SCompression: Enhancing Database Knob Tuning Efficiency Through Slice-Based OLTP Workload Compression

Baoqing Cai
Yu Liu
Huazhong University of Science and Technology
{bqcai,liu_yu}@hust.edu.cn

Lin Ma
University of Michigan
linmacse@umich.edu

Pingqi Huang
Huazhong University of Science and Technology
m202273593@hust.edu.cn

Bingcheng Lian
Huazhong University of Science and Technology
bingchenglian@hust.edu.cn

Ke Zhou
Huazhong University of Science and Technology
zhke@hust.edu.cn

Jia Yuan
University of Arizona
jiayuan@arizona.edu

Jie Yang
Tencent
edgeyang@tencent.com

Xiaofan Cai
Peijun Wu
Tencent
{anthonycai,kals}@tencent.com

ABSTRACT

Workload execution can account for 90% of the total database knob tuning time, which is often the bottleneck for efficient knob tuning in practice. Reducing the tuning time by using a compressed workload is a natural solution. However, many existing workload compression methods are designed for OLAP workloads, which reduce the number of queries needed for analysis tasks by sampling a small subset of queries. These methods are less effective for OLTP workloads in knob-tuning tasks, as they often disregard essential contextual details, including query sequence and concurrency. As a result, configurations that perform well on the compressed OLTP workload may not deliver similar competitive performance on the original workload. To address these challenges, we first define the objective of OLTP workload compression for knob tuning. We then propose a slice-based compression method, *SCompression*, which compresses workloads by slicing based on time intervals while preserving concurrency. *SCompression* achieves the objective by focusing on generating a compressed workload that (1) executes faster than the original workload and (2) produces performance variations similar to the source workload under different configurations. *SCompression* works in three steps: (1) dividing the workload into segments to capture regular performance fluctuations, (2) slicing each segment to preserve concurrency and transaction context, and (3) sampling slices under execution time constraints using a cluster-based approach to ensure representativeness. Finally, *SCompression* replays the compressed workload to produce the performance that mirrors the source workload. Extensive experiments on real-world and benchmark OLTP workloads show that *SCompression* is a cost-effective solution for knob tuning, accelerating tuning by up to 40× with only a 5% performance reduction.

PVLDB Reference Format:
Baoqing Cai, Yu Liu, Lin Ma, Pingqi Huang, Bingcheng Lian, Ke Zhou, Jia Yuan, Jie Yang, Xiaofan Cai, and Peijun Wu. SCompression: Enhancing Database Knob Tuning Efficiency Through Slice-Based OLTP Workload Compression. PVLDB, 18(6): 1865 - 1878, 2025.
doi:10.14778/3725688.3725712

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725712

Baoqing Cai and Yu Liu contribute equally to this paper.
Ke Zhou is the corresponding author.

1 INTRODUCTION

Database systems can have up to hundreds of configuration knobs that can significantly impact their performance. Recent works have proposed using machine-learning (ML) models to tune these knobs by intelligently exploring their configuration space [4, 6, 25, 27, 28, 42, 43, 46, 48, 49]. These approaches typically employ an iterative process that repeatedly (1) recommends configurations, (2) evaluates the performance, and (3) fits the model. They aim to find configurations that enhance the system's performance automatically.

However, these approaches often face a critical challenge in real-world deployments: the workload execution cost. The tuning process typically requires many iterations for the model training and convergence [6, 49]. Each iteration requires executing the representative workload, which can be expensive in production environments. While some studies use advanced ML techniques to reduce the number of iterations required for convergence [49], each iteration still inevitably involves a time-consuming workload execution. Others employ parallel learning and exploration strategies, which can be time-efficient but resource-intensive [6]. Thus, workload execution cost often remains a bottleneck in the tuning process.

We tackle this problem by investigating workload compression techniques to reduce the workload execution time for knob-tuning. We focus on Online Transaction Processing (OLTP) workloads, given their wide applications [16, 29, 30, 44]. Workload compression finds a substitute workload of a smaller size that preserves the information from the source workload to speed up the target task [7]. Previous studies mostly focus on compressing Online Analytical Processing (OLAP) workloads to reduce the number of queries for workload analysis and index recommendation [7, 11, 34, 42]. They effectively reduce the analysis time for OLAP workloads since they are suitable for query-by-query execution or analysis. Nevertheless, these approaches demonstrate limitations when applied to OLTP workload compression for knob-tuning purposes, as they potentially compromise essential query context information (*e.g.*, the sequence and conflicts between queries) inherent in OLTP workloads. Specifically, the loss of sequence and concurrency data can impair the accurate reproduction of OLTP workload performance during query replay operations. According to the characteristics of OLTP workloads and tuning requirements, an effective solution requires navigating four principal challenges.

(C1) Contextual Information Integrity. In OLTP workloads, performance features relate to a series of queries rather than a particular one. Since queries are executed concurrently and influence each other when running simultaneously, the compression method should maintain the context of the queries (*e.g.*, the sequence and conflicts between queries) and compress accordingly. However, existing query-based compression methods select queries independently, treating them as a set and ignoring the interactions between queries during workload execution. Therefore, they fail to maintain the sequence and concurrency of queries, making it difficult to preserve contextual information in compressed workloads.

(C2) Time-oriented Compression. For analysis tasks, query-based compression methods aim to reduce the number of queries. However, fewer queries do not necessarily result in shorter execution times for tuning OLTP workloads. The tuning system needs to replay the workload based on query timestamps to ensure performance fidelity, which makes traditional compression methods less effective at reducing replay time.

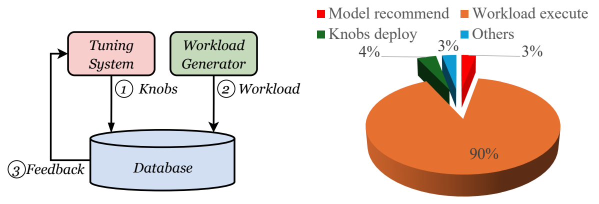
(C3) Large Query Volume. OLTP workloads generally contain a larger number of queries compared to OLAP workloads. For example, the highest number of queries in OLAP workload compression studies [7, 11, 34, 42] is around 2,200, whereas OLTP workloads may exceed 8 million queries. Compressing such a large volume of queries with query-based methods is both time-consuming and resource-intensive. For example, QCSA [42], a recent workload compression method for knob tuning, requires about 150 minutes to compress a 5-minute workload, as detailed in Section 9.2.

(C4) Replay Strategy. Even if the system can obtain a representative compressed workload, achieving similar performance when replaying the workload with different configurations remains challenging. Since a compressed workload inevitably loses some information from the source workload, minimizing the impact of queries that exhibit abnormal performance deviations is crucial.

Our Approach. To address these challenges, we propose a novel OLTP workload compression method for knob tuning, called *SCompression*. The core idea of *SCompression* is to reduce the number of *slices* rather than individual queries in the workload, where a slice is defined as a small time interval within the workload that contains all transactions starting during that interval. *SCompression* preserves the query context by retaining the information within sampled slices.

SCompression first slices the source workload into *slices*. Since each slice contains all queries that start within its designated time interval, it preserves the query sequence and concurrency, addressing **C1**. To improve the slice quality, *SCompression* employs a dynamic slicing strategy supported by workload segmentation. This segmentation divides the source workload into multiple regular patterns using Greedy Gaussian Segmentation (GGS) [15].

Next, *SCompression* implements a cluster-then-sample approach that combines clustering and sampling techniques to identify representative slices, which are then integrated into a compressed workload. Given that slices are delineated by temporal intervals, this methodology naturally incorporates the execution duration of the compressed workload, thereby satisfying requirement **C2**. To reduce computation costs during clustering, *SCompression* aggregates the query features within each slice into a single slice feature, ensuring high processing speed and addressing **C3**.



(a) Workflow of knob tuning. (b) Time Distribution (TPC-C).

Figure 1: Example of knob tuning.

Finally, *SCompression* employs a slice-based workload replay strategy to reproduce performance characteristics that closely mirror those of the source workload, addressing **C4**.

To evaluate *SCompression*, we perform extensive experiments on real-world and benchmark OLTP workloads with state-of-the-art tuning algorithms. Our empirical analysis demonstrates that *SCompression* exhibits exceptional efficiency, delivering a 40× acceleration in tuning while maintaining peak performance within a 5% margin of degradation. Our main contributions are as follows:

- We propose a novel context-aware OLTP workload compression method. It partitions a workload based on slices and splices sampled slices to build a compressed version, resulting in similar performance between the compressed and source workloads under the same configurations.
- We propose a dynamic slicing approach that preserves the contextual integrity of queries.
- We design a slice sampling strategy based on clustering, preserving workload diversity.
- We implement a replay strategy based on slices for compressed OLTP workloads to achieve fast evaluation with minimal performance distortion.
- Our experimental results demonstrate that *SCompression* provides a cost-effective approach to optimize knob-tuning performance.

2 PROBLEM SETTING

2.1 Preliminaries

Revisiting the problem of knob tuning, let W and C denote a workload (formed by query logs, as detailed in Section 2.3) and a database configuration (a set of knobs), respectively. $P(W, C)$ represents the performance of the database when executing W under C, where QPS (Queries Per Second) is a commonly used performance metrics [6, 18, 40, 42] and is adopted as the metric for P in this paper. The objective of knob tuning is to identify a C within the search space $\mathbb{C}$ that maximizes $P(W, C)$:

$$\arg\max_{C \in \mathbb{C}} P(W, C).$$

Table 1: Details of Query Logs and Monitor Metrics.

Name		Details
Query Logs	Basic Information	Timestamp, Sql, ThreadId, TrxLivingTime
	Additional Features	AffectRows, ExecTime, SentRows, CheckRows, CpuTime, IOWaitTime, LockWaitTime
Monitor Metrics	Features	CPU Utilization, IO Operations, Network Traffic, Buffer Hit Rate, QPS.

Example. As shown in Figure 1a, a knob-tuning application typically follows an iterative process: (1) the tuning system suggests C to deploy on the database, (2) the workload generator creates W and replays it with C to generate $P(W, C)$, and (3) the tuning system fine-tunes the model with the performance results, enabling accurate configuration recommendations in subsequent iterations.

In this process, the system relies on workload execution to estimate $P(W, C)$. The workload execution time can account for over 90% of the tuning time in previous studies [28, 47], as shown in Figure 1b. As a result, reducing workload execution time is essential for improving knob-tuning efficiency.

2.2 Problem Statement

Let W and $\overline{W}$ denote the source workload and its compressed workload, respectively. Given a user-specified target compression rate cr, we define our compression rate $CR(\overline{W})$ of $\overline{W}$ as the ratio of the execution time of W to that of $\overline{W}$, denoted as $E(W)$ and $E(\overline{W})$, respectively. Formally:

$$\frac{E(W)}{E(\overline{W})} = CR(\overline{W}) \geq cr.$$

For W, $E(W)$ corresponds to the actual execution time. However, we need to calculate $E(\overline{W})$ based on the query logs for $\overline{W}$, excluding idle periods (*i.e.*, intervals where no queries are executed), since $\overline{W}$'s actual execution time cannot be determined until the workload is executed. This calculation utilizes $q_{\text{Timestamp}}$ and $q_{\text{ExecuteTime}}$ features to accurately identify and eliminate periods of inactivity, as detailed in Section 2.3.

Meanwhile, the performance (*e.g.*, QPS) of executing on $\overline{W}$ should be close to that of W under the same configuration. As a result, the objective is to find a $\overline{W} \subseteq W$, satisfying: $\forall C \in \mathbb{C}$ such that $\min |P(W, C) - P(\overline{W}, C)|$, subject to: $\frac{E(W)}{E(\overline{W})} \geq cr$. To solve this problem, we formulate it as a constraint optimization problem and apply appropriate approximation techniques. Formally,

$$\underset{\overline{W} \subseteq W}{\arg\min} \sum_{C_i \in \mathbb{C}} \left| P(W, C_i) - P(\overline{W}, C_i) \right|, \text{subject to: } \frac{E(W)}{E(\overline{W})} \geq cr.$$

2.3 Notation

In this section, we first introduce the data obtained from the cloud for analysis, along with the relevant concepts and entities used to construct the workload. Based on these concepts, we then provide detailed definitions for **Slice**, **Segment**, and **Workload**.

In cloud databases, the data related to workloads primarily consists of Query Logs and Monitoring Metrics.

Query Logs. The cloud database maintains comprehensive query logs, also called audit logs, to track and document each query execution. As shown in Table 1, these logs capture both basic information and additional features. The *basic information*, essential for reconstructing the workload, includes the start time in nanoseconds (*Timestamp*), details of the SQL statement (*Sql*), a unique identifier for each connection (*ThreadId*), and transaction duration in milliseconds (*TrxLivingTime*). The *additional features* that assist in workload compression include the number of affected rows (*AffectRows*), execution time in milliseconds (*ExecTime*), number of rows returned to the client (*SentRows*), number of rows scanned (*CheckRows*), CPU time in milliseconds (*CpuTime*), IO wait time in milliseconds (*IOWaitTime*), and lock wait time in milliseconds (*LockWaitTime*).

Formally, let q denote a query and q_{feature} represent the value of a specific feature for query q. For instance, $q_{\text{AffectRows}}$ indicates the number of affected rows of q. In addition, a transaction is denoted as T, which is obtained from the query log. Each T consists of a sequence of qs that are grouped and executed as a single unit.

Monitor Metrics. A performance monitor gathers database performance metrics every 5 seconds. These metrics include CPU utilization (*CPU Utilization*), the number of IO operations (*IO Operations*), network traffic (*Network Traffic*), the hit rate of the InnoDB buffer pool (*Buffer Hit Rate*), and the number of queries per second (*QPS*), as shown in Table 1. Formally, m denotes a monitoring metric. As a result, for example, $m_{\text{CPU\_Utilization}}$ refers to the CPU utilization recorded by the performance monitor.

Connection. A database connection is a communication link established between a database management system (DBMS) and a client, enabling the client to interact with the database. Each connection sends queries sequentially to the DBMS, while different connections can execute concurrently. Formally, we denote a connection as N. The set $N(t_i, t_j)$ contains all queries in N, where the start time of q denotes as $\text{StartTime}(q)$ that falls within the interval $[t_i, t_j)$. Similarly, its end time is $\text{EndTime}(q)$.

Slice. A slice corresponds to a time interval of a workload, which contains all queries that start within its interval. Formally, when a slice corresponds to the interval $[t_i, t_j)$, it is defined as $S(t_i, t_j) = \{N_1(t_i, t_j), \ldots, N_m(t_i, t_j)\}$, where $N_1, \ldots, N_m$ denotes all the connections in the workload. Note that S can be viewed as a minimal subset of a workload in this paper.

Segment. A segment consists of a sequence of slices, which usually denotes an interval with regular performance. Formally, a segment is defined as $SG(t_i, t_j) = [S(t_i, t_b), S(t_b, t_c), \ldots, S(t_y, t_j)]$, where $t_i < t_b \cdots < t_y < t_j$.

Workload. A workload is spliced by all segments. Formally, a workload can be represented as $W = [SG_1, SG_2, \ldots, SG_m]$, where overlap is not allowed between SG_i and SG_j. Similarly, since S_i and S_j cannot overlap, the workload can also be expressed as $W = [S_1, S_2, \ldots, S_n]$.

Example. As shown in Figure 2, $S_i = S(t_i, t_{i+1})$ contains all concurrent connections $\{N_1, N_2, N_3\}$ within the interval $[t_i, t_{i+1})$, each of which consists of a sequence of qs (or transactions Ts). A sequence of Ss forms a SG, and all SGs splice a W.

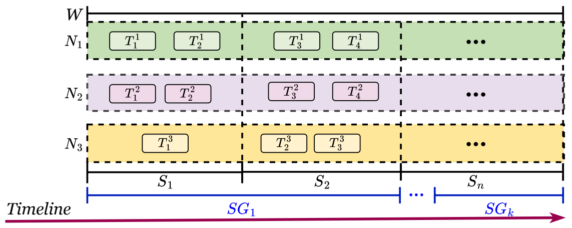


Figure 2: Workload constructed by slices.

3 SOLUTION OVERVIEW

3.1 Our Solution

Based on the above definition, the problem can be further formulated as follows: Given a $W = [S_1, \ldots, S_n]$, the objective is to find a $\overline{W} = [S_j, \ldots, S_k] \subseteq W$ such that $E(W)/E(\overline{W})$ is not less than cr. Formally,

$$\underset{[S_j,\ldots,S_k]}{\arg\min} \sum_{C_i \in \mathbb{C}} \left| P([S_1, \ldots, S_n], C_i) - P([S_j, \ldots, S_k], C_i) \right|,$$

$$\text{subject to: } \frac{E([S_1, \ldots, S_n])}{E([S_j, \ldots, S_k])} \geq cr.$$

A naïve approach involves randomly sampling $S_i \in W$ to form the compressed workload $\overline{W} \subseteq W$. However, random sampling may result in a lack of representativeness of the sampled slices, which affects the fidelity of the compressed workload.

To address this problem, *SCompression* adopts a cluster-then-sample strategy, as detailed in Section 6. *SCompression* first groups slices into clusters based on their features. Then, it samples representative slices from each cluster to construct $\overline{W}$. This strategy ensures that $\overline{W}$ reflects the distribution and characteristics of the W.

For knob tuning, executing the compressed workloads is essential to evaluate performance across various configurations, highlighting the importance of accurately mirroring the performance characteristics between the source and compressed workloads. To achieve this, it is crucial to retain performance-critical information from the source workload and carefully select slices using well-chosen split points. To maintain the integrity of concurrency conflicts, we avoid partitioning during high-concurrency periods while ensuring these conflicts are preserved in the compressed workload.

To this end, we design a dynamic slicing strategy for determining the appropriate slice length. While adding more split points can improve the compression rate, it also increases the risk of losing concurrent conflicts. Leveraging the Analyzer (Section 4) and Slicer (Section 5), *SCompression* can implement it accurately. This effort minimizes instances where query execution periods are across two slices, thereby reducing cases where concurrent queries are executed sequentially during replay, improving the accuracy of performance evaluation on compressed workloads.

3.2 Architecture

We now elaborate on the slice-based compression solution named *SCompression*. It is a compression framework for OLTP workloads used for knob tuning. It aims to reduce workload replay time while preserving the concurrency of queries in the workload, ensuring that the same configuration produces similar performance variations on the compressed and source workloads.

We show the overview architecture of *SCompression* in Figure 3. It consists of three components: *Analyzer*, *Slicer*, and *Compressor*. In addition, we develop a *Replayer* based on the proposed slicing strategy to execute the compressed workload.

Analyzer aims to segment a workload with varying stages into segments with regular patterns by adopting Greedy Gaussian Segmentation, a multidimensional time series segmentation algorithm. It initially records query log data along with corresponding monitoring metrics, subsequently analyzing these metrics to determine split points for workload segmentation.

Slicer divides each workload segment into numerous slices while preserving the concurrency in each slice to maintain the characteristics of the source workload. To this end, *SCompression* applies a dynamic slicing strategy to calculate the slice length for each segment, maintaining the integrity of context in a slice.

Compressor aims to sample representative slices under the constraint of the workload execution time to build the compressed workload. Instead of analyzing the complete queries, *SCompression* summarizes the feature of transactions inside a slice and then clusters slices. It samples slices from different clusters based on the cluster's scale to ensure representativeness.

Replayer executes the compressed OLTP workload. The above components aim to preserve the features of the source workload, while *Replayer* aims to reflect these features in workload performance by executing transactions with proper orders. To achieve this purpose, we design a slice-based workload replay strategy.

Workflow. The compression process starts with selecting an interval (*e.g.*, 10 minutes) from the query log for compression. We designed *SCompression* for a cloud database that collects query logs and performance monitoring metrics (details in Table 1). *Analyzer* first processes the query logs, organizing them into a source workload. It then segments the workload based on performance monitoring metrics, ensuring each segment reflects a regular performance pattern, which improves slicing quality (as discussed in Section 5). For each segment, *Slicer* divides it into numerous slices, which are then passed to the *Compressor*. *Compressor* analyzes these slices and selects representative ones based on a given workload execution time, forming a compressed workload. Finally, *Replayer* executes this compressed workload, delivering comparable performance to running on the source workload. This approach saves time on knob tuning by significantly reducing the workload execution time.

4 ANALYZER

To implement the dynamic slicing strategy, we integrate *Analyzer* and *Slicer*. *Analyzer*, a solution designed to generate representative segments with a regular pattern in each segment systematically for *Slicer*. It helps acquire a balanced number of slices across different patterns in the workload and maintains contextual information between transactions. We will further discuss the effectiveness of *Analyzer* for slicing in Section 5.

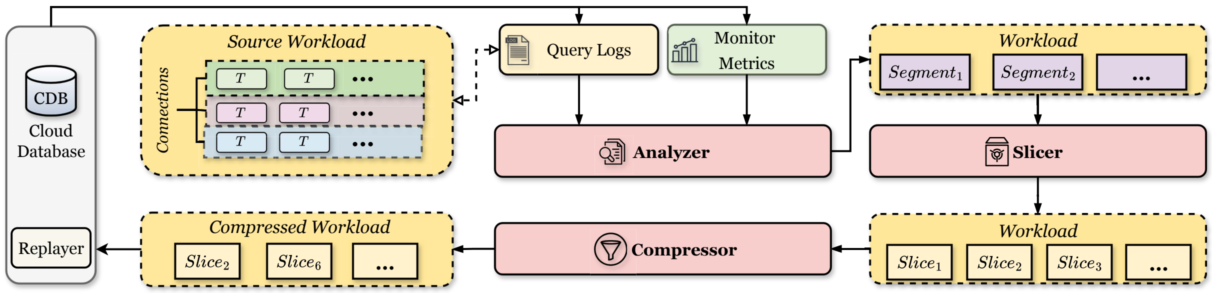


Figure 3: Overview Architecture and Workflow of *SCompression*.

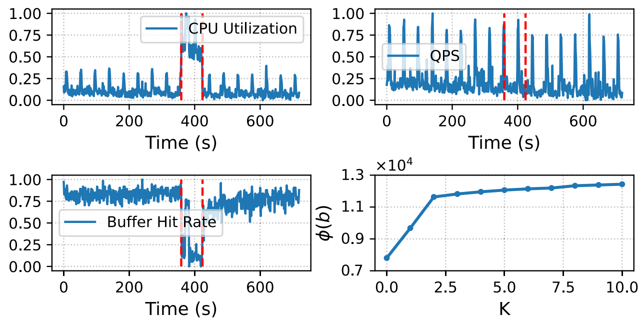


Figure 4: The segmentation results on metrics via GGS.

We select the monitor metrics shown in Table 1, based on recommendations from cloud database engineers, to determine segments. To analyze these multidimensional time series, we employ Greedy Gaussian Segmentation (GGS) [15], using the heuristic method to model segmentation as a maximum likelihood problem regularized by covariance. GGS segments a multivariate time series $\{m_t\}_{t=1}^{T}$, where m_t represents monitor metrics at time t, into $K+1$ contiguous segments. Each segment is modeled as independent samples from a multivariate Gaussian distribution with mean μ and covariance Σ, defined by K split points $\{b_k\}_{k=1}^{K}$. Despite the segmentation problem being NP-hard, GGS approximates the solution with linear time complexity. It maximizes a covariance-regularized log-likelihood:

$$\varphi(b,\mu,\Sigma)=\ell(b,\mu,\Sigma)-\lambda\sum_{i=1}^{K+1}\mathrm{Tr}(\Sigma_i^{-1}),$$

where $\ell(b,\mu,\Sigma)$ is the log-likelihood of the segmented time series, λ is a regularization parameter, and $\mathrm{Tr}(\Sigma_i^{-1})$ denotes the trace of the inverse covariance matrix for segment i. A higher $\varphi(b,\mu,\Sigma)$ value indicates better segmentation quality.

For example, Figure 4 shows Normalized CPU Utilization, Normalized Buffer Hit Rate, and Normalized QPS over approximately 750 seconds. During this period, the QPS plot displays periodic spikes throughout. The CPU Utilization plot exhibits periodic spikes before 350 seconds and after 430 seconds, with the Buffer Hit Rate following a similar pattern. The section between the red dashed lines clearly demonstrates a larger variance in metrics compared to the rest of the workload. Consequently, we divide the workload into three segments to maintain stable variance within each segment.

We use the elbow method to determine the appropriate number of split points [37]. Using the time series data used in Figure 4 as an example, we plot $\phi(b)$ against K and analyze its objective values in different split point counts, with $\lambda = 10^{-5}$. A significant change in $\phi(b)$ at $K = 2$, as shown in Figure 4 (the bottom right). Therefore, we should set $K = 2$ for the data in Figure 4. This conclusion is consistent with the reasonable result we observed in Figure 4, where two red dashed lines illustrate how GGS segments the time series data. It demonstrates that GGS can effectively recommend appropriate split points and counts.

Time complexity. According to GGS [15], the time complexity is $O(K^{max}n^3T^2)$, where n is the dimension of the metrics and T is the number of points of metrics. In this paper, $n = 5$ because there are five features in the monitor metrics, as shown in Table 1. Meanwhile, monitor metrics use 5-second intervals, giving T a small range of up to 130. For the longer time series, downsampling can be applied to mitigate computational overhead.

5 SLICER

When we define a workload W as being composed of $[S_1, ..., S_n]$, identifying the split points for slices to divide the workload remains a significant challenge. As discussed in Section 3.1, splitting the workload at a moment of high contention can compromise the concurrency integrity. To address this problem, the key idea is to **set split points at moments when there are few concurrences**. Following this principle, *Analyzer* implements a coarse-grained partitioning strategy on the workload from a global perspective, getting segments. On each resultant segment, we maintain this principle to execute a **dynamic slicing strategy**.

We utilize naïve slicing approaches - minimized, maximized, and fixed-length slicing (with uniform slice durations of 30 ms) - to demonstrate the significance of the principle. To compare these strategies with ours, we first extract slices from the source workload and then replay them using the replay strategy outlined in Section 7, without sampling. During replay, QPS and CPU utilization are collected. The baseline metrics for the source workload are presented in Figure 5. A slicing strategy that produces metrics aligned to the baseline is considered more effective.

Using the monitor metrics shown in Table 1, we illustrate the distinct QPS and CPU utilization metrics produced by these strategies, as shown in Figure 5. For comparison, Figure 5 presents the baseline metrics for the source workload, which spans approximately 650 seconds. The baseline CPU utilization exhibits regular spikes

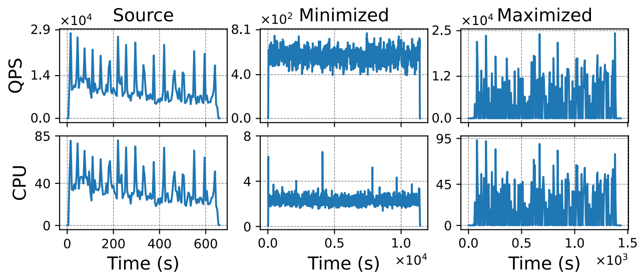


Figure 5: Different slice lengths.

between 30% and 85%, while QPS ranges from 5×10^3 to 3×10^4. Additionally, CPU utilization is positively correlated with QPS.

Using the minimized slice length (*i.e.*, each slice contains a single query, as shown in Figure 5), CPU utilization drops to around 3%, QPS to approximately 600, and the total runtime exceeds 3 hours. The positive correlation between CPU and QPS is no longer significant. We attribute this performance degradation to sequential query execution, which circumvents concurrency and conflict information, resulting in prolonged execution time and loss of baseline characteristics. A similar performance distortion occurs when using the maximized slice length, *i.e.*, all queries are in one slice, causing all connections to run concurrently. As shown in Figure 5, CPU utilization frequently spikes to nearly 100% and drops to about 0% in short intervals. QPS mirrors this trend with a lower average than the baseline, and the total runtime exceeds twice the baseline. Excessive concurrency overloads the CPU, causing execution anomalies and performance shocks. Recovery from these overloads further increases execution time.

The above results suggest that extreme slice lengths are impractical, prompting us to test predefined slice lengths. Figure 6a shows relatively desirable results with the fixed slicing approach, as it preserves trends in performance changes. However, compared to the baseline, both metrics decrease while execution time increases. We believe this issue stems from slicing without maintaining the context of queries, causing queries from the same or concurrent transactions to be split into multiple independent slices for sequential execution. This reduction in concurrency decreases CPU utilization and increases execution time.

To address this problem, we stipulate a dynamic slicing strategy by selecting the split points with few concurrent conflicts on each segment. To achieve it, we define a concept of instantaneous concurrency (IC), *i.e.*, the number of transactions being executed in an interval Δ. Formally, $IC(t) = Count(T_{active} \in [t, t+\Delta))$, where t is the start time, and we set $\Delta = 3$ms according to DBA's advice. A low $IC(t)$ indicates that connections are generally idle during the $[t, t+\Delta)$, suggesting that slicing within this interval could minimize the loss of concurrence. Specifically, *Slicer* slices at $t + \Delta/2$ when $IC(t)$ is low, resulting in two slices from one slicing operation.

To determine the threshold of IC for slicing, we compare two slicing strategies: the dynamic slicing strategy, which slices each segment based on a threshold of IC calculated individually for that segment, and the static slicing strategy, which applies a single IC threshold derived from the entire workload. We analyze the distribution of IC across all segments, shown at the bottom of Figure 6b. The top of Figure 6b shows the locations and corresponding values of IC during the first second of the workload. Although slicing at $IC(t) = 0$ is ideal (indicating the database is idle during $[t, t+\Delta)$), it occurs infrequently and can distort performance by pushing the slice length toward the maximum. Therefore, following the Pareto principle [33] and the histogram in Figure 6b, we measure IC at each Δ and perform slicing when IC falls in the bottom 20% of the histogram. Nevertheless, the results shown in Figure 6c closely resemble those in Figure 6a. We attribute this to the bias caused by the static slicing approach. When performance for a workload fluctuates irregularly, the contextual information can vary significantly across segments, meaning the appropriate IC threshold might differ for each segment. We show the distribution of IC on a segment at the bottom of Figure 6d, where the segment is divided from the source workload by *Analyzer*. The top of Figure 6d shows IC values and locations during the segment's first second. These results confirmed our hypothesis. Therefore, in dynamic slicing, we implement segment slicing by utilizing values within the lowest quintile (20%) of each segment's respective histogram distribution.

Based on this dynamic slicing strategy, we get the new performance metrics through replay and show them in Figure 6e. Execution time, performance trends, and positive correlations between CPU utilization and QPS are maintained. Although both performance metrics are slightly below the baseline during certain periods, likely due to the loss of concurrent conflicts, this is currently the most accurate method for reproducing performance results.

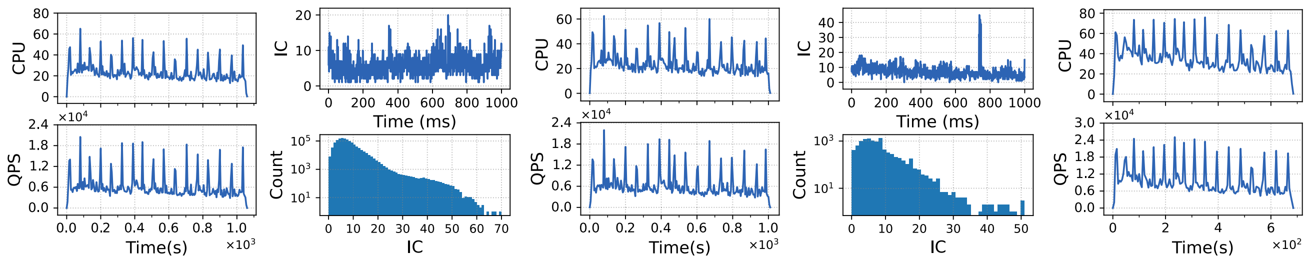
Time complexity. Given that n represents the number of instant time-points required to calculate IC, the time complexity is $O(n\log(n) + n)$, simplified as $O(n\log(n))$. $O(n)$ accounts for the time for calculating all IC values, while $O(n\log(n))$ corresponds to sorting time. In this paper, the longest workload contains approximately 2.16×10^5 instant time-points, given that $\Delta = 3$ ms.

6 COMPRESSOR

Slicing a workload produces numerous slices. To create a compressed workload that retains the diversity of the source workload, the *Compressor* samples these slices. Random sampling is inadequate as it often fails to identify representative slices. To address this, we adopt a cluster-then-sample strategy. The *Compressor* first clusters slices within each segment based on their features. It then samples slices from different clusters, ensuring a balanced representation of the source workload's characteristics. This approach enables comparable performance between the compressed and source workloads when using the same configurations.

6.1 Slice cluster

To vectorize a slice based on its multidimensional performance features, we use the additional features shown in Table 1. Generally, because each slice contains a finite number of queries, we define the feature of a slice as the sum of features of all queries within it. Formally, $F_c(S) = \sum_{sql \in S} F_c(sql)$, where $F_c(sql)$ represents the value corresponding to the feature c of a query. For example, $F_{ExecTime}(sql_i)$ denotes the execution time of sql_i. Thus, the workload W can be represented as $W = [F(S_1), F(S_2), ...]$, where $F(S_i) = [F_{c_1}(S_i), F_{c_2}(S_i), ...]$. For clustering, we normalize each $F(S_i) \in W$ using $F_c(S_i) = \frac{F_c(S_i) - \min(F_c)}{\max(F_c) - \min(F_c)}$, where $F_c = \{F_c(S_i), F_c(S_j), \dots\}$



(a) Fixed slicing. **(b) IC of all segments.** **(c) Static slicing.** **(d) IC of a segment.** **(e) Dynamic slicing.**

Figure 6: Workload replay with a dynamic slicing strategy.

represents the set of feature values for all slices corresponding to feature c. We evaluate varying query feature aggregation methods, including sum and max, and adopt sum as it achieves high empirical performance. We leave investigations of better query feature aggregation methods as future work.

Based on $F(S_i)$, we employ the *Hierarchical Clustering Algorithm* (HCA) [31, 32] to group slices, resulting in an effective sampling basis. HCA is a state-of-the-art unsupervised ML clustering algorithm that does not require predefining the number of clusters. It builds a hierarchy of clusters by grouping objects based on their features, forming a tree-like structure called a dendrogram. This dendrogram can represent the nested grouping of objects and the sequence in which clusters are merged or split, adapting to varying workload compression rates without re-clustering. Note that *SCompression* uses a "bottom-up" strategy to build this hierarchical structure. It starts with each object in its own cluster and then successively merges the closest pairs of clusters, with proximity measured using the *Euclidean Distance* in this study, until all objects are in a cluster, where each cluster starts with one slice.

Time complexity. The time complexity for clustering [31, 32] is $O(n^2 \log n)$, where n is the number of slices. In this paper, the maximum number of slices is approximately 1.4×10^4.

6.2 Slice sampling

Based on the hierarchical tree, where each leaf node contains a slice and other nodes denote clusters, we propose a "top-down" sampling strategy to obtain a specified length of slices.

Our algorithm recursively samples slices, starting from the root node (Algorithm 1). Given a node *node* and a given length (representing the execution time, as detailed in Section 2.2) of slices L, it first adds all the leaf nodes of the given *node* to a pool of candidates $\overline{W}$. If there is a node child that is not a leaf node, the algorithm calculates the length $\overline{L}$ and adds the candidate nodes *leaves* from the node *child* into $\overline{W}$. The length $\overline{L}$ of *child* is calculated by the length of all leaves of both *child* and *node*:

$$\overline{L} = \alpha \cdot L \cdot Length(child.all\_leaves)/Length(node.all\_leaves).$$

To prevent $\overline{L}$ from being too small to contain any slices, we use α to control its value. We set $\alpha = 1.3$ to obtain more samples from *child* nodes without influencing the length of the result.

Finally, the algorithm iterates through the candidate pool $\overline{W}$ and randomly excludes one node from the closest pair each time until the length of candidate nodes is less than the target length L.

Algorithm 1: SampleFromNodes

```
Input: Node node, given length of slices L
Output: Sampled Slices $\overline{W}$
$\overline{W}$ = [ ]
foreach child ∈ node.nodes do
    if $child.is\_leaf = True$ then
        $\overline{W}$.append(child)
    else
        $\overline{L} \leftarrow \alpha \cdot L \cdot \frac{Length(child.all\_leaves)}{Length(node.all\_leaves)}$
        $leaves \leftarrow SampleFromNodes(child, \overline{L})$
        $\overline{W}$.append(leaves)
    end
end
while $Length(\overline{W}) > L$ do
    $\arg\min_{i,j} Distance(S_i, S_j), (S_i, S_j \in \overline{W}, i \neq j)$
    Remove one of the leaves in $\{S_i, S_j\}$ randomly
end
return
```

Time complexity. The time complexity for sampling is $O(n \log m)$, where n is the number of slices in the compressed workload and m is the total number of slices in the source workload, usually $n \ll m$.

7 REPLAYER

Given a compressed workload $W = [S_i, .., S_j]$, we propose a replay strategy based on the concept of slices. The principle of replay is to produce performance according to the sequence and concurrency of transactions preserved in the W. To this end, we establish an intra-slice replay policy and an inter-slice replay policy, respectively.

For the intra-slice replay policy, *i.e.*, replay a slice S_i, we use the transaction's start time as a reference. We execute transactions within the same connection according to their start times. For transactions that overlap in time in different connections, we prioritize the one whose first query has an earlier start time.

A naive inter-slice replay policy involves replaying slices sequentially. For example, when handling consecutive slices $[S_i, S_{i+1}]$, the system initiates S_{i+1} only upon completion of S_i. However, this policy may result in low QPS during replay operations, as some queries may have extended execution durations on the compressed

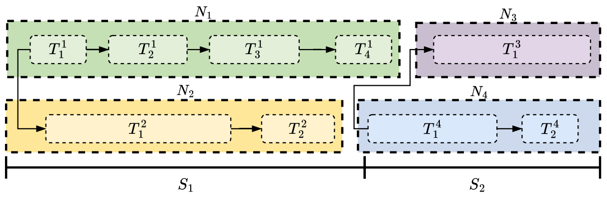


Figure 7: Toy example of slice replay.

workload. Additionally, slice execution periods often demonstrate temporal overlap, even when using the dynamic slicing strategy, further impacting performance.

To address this issue, we introduce a metric called the **SQL Finish Rate**, denoted as $SFR_W(S,t)$, which is formally defined as

$$SFR_W(S,t) = \frac{|\{q \in S|\text{EndTime}(q) < t\}|}{|S|},$$

where $|S|$ is the total number of queries in S. The metric represents the proportion of queries in S that have completed execution in workload W by time t.

We now present our inter-slice replay policy, involving replaying slices $[S_i, S_{i+1}]$ on $\overline{W}$. During the replay of the S_i, S_{i+1} is initiated at time t, when $SFR_{\overline{W}}(S_i,t) \geq \alpha_{S_i}$. The parameter $\alpha_{S_i} = SFR_W(S_i, t_{i+1})$ is calculated from W, where $S_i = S(t_i, t_{i+1})$. By implementing α_{S_i}, we ensure the precise timing of query execution for those queries that begin before the completion of queries in the previous slice, enabling accurate replication of concurrent operations.

To illustrate our replay strategy concretely, we provide a toy example of compressed workload $\overline{W}$ in Figure 7. Transactions in S_1 start first. Within S_1, N_1 and N_2 run concurrently, where N_1 begins by executing T_1^1, followed by T_1^2 starting in N_2. S_2 runs its transactions similarly and starts at t when $SFR_{\overline{W}}(S_1,t) \geq \alpha_{S_1}$.

8 DISCUSSION AND FUTURE WORK

Discussion. To the best of our knowledge, this is the first context-aware approach to compress OLTP workloads for knob tuning, though it is not the only pathway to accelerate this process. For instance, researchers [18, 25, 28] have improved knob tuning efficiency by reducing iterations through knob selection and parameter range optimization. *SCompression* can be seamlessly integrated with these approaches to further improve efficiency.

Future Work. While *SCompression* demonstrates significant effectiveness in compressing OLTP workloads for knob tuning, there remain opportunities for enhancement and further development.

Distance Function. We compared Euclidean and Cosine distance functions for calculating slice distances and found that Euclidean distance performs slightly better. While this approach, leveraging Euclidean distance, is highly effective for knob tuning (as demonstrated in Section 9), its performance diminishes in other tasks, such as index tuning, particularly at higher compression rates (see Section 9.5). Developing task-specific distance functions remains a critical challenge for future work.

Low-Frequency Queries. A notable consideration in sampling is the careful handling of underrepresented low-frequency samples. While we can implement sophisticated strategies to retain valuable low-frequency samples, the inherent trade-offs necessitate a balanced approach tailored to specific use cases. This presents an opportunity for continued research and refinement. Nevertheless, our experimental results in Section 9 demonstrate robust performance metrics, confirming the viability of our approach.

Surrogate model. An alternative to accelerate knob tuning is developing a surrogate model that maps configurations to system performance, enabling model-based optimization and minimizing direct system testing. However, training a surrogate model requires a large number of samples [3, 5, 49], inevitably involving multiple executions of the workload. As a result, workload execution is unavoidable, making workload compression essential for reducing knob tuning time. Developing a comprehensive, explainable surrogate model with workload compression remains an open challenge.

9 EXPERIMENTS

We present *SCompression*'s capability to compress OLTP workloads for knob tuning. *SCompression* generates a compressed workload that mirrors the performance changes of its source under identical configurations, significantly reducing knob-tuning time.

9.1 Experiment Setup

Hardware. We conducted experiments on a cloud-based MySQL database (v5.7) and index recommendation on PostgreSQL (v14.14), both configured with 8 CPU cores and 32 GB of RAM.

Workloads. We use four types of workloads, as detailed in Table 2: YCSB-A, TPC-C, Synthesis (by Sysbench), and a real-world OLTP workload referred to as Production. Synthesis is generated by Sysbench, alternating concurrency levels between 16 and 64 threads every 30 seconds, repeating this pattern until the test ends. The production workload, collected from a cloud database, contains approximately 8,500 distinct query templates. Performance metrics, including QPS and CPU usage, are presented in Figure 5. Since the query log does not specify connection establishment or release time, real-time concurrency is approximated by counting the number of unique *ThreadId*s per second.

Competitive Methods. We compare *SCompression* with state-of-the-art workload compression methods, including Query Configuration Sensitivity Analysis (QCSA) and Index-based Workload Summarization (ISUM). QCSA is a tool used in LOCAT [42] that compresses OLAP workloads by selecting queries that show the biggest difference in performance. According to [42], we evaluate 30 configurations to calculate the performance variance. ISUM [34] identifies queries with both high-performance improvement potential and significant influence on others based on their costs, selectivity, and similarity in indexable columns.

Table 2: Workload Information

Name	Queries	Concurrency	Table Size	Length (mins)
YCSB-A	6.9 M	32	179 MB	5
TPC-C	14.4 M	16	3 GB	5
Synthesis	2.3 M	16 - 64	5 GB	5
Production	8.2 M	140* - 464*	2.1 TB	10.8

Knob Tuning Setting. For the knobs to be tuned, we select 30 knobs in MySQL 5.7 according to senior DBAs' advice. We evaluate different compression techniques with three state-of-the-art tuning methods and observe similar performance trends. Due to space constraints, we only show results with the Bayesian Optimization (BO)-based method in most of our evaluations since it's usually slightly better [18]. Evaluations of *SCompression* with other tuning methods are provided in Section 9.5. For tuning, we conduct 150 iterations and identify the configuration yielding the highest throughput as the optimal one. *Bayesian Optimization (BO)* [13] excels at optimizing objective functions that are costly to evaluate and tolerate stochastic noise. It builds a surrogate model of the objective function and uses an acquisition function to guide sampling, enabling effective knob tuning through targeted optimization.
Compression Rate. For knob tuning, workload execution duration serves as the critical performance metric. We utilize time-based measurements rather than query count to establish the compression rate. As discussed in Section 2.2, given W and $\overline{W}$, the compression rate is defined as $CR(\overline{W}) = E(W)/E(\overline{W})$, where $E(W)$ and $E(\overline{W})$ represent the execution times of the source and compressed workloads, respectively. In this paper, we set compression rates at 5, 10, 20, 50, and 80.
Target Metric. When comparing performance across different configurations on the same workload, we use QPS as the primary performance metric, as shown in Section 9.2. To compare the performance trends of the source and compressed workloads, we employ min-max normalized QPS as the key metric, where the calculation of normalized QPS is described in Section 9.3.

9.2 Efficiency Evaluation

End-to-end performance comparison. We evaluate QPS of source workloads (*e.g.*, YCSB-A, TPC-C, Synthesis, and Production) by deploying optimal configurations learned from compressed workloads with 5 different compression rates (*i.e.*, 5, 10, 20, 50, and 80). Table 3 shows the tuning time (150 iterations) and the optimal QPS achieved using OtterTune on source workloads. Figure 8 shows the QPS of the source workload by deploying optimal configurations from different compression rates, where a higher QPS indicates better performance.

In Figure 8, the red dotted line represents the QPS of the optimal configuration tuned on the source workload and serves as a baseline, with corresponding details shown in Table 3. QCSA and ISUM struggle to achieve high QPS with compressed workloads compared to the baseline. As the compression rate increases, the QPS further declines, especially for Synthesis and Production workloads. Notably, QCSA performs slightly better than ISUM, as ISUM compresses the workload based on the indexable columns, which is specifically designed for index recommendation rather than knob tuning. In contrast, *SCompression* is notably better, maintaining relatively high QPS results. While their performance slightly degrades as the compression rate increases, it remains significantly higher than that of QCSA and ISUM and stays close to the baseline.
End-to-end speedup comparison. We measure the end-to-end speedup across different compression rates and use it as a key indicator to appraise the tuning efficiency. A higher speedup reflects a shorter tuning time.

Table 3: Comparison across workloads with 150 iterations.

Workload	YCSB-A	TPC-C	Synthesis	Production
Tuning Time (h)	13.9	14.3	14.1	25.2
Optimal QPS	37450	51716	59078	16799

Table 4: Overheads. "P" for "Prepare" and "C" for "Compress".

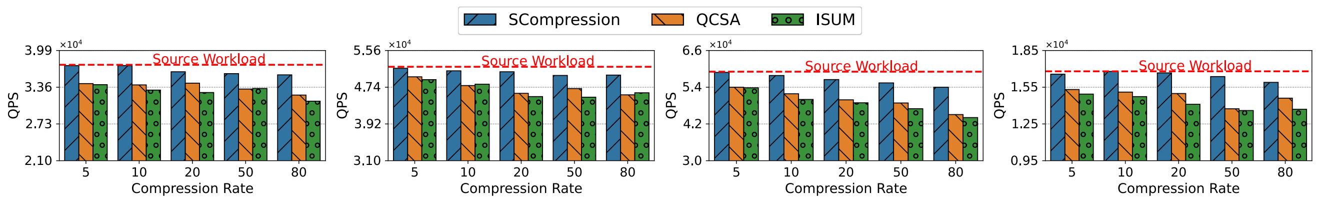
Workload		YCSB-A	TPC-C	Synthesis	Production
SCompression	P	0	0	0	0
	C	25 s	230 s	50 s	190 s
QCSA	P	150 mins	154 mins	157 mins	326 mins
	C	5s	8s	4s	6s
ISUM	P	0	0	0	0
	C	18 mins	52 mins	4 mins	23 mins

The results about tuning time and speedup on different workloads are shown in Table 3 and Figure 9, respectively. The results in Figure 9 show that increasing the compression rate reduces workload execution time, with the speedup rate growing as the compression rate increases. QCSA and ISUM methods achieve similar speedups to slice-based methods at compression rates below 20, around 13x. However, at a compression rate of 50, their speedups drop significantly, with QCSA reaching only 20x and ISUM just 15x, compared to *SCompression* methods at 45x. This observation can be attributed to query-based methodologies which, despite producing compressed workloads with reduced query counts, fail to adequately address query concurrency and demonstrate a tendency to extract extended query sequences within connections. In contrast, slice-based methods like *SCompression* show a consistent proportional increase in speedup as the compression rate rises, though the speedup gains become minimal when the compression rate surpasses 50.
Overhead of workload compression. Table 4 represents the average time cost of *SCompression*, QCSA, and ISUM at different compression rates. Unlike QCSA, *SCompression* and ISUM do not need 30 workload executions for query variance calculation. We designate this additional execution phase as **Prepare**, while the implementation of compression methods is designated as **Compress**.

Given that both *SCompression* and QCSA perform comprehensive workload analysis independent of compression rate, we focus on measuring the average overhead, as processing time maintains consistency across varying compression rates. For ISUM, where overhead decreases as the compression rate increases, we also use the average overhead as the metric for simplicity.

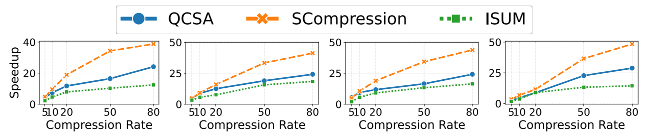
As shown in Table 4, QCSA demonstrates notably higher overhead compared to *SCompression* and ISUM, attributable primarily to extended Prepare time requirements. However, upon examination of knob tuning duration presented in Table 3, all methods maintain reasonable compression overhead levels, with *SCompression* exhibiting enhanced operational efficiency.

These results show that *SCompression* can generate compressed workloads from the source workload while ensuring that the optimal configurations produced on these workloads yield similar



(a) QPS on YCSB-A **(b) QPS on TPC-C** **(c) QPS on Synthesis** **(d) QPS on Production**

Figure 8: QPS on source workloads using optimal configurations generated on compressed workloads.



(a) YCSB-A **(b) TPC—C** **(c) Synthesis** **(d) Production**

Figure 9: Speedup for 150 iterations.

performance on the source workload. Given the speedup and overhead achieved by *SCompression*, we conclude that it significantly cuts workload execution time, making knob tuning more efficient and cost-effective.

9.3 Performance Comparison with the Same Configurations

An effective compression method should ensure that the source and compressed workloads achieve similar performance when using the same configurations. In other words, workloads with similar characteristics should exhibit consistent trends across different knob settings. In this section, we analyze performance similarity by selecting 15 configurations and measuring QPS for both the source and compressed workloads on TPC-C and Production workloads. To examine performance trends across different knobs, we normalize the QPS values of a workload compressed by a specific method and rate to the range [0, 1] using min-max normalization, defined as: $P_{normalized} = \frac{P-\min(P)}{\max(P)-\min(P)}$.

As a baseline, Figure 10a and Figure 10e display the normalized QPS results collected from source workloads (*i.e.*, TPC-C, Production) across 15 configurations. As shown in Figure 10b, 10c, and 10d, query-based methods like ISUM and QCSA get reasonable normalized QPS on TPC-C, closely resembling the baselines at lower compression rates (*i.e.*, below 10). However, as the compression rate increases, these methods struggle to maintain similarity with the source workload. In contrast, the slice-based method, like *SCompression*, demonstrates impressive similarity even at higher compression rates (*i.e.*, 50). Yet, their effectiveness diminishes when the compression rate becomes too high, such as at 80. TPC-C is a stable workload, with its QPS remaining nearly constant throughout execution. This results in minimal variation across different slices, allowing both query-based and slice-based methods to perform well when the compression rate is low.

As shown in Figure 10f, 10g, and 10h, on the Production workload, the similarity exhibits varying trends across different compression methods. We attribute it to the fact that Production is more complex than TPC-C. As shown in Figure 5, its QPS is uneven and varies during the whole period.

Our analysis reveals three key findings. First, ISUM demonstrates poor similarity performance even at a minimal compression rate of 5, and its performance deteriorates further at higher rates. We attribute it to the fact that ISUM only looks at indexable columns and ignores query concurrency, making it miss important high-contention queries. Second, while QCSA works well at the compression rate of 5, its similarity accuracy drops significantly at higher rates. Queries with high contention are more susceptible to configuration changes, giving QCSA a greater potential to maintain concurrency compared to ISUM. Third, *SCompression* maintains consistent QPS similarity across various compression rates, even outperforming others at higher compression rates (*i.e.*, 50).

These results demonstrate that both query-based methods and *SCompression* maintain performance similarity between compressed and source workloads at low compression rates for simple workloads like TPC-C. However, *SCompression* performs significantly better at higher compression rates (*e.g.*, above 20) and consistently outperforms query-based methods in preserving similarity. Query-based methods are less effective on real-world workloads, where compressed workloads exhibit different QPS trends under the same configurations. While QCSA performs well at low compression rates by selecting configuration-sensitive queries, it struggles as the compression rate increases. In contrast, *SCompression* excels by effectively preserving workload characteristics, capturing representative slices across various compression rates, even up to 50.

9.4 Ablation Study

To compare the effectiveness of designs in *SCompression*, we conduct an ablation study on the following components: *Analyzer* (**None** denotes skipping analysis; **GGS** denotes segmentation using GGS), *Slicer* (**Fixed** denotes fixed-length slicing, *e.g.*, 30ms; **Static** denotes use a static threshold from the whole workload; **Dynamic** denotes dynamic slicing), and *Compressor* (**Random** denotes random sampling; **Cluster** denotes cluster-based sampling). All results are shown in Table 5 and Table 6, where the numbers (*e.g.*, 10, 50, and 80) below QPS and Tuning Time are compression rates.

For TPC-C, as shown in Table 5, all strategies perform similarly when the compression rate is low (*e.g.*, 10). However, as the compression rate increases (*e.g.*, 50), the dynamic slicing and cluster-based sampling strategies exhibit a clear advantage in maintaining higher performance. The static slicing strategy generally performs slightly better than the fixed slicing strategy, while the benefits

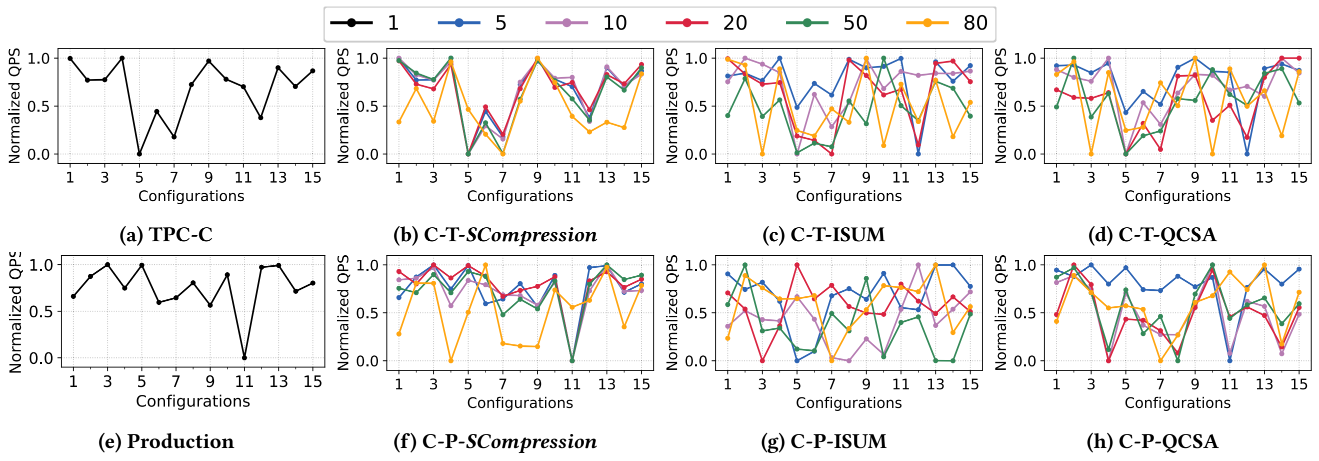


Figure 10: Normalized QPS on workloads (*i.e.*, TPC-C, Production, and their compressed versions) across varying compression rates and configurations. “C-T” represents “compressed TPC-C”. “C-P” represents “compressed Production”.

of dynamic slicing become more pronounced at higher compression rates (*e.g.*, 80). This highlights the effectiveness of dynamic slicing in preserving performance under significant compression. We attribute it to TPC-C being a stable workload with nearly constant QPS. This stability results in a minimal variation in monitor metrics over time, making it challenging for **GGS** to effectively segment the workload. However, an aggressive compression rate will compromise this stability by disrupting concurrency integrity in fixed and static slicing and losing representative slices in random sampling. The advantages of dynamic slicing and cluster-based sampling become pronounced, demonstrating their effectiveness in extreme situations.

The results on the Production workload confirm the above conclusion, as Production is more complex and unstable than TPC-C. The results in Table 6 highlight key differences in performance. Even at a low compression rate (*i.e.*, 10), the random sampling strategy underperforms compared to the cluster-based sampling strategy. Although dynamic slicing improves performance, combining it with the random sampling strategy yields worse results than using fixed-length slicing with the cluster-based sampling strategy. These results demonstrate that the cluster-based sampling strategy is crucial for performance, particularly when handling complex workloads.

Our analysis demonstrates that implementing any combination of the discussed technologies (*e.g.*, GGS in Segment, Dynamic Slicing, and Cluster-based Sampling) enhances system performance relative to baseline approaches of fixed or static slicing and the random sampling strategy. The optimal performance is achieved through the integrated application of all three techniques. The analysis further reveals that as compression rates increase, the cluster-based sampling strategy maintains significantly better performance compared to the random sampling strategy, primarily due to its superior ability to preserve workload characteristics. The data show that cluster-based sampling outperforms dynamic slicing, underscoring its key role in preserving workload representativeness and enhancing system efficiency.

Table 5: Ablation Study on TPC-C.

Module			QPS ($\times 10^4$)			Tuning Time (h)		
Analyzer	Slicer	Compressor	10	50	80	10	50	80
None	Fixed	Random	5.08	5.01	4.87	1.6	0.6	0.6
None	Fixed	**Cluster**	5.10	5.07	4.91	1.5	0.4	0.4
None	Static	Random	5.08	5.02	4.87	1.6	0.7	0.6
None	Static	**Cluster**	5.09	5.08	4.95	1.5	0.5	0.4
GGS	**Dynamic**	Random	5.09	5.07	5.00	1.9	0.6	0.6
GGS	**Dynamic**	**Cluster**	5.10	5.09	5.02	1.5	0.4	0.3

Table 6: Ablation Study on Production.

Module			QPS ($\times 10^4$)			Tuning Time (h)		
Analyzer	Slicer	Compressor	10	50	80	10	50	80
None	Fixed	Random	1.56	1.48	1.36	3.8	1.0	0.8
None	Fixed	**Cluster**	1.63	1.58	1.50	3.6	0.8	0.7
None	Static	Random	1.56	1.50	1.45	4.2	0.9	0.8
None	Static	**Cluster**	1.64	1.60	1.53	3.9	0.7	0.6
GGS	**Dynamic**	Random	1.58	1.52	1.48	4.1	0.9	0.7
GGS	**Dynamic**	**Cluster**	1.66	1.63	1.59	3.7	0.7	0.5

9.5 Performance of *SCompression* with Different Tuners and Effects for Index Recommendation

Different tuners. To evaluate *SCompression*’s effectiveness with different tuners, we test it using three popular tuners: OtterTune (BO-based), CDBTune (RL-based), and LlamaTune (BO-based). We measured the QPS of the source workload by deploying optimal configurations tuned on the compressed workloads (*e.g.*, TPC-C and Production) over 150 iterations. The results are shown in Table 7. The results indicate similar optimal throughput across the evaluated tuners, highlighting the broad applicability of *SCompression*. Notably, LlamaTune identifies the optimal configuration first.

Table 7: Performance of *SCompression* with different tuners. "T" denotes TPC-C. "P" denotes Production.

	Tuner	OtterTune			CDBTune			LlamaTune		
	cr	5	20	50	5	20	50	5	20	50
T	QPS ($\times 10^4$)	5.17	5.10	5.09	5.16	5.08	5.06	5.16	5.09	5.08
P	QPS ($\times 10^4$)	1.66	1.65	1.63	1.67	1.65	1.64	1.67	1.66	1.64

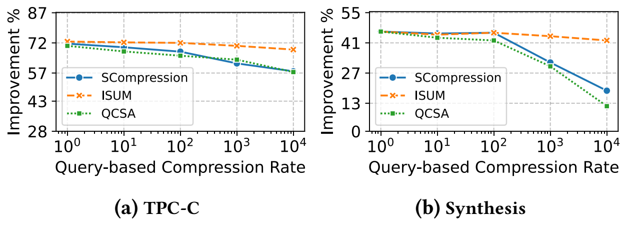


Figure 11: Improvement of index recommendation.

Index recommendation. We evaluate *SCompression*, ISUM, and QCSA on PostgreSQL for index recommendation, benchmarking their performance improvement on TPC-C and Synthesis workloads. For evaluation, we use the Database Engine Tuning Advisor (DTA) [1, 8, 20]. It is an advanced version of the AutoAdmin index selection tool for Microsoft SQL Server, identifying index candidates for individual queries and determining the optimal configuration for the entire workload using a greedy enumeration approach.

Since index advisors process workloads as sequentially executed queries [20], we adopt the query-based compression rate [11, 34], defined as $\frac{|W|}{|\overline{W}|}$, where $|W|$ and $|\overline{W}|$ represent the quantities of queries in the source and compressed workload, respectively. The improvement (%) is calculated as the reduction in the source workload execution time when using recommended indexes based on the tuning of the compressed workload, relative to the source workload execution time. As shown in Figure 11, at lower query-based compression rates (*e.g.*, below 10), all methodologies demonstrate satisfactory performance due to the increased retention of essential queries. Nevertheless, when compression rates exceed 100, both *SCompression* and QCSA exhibit diminished effectiveness. The performance degradation is due to their focus on knob-tuning over index selection, reducing their effectiveness in this context.

10 RELATED WORK

Existing studies on SQL-level workload compression primarily focus on reducing query count, particularly for OLAP workloads, which aim to accelerate workload analysis processes [7, 34].

Query-based Workload Compression. Workload compression for queries was first explored by Chaudhuri [7, 9], who employed KMED, All-Pairs, and Random Sampling to identify representative queries for tasks like index recommendation. Later studies [17] introduced NLP techniques to learn vector representations of SQL queries, treating them as textual data to support generalized workload analytics. Ettu [21, 22] is a system designed to analyze SQL query logs, assisting database administrators (DBAs) in identifying patterns and detecting security issues. It converts SQL queries into abstract syntax trees (ASTs) and leverages the Weisfeiler-Lehman (WL) graph algorithm to extract critical features. To cluster SQL queries based on structural similarity, [21–23] rely on query structure, avoiding the need for database data or schema. They enhance clustering quality through a regularization process that standardizes SQL queries by canonicalizing names, applying equivalence rules, and transforming query structures (*e.g.*, syntax desugaring and nested query flattening). GSUM [11] formalizes workload representativity and coverage, improving feature (*e.g.*, column) coverage and aligning compressed workloads with the original distribution with an efficient greedy algorithm. ISUM [34] selects queries that significantly impact performance, particularly those involving indexable columns. LOCAT [42] introduces Query Configuration Sensitivity Analysis (QCSA) to eliminate queries insensitive to configuration parameter changes.

DBMS Configuration Tuning. DBMS configuration tuning is an active research area [2, 19, 49, 50], with approaches generally falling into rule-based and model-based categories. Rule-based methods, such as IBM's DB2 self-tuning memory manager and DB2 Performance Wizard [24, 35, 38], automate initial parameter settings, while tools like BestConfig [51] optimize configurations under resource constraints, and COMFORT [41] applies control theory to single-knob tuning, though it struggles with interdependent knobs. SARD [10] ranks knobs by performance impact using Plackett-Burman design. Model-based methods use machine learning to recommend configurations, with examples like Tran *et al.* [39] employing regression models for buffer tuning and iBTune [36] utilizing deep neural networks to optimize buffer pool size. Other tools [12, 26, 40, 45, 48] adopt Bayesian Optimization or Reinforcement Learning for automated tuning. Recent advances [6, 14, 25, 49] reduce tuning costs by integrating black-box and white-box models, parallel strategies, and large language models to provide efficient configuration recommendations.

11 CONCLUSION

In this paper, we propose *SCompression*, a slice-based OLTP workload compression method for knob tuning. We treat the workload as a sequence of slices that contain transactions, slicing the workload to maintain high concurrency conflicts. *SCompression* uses a dynamic slicing strategy, a cluster-based technique, and a replay strategy to generate and execute compressed workloads, resulting in similar performance between the compressed and source workloads on the same configurations. Extensive experiments demonstrate that *SCompression* is a generic auxiliary approach. Deploying *SCompression* offers a cost-effective solution for knob tuning.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their insightful feedback and constructive suggestions, and we gratefully acknowledge Tencent for their support. This work was achieved in Key Laboratory of Information Storage System and Ministry of Education of China. This work was supported by the National Key Research and Development Program Grant No.2023YFB4502701, the National Natural Science Foundation of China Grant No.62232007, and the Natural Science Foundation of Hubei Province No.2022CFB060.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 1110–1121. https://doi.org/10.1016/B978-012088469-8.50097-8

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft SQL server 2005: demo. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 930–932. https://doi.org/10.1145/1066157.1066292

[3] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2011. The Case for Predictive Database Systems: Opportunities and Challenges.. In *CIDR*, Vol. 2011. Citeseer, 167–174.

[4] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1009–1024. https://doi.org/10.1145/3035918.3064029

[5] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. Tastes great! Less filling! High performance and accurate training data collection for self-driving database management systems. In *Proceedings of the 2022 International Conference on Management of Data*. 617–630.

[6] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: an online cloud database hybrid tuning system for personalized requirements. In *Proceedings of the 2022 International Conference on Management of Data*. 646–659.

[7] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. 2002. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 488–499.

[8] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime algorithm of database tuning advisor for microsoft sql server.

[9] Surajit Chaudhuri, Vivek Narasayya, and Prasanna Ganesan. 2003. Primitives for workload summarization and implications for SQL. In *Proceedings 2003 VLDB Conference*. Elsevier, 730–741.

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

[11] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey Naughton, and Stratis Viglas. 2020. Comprehensive and efficient workload compression. *arXiv preprint arXiv:2011.05549* (2020).

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

[13] Peter I Frazier. 2018. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018).

[14] Victor Giannakouris and Immanuel Trummer. 2024. Demonstrating λ-Tune: Exploiting Large Language Models for Workload-Adaptive Database System Tuning. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 508–511. https://doi.org/10.1145/3626246.3654751

[15] David Hallac, Peter Nystrup, and Stephen Boyd. 2019. Greedy Gaussian segmentation of multivariate time series. *Advances in Data Analysis and Classification* 13, 3 (2019), 727–751.

[16] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. 2018. OLTP through the looking glass, and what we found there. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 409–439.

[17] Shrainik Jain, Bill Howe, Jiaqi Yan, and Thierry Cruanes. 2018. Query2vec: An evaluation of NLP techniques for generalized workload analytics. *arXiv preprint arXiv:1801.05613* (2018).

[18] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-efficient DBMS configuration tuning. *arXiv preprint arXiv:2203.05128* (2022).

[19] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *Proc. VLDB Endow.* 15, 11 (2022), 2953–2965. https://doi.org/10.14778/3551793.3551844

[20] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395. http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf

[21] Gokhan Kul, Duc Luong, Ting Xie, Patrick Coonan, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. 2016. Ettu: Analyzing query intents in corporate databases. In *Proceedings of the 25th international conference companion on world wide web*. 463–466.

[22] Gokhan Kul, Duc Luong, Ting Xie, Patrick Coonan, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. 2016. Summarizing large query logs in ettu. *arXiv preprint arXiv:1608.01013* (2016).

[23] Gokhan Kul, Duc Thanh Anh Luong, Ting Xie, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. 2018. Similarity metrics for SQL query clustering. *IEEE Transactions on Knowledge and Data Engineering* 30, 12 (2018), 2408–2420.

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

[25] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1939–1952.

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

[27] Wan Shen Lim, Matthew Butrovich, William Zhang, Andrew Crotty, Lin Ma, Peijing Xu, Johannes Gehrke, and Andrew Pavlo. 2023. Database Gyms. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p27-lim.pdf

[28] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. 2024. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3680–3693.

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

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

[31] Fionn Murtagh and Pedro Contreras. 2012. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2, 1 (2012), 86–97.

[32] Fionn Murtagh and Pedro Contreras. 2017. Algorithms for hierarchical clustering: an overview, II. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 6 (2017), e1219.

[33] Robert Sanders. 1987. The Pareto principle: its use and abuse. *Journal of Services Marketing* 1, 2 (1987), 37–40.

[34] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. Isum: Efficiently compressing large and complex workloads for scalable index tuning. In *Proceedings of the 2022 International Conference on Management of Data*. 660–673.

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

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

[37] Robert L Thorndike. 1953. Who belongs in the family? *Psychometrika* 18, 4 (1953), 267–276.

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

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

[40] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1009–1024.

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

[42] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. Locat: Low-overhead online configuration auto-tuning of spark sql applications. In *Proceedings of the 2022 International Conference on Management of Data*. 674–684.

[43] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 674–684. https://doi.org/10.1145/3514221.3526157

[44] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1567–1581. https://doi.org/10.1145/2882903.2915222

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

[46] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 415–432. https://doi.org/10.1145/3299869.3300085

[47] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 15, 9 (2022), 1808–1821. https://doi.org/10.14778/3538598.3538604

[48] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the 2021 International Conference on Management of Data*. 2102–2114.

[49] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 631–645. https://doi.org/10.1145/3514221.3526176

[50] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. 2023. An Efficient Transfer Learning Based Configuration Adviser for Database Tuning. *Proc. VLDB Endow.* 17, 3 (2023), 539–552. https://www.vldb.org/pvldb/vol17/p539-zhang.pdf

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