Automatic Database Knob Tuning: A Survey

Xinyang Zhao, Xuanhe Zhou, Guoliang Li

Abstract—Knob tuning plays an important role in database optimization, which tunes knob settings to optimize the database performance or improve resource utilization. However, there are several common challenges in knob tuning. First, databases have hundreds of configuration knobs, and it is hard to determine the knobs that cause the performance/resource bottlenecks. Second, most knobs are of continuous values and cause large search space, where heuristic knob tuning may not find high-performance knob settings within limited time. Third, it is increasingly tricky to conduct knob tuning with the proliferation of cloud services, where we need to tune a large number of database instances for various scenarios (*e.g.,* different applications, datasets, and hardware). Recently, many learning-based knob tuning methods are proposed to alleviate those problems. The *core idea* of learning-based knob tuning is that, with the help of machine learning techniques, it is reasonable to collect knob tuning data, leverage these data to train a knob tuning model, and utilize the tuning model to recommend knob settings for new similar scenarios, so as to achieve the optimization objectives. In this paper, we provide a comprehensive survey on database knob tuning. The pipeline of knob tuning includes *knob selection*, *feature selection*, *tuning methods*, and *transfer techniques*. First, for *knob selection*, we introduce the main categories of database knobs and summarize existing knob selection algorithms. Second, for *feature selection*, we introduce commonly-used tuning features and explain existing feature selection techniques (*e.g.,* runtime metric selection and workload encoding). Third, for *tuning methods*, we compare four classes of tuning methods, *i.e.,* heuristic methods, Bayesian-optimization methods, deep-learning methods, and reinforcement-learning methods. In particular, we summarize the challenges and discuss how existing methods address those challenges. Moreover, we discuss some transfer techniques that utilize historically well-trained tuning models in new scenarios. Fourth, we discuss the implementation of automatic knob tuning methods in typical systems (*e.g.,* commercial relational databases and big data analytics systems). Lastly, we provide some research challenges and future research opportunities. We believe this survey can help researchers better understand the knob tuning problems and existing approaches and further encourage them to solve the remaining problems in automatic knob tuning.

Index Terms—Knob Tuning, Machine Learning, Knob Selection, Feature Selection, Tuning Methods

1 INTRODUCTION

CONFIGURATION knobs control many aspects of database systems (*e.g.,* user connections, query optimizer, underlying resource management), and different combinations of knob values significantly affect the system robustness, performance, and resource usage. In general, knob tuning aims to judiciously adjust the values of knobs so as to meet some tuning objectives. For example, the optimizer in relational databases is affected by various knobs like physical operators (*e.g.,* whether allowing to use nested loops in query plans), planning strategies (*e.g.,* heuristic, genetic algorithms), and cost weights (*e.g.,* the cost of random page read). It is vital to set proper values for the knobs in the optimizer so as to generate effective query plans and optimize the performance. Besides, database memory management is also controlled by various configuration knobs, *e.g.,* the space for user connections, the shared buffer size, and the space for background processes. Properly tuning these knobs can avoid frequent disk I/O and improve data access efficiency. Traditionally, knob tuning relies on database administrators (DBAs) to manually try out typical knob combinations, which is highly time-consuming and cannot adapt to a large number of database instances (*e.g.,* tens of thousands of database instances on the cloud). To solve these problems, researchers attempt to design tuning methods that can automatically explore suitable knob settings for different scenarios (*e.g.,* capturing the knob-performance relations with probability models; tuning with the exploration-and-exploitation strategy).

• *Xinyang Zhao, Xuanhe Zhou, Guoliang Li are with the Department of Computer Science, Tsinghua University, Beijing, China. E-mail: {xy-zhao20, zhouxuan19}@mails.tsinghua.edu.cn, liguoliang@tsinghua.edu.cn*
Xinyang Zhao and Xuanhe Zhou are co-first authors and make equal contributions. Corresponding author: Guoliang Li. The code repository is at https://github.com/evolveDB/tuning-survey

1.1 Challenges of Knob Tuning

The knob tuning problems can be categorized into the following questions: (1) what are the tuning objectives, (2) what to tune, (3) with what to tune, and (4) how to tune – these four fundamental problems, as shown in Figure 1.

Question 1: What are the tuning objectives? – Performance evaluation. There are two aspects of tuning objectives. On the one hand, users require to configure the databases for high performance, *i.e.,* (i) throughput, which indicates how efficiently the database executes concurrent queries, and (ii) latency, which indicates how fast the database responds to single queries [45], [48], [49], [52], [69], [89], [94], [56]. On the other hand, database vendors require to improve resource utilization (*e.g.,* I/O and memory usage) or reduce maintenance costs without sacrificing performance. Moreover, a tuning method is generally evaluated from four aspects: (i) performance (how well the method achieves the objectives in a given scenario), (ii) overhead (how much time or system resource the method requires to recommend knob settings), (iii) adaptivity (how well the method achieves the objectives in new scenarios), and (iv) safety (whether the method can avoid selecting knob settings that cause performance degradation). Performance and overhead are most important in offline tuning stage (*e.g.,* tuning on the

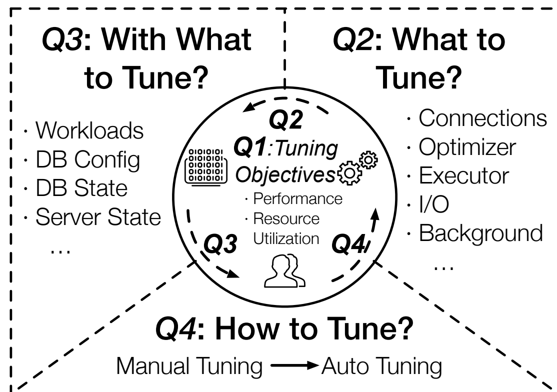

Fig. 1: Key Problems in Knob Tuning.

clone machine); while adaptivity and safety are more critical for online tuning (*e.g.,* tuning on the production machine).

Question 2: What to tune? – Huge search space with numerous knobs. There are hundreds of knobs in a database system, and tuning all the knobs may consume much time and system resources. Thus it is vital to select important knobs that can significantly affect the performance. However, there are two main challenges. (i) *How to evaluate the knob-performance relations?* The relations can be very complicated, and it is time-consuming to test each knob; (ii) *How to select important knobs for different tuning tasks?* Knobs may have different or even opposite effects on the performance, and it is hard to find suitable knob subsets. For example, maintenance-related knobs play an important role in high-update-ratio workloads but take minor or even negative effects on read-only queries. Furthermore, too frequent dirty-page cleaning may compete with read-only queries for system resources. We will review existing knob selection techniques [19], [22], [39], [69], [73], [76] in Section 3.

Question 3: With what to tune? – Diversified features for characterizing the tuning requirements. With proper knobs, another problem is how to select tuning features (*e.g.,* workloads, database state, hardware environments), which can reflect the database execution behaviours (*e.g.,* dirty page update for write operations, buffer sharing for read operations) and potentially affect the tuning performance. It is challenging to characterize the tuning requirements with proper features [25], [29], [79]. First, tuning features are in high dimensional space (*e.g.,* hundreds of runtime metrics). Although numerous features can reflect the database state, many features capture similar tuning characteristics (*e.g.,* accessed data blocks in different granularity), making it hard to filter out the redundant features. Second, many tuning features are in diversified domains (*e.g.,* query features and state metrics). It is tricky to combine (embed) useful features into the same input domain. We will review existing feature selection techniques in Section 4.

Question 4: How to tune? – Huge configuration search space. After filtering out most knobs and non-relevant features, the configuration space (*i.e.,* all the candidate settings of selected knobs) can still be large. Because there are various knob combinations and knob tuning is an NP-hard problem [38]. However, traditional empirical methods may only find sub-optimal knob settings that cause performance regression. There are two main challenges. (i) *The search space is huge.* Even if the knob number has been significantly reduced, most knobs are of continuous ranges, and the configuration space is too large to enumerate; (ii) *Obtaining the tuning performance is costly.* For each knob setting, generally, we need to execute workloads to gain the feedback and utilize the feedback to update the tuning model, which may take a long time. Moreover, the relations between knobs and the database performance could be too complex for simple regression methods like linear equations. Besides, traditional methods are unaware of various tuning objectives and constraints (*e.g.,* some knobs are not allowed to adjust) in real scenarios. For example, they utilize a coarse-grained rule to allocate 25% RAM as working space [77], which can be too large for short queries and causes space waste; or too small for queries with large intermediate data. Thus, it is vital to *intelligently recommend knob settings based on scenario characteristics and tuning objectives*. We will review existing learning-based tuning methods in Section 5.

Configuration Transferring – Adapt to different scenarios. Additionally, it is vital but challenging to adapt to workload changes or new tuning tasks in real scenarios [26], [52], [92]. Generally, a single knob setting cannot be optimal for all tuning tasks, because the knob-performance relations and tuning requirements may change on the new scenarios, and the tuning models trained on historical tuning tasks cannot be directly used (fine-tune or retrain new models from scratch). Besides, new tuning features make it even harder to adapt to new scenarios. Most existing machine learning models (*e.g.,* reinforcement learning) cannot directly recommend high-quality knob settings for new tuning tasks and take a long time to train from scratch. Hence, we will review transfer techniques for migrating well-trained models on historical tuning tasks to new tasks in Section 5.5.

1.2 Contributions

We review a wide spectrum of existing tuning methods [19], [45], [48], [49], [52], [69], [73], [74], [79], [82], [89], [94], [98] and show how they can answer the above four questions and solve the challenges. As shown in Figure 2, for *knob selection*, we formalize the definition of configuration knobs, showcase common knobs, and explain two knob selection methods, *i.e.,* empirical-based approaches and ranking-based approaches (Section 3). For *feature selection*, we explain tuning features in existing tuning methods (*e.g.,* workload features, database features) and show two feature selection methods, *i.e.,* feature discretization and feature clustering (Section 4). For *knob tuning*, we explain the core ideas and existing works of four typical tuning methods, *i.e.,* heuristic approaches, Bayesian optimization approaches, deep learning approaches, and reinforcement learning approaches (Section 5). For *transfer techniques*, we introduce common feature extraction and adaptive weight transferring (Section 5.5).

2 Knob Tuning Overview

As shown in Figure 3, a general knob tuning workflow includes four main modules, *i.e.,* knob selection, feature selection, tuning methods, and transfer techniques.

(1) Knob Selection. Database systems contain hundreds of knobs that (1) have various effects on tuning performance (*e.g., working memory size* is vital to memory-intensive queries, *maximum IO concurrency* is vital to IO-intensive

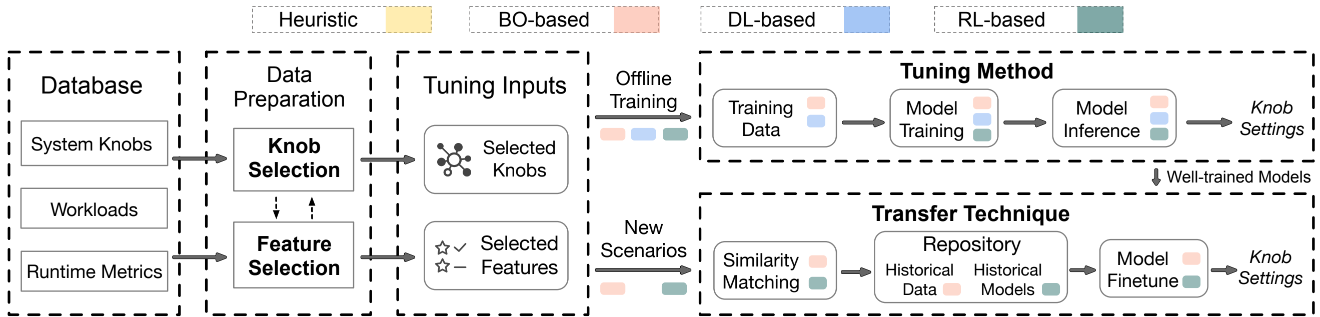

Fig. 2: The Workflow of Automatic Knob Tuning.

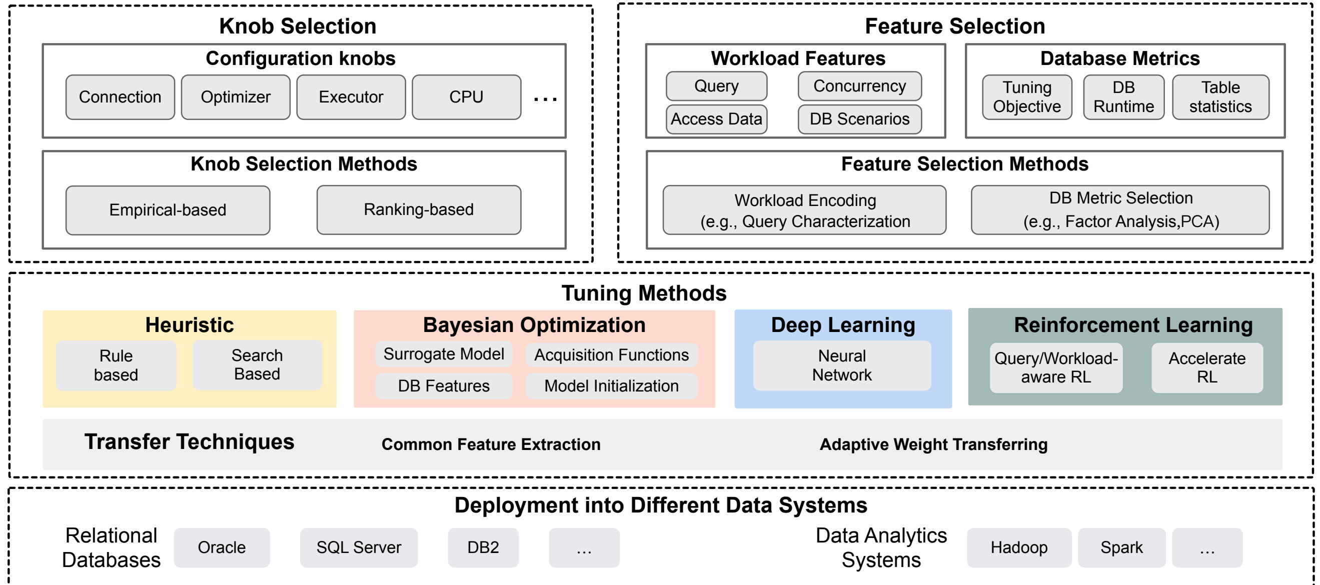

Fig. 3: The Framework of Knob Tuning.

TABLE 1: Example Knobs (*Reconfiguration Cost* denotes the time for the new knob value to take effect).

Knob Name	Class	Type	Restart	Reconfiguration Cost
max_connections	AC	integer	✓	high
seq_page_cost	QP	float	×	low
force_parallel_mode	QE	enum	×	medium
vacuum_cost_limit	BP	integer	×	low
shared_buffers	RM	integer	✓	high

queries), and (2) correlations between knobs can significantly affect the performance (*e.g.*, the maximum number of user connections is limited by the allocated resources). As shown in Table 1, we first showcase knobs in typical categories, *e.g.*, access control (AC), query optimization (QP), query execution (QE), background processes (BP), and resource management (RM). For reconfiguration costs, the knobs that require database restarting are labeled with *high* reconfiguration cost (*e.g.*, max_connections); knobs that do not require restarting but significantly change the underlying mechanisms are labeled with *medium* reconfiguration cost (*e.g.*, force_parallel_mode); and the rest knobs are labeled with *low* reconfiguration cost (*e.g.*, seq_page_cost).

Next, we summarize existing knob selection techniques, including empirical-based strategies [69], [73] and ranking-based strategies [69], [73], which aim to filter unimportant knobs and reduce the configuration space. The *empirical-based strategy* requires DBAs to select important knobs by utilizing their experience. However, relying on DBAs to select from hundreds of knobs is laborious. Therefore, the *ranking-based strategy* evaluates the importance of knobs to the database performance and selects knobs with the highest importance. The above two strategies could significantly reduce the configuration space by filtering unimportant knobs. However, there are some problems. First, they rely on some simple assumptions (*e.g.*, assuming the linear relations between knobs and performance [19]). Second, they cannot capture the correlations of knobs. Third, they may miss valuable knobs. Fourth, they rely on a large number of samples (*e.g.*, the lasso algorithm [79]) to find important knobs and may cause performance regression when the workloads change.

(2) Feature Selection. Various features can represent database characteristics (*e.g.*, workload types, runtime state) and provide important hints on the tuning requirements. We broadly divide tuning features into workload features and database metric features, widely used in existing tuning works [26], [52], [92]. Workload features can reflect the basic information of the workloads (*e.g.*, typical operators, execution costs). Database metrics are extracted from executions and can infer the database state (*e.g.*, the number of read-/written blocks, database performance) under the workload. Additionally, the database metrics, especially tuning objective metrics (*i.e.*, latency and throughput), can also be used to evaluate and select important knobs [63], [39]. Next, we separately discuss existing techniques for these two features, *e.g.*, discretization for workload feature encoding and clustering algorithms for metric feature selection.

(3) Tuning Methods. As the knob tuning problem is NP-hard, one tuning method cannot perform well un-

der all the scenarios (*e.g.,* heuristic methods achieve low tuning overhead; while learning-based methods can utilize historical tuning experience and optimize the performance). Specifically, we classify existing tuning methods into four categories, *i.e.,* (i) *heuristic methods*, (ii) *Bayesian optimization-based methods*, (iii) *deep-learning-based methods*, (iv) *reinforcement-learning-based methods*.

First, *heuristic tuning methods* [69], [98] aim to automatically explore suitable knob settings under a limited budget (*e.g.,* within hours). The advantage is that heuristic methods are easy to implement. However, there are three main challenges: (1) heuristic methods randomly sample a few knob settings in each iteration, and it is challenging to find promising knob settings among large configuration space within limited tuning time; (2) heuristic methods cannot utilize historical tuning data and prior knowledge, so each time, they need to restart tuning from scratch, even if the new tuning task has similar optimal configurations to historical tasks; (3) heuristic methods may waste time in evaluating the search space of inferior knob settings.

Second, *Bayesian optimization(BO)-based tuning methods* [74], [79], [45], [92], [93] are proposed to recommend suitable knob settings by iteratively sampling knobs and approximating the knob-performance relationships. The core idea is that they utilize a probability surrogate model to map the relationship between knob settings and database performance. During the tuning process, as the number of sampled knob settings increases, it evaluates and updates the model until it finds the optimal settings. Many probability models could be used in BO-based tuning methods, *e.g.,* the Gaussian process (GP), SMAC, and Tree-structured Parzen estimator (TPE) (see Section 5.2). Among these models, GP natively supports continuous knobs and is most widely used as the surrogate model [74], [79], [45], [92], [10], [93]. The advantage of BO-based methods is that they can quickly find high-quality knob settings when the knob number is small (*e.g.,* 5-20 knobs) [80], [91]. However, BO-based methods are based on probability distribution (*e.g.,* assuming the knob-performance relations following the Gaussian distribution) and cannot efficiently represent and explore large configuration space.

Third, *DL-based tuning methods* [73], [80] aim to improve the capability of representing large configuration space by replacing the GP models with neural networks, which can help to find better knob settings. However, DL-based methods require a large number of high-quality samples to train the neural networks.

Fourth, *Reinforcement-learning (RL)-based tuning methods* [9], [52], [89], [77], [82] are proposed to explore suitable knob settings by the interaction between the agent (the tuning model) and the environment (the target database). Both BO-based methods and RL-based methods generate training data through guided exploration and do not require prepared training data. In addition, RL-based methods have no limitation on the size of the configuration space. However, the tuning overhead of RL-based methods is much higher than BO-based methods, and the performance greatly depends on the initially sampled knob settings.

(4) Transfer Techniques. With well-learned tuning models on historical tasks, *transferring those models to new tuning tasks* is also a challenging problem (*e.g.,* tuning for workloads with completely new operators, datasets, or even hardware environments). Moreover, it is vital to achieve practical learned tuning methods in real scenarios. Thus, transfer techniques are proposed to support dynamic scenarios at different levels (*e.g.,* various workloads and databases).

First, *common feature extraction* directly extracts the common features of different workloads (*e.g.,* operator types, operator costs, write/read ratios) [26], [52], [92]. It embeds the extracted features with deep learning models (*e.g.,* dense network [52], self-attention [26]), which helps the tuning model to generalize to different workloads efficiently. There are two main challenges: (1) Workloads may have different access patterns. It is hard for the neural network to embed for unseen workloads (*i.e.,* out of distribution). (2) They need to design an extra workload embedding model, which requires much prepared training data.

Second, *adaptive weight transferring* aims to adapt to new scenarios by preparing a suite of tuning models that are well-trained under historical workloads [79], [92]. Intuitively, for a new workload, they [79] map the workload to the historical data whose corresponding workload has the highest similarity (*e.g.,* within the same range of write ratio). They utilize the historical data to train a new tuning model for the new workload. Moreover, to adapt to different databases [92], they prepare some base models trained on typical workloads from different database instances. For a new database instance, they generate a new tuning model based on the weighted sum of prepared models and fine-tune the weights of the tuning model on the new instance.

3 Knob Selection

In this section, we focus on knob selection, *i.e., what to tune (configuration knobs)*. As mentioned in Section 1, among the numerous system knobs, some knobs can significantly affect performance via different mechanisms. For example, some knobs (*e.g.,* concurrency control) can directly affect the database performance, while some knobs (*e.g.,* background process management) may indirectly affect the performance by reducing unnecessary maintenance operations. Thus, in this section, we first introduce the typical knobs in database systems (Section 3.1). Next, we explain existing knob selection techniques in Section 3.2.

3.1 Configuration Knobs

As shown in Figure 4, configuration knobs in different database modules potentially affect the performance. Hence, we first formalize the definition of configuration knobs, then we introduce seven categories of configuration knobs shared by mainstream databases like Postgres and MySQL (Table 2), and discuss their effects on database performance.

3.1.1 Formalization

In a broad sense, we use the term "configuration knobs" to denote the knobs in databases whose values can be adjusted.

Definition 1 (Configuration Knobs). For any configuration knob $\mathcal{K}$, the knob $\mathcal{K}$ is associated with a value $V_{\mathcal{K}}$, which can be adjusted within a value range (V^{min}, V^{max}) (continuous knobs) or a set of candidate values $\{V^1, \cdots, V^k\}$ (enumeration/categorical knobs).

TABLE 2: Categories of Configuration Knobs.

	Category	Functionality	Example (Postgres)	Example (MySQL)
1	Access Control	Connections	max_connections	innodb_thread_concurrency
		Transactions	deadlock_timeout	innodb_table_locks
2	Query Optimizer	Query Plan	join_collapse_limit	rewriter_enabled
		Cost Values	seq_page_cost	join_buffer_size
3	Query Executor	Persistence	full_page_writes	replica_pending_jobs_size_max
4	Background Processes	Logging	log_rotation_size	binlog_cache_size
		Others	checkpoint_timeout	innodb_log_file_size
5	Resource (CPU)	CPU Usage	max_files_per_process	innodb thread concurrency
6	Resource (Memory)	Memory Space	shared_buffers	innodb_buffer_pool_size
7	Resource (Disk)	Disk IO/Caches	temp_file_limit	max_sort_file_size

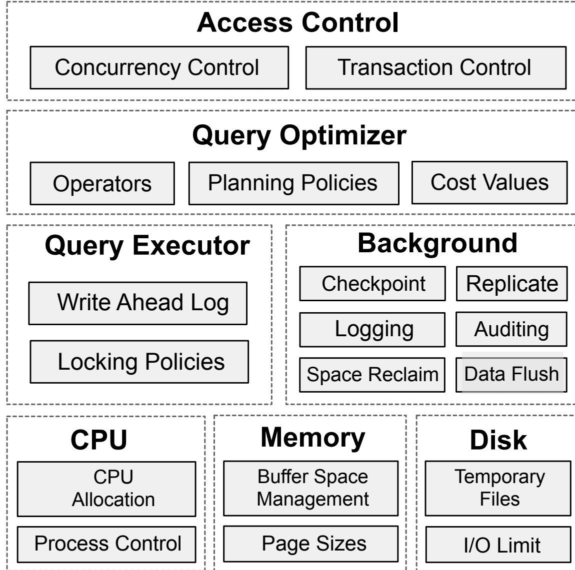

Fig. 4: General Configuration Knobs.

If the value is adjusted into V_K', the database performance may be affected.

Example 1. For the continuous knob *work_mem* in Postgres, it sets the maximum memory space that can be used for query operations (*e.g.*, a sort, hash join), which helps to reduce the disk I/O frequency and improve the execution efficiency. The *work_mem* value can be adjusted from 65kB to the maximal allowed memory size. Similarly, there are memory-allocation knobs in databases like MySQL (*innodb_buffer_pool_size*), Oracle (*memory_max_target*), and DB2 (*database_memory*).

Example 2. There are categorical knobs for query optimization. For example, in Postgres, *enable_memorize* ({on, off}) controls whether the memorized plans can be used to directly access the cached results; in MySQL, *rewriter_enabled* controls whether the logical query transformation function is adopted, which may increase the query planning time but enhance the execution with the simplified plans after query rewrite.

3.1.2 Access Control

Access control knobs mainly affect (i) the concurrent accesses (*e.g.*, *max_connection* in Postgres, *innodb_thread_concurrency* in MySQL) or (ii) transactions to the database (*e.g.*, *deadlock_timeout* in Postgres, *innodb_table_locks* in MySQL). Access control knobs affect the database performance by balancing concurrency and execution efficiency. Intuitively, the more connections (*e.g.*, higher concurrency) or transactions (*e.g.*, longer deadlock checking time) are accepted, the fewer resources can be allocated to each connection. Fewer resources may cause a bottleneck and slow down the overall performance. On the other hand, fewer concurrent connections can improve the performance of these connections but may sacrifice the throughput. Thus it is challenging to select concurrency knobs to balance the latency and throughput.

3.1.3 Query Optimizer

Knobs for query optimizer control the generation of query plans (*e.g.*, enabled operators, plan optimization, and cost values). For example, for query plan optimization, there are *join_collapse_limit* in Postgres and *rewrite_enabled* in MySQL; for cost values, there are *seq_page_cost* in Postgres and *join_buffer_size* in MySQL. Knobs like *seq_page_cost* can significantly affect the physical operator selection (*e.g.*, lower *seq_page_cost* may cause the optimizer to select seq_scan rather than index_scan; lower *random_page_cost* may cause the optimizer to utilize indexes instead). By judiciously adjusting these knobs, we can generate efficient query plans that significantly improve the query performance. For example, the knob controls whether the optimizer can utilize nested loop operations to carry out the logical table joins. Sometimes, utilizing nested loop can gain lower query cost (*e.g.*, the driven table has low selectivity) than other equivalent operators like hash join [61].

Query-optimizer-related knobs affect the quality of generated query plans from three main aspects. First, some knobs determine which physical operators can be utilized in the generated query plan. For example, sequential scan operators may outperform index scan operators when many tuples cannot be filtered (*e.g.*, low selectivity), and nested loop may outperform hash join when the outer table is large. It is vital to judiciously configure those knobs so as to utilize proper physical operators. Second, some knobs specify the adopted planning policies, *e.g.*, greedy or genetic algorithms. The greedy algorithm may efficiently find good query plans for queries with few joins, while genetic algorithms may find better query plans for queries with multiple joins. Third, the cost weights for different factors approximate the latency of basic physical operations.

Recent studies [46], [47], [55], [70], [71], [81], [97] have shown that cost estimation is vital to query plan selection, and poorly estimated costs usually lead to suboptimal query plans. Thus, adjusting the cost weights based on the underlying physical settings can improve cost estimation accuracy and performance.

3.1.4 Query Executor

Knobs for query executor aim to configure physical execution mechanisms (*e.g.*, physical operator replacement) and locking mechanisms. First, some query execution knobs

may change the query plan based on the factors like query complexity. For example, in Postgres, for any sort operator whose input tuples are fewer than the value of *replacement_sort_tuples*, the database utilizes replacement-selection sort rather than quicksort to improve the sorting efficiency under limited memory. Second, similar to transaction control knobs, there are locking-related knobs that affect the locking policies for concurrent read/write queries. Frequent deadlock checking may cause high overhead, while the occurrence of deadlocks may slow down query execution. Thus, in MySQL, for high concurrency workloads, we can tune *innodb_lock_wait_timeout* to control the waiting time for a lock before being rolled back.

3.1.5 Background Processes

There are numerous background processes that carry out various maintenance tasks (*e.g.,* logging, checkpoint, auditing, replicating, and automatic vacuuming), which are vital to ensure the reliability of database instances. The critical issue is to balance the efficiency (*e.g.,* recovery) and overhead (carrying out the background tasks). First, logging mechanisms (*e.g.,* write ahead log) are vital to ensure important database reliability, like data consistency. Background-process knobs control various aspects of system logging, like the logging level (*e.g.,* whether skipping some bulk operations), the indicator of execution success, and whether enabling slow query logging. Relevant knobs include *log_rotation_size* in Postgres and *binlog_cache_size* in MySQL. Second, mechanisms like checkpoint, auditing, and replication are also vital to ensure data consistency. However, similar to logging, too frequent checkpoints may affect database performance. We can control it by tuning knobs like maximum waiting time between two checkpoints. Third, dirty-page-cleaning aims to synchronize updated pages in the memory to disk and clean up the corresponding pages. And there are many knobs (*e.g., log_autovacuum_min_duration* in Postgres, *innodb_log_file_size* in MySQL) that help to balance memory utilization and cleaning overhead.

3.1.6 Resource Management (CPU)

CPU-related knobs control the kernel resources allocated to each user connection so as to ensure database stability and improve query concurrency (*e.g., max_file_per_process* in Postgres, *innodb_thread_concurrency* in MySQL). First, they need to ensure a safe resource limit for each user. Second, some knobs control the maximum parallelism for executing single queries, which helps to balance execution efficiency and resource utilization.

3.1.7 Resource Management (Memory)

Memory-related knobs are vital to database performance, especially for OLAP queries that access large input tables or product large intermediate tables. There are two main categories of memory knobs, including memory allocation and configuration. For memory allocation, some knobs control the memory usage in multiple levels (*e.g.,* shared_buffers in Postgres, *innodb_buffer_pool_size* in MySQL). Note that knobs like shared buffer sizes require restarting databases before the knob adjustment takes effect, which may cause large tuning overhead, while knobs like local memory allocation (operator level) can immediately take effect and are vital to online knob tuning. Hence, we can select different knobs to tune based on the tuning requirements. Second, for memory configuration, some knobs control the memory settings, like the size of single pages. Scenarios like large table access may require huge pages to reduce the IO and computation time, and judiciously tuning those knobs can improve the performance.

3.1.8 Resource Management (Disk)

Disk-related knobs specify the configuration of disk settings so as to fully utilize I/O bandwidth and disk space for various tasks in the database, *e.g.,* user connections, and temporary file processing. There are disk knobs like *temp_file_limit* in Postgres and *max_sort_file_size* in MySQL. Disk knobs mainly affect the database performance by controlling IO. Since the IO bandwidth is limited, many IO knobs limit the concurrent IO operations or determine the types of background write-back operations. Note that we can find that disk knobs mainly take effect for workloads with a high write ratio and random access. In those cases, we need to pick knobs based on workload characteristics and database states.

3.2 Knob Selection Methods

Most existing databases have hundreds of knobs, and many knobs can potentially affect the performance. However, only a subset of knobs could play a major role in database performance. Kanellis et al. [39] claim that pre-selecting several knobs for tuning could achieve good performance. There are two main challenges for knob selection: selecting important knobs and selecting suitable knobs. The former means that only those knobs have a major impact on performance should be considered in the tuning process. For example, many methods [9], [19], [39], [74], [79] identify important knobs before actual knob tuning to reduce the tuning overhead. The latter means selecting knobs that are suitable for different scenarios.

Thus, traditional knob selection methods rely on DBAs to select important or suitable knobs (*e.g.,* some important knobs are not allowed to change in case the database crashes) [9]. However, this leads to a huge burden for DBAs to handle hundreds of knobs. Thus, some tuning methods [22], [66], [69], [73], [76] propose a knob selection module in their framework. Here we introduce three categories of knob selection methods, *i.e.,* empirical-based method, ranking-based method, and learning-based method. Table 3 presents a classification of knob selection methods based on the current tuning work.

3.2.1 Empirical-based Knob Selection

Empirical-based knob selection requires humans (*e.g.,* database users, DBAs) to select knobs based on limited tuning documents or empirical experience. On the one hand, database users can learn to select knobs according to the configuration manual; on the other hand, DBAs can select knobs based on the tuning knowledge accumulated in real practice [45], [73]. For example, memory is usually the resource bottleneck of the cloud database performance, and the optimal buffer pool sizes may change for different instances. Therefore, to optimize individual resource

utilization, DBAs could manually select the *buffer pool size* knob as the tuning objective [73]. In addition, for a high-concurrency scenario, Sullivan et al. [69] recommended tuning four knobs (relevant to the page size, write threads, and lock policies) to maximize resource usage and improve performance.

Remark. The advantage of empirical-based methods is that they are relatively simple and more trustworthy since each knob is verified manually. However, they have some limitations. First, it is hard for humans to identify the complex correlations between various knobs (*e.g.,* the maximum values of some knobs may rely on other knobs). Besides, simply relying on the DBA's experience may also ignore potentially effective knobs on system performance. Second, empirical knob selection is generally based on some heuristic rules (*e.g.,* determining knobs based on workload types like OLTP/OLAP). These rules may not work well for complex scenarios like HTAP workloads. Third, the tuning experience may not catch up with the frequent version updates of database systems (*e.g.,* the number of knobs and the knob effects may significantly change on the new version).

3.2.2 Ranking-based Knob Selection

Empirical-based methods select knobs mainly based on experts' experience, and they usually do not evaluate the relationship between knobs and performance. Instead, ranking-based methods sort the knobs based on their impact on the database performance and select the knobs with large impacts [22], [66], [76]. Ranking-based methods include configuration sample collection (*e.g.,* a configuration and its corresponding impact on the database) and knob ranking. The first step collects some configuration settings and their related performance on a workload. Assume there are 100 knobs, and each knob has two options. There will be 2^{100} possible samples. Thus, the first step selects representative samples. The second step ranks the knobs based on the samples. Next, we discuss the methods for these two steps.

Sample Collection. It aims to provide high-quality samples (knob settings) for knob ranking. A simple and easy method is to randomly select knob settings as samples. However, random sampling may lead to bias and miss many combinations of effective knob values.

Another popular sample collection method is Latin Hypercube Sampling (LHS) [39], [45], [74], [79], [92]. Specifically, suppose there are n knobs, which splits each knob value range into d intervals. Then the sample space will be d^n. It is prohibitively expensive to enumerate this space. To address this issue, LHS selects d samples such that for each interval, there is exactly one sample covering the interval. The advantage of LHS is that it can thoroughly and uniformly cover the configuration space. However, LHS cannot utilize prior knowledge to filter inferior knob settings, which lead to many ineffective samples.

The third sample collection method is a hybrid method used in HUNTER [9]. This hybrid sampling method helps to generate legal and high-quality samples based on both the empirical rules (*e.g.,* the allowed value ranges for specific knobs) and efficient search algorithm (*i.e.,* Genetic Algorithm). However, the empirical rules still rely on manual updates, which are hard to generalize to different scenarios.

TABLE 3: Knob Selection Methods.

Category	Selection Method	Brief Description
Empirical based	Database Manual	Database users select knobs according to database configuration manual.
	Tuning Experience[45][73]	DBAs select knobs based on their accumulated tuning experience.
Ranking based	Plackett& Burman[19]	Collect knobs-performance samples and use a statistical method to select knobs.
	Lasso [79]	Collect knobs-performance samples and use a regression model with a penalty function to rank the knobs.
	CART [39]	Collect knobs-performance samples and use a random forest to rank knobs.
	Sensitivity Analysis[74]	Collect knobs-performance samples and calculate the effect of input knobs to eliminate low-effect knobs.

Knob Ranking. Knob ranking aims to rank the knobs based on collected configuration samples. There are multiple methods to rank the knobs based on the samples.

The first is a statistical method in SARD [19], which uses the statistical technique so called Plackett & Burman (P&B). For each sample, it assigns "+1" ("-1") for each knob, separately denoting values slightly higher or lower than normal value ranges. If the performance gets better (worse) than the initial setting, it uses the P&B method to sum up the multiplication of "+1" ("-1") with the performance in each sample to rank the knobs. The pros of P&B are simple and easy to implement, while the quality of the sorting results is limited.

The second is a linear regression method called Lasso [76] used in OtterTune [79], which takes samples as independent variables (X) and takes the selected performance metrics as dependent variables (y). It employs a regularized version of least squares to rank the knobs. Lasso is a regression model that obtains a more refined model by constructing a penalty function that shrinks the weights of the knobs towards those with high-performance impact. By automatically adjusting the value of parameter $lambda$, Lasso can control the number of knobs it selects. Initially, $lambda$ is set to a large value so that no knobs are selected. Gradually, the size of $lambda$ is reduced, and more knobs are introduced, with the order in which each knob first appears indicating its importance together with other added knobs. Lasso ignores the non-linear relationships between knobs so that the selected knob affects the ordering of the other knobs.

The third is an ensemble of regression trees (*i.e.,* random forest), used in [9], [39], which builds a random forest for the samples and the corresponding performance. Then it selects important knobs based on the random forests. The regression tree has three advantages: capturing non-linear relationships between knobs, the interpretability of selected knobs, and the fast training of the model.

The fourth is Sensitivity Analysis (SA) [66] used in iTuned [22], which identifies and eliminates unimportant knobs. It defines an empirical equation (`main effect`), which represents the extent to which the system performance y changes when any input knob x_i is changed, *i.e.,* $V = Var(E(y|x_i))/Var(y)$, where $E(y|x_i)$ denotes the expected performance, and $Var(y)$ denotes the variance of the performance. Specifically, if the computed `main effect` V is high, it means that the change in knob x_1 has a

large effect on the performance, and so it is vital to select the knob x_1 to tune; otherwise, could ignore those knobs with low `main effects` for the process of configuration tuning. Sensitivity analysis is a relatively simple and easily applied method to identify and eliminate low-effect knobs. However, it has a limited effect on non-linear models and inputs with correlation [15]. Therefore, this method could not select suitable knobs for the database system.

Remark. The advantage of knob selection techniques is that they can find several knobs significantly affecting the database performance. However, there are still several challenges. First, ranking-based knob selection methods require to find the relations between the knobs and performance. However, since there are hundreds of knobs, only a small number of knobs are involved in knob selection, which may introduce bias. Second, if the ranking-based methods miss some important knobs, they will affect the tuning performance. Third, pre-selecting knobs could not handle dynamic scenarios, as different scenarios may require different important knobs.

4 Feature Selection

In this section, we discuss the feature selection problem, *i.e., deciding what features to tune*. The main challenge for feature selection is selecting suitable tuning features to reflect the database execution behaviours. Note that various features like workload characteristics could affect the database performance, and it is vital to select tuning features (*e.g.,* workloads, database state, and the hardware environment) to characterize the target tuning requirements. In this section, we first provide a brief overview of the available tuning features and their effects on both the tuning performance and transfer capabilities in Section 4.1. Next, we discuss the feature selection methods commonly used in existing tuning works in Section 4.2.

4.1 Tuning Features

Tuning features are measurable properties that can profile the database state and help to find proper knob settings. The tuning features include workload features and database runtime status. The former captures the workload characteristics which can be used to enable workload-aware tuning. The latter captures the underlying database characteristics, which captures the database states when tuning knobs.

4.1.1 Workload Features

The performance of different workloads can be affected by different components. For example, for memory-intensive workloads, tuning memory-relevant knobs may improve the performance, while other knobs, like dirty page cleaning, may take minor effects. Thus, it is vital to characterize workloads so as to locate the problematic components and optimize the corresponding knobs (*e.g.,* increase the concurrency thresholds for high-parallelism queries).

Based on existing tuning works [52], [26], [92], [45], we first introduce three main characteristics that determine the tuning requirements of different workloads, *i.e.,* query features, concurrency features, and data features.

Query Features include the used query operators, keywords, query structures (e.g., simple SPJ queries, complex queries with multi-joins, correlated subqueries), query costs (*e.g.,* IO usage, memory usage, CPU usage), and the ratio of read-write queries [52], [26], [92]. These features reflect the requirements to execute single queries.

Concurrency Features include the parallelism level and read-write ratios [26], which implicitly indicate the level of access conflicts and require adjustment of the access-control knobs.

Accessed Data Features include the sizes of the base and intermediate tables and the data statistics like tuple selectivities [52], [45], which are vital to determine the resource-related knobs.

Database Scenarios. Based on the above common workload features, we discuss the typical database scenarios (*i.e.,* OLTP, OLAP, and HTAP), together with the workload characteristics and tuning requirements.

(1) *OLTP workloads* denote transactional queries where both point read and write operations may occur. A typical scenario of OLTP tuning is IO-intensive, where many write operations may compete for disk IO. In this scenario, we may need to tune the disk IO-relevant knobs like `effective_io_concurrency`. Besides, we may also need to manage the background write processes (*e.g.,* allocated resources, access frequency) since they may also compete disk IO with user tasks.

(2) *OLAP workloads* involve analytical queries, which may have complex computational patterns and produce large intermediate tables. For those queries, many configuration knobs can significantly affect the query performance. First, for complex analytical queries with multiple joins or complex structures, it is vital to generate high-quality plans by tuning query planning knobs (*e.g.,* selecting proper planning algorithms and enabling effective physical operators). Second, analytical queries may consume many system resources, *e.g.,* costly computations (*e.g.,* aggregates and large-scale intermediate tables) that take up much memory. Third, concurrency control may also be critical when many parallel analytic tasks share different data tables and compete for system resources.

(3) *HTAP* is a mixed workload with transactional activities (*e.g.,* write operations) and analytical workloads over large-scale data. Hence, there can be even more knobs that may affect the workload performance (*e.g.,* both memory and IO allocation knobs). And the increased number of knobs makes tuning HTAP workloads more challenging. For example, we need to balance allocated resources for complex analytics and parallel accesses (*e.g.,* the number of maximum concurrent tasks and allocated buffer sizes).

4.1.2 Database Metrics

Most database systems provide numerous database metrics to reflect the execution state (*e.g.,* in the system views). We can broadly classify them into three categories: tuning objective metrics, database runtime metrics, and table statistics.

(1) *Tuning objective metrics* reflect the database performance (*e.g.,* latency and throughput) or resource usage (*e.g.,* I/O and memory usage). On the one hand, we can utilize tuning objective metrics to identify important knobs (the impact on the metric values). On the other, we rely on the tuning objective metrics to evaluate tuning methods.

(2) *Database runtime metrics* involve (i) database state metrics and (ii) query state metrics. First, database state

metrics reflect the execution state of the database (*e.g.*, the number of read disk blocks and the number of updated tuples). We can utilize the incremental values of the database state metrics before/after execution to reflect the workloads at database level. Second, query state metrics reflect the execution state of queries submitted to the database (*e.g.*, idle or under execution). We can utilize query state metrics for query-level tuning.

(3) *Table statistics* are used to reflect the data distribution, *e.g.*, the width of column entries and the distinct value ratios. We can utilize those metrics to profile the scale of access data, which is vital for knobs like resource allocation.

4.2 Feature Selection Methods

As shown above, there are various workloads and numerous runtime metrics. Some feature selection methods attempted to extract useful features from workloads and runtime metrics to characterize database states. Among the existing works, there are two main approaches: (1) Workload encoding, *i.e.*, extracting the cost/operator features from the query plans; (2) Database metric selection, *i.e.*, using the internal runtime metrics of the database. Table 4 presents a classification of feature selection methods based on the current tuning work.

TABLE 4: Feature Selection Methods.

Category	Selection Method	Brief Description
Workload Encoding	Read/Write ratio [26]	Classify the workload based on the read/ write ratio
	Query characterization [52]	Use query characteristics to represent the workload, i.e., query types/cost
	Occurrence Frequency [92]	Use the TF-IDF feature vector of queries, and classify queries.
Database Metric Selection	FA/PCA [79]	Reduce the dimensionality of the metrics.
	K-Means [79]	Cluster database metrics and remove duplicate metrics.

4.2.1 Workload Encoding

In real scenarios, the workloads are dynamically changing. Those changes should be captured by the characteristics of the workload. Hence, workload encoding aims to extract workload characteristics that could improve the tuning performance and analyze the tuning requirements based on these characteristics. Workload encoding includes two main strategies: utilizing the read/write ratios of the workload and extracting detailed query information to represent the workload execution features.

Read/Write Ratios. Any workload could be classified according to the read/write ratios [26], *i.e.*, the read-only, write-only, and read-write workloads. Furthermore, the read-write workload could also be further divided into fine-grained sub-workloads. For example, dividing the read-write workload into ten categories according to 10%, 20%, $\cdots$, 100% writes. This is a coarse-grained method to capture workload characteristics. Nevertheless, many different workloads may also have the same read/write ratio; thus, this method cannot distinguish such workloads.

Query Characterization. Besides the read/write ratios, the query features could also represent the workload. The challenge for query characterization is that the key features vary in different query statements (*e.g.*, queries may access different tables), which makes it challenging to characterize into fixed-length feature vectors. The query characterization methods mainly utilize a part of query keywords together with some execution information to represent the corresponding workloads. One way to extract the query feature is to build a vector based on the query plan and the cost estimation [52]. The query vector contains three parts, *i.e.*, the query types, the tables used by the query, and the query cost. First, different types of queries may have different tuning requirements (*e.g.*, update operations may involve vacuum mechanisms). And query types can be reflected by the SQL keywords (*e.g.*, insert, delete, select, and group by). Second, tables involved in the queries can determine the volume and schema of accessed data, which can significantly affect performance. Thus, the query vector contains a T dimension vector. The value T is the number of tables in the database. If the query contains these features, the corresponding value is 1; 0 otherwise. In addition, this vector utilizes the query plan generated by the query optimizer. It uses the estimated cost of each operation to represent the execution characteristics. By increasing the dimensionality of workload features, the query characterization approach could describe the workload more deeply. However, the limitation is that it may not be generalized to different workloads, because different workloads may have different tables and columns.

Occurrence Frequency. Even the query execution features (*e.g.*, query costs) may be unavailable. Hence, some workload encoding methods [92], [34] only utilize the SQL query features to conduct workload encoding. First, they selectively extract some keywords for queries that frequently appear (*e.g.*, SELECT, UPDATE) and utilize a probabilistic model called Term Frequency - Inverse Document Frequency (TF-IDF), which helps to identify the importance of extracted keywords (TF-IDF feature vector). Second, they infer the cost levels by utilizing random forest to classify queries based on their TF-IDF feature vectors. Finally, they utilize the TF-IDF feature vectors and average cost levels to represent the corresponding workload. However, it only uses limited features and thus cannot well capture the query characteristics.

Remark. Existing methods characterize the workload from different perspectives. With the above workload encoding techniques, we can characterize a workload into feature vectors. The workload feature vectors can be used in two scenarios. First, the tuning methods can use the vectors to find optimal configuration settings, since the feature vectors provide rich information of the tuning requirements in the queries (*e.g.*, space consumption) (see Section 5). Second, the transfer techniques could utilize the feature vectors to compute the similarity between the new and historical workloads and enhance knob tuning based on similar historical workloads (see Section 5.5).

4.2.2 Database Metric Selection

Besides workload features, runtime metrics are also vital to represent database states and reflect the tuning requirements. For example, Postgres records the state of queries under execution; and the InnoDB engine in MySQL provides execution statistics like read/written pages and data utilization frequencies [79], [89], [52]. There are also hundreds of metrics in the database, most of which may not

effectively represent the database state and cause a great tuning burden. Moreover, among those metrics, some metrics provide useless or duplicated information. Thus, metric selection methods aim to reduce the metrics to a smaller set based on the correlation (similarity) between metrics.

Metric selection methods mainly include two main steps, *i.e.,* reducing the dimensionality of metrics and removing duplicated metrics. First, they utilize dimensionality reduction techniques, *e.g.,* factor analysis (FA) used in OtterTune [79], or Principal component analysis (PCA) used in HUNTER [9], to reduce the dimensionality of metrics. In FA, it combines the knob configurations with similar coefficients, computed based on metric values under different configurations. In PCA, they use the linear combination to convert metrics into multiple uncorrelated principal components. Next, for the processed metrics, some metrics have similar functionality for knob tuning (*e.g.,* page number metrics in different scales). To remove the redundant metrics, they utilize clustering algorithms (*e.g.,* K-Means) to cluster similar metrics and only pick one metric from each cluster to represent the database state. Existing metric selection methods could reduce the configuration search space and accelerate the training and tuning process of tuning methods. However, pre-selected metrics could not handle the dynamic changing workloads and database scenarios.

Remark. Selected database runtime metrics, *i.e.,* the value changes of those metrics before/after executing the workloads, can be used to represent the workloads. Moreover, transfer techniques could also use these metrics to map similar historical workloads and reuse prior knowledge (*e.g.,* reusing historical sampled knob settings).

5 TUNING METHOD

Configuration knob tuning aims to find the optimal knob settings that meet user requirements. We can formalize the knob tuning problem into an optimization problem, which given a workload and a suite of adjustable knobs, select the optimal knob settings under predefined objectives and constraints (*e.g.,* maximizing the throughput within limited memory space). Traditional tuning methods rely on DBAs to manually tune the knobs. DBAs first choose an initial configuration setting based on their experience. Then DBAs run a query workload with this setting and collect the performance metrics from the database system. If the performance is good enough, DBAs adopt this configuration setting and terminate the tuning process; otherwise, DBAs diagnose the performance, select another configuration setting based on their tuning experiences and repeat the above steps. However, these manual tuning methods are time-consuming and highly dependent on the DBA's experience.

Many tuning methods are proposed to address these limitations. We categorize existing studies into four categories: *(1) heuristic tuning methods*, *(2) Bayesian optimization-based tuning methods*, *(3) deep learning-based tuning methods*, and *(4) reinforcement learning-based tuning methods*. In this section, we review these tuning methods. Figure 5 illustrates the timeline of these tuning methods. Table 6 present the classification of database tuning methods and the details of these methods, respectively.

5.1 Heuristic Tuning Methods

We can broadly classify heuristics-based methods into rule-based methods and search-based methods. First, the rule-based methods borrow the idea from the manual tuning method, which constructs some rules from DBAs' experiences (or database manual) and use these rules to guide the tuning process [1]. The rule-based methods can accelerate the decision-making process of knob tuning compared with DBAs. Second, the search-based methods divide the configuration space into several subspaces, run a given workload with this configuration subspace, select the best subspace, and then search the neighbours of this selected subspace to find the knob settings.

Rule-based Method. The rule-based tuning methods use pre-specified tuning rules (or what-if questions) to tune the performance. For example, an example tuning rule from the manual of Postgres is "a reasonable starting value for *shared_buffers* is 25% of the memory in your system"; one tuning rule in the MySQL manual is that the value of *innodb_buffer_pool_size* could be set between 50 and 75 percent of the system memory. In addition, PG-Tune [1] makes configuration recommendations by asking users for basic information about the Postgres database they are using and the details about their hardware environment. Note that the information of the Postgres' version and the number of CPUs affects the setting of the knobs because a new version will introduce new knobs. For versions below 9.5, *max_worker_processes* is not available. Similarly, *max_parallel_workers_per_gather* supports versions higher than 9.5, and *max_parallel_workers* supports v10 and higher versions. The setting of the knob values also follows the rules. These rules include that the values of *max_worker_processes* and *max_parallel_workers* are equal to the number of CPUs and the value of *max_parallel_workers_per_gather* is half the number of CPUs.

Given a set of tuning rules, Probability Theory [69] is proposed to determine the knob settings to maximize the expected performance. To determine whether a rule can be used for a given workload, it reasons the characteristics of the workload and the system states. If it reasons that the performance can be improved, this rule can be used. For multiple rules with correlations, it uses conditional probabilities to do the reasoning. Based on probability theory and decision theory, it can quantify the uncertainty and determine the best possible decisions given the uncertainty [69].

The advantage is that they are simple and fast, as the heuristic tuning methods can improve the performance by replacing DBAs with effective tuning rules. However, they may not find the optimal configuration, because knob tuning is affected by many factors, and it is difficult to obtain the optimal setting through these simple rules. Moreover, they cannot adapt to the workload and data changes.

Search-based method. The search-based tuning methods employ a hierarchical search method to select the knob setting. Firstly, they partition the knob space into several subspaces, run a workload with each subspace, and select the subspace with the best performance. Then they search for the neighbour of this subspace by changing the values of the knobs. There are two search-based methods: an ensemble method combining multiple search methods and a

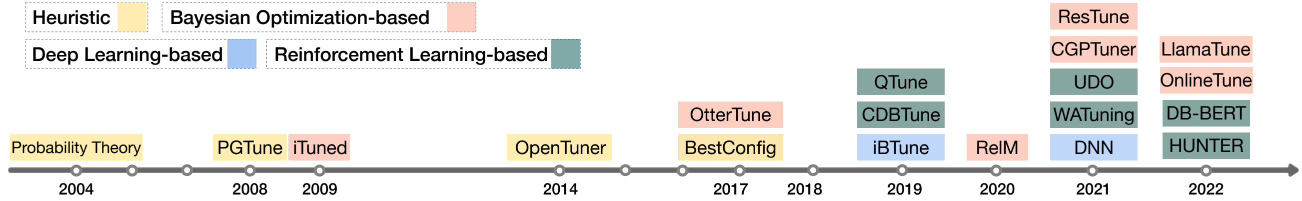

Fig. 5: Timeline of Existing Knob Tuning Methods.

sampling-optimization method.

(i) An Ensemble Method by Combining Multiple Search Methods. OpenTuner [5] is an automatic configuration tuning framework that assembles several search-based techniques to find the configuration with good performance. First, it selects some initial configuration samples. Then it uses a user-defined measurement function to evaluate the quality of these configuration samples. Next, based on the estimated performance, it uses several configuration search methods, *e.g.,* multi-armed bandit and particle swarm optimization, where each search method can recommend the next search example (*e.g.,* by learning a curve between configuration and performance). Based on the quality of recommended search samples for each method, it can evaluate the quality of each search method and allocate the search methods with better performance more chances to recommend samples, while those performing poorly are allocated fewer chances. Iteratively, OpenTuner can recommend a good configuration setting. However, the quality of the ensemble search algorithm is highly dependent on the quality of the initial configuration sample and requires high-quality configuration settings.

(ii) A Sampling-Optimization Method. BestConfig [98] adopts a sampling-optimization method to find the knob configuration. It collects some configuration samples and optimizes these good samples by the search algorithm. It first uses a Divide and Diverge Sampling (DDS) method to select configuration samples and evaluate them (by running a workload or user-defined measurements). Then, it uses the Recursive Bound and Search (RBS) algorithm to optimize these samples by searching their nearby configurations. It repeatedly calls the above steps within a given resource limit. However, the sampling process is time-consuming and may also miss better knob settings because some search space may be ignored.

Remark. Although the search-based tuning method cannot guarantee a globally optimal knob setting, it can find a near-optimal configuration that is usually close to the best possible database knob settings. Compared with the rule-based tuning method, the search-based methods do not rely on historical experience and can adapt to different database systems. Nevertheless, they have some non-negligible drawbacks. First, the process of sampling and searching is time-consuming. Second, the sampling process may also miss better configuration settings, because some search space may be ignored.

5.2 Bayesian Optimization-based Tuning Methods

To obtain high-quality configuration settings, some works propose to combine traditional machine learning in knob tuning. Compared with heuristic methods, they adopt the Bayesian optimization approach to estimate tuning benefit and better guide the tuning procedure [22], [43], [79], [88].

Basic Idea of Bayesian Optimization. Bayesian Optimization (BO) [24] is a sequential model-based iterative algorithm. It uses a surrogate model to approximate the objective function and sequentially update this model with the new data points observed via iterations. Typically, BO initializes the surrogate model with some sampled data points. The sampling methods include random sampling and Latin hypercube sampling. For each iteration, BO uses an acquisition function to decide where to sample new points with the exploration–exploitation policy.

For the knob tuning problem, the exploration-exploitation policy in BO could be illustrated as follows:

(i) Exploitation: Make slight modifications to the knob values based on the current configuration.

(ii) Exploration: Try to explore some knob values based on unexplored configuration space.

The BO-based methods use the sampled knob settings and their performance as the starting points to kick off the tuning process. Then, it repeats the modelling and updating steps until finding a satisfactory knob setting or reaching the termination condition (time/resource constraints).

Key Modules in BO-based Knob Tuning. Apart from the *Surrogate Model* and *Acquisition Function*, in BO-based tuning methods, there are two other key modules. (i) *Model Initialization*, which initializes the tuning model with some knob setting samples, which will affect not only the convergence speed but also the selected knob setting quality; (ii) *Database Features* involve knob settings and possibly other database metrics (*e.g.,* profiled execution statistics), where the surrogate model could be utilized to approximate the knob-performance relationship. Figure 6 illustrates the detailed key modules in different BO-based tuning methods.

(1) Surrogate Model. First, BO-based methods need to decide the suitable surrogate model, which maps the relationship between knob settings and database performance. Among BO-based tuning works, Gaussian process (GP) is the most widely used surrogate model [74], [79], [45], [92], [10], [93] (easy to approximate different distributions). In addition, Zhang et al. [91] try out another two surrogate models, *i.e.,* SMAC (a random forest-based surrogate) and Tree-structured Parzen estimator (TPE). And LlamaTune [40] separately evaluates the sample efficiency of BO with GP and SMAC. For these surrogate models, both GP (for integer knobs) and SMAC (for both integer and categorical knobs) model the knob-performance relationship by identifying the predictive distribution; while TPE relies on the density functions to build probability distributions for high-performance knob settings and bad knob settings [85].

(2) Acquisition Function. The acquisition function could

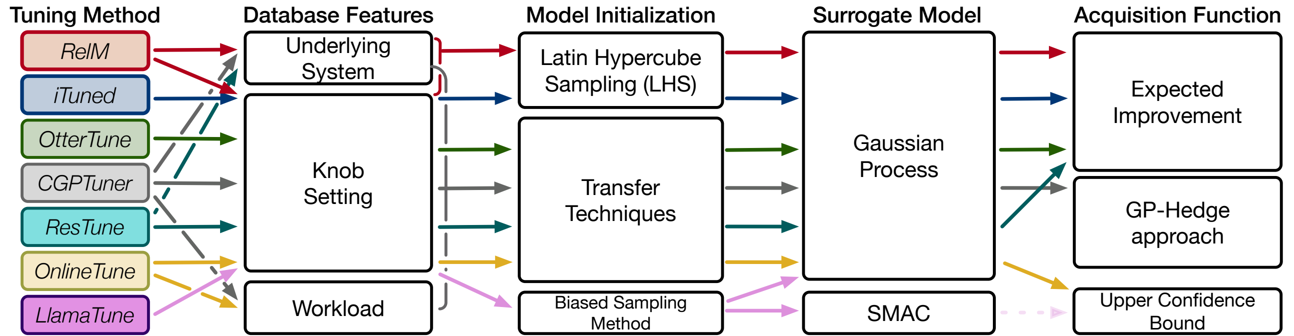

Fig. 6: The Workflow of Bayesian Optimization-based Tuning Methods.

decide where to sample. It balances the prediction and the uncertainty of the surrogate model, which ensures the exploration of the unobserved configuration space and the exploitation of the well-performed configuration space. There are some commonly used acquisition functions, *i.e.*, the probability of improvement (PI), expected improvement (EI), and upper confidence bound (GP-UCB). The PI function aims to maximize the probability of improvement, and it is biased toward exploitation. Expected improvement is a commonly used acquisition function in many BO-based methods [22], [79], [45], [92]; it not only considers the probability of improvement (exploitation) but also the potential for improvement (exploration). GP-UCB selects knob settings in the area of maximum confidence, and OnlineTune [93] uses GP-UCB to achieve the exploitation-exploration trade-off. Apart from using a single acquisition function, CGPTune [10] also utilizes GP-Hedge [37] to ensemble different acquisition functions (adaptively select the best acquisition function in each iteration) so as to improve the performance.

(3) Model Initialization. To initialize the surrogate model, BO typically begins with some sampled knob settings. As random sampling may not get high-quality samples with high dimensional space, iTuned [22] and RelM [45] utilize the Latin hypercube sampling to produce more stable samples than random sampling (see Section 3.2.2) [30]. Note that, since the sampling methods may not find good initial samples, the transfer techniques could help to select high-quality initial samples based on historical workloads (see Section 5.5). In addition, LlamaTune [40] designs a biased sampling-based method to handle knobs with special values. For example, in MySQL, the value 0 of *innodb_thread_concurrency* indicates no limitation of the supported parallel threads, which is frequently used as the default value. LlamaTune assigns a biased sampling probability for the value 0 of *innodb_thread_concurrency* so that 0 could be easily sampled during the tuning process.

(4) Database Features. To model the knob-performance relationship, a straightforward method is to use the knob settings as the input data. However, the relationship and tuning requirements may change in new tuning scenarios. Thus, some additional database features are considered. Based on existing BO-based methods (Figure 6), we summarise three adopted database features: (i) *Knob Setting Samples*, (ii) *Workload/Runtime Features*, (iii) *Underlying System Features*.

(i) Knob Settings. Knob settings involve the selected knobs and their values. In some early tuning methods (*e.g.*, iTuned [22], OtterTune [79]), they only use the knob setting samples and their performance to build the tuning models.

(ii) Workload/Runtime Features. Different workloads and their execution state may lead to different optimal knob settings. To capture the feature of changing workloads, OnlineTune [93] and CGPTuner [10] separately take the query (*e.g.*, query arrival rate, query type ratio) and runtime metrics as the additional input of the tuning model. The query features are valuable to reflect workload trends, but the query statements may not be available (privacy issues); while the runtime metrics characterize the physical behaviours of the workloads, but they are costly to collect and hard to generalize to different systems.

(iii) Underlying System Features. In addition to the changing workloads, the difference between the production environments of databases (*e.g.*, hardware, software versions, and data features) may also affect the performance of tuning methods. Therefore, some BO-based tuning methods [10], [45] attempt to use the knowledge learned from the environment to train tuning models. First, CGPTuner [10] and RelM [45] consider the features and interactions of different system levels (*e.g.*, memory control across workloads, container, and JVM). Second, ResTune [92] uses resource utilization (*e.g.*, CPU, memory, and I/O usage), and OnlineTune [93] characterizes the data distributions (*e.g.*, involved data tuples, used indexes) to handle environment changes.

Remark. BO-based tuning methods can efficiently find high-quality knob settings with the exploration-exploitation strategy, and generally outperform the heuristic methods [80], [91]. However, BO-based methods (especially those that adopt the Gaussian process) cannot efficiently scale to large configuration space and fall into sub-optimum. Besides, for the surrogate models in BO-based tuning methods, the Gaussian process is mainly suitable for optimizing continuous knobs; while SMAC can support both continuous and category knobs and performs better than the Gaussian process for relatively large configuration space [91]. However, GP can efficiently explore the untouched configuration space, for which the tree model in SMAC could not promise high predictive quality.

5.3 Deep Learning-based Tuning Methods

In addition to BO-based tuning methods, deep-learning-based tuning methods could also iteratively estimate and improve the tuning performance. In the basic deep-learning-based method [80], they utilize a deep neural network

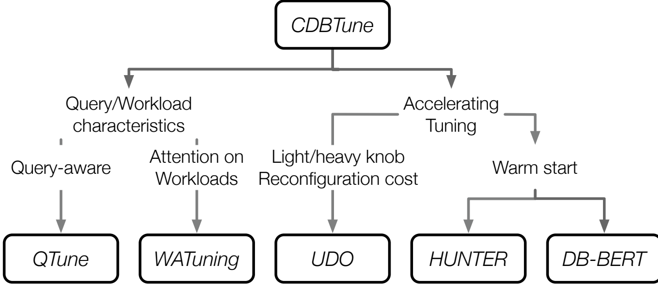

Fig. 7: Reinforcement Learning Tuning Methods.

TABLE 5: Reinforcement Learning Methods.

Method	State	Policy
CDBTune [89]	Runtime metric	Deep Deterministic Policy Gradient
QTune [52]	Runtime metric, Predicted metric change	Double-State Deep Deterministic Policy Gradient
WATuning [26]	Weighted runtime metric	Deep Deterministic Policy Gradient using pre-trained model
UDO [82]	Performance metric	Delayed Hierarchical Optimistic Optimization (a mcts algorithm)
DB-BERT [77]	Runtime metric	Double Deep Q-Network algorithm using pre-trained language model
HUNTER [9]	Runtime metric	Deep Deterministic Policy Gradient

(DNN) as the tuning model. The DNN they use has two hidden layers (fully connected), and each layer has 64 neurons and utilizes ReLU as the activation function. To avoid suboptimum, they add Gaussian noise to the neural network to control the possibility of exploring unseen configurations.

Moreover, previous BO-based methods do not consider the resource utilization problem, which is vital for cloud databases. Hence, iBTune [73] proposes a deep learning-based estimation model to address this problem. It relies on a neural network to tune the buffer pool size. Given a database instance, it first selects some candidate buffer pool sizes from similar instances (*e.g.,* similar logic reads, CPU usage), which own a higher cache miss ratio and worse performance. The reason is that those buffer pool sizes are possibly (1) smaller than the instance's buffer pool size (improving resource utilization) and (2) may keep the performance within a threshold (*e.g.,* worse than the default performance by 5%). Next, it uses the deep learning model to predict the performance for the selected buffer pool sizes, and finally selects the minimal size whose predicted performance is within the threshold. The limitation of iBTune is that it can only predict the buffer pool size and cannot be generalized to other tuning objectives.

Remark. Compared with BO-based methods, deep learning-based methods (i) can effectively estimate the tuning performance, and (ii) do not need to repeatedly run workloads to evaluate the selected settings, which can reduce the tuning overhead. However, DL-based methods require a large number of prepared training samples to well train the tuning model.

5.4 Reinforcement Learning-based Tuning Methods

To improve the performance for large configuration space without much historical data, reinforcement learning (RL) based methods have been proposed [89], [52], [26], [82], [86], [90], [11], [77], which explore promising configuration space with trial-and-error strategies without limiting the number of knobs. Figure 7 demonstrates the technical comparisons of existing RL-based tuning methods. These methods use deep neural networks to enable reinforcement learning (RL) to scale to large configuration space.

Basic Idea. In reinforcement learning (RL), there are six modules (*i.e.,actions, rewards, agent, environment, policy, state*), and the essential of RL is learning the tuning strategy by the interactions between environment and agent. RL-based methods map the knob tuning problem into the six modules in the reinforcement learning framework. That is, they take the database as the *Environment*, runtime metrics as the *State*, the tuning model as the *Agent*, knob tuning as the *Action*, the performance change after tuning as the *Reward*, and the tuning evaluation model (*e.g.,* deep learning) as *Policy*. Iteratively, the agent recommends tuning actions based on the state features and updates the tuning policy by the reward to optimize the database performance.

Therefore, as shown in Table 5, there are two main challenges for RL-based tuning methods: (1) how to design the *Policy* to find high-quality knob settings, and (2) how to design the *State* to reflect the database execution statistics. The design of the *Policy* decides the quality and the efficiency of the tuning model, and the selection of *State* determines how accurate the database state could be.

In CDBTune [89], it uses the deep deterministic policy gradient method (DDPG) to optimize the knob settings in high-dimension configuration space. Specifically, it selects 64 database runtime metrics (*e.g.,* the number of of read-/write disk pages) as the *State* to represent the database state and uses the DDPG algorithm as the *Policy* to conduct knob tuning, where the actor function decides the next tuning action (*i.e.,* selecting more promising knob settings) and the critic function estimates the long-term tuning benefit to guide the learning of the actor function. Although CDBTune can work well for static workloads (by characterizing the database states), CDBTune cannot capture the incoming query/workload information and the recommended knob settings (on out-of-date metrics) may not lead to high performance on the new workloads.

Query-aware RL-based Tuning. To adapt the RL tuning model to changing queries/workloads, Qtune [52] and WATuning [26] have integrated query/workload characteristics into the RL tuning model.

(i) **Predict the database metric.** QTune [52] builds a feature vector as the *State*, containing the query information (*e.g.,* query types; used tables) and execution information (*e.g.,* estimated operator costs) to illustrate the query characteristics. QTune builds a deep neural network to predict this ΔS based on the feature vector during the processing of query implementation. QTune adopts a modified DDPG model as the *Policy* to enable knob tuning based on both the current runtime metrics S and predicted ΔS (*State*). QTune integrates the encoding of incoming queries into the tuning model to improve the tuning performance. But it has a limitation on adaptivity since it cannot adapt to new data tables and columns.

(ii) **Pre-trained RL model.** WATuning [26] uses an attention-based network to adapt to different workloads. It uses a controlled module and divides workloads into multiple

categories according to the read-write ratio of workloads. For each category, it pre-trains a tuning model. It uses the attention-based model to make the runtime state of the database system get different weight values for different workloads and thus make it possible to reference finer-grained tuning. When a new workload comes, WATuning calculates the read/write ratio and selects a pre-trained tuning model with a similar ratio to recommend the configuration setting. However, workload characteristics are not only related to the read/write ratio but other more important information (*e.g.,* predicates and costs).

Accelerating RL Tuning. Traditional RL-based tuning methods require many iterations during model training so as to learn an optimization strategy. Due to the complexity of database system, it is time-consuming for the RL-based tuning model to learn the optimization strategy. To accelerate the tuning process of the RL-based tuning model, UDO [82], HUNTER [9], and DB-BERT [77] are proposed.

(i) **Overhead-aware RL Tuning.** Tuning some knobs requires the database restart, while others do not. This characteristic of knobs may affect the performance of the tuning model. Therefore, UDO [82] attempts to reduce the restarting overhead by classifying knobs into heavy knobs (which require database restart), and light knobs (which do not require the restart). UDO uses a delayed RL algorithm, named delayed Hierarchical Optimistic Optimization (delayed-HOO), to select the knob setting for heavy knobs. It collects and reorders those knob settings with similar heavy knobs. Therefore the reconfiguration cost of tuning one heavy knob could be shared among multiple similar knob settings. After the heavy knobs are determined, UDO uses the RL algorithm without the delayed feedback to find the knob settings for light knobs. The limitation may affect the tuning performance as delayed RL may miss some important settings. Note, different from the DDPG algorithm, delayed-HOO selects knob settings that can maximize the *upper confidence bound* function, which considers factors like the average of available tuning rewards (including delayed rewards) and the selection frequency.

(ii) **Warm Start RL Tuning.** The RL model needs to retrain a new model for each tuning task and does not use the existing tuning data/experience. To address this problem, HUNTER [9] uses generated configuration samples, and DB-BERT [77] uses the extracted tuning rules to warm-start the RL model. HUNTER uses some rules extracted from user requirements and the genetic algorithm (GA) to generate configuration samples. Then, HUNTER uses the DDPG to optimize the generated configuration. In addition, HUNTER also uses the knob selection method (random forest) and feature selection method (PCA) to reduce the dimensionality of the configuration space. DB-BERT uses a pre-trained language model (BERT model) to analyze the text document (*e.g.,* database manual, search engines) and extract some useful tuning hints. Then, it uses these tuning hints to assist the Double Deep Q-Network (DDQN) algorithm in selecting the knobs as well as the suggested values. Note that, they adopt DDQN (a simpler RL algorithm than DDPG) because the hint space is much smaller than the origin configuration space. As both HUNTER and DB-BERT rely on tuning rules/hints, the collection of high-quality rules/hints is expensive.

Remark. RL-based tuning methods learn the tuning strategy by the interactions between the database and tuning model, which are suitable for high-dimensional configuration space exploration. However, RL-based methods require much higher tuning overhead than BO-based tuning methods. For example, in [80], within 150 iterations, RL-based methods could not fully optimize these knobs and perform worse than BO-based methods, because BO-based methods avoid approximating the complex metric-knob relationships and efficiently explore suitable knob settings. However, with higher tuning overhead (*e.g.,* 200 iterations in [91]), BO-based methods stop at sub-optimal knob settings, while RL-based methods could find better knob settings.

5.5 Transfer Techniques

In real scenarios, the workloads could dynamically change. For example, the workload pressure may change significantly over time (morning to evening, weekday to weekend, or workday to holiday). Hence, there is no single knob setting that can be optimal for diverse workloads. However, learning-based methods like classic reinforcement learning may not efficiently transfer to new workloads. In other words, whenever the workload changes, these methods need to train from scratch, which is time-consuming and wastes many system resources. To adapt to dynamic workloads, transfer techniques aim to improve tuning efficiency by migrating the knowledge learned from historical tuning tasks to new tuning tasks. Currently, three transfer techniques have been proposed in knob tuning [79], [52], [26], [88], [26], [10], including *workload mapping*, *learned workload embedding*, and *model ensemble*. These transfer techniques could enhance the adaptivity of tuning methods for dynamic workloads.

Workload Mapping. Traditional Gaussian process methods randomly sample configurations for a new workload, which takes much time to explore many configurations. Instead, to enhance the cold-start tuning, OtterTune [79] and CGPTuner [10] store historical workloads in the repository and matches the most similar historical workload to build the initial tuning model. Specifically, the workload mapping technique uses the feature vector extracted from runtime statistics to represent the workload and calculates the similarity (*e.g.,* Euclidean distance) between the feature vector of the target workload and the vectors of historical workloads in the repository. With the matched workload, it takes the corresponding tuning data (*e.g.,* high-quality configurations of the matched workload) to learn a new GP model for the new workload.

Although workload mapping can directly match most similar historical workloads, it has two limitations. First, it may not find any similar historical workload from the tuning repositories, and in this case, it needs to train a new tuning model from scratch. Second, it requires executing the workload to obtain the statistics information for workload mapping, which is relatively expensive.

Workload Embedding. Another transfer technique is to extract the workload features and integrate the features into the tuning methods [52], [26]. In this way, it can utilize

TABLE 6: Summary of Knob Tuning Methods.

Category	Work	Tuning Objective	Knob Selection	Feature Selection	Tuning Method	Transfer Techniques
Heuristic (Rule-based)	Probability Theory [69]	Performance	Tuning experience	N/A	Probability theory and decision theory	N/A
	PGTune [1]	Performance	Tuning experience	N/A	Empirical rules	N/A
Heuristic (Search-based)	OpenTuner [5]	Performance	Tuning experience	N/A	E.g., multi-armed bandit, particle swarm optimization	N/A
	Bestconfig [98]	Performance	Tuning experience	N/A	Configuration sampling (DDS) + search algorithm (RBS)	N/A
BO-based	iTuned [73]	Performance	Sensitivity analysis	N/A	Gaussian process	N/A
	OtterTune [79]	Performance	Lasso	Factor Analysis K-Means	Gaussian process	Workload mapping
	RelM [45]	Performance	Tuning experience	N/A	Gaussian process	N/A
	ResTune [92]	Performance Resource utilization	Tuning experience	Occurrence frequency	Gaussian process	Model ensemble
	CGPTuner [10]	Performance	Tuning experience	N/A	Contextual Gaussian process	Workload mapping
	OnlineTune [93]	Performance	Tuning experience	Query/Data characterization	Contextual Gaussian process	Model ensemble
	LlamaTune [40]	Performance	Tuning experience	N/A	Gaussian process; SMAC	N/A
DL-based	iBTune [73]	Performance	Tuning experience	N/A	Convolutional Neural Network	N/A
	DNN [80]	Performance	Lasso	Factor Analysis K-Means	Deep Neural Network	N/A
RL-based	CDBTune [89]	Performance	Tuning experience	Read/Write ratio	Deep Deterministic Policy Gradien (DDPG)	N/A
	QTune [52]	Performance	Tuning experience	Query characterization	Double-State DDPG	Workload embedding
	UDO [82]	Performance Resource utilization	Tuning experience	N/A	A MCTS algorithm with delayed feedback	N/A
	WATuning [26]	Performance	Tuning experience	Read/Write ratio	Attention + DDPG	Workload embedding
	DB-BERT [77]	Performance	NLP (BERT)	N/A	Language model + DDQN	N/A
	HUNTER [9]	Performance	Random forest	PAC	Genetic algorithm + DDPG	N/A

learning models to learn the tuning requirements of different workloads. To this end, Li et al. [52] proposed to design a workload embedding model which takes as input the common query features of workloads (*e.g.,* operator types, query costs) and outputs the predicted database state changes (*e.g.,* the number of read blocks). The workload embedding model is based on a feed-forward neural network that extracts the tuning-related features based on the supervised losses and embeds those features into database state changes. Similarly, WATuning [26] includes an attention neural network in the reinforcement learning framework that captures the workload characteristics and enhances the convergence speed on new workloads.

The advantage of learned workload embedding is that it can directly extract tuning features of different workloads rather than repeatedly executing them, which is generally unaffordable in real scenarios. But the weakness is that the embedding model still needs a large number of prepared training samples to learn the mapping from workloads to future database state changes.

Model Ensemble. Instead of matching with historical workloads/models [88], [26], another transfer technique aims to prepare a cluster of well-trained tuning models (base models) on typical workloads and assemble those models to generalize for unseen workloads [92]. In the process of adapting to a new workload, they adopt two strategies, *i.e.,* static weight learning, and dynamic weight learning. In the *static weight learning* phase, they have no execution feedback, and they initialize a tuning model based on the weighted sum of all the base models, where the weights are equal to the similarity in workload features. This phase helps to utilize historical knowledge to solve the cold-start problem. Moreover, in the *dynamic weight learning* phase, after obtaining enough observations on the new workload, they assign weights for each base model based on the accuracy that the base model predicts for the new workload.

The advantage of the *model ensemble* is that they rely on the weighted sum of historical models to generalize to new workloads, which is more robust than matching with the most similar workload. However, the performance of *model ensemble* significantly relies on the selected base model, *i.e.,* whether those models are representative enough. It needs to execute workloads for several iterations after assembling the base models (dynamic weight learning). It is vital to transfer the tuning model without repeatedly executing workloads.

Remark. The above methods can only adapt to different workloads. If the database kernel (*e.g.,* from Postgres to MySQL) or schema (*e.g.,* add or remove columns) changes, these transfer techniques may not work, because the historical knowledge is rarely usable for the new database or data distributions. Thus it calls for new transfer techniques to support such cases. Moreover, if the hardware changes, it is also important to adapt to new hardware. Intuitively, if the tuning models use the basic features, e.g., IOPS, read/write latency, then they can be easy to transfer, as these knowledge can be reused in different scenarios; on the contrary, if the tuning models use more specific features, e.g., query features, database features, then the models are hard to transfer, as different workload may have different query features. But these models may get much better tuning performance as they use more features. Thus we should leverage different features to achieve different tuning objectives.

TABLE 7: Example Tuning Techniques in Relational Database Products.

	Tuning Objective	Knob Selection	Feature Selection	Tuning Methods	Deployment
Oracle [20], [44]	Performance; Robustness	DBAs	Metric Monitor	Hybrid	Kernel
SQL Server [12]	Performance; Robustness	DBAs	Data Statistics	Rules	Kernel
DB2 [68]	Performance	DBAs	Data Statistics	heuristic	Plugin

6 DEPLOYING CONFIGURATION TUNING INTO DIFFERENT DATA SYSTEMS.

In this section, we first discuss how to deploy existing knob tuning techniques into real systems, *e.g.*, relational databases (see Section 6.1) and big-data systems (see Section 6.2). Next, to extend the scope of this survey, we discuss three configuration tuning problems beyond knobs, including index tuning, view tuning, and partition tuning (see Section 6.3).

6.1 Relational Databases

Deployment Challenges. First, relational databases support various dynamic workloads, and it needs to characterize the workloads so as to decide the tuning requirements, *i.e.*, choosing the proper knobs to optimize relevant modules. Second, the tuning methods should be lightweight (low tuning overhead) and adaptive (supporting different new applications and hardware environments).

To address these issues, many database vendors adopt various tuning methods (*e.g.*, empirical rules, hybrid algorithms) as either plugin tools [12], [68], [95], or components in the database kernel [44], [50]. As shown in Table 7, here we showcase the techniques in four relational database products (*i.e.*, Oracle, SQL Server, DB2, openGauss), which have accumulated much tuning experience and practiced the tuning tools in real scenarios [17], [68], [75], [12], [50].

Knob Selection. First, most databases only select a small part of important knobs for specific scenarios. They adopt the empirical methods which rely on the DBAs to observe the scenario characteristics (*e.g.*, workload types, hardware settings) and select relevant knobs based on the tuning experience (see Section 3). For database products, they generally tune 10-15 knobs that are taken as most effective by experience [80]. For example, in Oracle [17], since databases are deployed to support various workloads, the workloads may contain operators with different memory requirements (*e.g.*, hash join and sort operators that require much memory, and OLTP queries that process small input data). Thus, in some scenarios, they mainly tune the memory sizes (*e.g.*, *cache size*, *multi-pass size*) based on the workload characteristics. Similarly, in DB2 [68], [75], since tuning memory knobs can significantly improve the performance, they provide an empirical model that evaluates the performance (*e.g.*, latency caused by missed pages in the buffer pool) after increasing the buffer pool size and heuristically recommends suitable sizes with maximum benefits within a given threshold.

Feature Selection. Besides, databases rely on runtime metrics to tune the knob values (see Section 4). For example, to proactively optimize the performance, Oracle [20] provides an automatic metric monitoring module, which collects critical query runtime metrics and knobs settings (*e.g.*, allocated memory sizes, number of CPUs, and query optimizer settings), analyzes abnormal metrics, and tunes relevant knobs. Besides, to tune memory-management knobs, databases like SQL Server [12] collect both the data statistics (*e.g.*, access frequency) and memory-access costs, based on which it can predict whether the current query execution time has degraded and adjust the memory usage (*e.g.*, allocating more memory for joins, decreasing the size of shared memory) when the performance goal is not satisfied.

Tuning Methods. Lastly, traditional databases adopt heuristic tuning methods (see Section 5). For rule-based tuning, databases (like Oracle [44], SQL Server [12]) deploy some what-if rules inside the kernels, which can automatically adjust knobs for typical scenarios (*e.g.*, recommend the memory sizes based on simple features like table sizes). However, the rule-based method is coarse-grained, *e.g.*, the memory allocated based on maximum table size may be inadequate for workloads with multi-join queries. For search-based tuning, databases like DB2 [68] adopt bounded search to heuristically search local-optimal configuration.

However, heuristic methods have limited optimization ability for complex scenarios (*e.g.*, involving multiple modules) and need to be manually maintained. Hence, some databases [50], [62], [63], [60], [4] deploy both rule-based and machine-learning methods. For example, openGauss [50], [95] supports two-phase online tuning. First, at the initial phase, there are rare historical tuning data, and thus it utilizes rule-based tuning methods to detect the problems in existing knob settings using heuristic rules (*e.g.*, *too high max_connections with a small number of CPU cores*). Second, it collects the execution information as training samples and utilizes the samples to train the ML model (*e.g.*, reinforcement learning). After the model has converged, it will replace with the knob settings recommended by the ML model, which is trained for the current workload and has a higher possibility of finding better knob settings.

Moreover, there are also some proposals (*e.g.*, SageDB [42], Xuanyuan [51]) that provide insights into how to deploy configuration tuning with machine learning. First, SageDB [42] aims to conduct knob tuning based on the learned distribution of the dataset and workloads. Second, Xunayuan [51] proposes to utilize deep reinforcement learning to tune various aspects of database configurations (*e.g.*, system knobs, software upgrading, partition scheme) in different granularities (*e.g.*, query-level, workload-level).

Remark. Because of the limitations in machine learning (*e.g.*, unexplainable "black box" model, poor adaptability for various datasets), most databases still integrate heuristic tools or rely on DBAs to conduct knob tuning, but also suffer from the sub-optimal problem (*e.g.*, allocating too large memory space and causing resource waste). Besides, most deployed tuning methods are in the form of plugin tools, which are not as efficient as deploying them into the database kernels. Hence, in the future, it is vital to support more lightweight (*e.g.*, tuning within seconds or using limited resources on the replica nodes) and intelligent (*e.g.*, learning from historical data) tuning methods and implement them into the database kernel with the support of in-database ML algorithms. Moreover, proactively mon-

itoring the performance is also vital to knob tuning and other optimization problems, since it discovers and analyzes abnormal metrics and helps to make optimization decisions.

6.2 Big Data Analytics Systems

In addition to database systems, there are also some tuning methods for big data analytics systems [32], [56], [45], [54].

Deployment Challenges. Similar to databases, big data analytics systems (*e.g.,* Hadoop, Spark) also have numerous tunable knobs, which can significantly affect the system performance and stability. However, they have three main features that are different from database systems. First, they adopt a more complex multi-level distributed architecture. For example, they need to tune memory management knobs on multiple levels (*e.g.,* applications, containers, and virtual machines) [45]. Second, the code-based applications on Spark are more complex than SQL-based database queries, which brings challenges to feature encoding. Third, they separate computation nodes and storage nodes. And the nodes may have different knobs and tuning requirements.

Code-based Feature Selection. To deploy knob tuning in big data systems, it is vital to characterize application features so as to predict performance and provide tuning ability. However, compared with database queries, applications have two characteristics. First, they use Spark codes to compute results, which are more complex than SQL statements. Second, since Spark generally processes large-scale data in memory, it is more sensitive to CPU and memory bandwidth [7], [54]. Thus, knob tuning in Spark needs to be aware of configurations that may affect the application performance (*e.g.,* the sizes of shuffle partitions, the caching policies), and can extract useful information from the applications (*e.g.,* map-reduce operations, code structures, and data sizes) to optimize different spark applications.

Multiple-level Tuning. It aims to support multi-level tuning, *i.e.,* job-level tuning, workflow-level tuning, workload-level tuning. Starfish [33] is a self-tuning analytics system built on the Hadoop stack. The goal of Starfish is to ensure Hadoop could have a good performance without the need for users to understand and manipulate the knobs. Starfish considers multiple levels of workloads running on Hadoop and proposes multiple-level tuning that optimizes different modules with empirical rules, *i.e.,* job-level tuning, workflow-level tuning and workload-level tuning. Through the interactions with those modules, Starfish acquires the capability of self-tuning.

Remark. In big data analytic systems, most automatic knob tuning methods use rule-based methods. One of the most important reasons is that executing the applications generally requires much time, and it is hard to collect sufficient training samples for machine learning-based methods. Thus, it is vital to design prediction models that can provide relatively accurate performance estimation on small sets of training samples.

6.3 Configuration Tuning Beyond Knobs

Besides knob tuning, there are some other configuration problems that are vital to the database performance, including index configuration, view configuration, and partition configuration. Next, we review existing studies.

Index Configuration. First, building indexes on appropriate columns can improve query performance. However, it is expensive to recommend and build indexes with a large number of columns. Generally, there are two sub-problems in index configuration: (1) Index benefit estimation aims to evaluate the benefit/cost of building an index; (2) Given a set of candidate indexes (*e.g.,* some possible columns), index configuration selects some candidates to build indexes within a given index size budget. The former is a regression problem, and the latter is an NP-hard optimization problem. Existing index configuration methods can be broadly classified into heuristic or exact methods [13], [78], [14], [18] and learning-based methods [58], [41].

First, for *index benefit estimation*, it is challenging to improve the estimation quality. To this end, Ding et al. [21] proposed to design a neural network that estimates index benefits based on the plans before/after creating indexes, based on which they recommend indexes with large benefits. However, it is still challenging to conduct workload-level tuning, which involves complex factors like the correlations between indexes and queries (*e.g.,* the maintenance cost may be higher than reduced lookup cost).

Second, for *index selection*, it is an NP-hard problem, and there are traditional methods (*i.e.,* exact algorithms like [18], [78], [13] and heuristic algorithms like [14], [67]), learning-based methods [65], [64], [96], [83]. Traditional methods [14], [78], [2] rely on exact (*e.g.,* dynamic programming) or greedy algorithms (*e.g.,* top-k, hill-climbing) to build indexes on columns that are frequently accessed by historical queries. There are two problems. (1) Within limited tuning budget, they may find suboptimal solutions when there are numerous candidate indexes; (2) They ignore the negative effects caused by building indexes (*e.g.,* indexing overhead, update costs). For the first problem, there are some learning-based methods that utilize reinforcement learning to learn the combined benefits of indexes and judiciously select the most promising indexes to build [65], [64]. For example, Sadri et al. [65] propose a DRL-based index selection method. First, without expert rules, they denote workload features as the arrival rate of queries and denote the column features as the access frequency and selectivity of each column. Second, they use deep reinforcement learning to learn from the workload and column features and output a set of actions, representing the recommended indexes. However, DRL mainly supports index addition (hard to rollback to historical state). And it is still hard to consider the removal of redundant/negative indexes, which can significantly affect the performance.

View Configuration. It is important in databases that utilize views to improve the query performance based on the space-for-time trade-off principle. Judiciously selecting materialized views can significantly improve the query performance within an acceptable overhead. However, traditional methods rely on DBAs to generate and maintain materialized views. Unfortunately, DBAs cannot handle a large number of database instances, especially for cloud databases that have millions of database instances and support millions of users. Thus, it calls for the view advisor, which automatically identifies the appropriate views for a given query workload [41], [53], [87], [28]. There are two sub-problems

in view configuration. First, view benefit evaluation aims to evaluate the benefit/cost of building a view. Second, give a set of candidate views (*e.g.,* frequent sub-queries), and view selection selects some candidates to build views within a given view size budget. The former is a regression problem, and the latter is an NP-hard optimization problem.

For view benefit estimation, traditional methods mainly rely on the cost model of the database optimizers and cannot effectively estimate the benefits of using MVs to answer a query [41], [53]. Thus, Han et al. [28] propose an encoder-reducer based model to estimate the benefit of different combinations of MV candidates and queries. This model can support variable-length input for dynamic workloads with different candidate views. For view selection, traditional methods [3], [6], [27], [8] rely on iteration-based heuristic methods. They cannot utilize the historical query data to optimize the MV configurations. Thus, Yuan et al. [87] model *view selection* as an integer linear programming (ILP) problem. They utilize a deep RL method (Deep Q-Network Learning) to judiciously select promising MV combinations based on the input workload features and the feedback of workload performance.

Partition Configuration. Data partitioning is vital in distributed databases which aims to allocate the data into different nodes by applying some partition functions (*e.g.,* hash, range, list) on some columns. Partition configuration aims to find appropriate columns (partition keys) to partition the data. As the partition configuration problem is NP-hard, some heuristic search methods [72], [16], [57], [23], [72], [31] and RL methods [35] are proposed. Traditional methods model the data and query patterns in vector [72], graph [16] or other hierarchy structures [57], [23], [72], [31], and heuristically select columns as partition keys (single column mostly). However, they cannot effectively trade-off between data load balance and access efficiency. To address the problem, some work [35] proposes to utilize a reinforcement learning model to explore different partition keys and implement a fully-connected neural network to estimate partition benefits. However, there are still some challenges. (1) Existing methods mainly consider foreign key constraints. Other important factors like data features (e.g., distinct values within columns) or query features (e.g., range queries, multiple joins) are hard to handle (*e.g.,* in graph model), but are vital to the partition performance. For example, a frequently-joined column should be considered even if it is not a foreign key, because it is promising to reduce remote joins. (2) In order to train a deep learning model, it is costly to really partition the databases and then evaluate the performance. Existing cost model is unaware of the data and query distribution under the selected partition keys and leads to unreliable estimation. Thus it calls for new partition methods to address these challenges.

7 Challenges and Open Problems

7.1 Knob Selection: Capturing Knob Correlations

Knob selection can be extremely important for tuning database systems with hundreds of knobs. However, many learning-based methods propose to tune most knobs without pre-selecting some knobs before tuning, which is because there are still several challenges. First, some existing knob selection methods simplify the knob-performance relation to a linear model, which can cause errors, especially in some cases (*e.g.,* when the knob value is larger than a number, the effect will not change). Thus, we need to evaluate single knobs with more advanced (hybrid) probabilistic methods. Second, ranking-based methods do not consider the relations between knobs, while the knob correlations can significantly affect the tuning performance (*e.g.,* one knob may be the threshold of another knob). Thus, we need to select knobs by considering both the single knob characteristics and their co-effects.

7.2 Feature Selection: Justify the Adopted Features

In existing tuning methods, most tuning features are selected based on human experience (*e.g.,* utilize query costs to reflect the computation requirements). However, these features are not always useful in many tuning cases (*e.g.,* read/write ratio could be more adaptive than query costs), and adopting all the features can cause great tuning overhead (*e.g.,* increase the training overhead of tuning models). Thus, it is vital to design an interpretable feature selection methodology that can automatically select tuning features with reasonable justifications. Besides, apart from historical workloads, there are some techniques that help to predict future workload trends, which can be more useful in tuning the configuration knobs [59] (*e.g.,* tune based on the memory consumption of incoming queries).

7.3 Knob Tuning: From High-Performance to Practical

Existing methods have achieved great progress in improving the tuning performance and migration capability. But there are still some open challenges.

Data Distribution Aware Configuration Tuning. Existing learning-based methods require query workloads to enrich the tuning features and utilize these features to facilitate configuration tuning. They assume that the workloads and run-time metrics are available. However, in real-world scenarios, databases may not keep query workloads, because logging them may sacrifice the performance. In those cases, we need to conduct configuration tuning based on limited database state metrics and data distributions. Thus, it is challenging to model data distributions and use distributions to tune configurations.

Incremental Configuration Tuning. In real scenarios, the database schema, data, and queries may continuously update. And it is very important to fine-tune the model with these updates. Existing configuration tuning cannot efficiently update knob values as the database updates. It is vital to ensure optimal overall performance and improve system robustness. There are two challenges in supporting incremental tuning. First, if the performance is not good, it is important to determine whether the data is out of distribution. Second, incremental tuning should be lightweight and take limited resources, and it cannot significantly sacrifice the performance during tuning.

Tuning Performance Prediction. Existing learning-based methods require repeatedly executing the workloads multiple times so as to obtain the execution results, which is vital to evaluate the tuning performance. However, rerunning the workloads can be time-consuming, and it may be unrealistic

in real-workload tuning scenarios (where users require to run the workload directly). Hence, it is vital to predict the tuning performance rather than actually executing the workloads. There are some similar tuning evaluation models like the critic network in the reinforcement learning algorithm *DDPG*, but it is hard to transfer to different workloads and give reliable evaluation. One of the key problems is to *evaluate whether a learned model is effective and outperforms non-learning methods*. For example, whether a knob tuning strategy really works for a workload? It requires designing a validation model to evaluate a learned model.

7.4 Transferring: One Model for A Batch of Scenarios

Although some tuning methods work well for specific scenarios (*e.g.,* query encoding for KV stores) and achieve generalization capability for similar scenarios, it is too heavy to train multiple models under different scenarios. Thus, it requires to design one tuning model that can handle typical scenarios. There are two challenges. First, there are different kinds of ML models (*e.g.,* forward-feeding, sequential, graph embedding), and it is inefficient to manually select tuning models and adjust the parameters. Second, it is hard to evaluate whether a learned model is effective in most scenarios, for which a validation model is required. One possible approach is to learn a general cost model and utilize the model to generalize to other database tasks like knob tuning and index selection [36], [84].

8 Conclusion

We have comprehensively reviewed existing knob tuning studies. We summarized the main components of the knob tuning pipeline, including knob selection, feature selection, tuning methods, and transfer techniques. For each component, we categorized the corresponding solutions and discussed their advantages and limitations. We also discussed how to deploy these techniques into real systems. We provided the research challenges and open problems in knob tuning. This survey provides researchers with a better understanding of how existing solutions were designed and will also provide practitioners with a means to decide how to deploy existing tuning solutions.

Acknowledgements

This paper was supported by NSF of China (61925205, 62232009, 62102215), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] https://pgtune.leopard.in.ua.
[2] S. Agrawal, S. Chaudhuri, L. Kollár, and et al. Database tuning advisor for microsoft SQL server 2005. In *VLDB*, 2004.
[3] R. Ahmed, R. G. Bello, and et al. Automated generation of materialized views in oracle. *Proc. VLDB Endow.*, 2020.
[4] A. C. Almeida, F. A. Baião, and et al. Tun-o_{cm}: A model-driven approach to support database tuning decision making. *Decis. Support Syst.*, 2021.
[5] J. Ansel, S. Kamil, K. Veeramachaneni, and et al. Opentuner: An extensible framework for program autotuning. In *Parallel architectures and compilation*, 2014.
[6] H. Azgomi and M. K. Sohrabi. A novel coral reefs optimization algorithm for materialized view selection in data warehouse environments. *Appl. Intell.*, 49(11):3965–3989, 2019.
[7] L. Bao, X. Liu, and W. Chen. Learning-based automatic parameter tuning for big data analytics frameworks. In *Big Data*, 2018.
[8] A. Boukra and S. Bouroubi. Selection of views to materialize in data warehouse: A cooperative approach. *Stud. Inform. Univ.*, 2011.
[9] B. Cai, Y. Liu, and et al. HUNTER: an online cloud database hybrid tuning system for personalized requirements. In *SIGMOD*, 2022.
[10] S. Cereda and et al. Cgptuner: a contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *Proc. VLDB Endow.*, 2021.
[11] Y. Chai, J. Ge, and et al. Xtuning: Expert database tuning system based on reinforcement learning. In *WISE*, volume 13080, 2021.
[12] S. Chaudhuri, E. Christensen, and et al. Self-tuning technology in microsoft SQL server. *IEEE Data Eng. Bull.*, 22(2), 1999.
[13] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *SIGMOD*, 2003.
[14] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB*, 1997.
[15] H. Christopher Frey and S. R. Patil. Identification and review of sensitivity analysis methods. *Risk analysis*, 22(3):553–578, 2002.
[16] C. Curino, Y. Zhang, and et al. Schism: a workload-driven approach to database replication and partitioning. *VLDB*, 2010.
[17] B. Dageville and M. Zaït. SQL memory management in oracle9i. In *VLDB*, pages 962–973. Morgan Kaufmann, 2002.
[18] D. Dash, N. Polyzotis, and et al. Cophy: A scalable, portable, and interactive index advisor for large workloads. *VLDB*, 2011.
[19] B. K. Debnath, D. J. Lilja, and et al. Sard: A statistical approach for ranking database tuning parameters. In *ICDE Workshop*, 2008.
[20] K. Dias, M. Ramacher, U. Shaft, and et al. Automatic performance diagnosis and tuning in oracle. In *CIDR*, 2005.
[21] B. Ding, S. Das, R. Marcus, and et al. AI meets AI: leveraging query executions to improve index recommendations. In *SIGMOD*, 2019.
[22] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *VLDB*, 2(1):1246–1257, 2009.
[23] G. Eadon, E. I. Chong, S. Shankar, and et al. Supporting table partitioning by reference in oracle. In *SIGMOD*, 2008.
[24] P. I. Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.
[25] B. Fruchter. Introduction to factor analysis. 1954.
[26] J.-K. Ge and et al. Watuning: A workload-aware tuning system with attention-based deep reinforcement learning. *JCST*, 2021.
[27] A. Gosain and et al. Handling constraints using penalty functions in materialized view selection. *Int. J. Nat. Comput. Res.*, 2019.
[28] Y. Han, G. Li, and et al. An autonomous materialized view management system with deep reinforcement learning. In *ICDE*, 2021.
[29] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 1979.
[30] J. C. Helton and F. J. Davis. Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems. *Reliability Engineering & System Safety*, 2003.
[31] H. Herodotou, N. Borisov, and S. Babu. Query optimization techniques for partitioned tables. In *SIGMOD*, 2011.
[32] H. Herodotou, Y. Chen, and J. Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Computing Surveys (CSUR)*, 53(2):1–37, 2020.
[33] H. Herodotou, H. Lim, and et al. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, 2011.
[34] D. Hiemstra. A probabilistic justification for using tf× idf term weighting in information retrieval. *International Journal on Digital Libraries*, 3(2):131–139, 2000.
[35] B. Hilprecht, C. Binnig, and U. Röhm. Learning a partitioning advisor for cloud databases. In *SIGMOD*, 2020.
[36] B. Hilprecht and et al. One model to rule them all: towards zero-shot learning for databases. *arXiv*, 2021.
[37] M. Hoffman, E. Brochu, N. De Freitas, et al. Portfolio allocation for bayesian optimization. In *UAI*, pages 327–336, 2011.
[38] C. Huang and et al. A survey of automatic parameter tuning methods for metaheuristics. *IEEE Trans. Evol. Comput.*, 2020.
[39] K. Kanellis, R. Alagappan, and S. Venkataraman. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs. In *HotStorage*, 2020.
[40] K. Kanellis, C. Ding, and et al. Llamatune: Sample-efficient dbms configuration tuning. *arXiv*, 2022.
[41] J. Kossmann, S. Halfpap, and et al. Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proc. VLDB Endow.*, 2020.

[42] T. Kraska, M. Alizadeh, and et al. Sagedb: A learned database system. In *CIDR*, 2019.
[43] A. Krause and C. S. Ong. Contextual gaussian process bandit optimization. In *Nips*, pages 2447–2455, 2011.
[44] S. Kumar. Oracle database 10g: The self-managing database. In *White Paper.*, 2003.
[45] M. Kunjir and S. Babu. Black or white? how to develop an autotuner for memory-based analytics. In *SIGMOD*, 2020.
[46] V. Leis, A. Gubichev, and et al. How good are query optimizers, really? *Proc. VLDB Endow.*, 2015.
[47] V. Leis, B. Radke, A. Gubichev, and et al. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.
[48] G. Li, X. Zhou, and L. Cao. Ai meets database: Ai4db and db4ai. In *SIGMOD*, pages 2859–2866, 2021.
[49] G. Li, X. Zhou, and L. Cao. Machine learning for databases. *Proc. VLDB Endow.*, 14(12):3190–3193, 2021.
[50] G. Li, X. Zhou, and et al. opengauss: An autonomous database system. *Proc. VLDB Endow.*, 2021.
[51] G. Li, X. Zhou, and S. Li. Xuanyuan: An ai-native database. *IEEE Data Eng. Bull.*, 42(2):70–81, 2019.
[52] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 2019.
[53] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic view materialization with deep reinforcement learning. *CoRR*, 2019.
[54] C. Lin, J. Zhuang, J. Feng, H. Li, X. Zhou, and G. Li. Adaptive code learning for spark configuration tuning. ICDE, 2022.
[55] G. Lohman. Is query optimization a "solved" problem. In *Workshop on Database Query Optimization*, 2014.
[56] J. Lu, Y. Chen, H. Herodotou, S. Babu, et al. Speedup your analytics: Automatic parameter tuning for databases and big data systems. *Proc. VLDB Endow.*, 2019.
[57] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. Adaptdb: Adaptive partitioning for distributed joins. *VLDB*, 2017.
[58] L. Ma, B. Ding, S. Das, and et al. Active learning for ML enhanced database systems. In *SIGMOD*, pages 175–191. ACM, 2020.
[59] L. Ma, D. Van Aken, and et al. Query-based workload forecasting for self-driving database management systems. In *SIGMOD*, 2018.
[60] L. Ma, W. Zhang, and et al. MB2: decomposed behavior modeling for self-driving database management systems. In *SIGMOD*, 2021.
[61] R. Marcus, P. Negi, H. Mao, and et al. Bao: Making learned query optimization practical. In *SIGMOD*, 2021.
[62] R. Marins, R. P. de Oliveira, and et al. Outer-tuning: an ontology-based extensible framework for supporting database automatic tuning. *J. Inf. Data Manag.*, 2021.
[63] A. Pavlo, G. Angulo, and et al. Self-driving database management systems. In *CIDR*, volume 4, page 1, 2017.
[64] R. M. Perera, B. Oetomo, and et al. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *ICDE*, 2021.
[65] Z. Sadri and et al. Online index selection using deep reinforcement learning for a cluster database. In *ICDEW*, 2020.
[66] A. Saltelli. Sensitivity analysis for importance assessment. *Risk analysis*, 22(3):579–590, 2002.
[67] K. Schnaitter, S. Abiteboul, and et al. On-line index selection for shifting workloads. In *ICDE*, 2007.
[68] A. J. Storm, C. Garcia-Arellano, S. Lightstone, and et al. Adaptive self-tuning memory in DB2. In *VLDB*. ACM, 2006.
[69] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. *SIGMETRICS*, 2004.
[70] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*, 13(3):307–319, 2019.
[71] J. Sun, J. Zhang, and et al. Learned cardinality estimation: A design space exploration and A comparative evaluation. *VLDB*, 2021.
[72] L. Sun, M. J. Franklin, S. Krishnan, and et al. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, 2014.
[73] J. Tan, T. Zhang, F. Li, and et al. ibtune: Individualized buffer tuning for large-scale cloud databases. *VLDB*, 2019.
[74] V. Thummala and S. Babu. ituned: a tool for configuring and visualizing database parameters. In *SIGMOD*, 2010.
[75] W. Tian, P. Martin, and et al. Techniques for automatically sizing multiple buffer pools in DB2. In *Collaborative Research*, 2003.
[76] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 1996.
[77] I. Trummer. DB-BERT: A database tuning tool that "reads the manual". In *SIGMOD*, pages 190–203. ACM, 2022.
[78] G. Valentin, M. Zuliani, and et al. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.
[79] D. Van Aken and et al. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, 2017.
[80] D. Van Aken, D. Yang, and et al. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proc. VLDB Endow.*, 2021.
[81] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A normalizing flow based cardinality estimator. *VLDB*, 15(1):72–84, 2021.
[82] J. Wang, I. Trummer, and et al. UDO: universal database optimization using reinforcement learning. *Proc. VLDB Endow.*, 2021.
[83] W. Wu, C. Wang, T. Siddiqui, J. Wang, V. R. Narasayya, S. Chaudhuri, and P. A. Bernstein. Budget-aware index tuning with reinforcement learning. In *SIGMOD*, pages 1528–1541. ACM, 2022.
[84] Z. Wu, P. Yang, P. Yu, R. Zhu, Y. Han, Y. Li, D. Lian, K. Zeng, and J. Zhou. A unified transferable model for ml-enhanced dbms. *arXiv preprint arXiv:2105.02418*, 2021.
[85] L. Yang and et al. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 2020.
[86] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *ICDE*, pages 1297–1308, 2020.
[87] H. Yuan, G. Li, L. Feng, and et al. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, 2020.
[88] B. Zhang and et al. A demonstration of the ottertune automatic database management system tuning service. *VLDB*, 2018.
[89] J. Zhang and et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, 2019.
[90] J. Zhang and et al. Cdbtune+: An efficient deep reinforcement learning-based automatic cloud database tuning system. *VLDB J.*, 2021.
[91] X. Zhang, Z. Chang, and et al. Facilitating database tuning with hyper-parameter optimization: A comprehensive experimental evaluation. *Proc. VLDB Endow.*, 2022.
[92] X. Zhang, H. Wu, and et al. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *SIGMOD*, 2021.
[93] X. Zhang, H. Wu, Y. Li, and et al. Towards dynamic and safe configuration tuning for cloud databases. In *SIGMOD*, 2022.
[94] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *TKDE*, 2020.
[95] X. Zhou, L. Jin, J. Sun, and et al. Dbmind: A self-driving platform in opengauss. *Proc. VLDB Endow.*, 2021.
[96] X. Zhou, L. Liu, and et al. Autoindex: An incremental index management system for dynamic workloads. In *ICDE*, 2022.
[97] X. Zhou, J. Sun, G. Li, and J. Feng. Query performance prediction for concurrent queries using graph embedding. *VLDB*, 2020.
[98] Y. Zhu and et al. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCCer*, 2017.

Xinyang Zhao received the bachelor's degree in Electronics and Engineering from the Beijing Institute of Technology, China, and the MSc degree in Data Science from University of Sheffield, UK, in 2017 and 2020, respectively. She is a PhD student in the Department of Computer Science, Tsinghua University. Her research interest is database configuration tuning.

Xuanhe Zhou received his bachelor's degree in Computer Science from the Beijing University of Posts and Telecommunications in 2019. He is currently working toward the PhD degree in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests lie in autonomous database systems.

Guoliang Li is currently working as a professor in the Department of Computer Science, Tsinghua University, Beijing, China. He received his PhD degree in Computer Science from Tsinghua University, Beijing, China in 2009. His research interests mainly include data cleaning and integration, spatial databases, crowdsourcing, and AI &DB co-optimization.