure 11 and we summarized them as following: a) Under 1 instance 10 hours condition, HUNTER gets the best performance. b) Under 3 instances 10 hours condition, HUNTER not only leads the way in performance, but the advantage under 1 instance 10 hours remains. c) Under 20 instances 5 hours, all methods get similar performance.

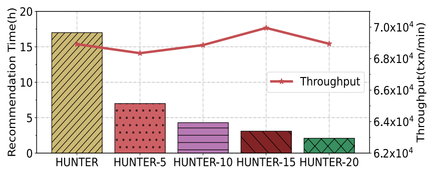
Figure 12(a) shows the changes of optimal throughput and recommendation time on MySQL with TPC-C by varying the number of cloned CDBs, where the number is 1, 5, 10, 15, and 20. As we can see, with the rise of the number of cloned CDBs, the recommendation time decreases. When the number reaches 20, the recommendation time decreases by 87.6% compared to the first case. Meanwhile, the HUNTER-* will terminate when its throughput exceeds 98% of the best performance of HUNTER. Thus, the change of throughput is relatively stable and parallelization would not significantly improve the performance of the optimal configuration.

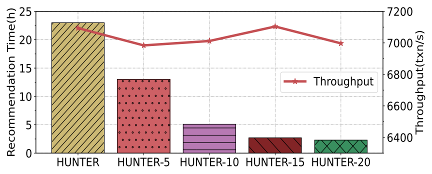
For another workload, *i.e.*, Sysbench RO, Figure 12(b) shows the same tendency that the recommendation time is reduced by 90% with 20 cloned CDBs compared to that with one cloned instance. For another database, *i.e.*, PostgreSQL, the results are similar. As shown in Figure 12(c), the recommendation time reduces by 90% with 20 cloned CDBs compared to that with only one cloned instance.

According to these results, we believe that the parallelization scheme is conducive to all methods. Based on the hybrid architecture, HUNTER can gain benefit from parallelization and achieve high performance faster with limited resources.

6.4 Effect of Online Model Reuse

In this section, we use two workloads, *i.e.*, Sysbench RW (4:1) and Sysbench RW (1:1), where the read/write ratios are 4:1 in Sysbench RW (4:1) and 1:1 in Sysbench RW (1:1). These two workloads have the same key knobs and dimension of compressed metrics, which


(a) MySQL(TPC-C)


(b) MySQL(Sysbench RO)

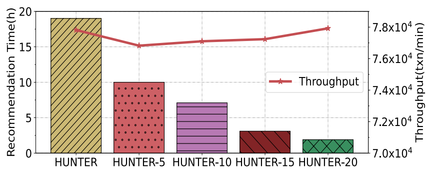
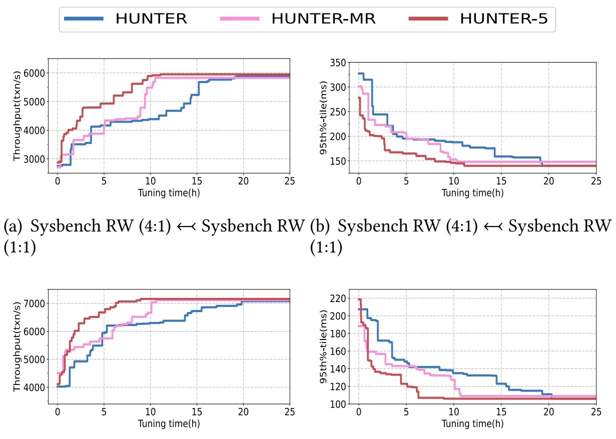

(c) PostgreSQL(TPC-C)

Figure 12: Throughput and Recommendation Time Evaluation – Show effect changes by varying numbers of cloned CDBs (*i.e.*, MySQL and PostgreSQL) with different workloads (*i.e.*, TPC-C and Sysbench RO).



(a) Sysbench RW (4:1) ⟻ Sysbench RW (1:1)
(b) Sysbench RW (4:1) ⟻ Sysbench RW (1:1)
(c) Sysbench RW (1:1) ⟻ Sysbench RW (4:1))
(d) Sysbench RW (1:1) ⟻ Sysbench RW (4:1)

Figure 13: Performance Evaluation on MySQL by Model Reuse Scheme – (a) and (b) test Sysbench RW (4:1) by the model trained with Sysbench RW (1:1). (c) and (d) test Sysbench RW (1:1) by the model trained with Sysbench RW (4:1).

Table 7: Cloud Database Instance Types

Instance Type	A	B	C	D	E	F	G	H
CPU (cores)	1	4	4	4	6	8	8	16
RAM (GB)	2	8	12	16	24	32	48	64

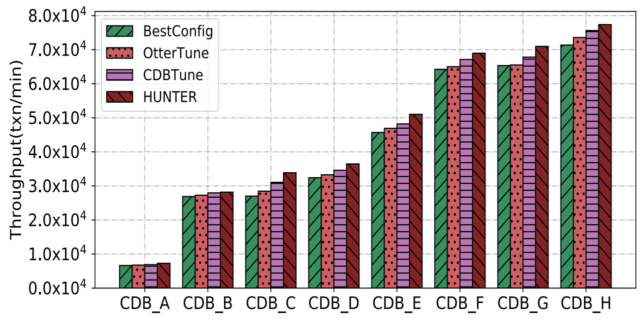


Figure 14: Performance Evaluation on different type of instances – Comparisons of state-of-the-art performance tuning systems on MySQL with TPC-C.

meets the requirements of the online model reuse scheme. Sysbench RW (X) ⟻ Sysbench RW (Y) represents that we fine-tune the model on Sysbench RW (X), where the model is trained on Sysbench RW (Y). We compare HUNTER with HUNTER-5 and HUNTER-MR (HUNTER runs with online model reuse scheme) in terms of performance on MySQL with these workloads.

Figure 13(c) and Figure 13(d) show the performance comparisons of HUNTER, HUNTER-5, and HUNTER-MR in the case of Sysbench RW (4:1) ⟻ Sysbench RW (1:1). As shown in Figure 13(c), although the peak value of HUNTER-MR is slightly lower than that of HUNTER, HUNTER-MR can achieve the optimal throughput faster than HUNTER by 8 hours, and approach the efficiency of HUNTER-5. The similar phenomenon also reflects on Figure 13(d). Figure 13(a) and Figure 13(b) show the same trends in the comparisons on Sysbench RW (1:1) ⟻ Sysbench RW (4:1). Although the optimal performance is weaker than that of HUNTER, the recommendation time for the optimal performance of HUNTER-MR is faster than that of HUNTER by 10 hours.

As a result, we think that the online model reuse scheme can further accelerate the tuning velocity. For the users lacking CDB resources and craving the velocity of tuning, it is a feasible solution. Furthermore, if there exists a swift method judging the similarity between workloads, the tuning time will further reduce.

6.5 Varying Instance Types

To further discuss on the impact of instance types on performance, we design experiments to tune configurations on the instances that have different types. As shown in Table 7, we list 8 kinds of instance types with different cores of CPU and capability of RAM. We select instance type F to train the model and fine-tune others on this model. The CDB_X represents CDB with the instance type X.

In this experiment, we train the model for 100 hours with workload TPC-C on CDB_F and use this model to tune instances in Table 7 with the same workload. For each instance, the tuning steps are 5 and the best performance is shown in Figure 14. Because the existing tuning systems have the same test conditions, we also show their model reuse effect under different instance types in Figure 14.

Compared with the existing tuning systems, HUNTER always has the performance advantage. Obviously, the richer resource of instance type owns, the better performance it has. Specifically, CDB_A shows a very low performance compared to other instance types. It is hard to improve performance by adjusting the knobs in that the loading workload has exceeded the capacity of CDB_*A*. With the improvement of instance's capability, the increments of performance will gradually be obvious. The further observation shows that the improvement of throughput is positively related to the number of CPU cores. Although the instance types of CDB_*F* and CDB_*G* are different, their performance is almost the same. The reason is that both CDB_*F* and CDB_*G* have enough capacity to load all the data into their memory on TPC-C workload, and the extra memory has almost no effect on the performance improvement. If the number of CPU cores increases, the throughput will increase again, *e.g.* CDB_*H*. However, the CPU utilization of CDB_*H* is only about 60% while the CPU utilization is about 100% in CDB_*F*. We believe that the resources of CDB_*H* are not fully utilized, resulting in unsatisfactory improvement with more CPU cores.

As a result, we believe that the model reuse scheme is susceptible to instance types. Although instances with better configurations may generate better performance, the workload characteristics and machine specification define performance's boundary.

7 RELATED WORK

Existing research on automatic DBMS [3, 9, 13, 18, 21, 23, 25, 26, 29, 30, 34, 37, 38, 46, 47] tuning can be classified into two categories: tuning the configuration knobs and tuning the physical design.

Database Configuration Tuning. IBM [32, 35] released a version of DB2 with a self-tuning memory manager that uses heuristics to allocate memory to the DBMS's internal components, and released the DB2 Performance Wizard tool [19] for automatically selecting the initial values for the configuration parameters. Tran *et al.* [36] used linear and quadratic regression models for buffer tuning. The DBSherlock tool [44] helps a DBA diagnose problems by comparing regions where the system was slow with regions where the system behaved normally based on the DBMS's time-series data on performance. The COMFORT tool [39] uses a technique from control theory that can adjust a single knob up or down at a time, but cannot discover the dependencies between multiple knobs. BerkeleyDB [33] uses influence diagrams to model probabilistic dependencies between configuration knobs, to infer expected outcomes of a particular DBMS configuration. The SARD tool [7] generates a relative ranking of a DBMS's knobs based on their impact on performance using a technique called the Plackett-Burman design. iTuned [10] is a generic tool that continuously makes minor changes to the DBMS configuration whenever the DBMS is not fully utilized, employing the Gaussian process regression for automatic configuration tuning. OtterTune [37] leverages the past experience data and collects new information to tune DBMS configurations. It exploits the Gaussian process regression and additionally introduces a feature-selection step to reduce the number of parameters which reduces the complexity of the configuration tuning problem. BestConfig [48] automatically finds a configuration setting that can optimize the performance of a deployed system under a specific application workload within a given resource limit. iBTune [34] employs a pairwise DNN that predicts request response time to tune the buffer pool size. CDBTune [46] first applies reinforcement learning to database parameter tuning, using end-to-end design to tune database knobs. QTune [21] also uses reinforcement learning to tune parameters, with the difference that it helps knobs tuning with the help of information from queries. Kanellis [17] proposed a method to reduce database knobs tuning time using database knobs importance ranking, which as an aid can improve the usability of parameter tuning. ResTune [47] employs meta-learning to optimize the resource utilization without violating SLA constraints on the throughput and latency requirements through knobs tuning.

Physical Design Tuning. Kraska *et al.* [18] propose that indexes are models, where the B+-tree index can be seen as a model that maps each query key to its page. FITing-tree [13] provides strict error bounds and predictable performance using piece-wise linear functions and supports two kinds of data insertion strategies. Alex [9] is flexible for balancing the trade-offs between space and efficiency by reserving scattered spaces for inserted keys which are put into the position predicted by the model directly. Wu *et al.* [43] propose a succinct secondary indexing mechanism called Hermit, which leverages a tiered regression search tree to capture the column correlations and outliers. Idreos *et al.* [5, 6] show that the LSM-tree can be constructed from fundamental components, and the learned cost model can guide the construction directions. Dutt *et al.* [11] propose a regression model on multi-dimensional numerical range predicates for selectivity estimation. Wu *et al.* [42] propose a learning-based method for workloads in shared clouds named CardLearner, which extracts overlapped subqueries from workloads, and classifies them according to the structure. Marcus [27] tries to use learning query optimizer instead of traditional optimizer and exceeds the performance of traditional optimizer in some scenarios. RTOS [45] uses a two stage training to generate better join order on latency with a well-designed neural network structure to address the problem that the above two methods cannot support updates on schemes.

8 CONCLUSION

In this paper, we propose an online cloud database hybrid tuning system called HUNTER, which recommends configurations for high performance to meet users' personalized requirements in a short time. As the core module of HUNTER, Hybrid Tuning System is devised to (1) generate high-quality samples, (2) optimize the search space by metrics compression and key knobs selection, and (3) achieve fast online configurations recommendation. In addition, we design a clone and parallelization scheme, resulting in fast configurations exploration without performance fluctuation for users. Our evaluation shows that, given the same time budget and resources, compared with state-of-the-art tuning systems, HUNTER can improve performance and greatly reduce recommendation time up to 2.8× and 22.8× speedups using 1 and 20 cloned CDBs respectively.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback and comments. We would also like to thank Tencent Technology (Shenzhen) Co., Ltd. for experimental environment. This work is supported by the Innovation Group Project of National Natural Science Foundation of China No.61821003 and the National Natural Science Foundation of China No.61902135.

REFERENCES

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. 2017. Hindsight experience replay. *arXiv preprint arXiv:1707.01495* (2017).

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine* 34, 6 (2017), 26–38.

[3] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *VLDB*, Vol. 97. Citeseer, 146–155.

[4] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically indexing millions of databases in microsoft azure sql database. In *Proceedings of the 2019 International Conference on Management of Data*. 666–679.

[5] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 505–520.

[6] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.

[7] Biplob K Debnath, David J Lilja, and Mohamed F Mokbel. 2008. SARD: A statistical approach for ranking database tuning parameters. In *2008 IEEE 24th International Conference on Data Engineering Workshop*. IEEE, 11–18.

[8] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle.. In *CIDR*. 84–94.

[9] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

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

[11] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.

[12] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 1126–1135.

[13] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1189–1206.

[14] David E Goldberg and John Henry Holland. 1988. Genetic algorithms and machine learning. (1988).

[15] Steven M Holland. 2008. Principal components analysis (PCA). *Department of Geology, University of Georgia, Athens, GA* (2008), 30602–2501.

[16] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[17] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. 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 (HotStorage 20)*.

[18] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[19] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. 2002. Automatic configuration for IBM DB2 universal database. *Proc. of IBM Perf Technical Report* (2002).

[20] Roger J Lewis. 2000. An introduction to classification and regression tree (CART) analysis. In *Annual meeting of the society for academic emergency medicine in San Francisco, California*, Vol. 14. Citeseer.

[21] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[22] 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).

[23] Ji Liu and Ce Zhang. 2021. Distributed learning systems with first-order methods. *arXiv preprint arXiv:2104.05245* (2021).

[24] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active learning for ML enhanced database systems. In *Proceedings of the 2020 International Conference on Management of Data*. 175–191.

[25] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.

[26] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1248–1261.

[27] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[28] Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas. 2012. How many trees in a random forest?. In *International workshop on machine learning and data mining in pattern recognition*. Springer, 154–168.

[29] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online index selection using deep reinforcement learning for a cluster database. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 158–161.

[30] Karl Schnaitter and Neoklis Polyzotis. 2010. Semi-automatic index tuning: Keeping dbas in the loop. *arXiv preprint arXiv:1004.1249* (2010).

[31] Jonathon Shlens. 2014. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100* (2014).

[32] Adam J Storm, Christian Garcia-Arellano, Sam S Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive self-tuning memory in DB2. In *Proceedings of the 32nd international conference on Very large data bases*. 1081–1092.

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

[34] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1221–1234.

[35] Wenhu Tian, Pat Martin, and Wendy Powley. 2003. Techniques for automatically sizing multiple buffer pools in DB2. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*. 294–302.

[36] Dinh Nguyen Tran, Phung Chinh Huynh, Yong C Tay, and Anthony KH Tung. 2008. A new approach to dynamic self-tuning of database buffers. *ACM Transactions on Storage (TOS)* 4, 1 (2008), 1–25.

[37] 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*. 1009–1024.

[38] 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. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1241–1253.

[39] Gerhard Weikum, Christof Hasse, Axel Mönkeberg, and Peter Zabback. 1994. The COMFORT automatic tuning project. *Information systems* 19, 5 (1994), 381–432.

[40] Darrell Whitley. 1994. A genetic algorithm tutorial. *Statistics and computing* 4, 2 (1994), 65–85.

[41] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.

[42] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.

[43] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1223–1240.

[44] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*. 1599–1614.

[45] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1297–1308.

[46] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.

[47] 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*. 2102–2114.

[48] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*. 338–350.