PinSQL: Pinpoint Root Cause SQLs to Resolve Performance Issues in Cloud Databases

Xiaoze Liu[†‡], Zheng Yin[§], Chao Zhao[†‡], Congcong Ge[†‡], Lu Chen[†], Yunjun Gao[†],
Dimeng Li[§], Ziting Wang[§], Gaozhong Liang[§], Jian Tan[§], Feifei Li[§]
[†]Zhejiang University, [‡]Alibaba-Zhejiang University Joint Institute of Frontier Technologies, [§]Alibaba Group, China
[†]{xiaoze, yuhao.zhao, gcc, luchen, gaoyj}@zju.edu.cn
[§]{yinzheng.yz, lidimeng.ldm, zizhou.wzt, gaozhong.lgz, j.tan, lifeifei}@alibaba-inc.com

2022 IEEE 38th International Conference on Data Engineering (ICDE) | 978-1-6654-0883-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDE53745.2022.00236

***Abstract*—Deploying database services on cloud systems has gained increasing popularity and has become a common practice in the industry. However, the complicated cloud environments make performance issues inevitable, which could violate the service level guarantee if not addressed in a timely manner. Among the various problems, anomalies in SQL queries are the most commonly reported sources that cause performance issues in database applications. These anomalous queries can be divided into High-impact SQLs (H-SQLs) and Root Cause SQLs (R-SQLs), representing the related SQLs that are correlated with the anomalies and the ones that are the root causes of the performance issue, respectively. In the presence of a large number of queries, to pinpoint the R-SQLs is far more difficult than to identify the H-SQLs. To address this challenge, we aim at automatically pinpointing the R-SQLs to resolve performance issues in cloud databases.**

This paper introduces PinSQL, an autonomous diagnosing system for Alibaba Cloud, which has four modules that are executed sequentially, including data collection and pre-processing, anomaly detection, root cause analysis, and repairing actions. First, the related performance metrics and query logs from monitored cloud database instances are collected and aggregated as the data sources. Then, based on these inputs, efficient anomaly detection is conducted in real-time. Upon the detection of an anomaly, the root cause SQLs are pinpointed through tracking the propagation chain of the involved SQLs. Finally, repairing actions are suggested and then executed on R-SQLs to address the anomalies. Extensive experiments on an Alibaba production system show that PinSQL can achieve an 80% accuracy for pinpointing the top-1 R-SQLs and successfully resolve the database performance issues resultantly.

***Index Terms*—cloud databases, root cause, performance anomaly**

I. Introduction

With the development of cloud computing technologies, many mission-critical services have been deployed on the cloud, where the services are usually hosted on cloud database systems such as Alibaba Cloud RDS [1], AWS RDS [2], Microsoft Azure SQL Database [3], and Google Cloud SQL [4]. However, the complicated cloud environments make performance issues inevitable, which could violate the service level guarantee if not addressed in a timely manner. Among the various problems, it has been reported that 70% of them are caused by database problems [5]. These performance anomalies could lead to potential service interruptions and thus adversely affect customers' business operations. In order to provide services with high elasticity, availability, and stability, cloud database vendors have paid much attention to efficiently diagnosing performance issues, such as drastic increases in CPU usage, spikes in the number of running threads, large fluctuations in business traffic flows, and so on.

To proactively prevent the performance issue, it is not sufficient to detect anomalies alone, and it is also crucial to diagnose the root cause of detected anomalies. Many studies have investigated the problem of root cause analysis (RCA for short) of performance anomalies on cloud databases, including classification-based, Top-SQL-based, and Autoregressive-based approaches. Specifically, classification-based approaches [6]–[9] divide the causes into a limited collection of types. Top-SQL-based approaches [2], [10]–[12] select the highest SQLs via sorting performance metrics. Autoregressive-based approaches [13]–[15] analyze causal dependency between variables on multivariate time-series data. In real-world operations and maintenance (O&M) scenarios, most performance anomalies in cloud databases are caused by a large number of concurrent and competing transactions [6]. Among those anomalous SQL queries, **Root Cause SQLs (R-SQLs)**, such as business scenario change (QPS sudden increase), poor SQL statements, and MDL locks/Row locks, are the keys to resolve the performance anomalies. Although existing studies have significantly reduced human labor for identifying anomalies, they cannot pinpoint the R-SQLs to efficiently resolve the performance issues. Moreover, many existing works [6]–[9] aim to optimize or tackle system problems. However, system problems caused by R-SQLs might not need special treatment. For example, for solving the CPU bottleneck caused by some CPU-intensive R-SQLs, it is not necessary to apply instance scaling. A more reasonable method is to perform targeted optimization on R-SQLs instead of system problems, which can reduce the impact of anomalies, thereby improving the overall stability of the database instance.

As a large number of SQL queries exist in cloud databases, it is difficult or costly for DBAs to *manually* find out the anomalous queries that directly or indirectly affect key performance metrics. Driven by this, we aim to pinpoint R-SQLs automatically. Considering various types of SQLs, we usually aggregate SQL queries into different SQL templates [16]–[19]. Hence, instead of finding out specific root cause queries, we focus on SQL templates in this paper. Performance anomalies caused by SQL queries are related to a high active session of the database instance [20] (to be detailed in Definition II.4),

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00236

where active session denotes the number of active SQL queries each timestamp, and an active session of a template consists of the number of active SQL queries that belong to this same template. Once a group of anomalous SQLs (i.e., R-SQLs) appear, High-impact SQLs (H-SQLs for short), being the R-SQLs themselves or SQLs affected by R-SQLs, will appear simultaneously. The H-SQLs directly affect the instance performance, incurring the anomalies of active session detected by the anomaly detector. We describe this entire process of generating performance anomalies as an anomaly propagation chain: R-SQLs→H-SQLs→active session. We aim to locate R-SQLs through the anomaly propagation chain. Three challenges exist as below when pinpointing R-SQLs.

Challenge I: *How to effectively obtain the individual active session of templates without degrading the database instance performance?* To model the impact of templates on the instance active session, we need to obtain the *individual* active session of each template. A straightforward method is to utilize the total response time of templates, as the response time of queries is positively correlated with the active session metric [20]. However, this method is inaccurate because the response time of SQL queries cannot fully represent whether they are active. In order to obtain accurate active session of templates, conventional approaches utilize database built-in monitoring systems [21]. However, such monitoring systems will produce performance overhead, especially in large-scale production environments. Hence, it is difficult to obtain accurate active sessions of templates without degrading the database instance performance.

Challenge II: *How to correctly model the impact of SQL templates on the instance active session?* After obtaining the individual active session of each template, we need to model the impact of each template on the instance active session in order to locate H-SQLs, as H-SQLs are those templates that directly cause the sudden change of active session. Templates that have a tremendous amount of traffic flow are often regarded as H-SQLs by Top-SQL-based approaches. However, these templates may not be H-SQLs. This is because stable traffic queries may not be affected by R-SQLs on large cloud database instances. For example, given R-SQLs UPDATE queries, they only block other queries that operate on the same table but will not affect queries that examine other tables. Motivated by this, templates that directly cause an anomaly should have both considerable traffic and a similar trend with the active session. Commonly used correlation coefficients only consider how similar the trends of two time-series data are, but ignore the scale. Thus, it is difficult to locate H-SQLs.

Challenge III: *How to distinguish R-SQLs from massive SQLs through the located H-SQLs?* Existing industrial solutions only provide a ranking of templates by metrics (e.g., total response time for the active session), leaving the task of finding the root cause to DBAs. However, the active session metric might not be affected too much by R-SQLs, while the H-SQLs affected by R-SQLs lead to high active session. Thus, it's tough for DBAs to pinpoint R-SQLs by simply ranking the related metrics of templates during the anomaly

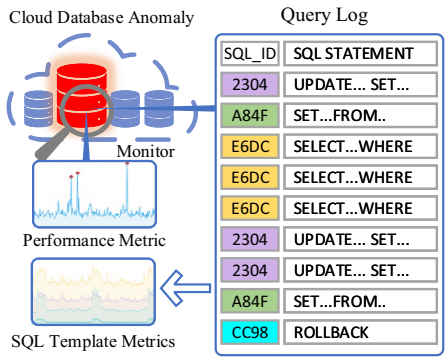


Fig. 1. Diagnosing DB Instance Anomaly with SQL Query Logs. The query log shows the unique SQL template ID (SQL_ID for short) and SQL Statements.

time. For example, assume a SALES table in the database instance. A set of SQL queries to UPDATE the table arrives, which adds exclusive locks on many rows in this table. Thus, running SELECT threads on this table are forced to wait due to exclusive locks, resulting in a sudden increase of active session metric. Finally, the increase of the active session metric is detected by anomaly detectors. In this case, the SELECT queries are H-SQLs, while UPDATE queries are R-SQLs. Therefore, it is difficult to find out R-SQLs, especially in database instances with complicated business queries from different applications.

To address the challenges stated above, we propose a system PinSQL. As depicted in Fig. 1, PinSQL analyses aggregated metrics from query logs to find R-SQLs that causing anomaly cases. PinSQL consists of four modules, i.e., *Data Collection And Anomaly Detection Module*, *High-impact SQL Identification Module*, *Root Cause SQL Identification Module*, and *Repairing Module*. First, Data Collection And Anomaly Detection Module collects and aggregates performance metrics and query logs from cloud database instances. It also detects performance anomalies in real-time by performance metrics. Once an anomaly is detected, following the anomaly propagation chain, High-impact SQL Identification Module is triggered to locate H-SQLs. Thereafter, based on the result of High-impact SQL Identification Module, Root Cause SQL Identification Module select possible R-SQLs through a clustering-based strategy. To react to the anomaly, PinSQL comes with a Repairing Module to suggest/execute actions on R-SQLs, to solve the anomalies. Furthermore, we integrate PinSQL into the Database Autonomy Service (DAS) of Alibaba Cloud [22]. We summarize the key contributions as follows:

- We develop PinSQL, an autonomous diagnosing system that includes two key features (i.e., root cause analysis and automatic repairing), to solve the problem of pinpointing root cause SQLs for the performance issues in cloud databases.
- We introduce a Data Collection And Anomaly Detection Module, which estimates the active session of each template with little performance overhead on database instances.
- We propose a High-impact SQL Identification Module, which fuses the multi-level impact of SQL templates on active session to effectively identify H-SQLs.
- We present a Root Cause SQL Identification Module, which

utilizes a clustering-based strategy to select possible R-SQLs via their trends on execution number. It accurately distinguishes R-SQLs based on H-SQLs by recognizing the trends of execution number.
- Comprehensive experimental results on real-world anomaly cases demonstrate the superiority of our proposed PinSQL for identifying and handling R-SQLs, compared against existing approaches.

II. Problem Statement

In this section, we formalize our problem of pinpointing R-SQLs and related concepts.

Definition II.1 (Time-series Data)**.** A time-series data is a sequence of data sample points: $X = \{x_1, x_2, ..., x_N\}, x_i\ (1 \leq i \leq N) \in \mathbb{R}$, where $x_i\ (1 \leq i \leq N)$ is an observation value at timestamp t_i, and thus, X is an process observation during the time period $[t_1, t_N]$ with a fixed time interval $\frac{t_N - t_1}{N}$.

We usually use 1 second or 1 minute as the time interval to record or synchronize the time-series data. To simplify the notation, when accessing an element by its subscription in a time-series data, we assume that directly using the timestamp is equivalent to using the index (i.e., the distance from the timestamp to the starting time divided by interval). In other words, we can use both X_{t_1} and X_1 to obtain x_1. Although these two methods are different in implementation, the transformation between the two methods is trivial.

Definition II.2 (Anomaly Case)**.** An anomaly case contains data for root cause analysis. It is denoted as $\mathcal{C} = (\mathcal{M}, \mathcal{Q}, a_s, a_e)$, where $\mathcal{M}$ is the set of performance metrics, $\mathcal{Q}$ is the set of SQL templates, and $[a_s, a_e)$ is the detected anomaly time period (a_s and a_e are the timestamps when the anomaly starts and ends, respectively).

An anomaly is defined as a collection of multiple anomalous phenomena of performance metrics, including spike up/down, level shift up/down [9] observed in the performance metric. Spike indicates a sudden increase/decrease then recover, while level shift indicates a sudden increase/decrease without recovering for a long period. The anomaly period is the period between the anomaly phenomena is detected and it recovers. In order to identify the root cause of the anomaly case $\mathcal{C}$, we collect the logs and metric data during the time period $[a_s - \delta_s, a_e)$ rather than $[a_s, a_e)$, where δ_s is an offset to collect more information. Here, δ_s is introduced as R-SQLs usually appear before the actual anomaly case. To simplify the notation, we define $t_s = a_s - \delta_s$ and $t_e = a_e$.

Definition II.3 (SQL Template)**.** A SQL template (or SQL digest) is a composite of multiple SQL queries that are structurally similar but might have different literal values.

The SQL template replaces hard-coded values in the SQL statement with a placeholder (e.g., '?'). For example, a SQL template `SELECT * FROM user_table WHERE uid = ?` includes the following SQL queries:
- `SELECT * FROM user_table WHERE uid = 123456`
- `SELECT * FROM user_table WHERE uid = 654321`
- `SELECT * FROM user_table WHERE uid = 123321`

For each SQL query q in the template Q ($\in$ $\mathcal{Q}$), we collect the query response time (or DB time, execution time) $t_{res}(q)$, the total number of examined rows by the query $\#\text{examined\_rows}(q)$, and the timestamp (in milliseconds) that the query reaches the database $t(q)$.

Definition II.4 (Performance Metric)**.** Each Performance Metric $M \in \mathcal{M}$, indicating one specific system performance, is a time-series data sampled every second by monitoring system from the database instance during the period $[t_s, t_e)$.

In this paper, we only focus on the active session metric (i.e., the number of active queries at the current timestamp), which is the most significant performance metric in the industry [20], [23] for troubleshooting performance issues. The reason is that most performance issues are always accompanied by anomalous features (e.g., spike or sudden increase) of the active session metric. To prove the importance of active session, we categorize the performance issues based on the experience of senior expert DBAs in Alibaba Cloud, and divide R-SQLs into three categories: (1) The performance issues of the first category are caused by business scenario changes (e.g., Business spike in Double 11 holiday or Black Friday), which will lead to workload changes (e.g., an increase of active session metric). (2) The performance issues belonging to the second category are caused by poor SQL statements (e.g., the large number of examined rows, multiple joined tables, unreasonable index, resource-intensive queries, etc.), which will lead to the instance resource bottleneck (i.e., cpu_usage, iops_usage, mem_usage, buffer_pool_usage). In this scenario, the active session will increase since more intermittent slow queries [9] will be pilled up. (3) The performance issues belonged to the third category are caused by lock-related problems: (i) Metadata locks that are produced mainly by Data Definition Language (DDL) statements (e.g., CREATE, ALTER, etc.), accompanied with the status of "Waiting for table metadata lock". In this scenario, as the entire database is locked, millions of affected queries will be pilled up, resulting in a significant increase of active session. (ii) Row locks accompany the spike of the rowlock metrics (e.g., Innodb_row_lock_waits and Innodb_row_lock_time in MySQL [24]). In this scenario, the conflicted queries are slowed down, increasing active session of the instance. Therefore, instead of using hundreds of performance metrics, we only focus on analyzing the anomalous phenomenon of active session metric, which can cover most performance problems.

Based on the above definitions, we model Pinpointing Root Cause SQLs as a ranking problem, which is defined below.

Definition II.5 (Pinpointing Root Cause SQLs)**.** Given an anomaly case $\mathcal{C} = (\mathcal{M}, \mathcal{Q}, a_s, a_e)$, we aim to find a ranked list (i.e., a subset of the total SQL templates) to store R-SQLs, where higher-ranking templates are more likely to be the root causes. In addition, we also aim to find another ranked list to store H-SQLs, where higher-ranking templates are more likely to be the direct causes of performance anomalies.

III. System Overview

Our system *PinSQL*, as shown in Fig. 2, automatically identifies and handles R-SQLs for performance anomalies in cloud databases. In PinSQL, *Data Collection And Anomaly*

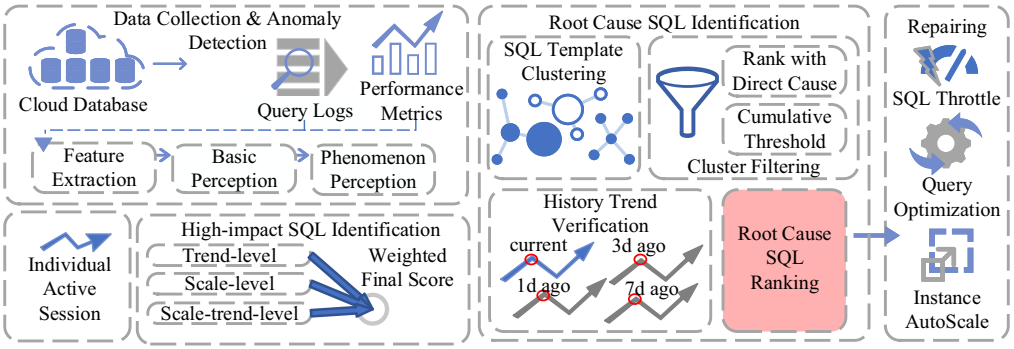


Fig. 2. Overview of PinSQL System

Detection Module firstly collects and pre-processes the streaming raw data (i.e., Performance Metrics data & Query Logs data) from millions of database instances in real-time. Note that, the pre-processed data will be stored persistently in our offline storage. During the pre-processing, the changes of performance metrics and the anomalous phenomenon will be detected as possible anomalies. The Data Collection And Anomaly Detection Module of PinSQL is executed in real time and uninterruptedly. The anomaly detection module evokes the root cause analysis modules when an anomaly is detected with the aggregated time-series data. Thus, the High-impact SQL Identification Module and Root Cause SQL Identification Module only execute when an anomaly is detected, and they are executed asynchronously. When an anomalous phenomenon is detected, following the anomaly propagation chain, *High-impact SQL Identification Module* is first triggered to locate H-SQLs, and then, *Root Cause SQL Identification Module* pinpoints R-SQLs through a clustering-based strategy. Finally, *Repairing Module* performs various autonomous actions to handle R-SQLs.

IV. Data Collection And Anomaly Detection Module

Data Collection And Anomaly Detection Module is divided into three components, as described below.

A. Data Collection & Pre-processing

In this component, we first deploy collectors into database instances to collect the streaming raw data, including Performance Metrics data [20] & Query Logs data. Performance Metrics are used for database instances, and Query Logs data are used for SQL queries. In terms of Query Logs data, all the information of each SQL query is collected. Specifically, the query information consists of the basic information (e.g., detail SQL statement, related tables), metric data (e.g., query response time t_{res}, the number of examined rows $\#examined\_rows$), and timestamp. The data is asynchronously stored into Alibaba Cloud LogStore [25] in real-time. As the data is asynchronously stored, it has little impact on database instances [26], [27]. PinSQL deletes the collected time-series data after a period to keep the log store's size within a specific limit. Specifically, the data will be invalidated after three days (or another user-customized expiration period).

After the Query Log is collected, Kafka and Flink provide stream data collection and aggregation to collect sufficient time-series data for anomaly detection algorithms. We apply Kafka to subscribe to the topics produced by those collectors simultaneously. Finally, Flink aggregates the streaming data according to different time granularities (i.e., 1 second and 1 minute), i.e., SQL queries are aggregated into SQL templates.

As massive SQL queries are aggregated into an SQL template, the amount of data will be greatly reduced, making it possible to save time-series data during a long period (e.g., 30 days) We can conveniently obtain SQL templates [16]–[19] by its definition, as discussed in Section II. Next, we detail how to aggregate the performance metrics for SQL templates. We hash the template via a unique SQL_ID and aggregate the query metric data (i.e., the response time t_{res} and the number of examined rows #examined_rows every second/minute) of all the queries that belong to this template. We use **sum**, **mean**, **count** functions to aggregate query metrics for each SQL template Q. Formally, we define the metric sequence of a SQL template as: $metric_{Q,t} = Aggregate(\{metric(q), \forall q \in Q \textbf{ where } t(q) \in [t, t + \triangle t)\})$, where $metric_{Q,t}$ denotes specific metric data for SQL template Q at time t, and $\triangle t \in \{1s, 1min\}$ is the time interval to determine the granularity of time series. We aggregate the data at 1-minute interval or 1-second interval depending on the demands.

In addition, we also utilize extra storage for real-time anomaly detection and efficient root cause analysis, including lower and upper bounds of performance metrics for real-time anomaly detection and individual active session for H-SQL identification. However, the amount of this data is far less compared with the size of detailed query logs. Overall, the storage space is sufficient and does not reach limitations.

B. Anomaly Detection

The Anomaly Detection component is responsible for detecting possible anomalies in the production. We integrate a variety of methods [9], [20], [28]–[30] and then build a real-time anomaly detection component. The Anomaly Detection component detects anomalies round-the-clock based on machine learning and fine-grained data monitoring. We deploy the anomaly detection system on the functional computing service of Alibaba Cloud [31].

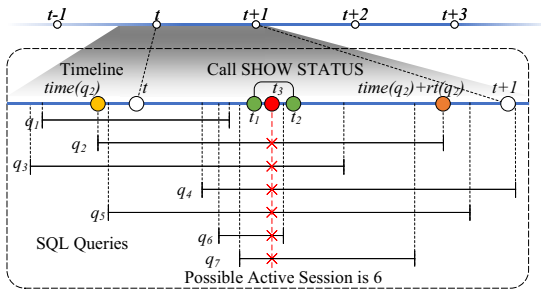


Fig. 3. Illustration of restriction in collecting Active Session value of $[t, t+1)$. There are 7 running queries $\{q_1, q_2, ..., q_7\}$ in current DB instance. Each query starts at $t(q_i)$ and ends at $t(q_i) + t_{res}(q_i)$. We assume that the collector calls SHOW STATUS in t_1 and gets response in t_2. The active session is collected by observing how many sessions are currently running at a time point t_3 between t_1 and t_2. At t_3, the collector will observe $q_2, q_3, q_4, q_5, q_6, q_7$ as active, so the current_active session is 6. In reality, t_3 is not known and could be any time point between $[t, t+1)$. Taking q_2 as an example, only when $t_3 \in [t, t(q_2) + t_{res}(q_2))$, q_2 will be observed as active.

The Anomaly Detection component consists of Basic Perception Layer and Phenomenon Perception Layer. After receiving the data from Data Collection & Preprocessingcomponent, the Basic Perception Layer can detect multiple anomalous features (e.g., spike up/down, levelshift, etc.). Then, the Phenomenon Perception Layer can be configured by the combination of anomalous features of different performance metrics to recognize different anomalous phenomena. It utilizes iSQUAD [9] to decide the type of the anomaly phenomenon. In the current implementation, we support recording and detecting anomalies via more than 40 performance metrics [32]. Users can customize which anomaly they concern the most from a variety of performance metric problems and which repairing action for a specific type of anomaly (to be discussed in Section VII). For example, the proliferation of Active Session can be configured as [active_session.spike], meaning if a spike of the active session metric is detected, it will be considered as an anomaly. The default configuration is to handle anomalies in three metrics, including active session, CPU usage, and IOPS usage (i.e., the percentage of the I/O per second w.r.t. the I/O capacity of instance). Once the configured anomaly is detected, Data Collection And Anomaly Detection Module will construct an anomaly case with the timestamp of the first detected anomaly and the current timestamp as the anomaly duration. Then it will trigger the next module. Users can also configure to ignore anomalies when their duration is less than a certain length of time. If multiple anomaly phenomena of the same type occur close in time (less than a configurable threshold). They will be merged into a longer anomaly.

C. Individual Active Session Estimation of Templates

Recall that the active session metric of a database instance (denoted as $session_t$) is the number of SQL queries being executed by the database instance at the current timestamp t. Similarly, we define *individual* active session of a template Q as the number of SQL queries of Q ($Q \in \mathcal{Q}$) that being executed at the current timestamp, denoted as $session_{Qt}$.

Individual active sessions can be precisely obtained by database built-in monitoring systems such as MySQL Performance Schema [33]. However, it will lead to performance overhead on the database instance. Our experimental results reveal that the performance degradation could be up to 30% (to be discussed in Section VIII-F). In addition, it is unnecessary to compute the statistics data inside the database instance constantly. As a result, we need a more lightweight alternative to estimate each active session. To find the lightweight alternative for estimating each active session, we turn our attention to query logs (as described in Section IV), since the response time of SQL queries is highly correlated with $session_t$ [20].

To be more specific, each SQL query q_i is active during [time(q_i) + $rt(q_i)$), where time(q_i) is the start time of q_i, and $rt(q_i)$ is the corresponding response time. Hence, it is very easy to estimate the active session (e.g., counting the number of active SQL queries) at each time. For example, the active session is 6 at t_1 but is 5 at t_2. Note that, we use SHOW STATUS statement to obtain $session_t$ (i.e., the active session) as the ground truth. However, as shown in Fig. 3, we do not know the accurate time (i.e., t_3) when SHOW STATUS executes. This means that $session_t$ could be obtained at any time point between $[t, t+1)$. In order to estimate t_3, we split 1 second into several buckets and we find the bucket where active session obtained by SHOW STATUS equals to the estimation using the response time. After obtaining the time (i.e., the bucket) when SHOW STATUS executes, we can estimate the current active session of each SQL template. For example, assuming that one SQL template contains q_1 and q_2, the active session of this template is 1 at t_3 when SHOW STATUS executes.

In the following, we provide more details about how to estimate $session_t$ according to query logs (i.e., the start time and the response time). For a time period $p = [t_s, t_e)$, the probability of a query q is observed to be active is as: $P(observed(p, q)) = \frac{|p \cap [t(q), t(q)+t_{res}(q))|}{|p|}$. As each active query contributes 1 to $session_t$, we calculate the expectation of $session_t$ in current time period $p = [t, t+1)$ as $\sum_{Q \in \mathcal{Q}} \sum_{q \in Q} P(observed(p, q))$. Our purpose is to make the estimated sum of active session close to the recorded value from the database instance. To reach a more accurate estimation of $session_t$, we split one second $[t, t+1)$ into K buckets $b_i = [t + \frac{i}{K}, t + \frac{i+1}{K})$ $(1 \le i \le K)$, and then calculate the expected active session for each bucket b_i as $\mathbb{E}[session_{b_i}|\mathcal{Q})] = \sum_{Q \in \mathcal{Q}} \sum_{q \in Q} P(observed(b_i, q))$. For every second, we select one bucket $b_i \subset [t, t+1)$ that the estimated active session is the closest to the observed value, indicating the value is most likely observed in the bucket b_i. Formally, $sel_t = \underset{b_i \in \{b_1, ..., b_K\}}{\arg\min} |session_t - \mathbb{E}[session_{b_i}|\mathcal{Q}]|$. Through splitting buckets, we can locate in which bucket the statement collects data. Take Fig. 3 as an example. Most of the inaccuracy comes from the fact that t_3 is randomly distributed. We can determine whether t_3 is in one bucket in this approach by comparing the estimated with ground truth. By limiting t_3 into one specific bucket, we can im-

prove the accuracy of the estimation by only considering one bucket. Thus, we can obtain the individual active session of each template using queries that take place during the bucket. Formally, we can estimate for each SQL template Q, the individual active session every second as: $session_Q = \{\sum_{q \in Q} P(observed(sel_t, q)), t \in \{t_s, t_s + 1, ..., t_e\}\}$.

Note that, in some cases, SHOW STATUS cannot finish in one second, i.e., $session\_t$ will be obtained at a time point outside [t, t + 1). If such situation emerges, we can extend our algorithm by splitting N ($N > 1$) seconds [t, t + N) into multiple bucket for each second t. However, it rarely happens, which can be ignored. Besides, if SHOW STATUS cannot finish in one second, it is most likely caused by an anomaly, which could be detected and recorded to respond before SHOW STATUS fails. Hence, in our paper, we assume that $session\_t$ is obtained at any time point between [t, t + 1). Although this assumption is not true, it will rarely affect the performance of PinSQL.

Discussions. Compared with DB built-in monitoring systems, this estimation is more general and is not limited to a specific DB engine version. Moreover, our method is more lightweight, contributed by two reasons below. First, we can deploy this calculation outside the database instance while DB built-in monitoring systems cannot. The additional performance consumption of the user database instance is only contributed by collecting the query log data asynchronously, which is neglectable [26], [27]. Second, the estimating process is triggered only when an anomaly is detected, reducing unnecessary performance overhead.

V. High-impact SQL Identification Module

In this section, we introduce the High-impact SQL Identification Module of PinSQL. The High-impact SQL Identification Module utilizes the estimated individual active session of each SQL template as a measuring metric and then ranks H-SQLs based on a weighted score to measure the impact of SQL templates on the active session. After obtaining the $session_{Qt}$ of each template Q, we continue to determine whether each template is a H-SQL. That is to say, we further calculate the possibility of active session anomaly caused by these templates. Specifically, whether a piece of SQL causing active session anomaly is determined by fusing the trend-level, scale-level, and scale-trend-level scores. We first briefly introduce the correlation coefficient, then we describe each level score. Finally, we describe how to fuse these scores.

Correlation Coefficient. In this paper, we use Pearson correlation coefficient [34], a common practice in time-series analysis to measure the correlation of two time-series X and Y: $corr(X, Y) = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\mathbb{E}[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}$.

Trend-level. For a template Q, if the trend of $session_{Qt}$ does not match $session_t$, then Q is not the SQL template that leads to a high active session. Trend-level score is used to help filtering uncorrelated templates. In anomaly cases, we should mainly focus on when the anomaly occurs. The trend-level score calculates the weighted Pearson coefficient between $session_{Qt}$ and $session_t$, i.e., $trend(Q) = corr(session_{Qt}, session_t; W)$ calculates the weighted covariance: $\mathrm{cov}(X, Y; W) = \frac{\sum_i w_i \cdot (x_i - \mathrm{m}(X;W))(y_i - \mathrm{m}(Y;W))}{\sum_i w_i}$. where $W \in [0,1]^N$ is a weight and $\mathrm{m}(X; W) = \frac{\sum_i w_i x_i}{\sum_i w_i}$ is the weighted mean. We want to emphasize the time-series data of the anomaly period, but the data near the anomaly period is also informative. Therefore, a smooth weight function is applicable, which makes the weight gently grows before the anomaly period, and retains a high value during the whole anomaly period, finally gently descents after the anomaly. We construct the weight through a Sigmoid-based function to highlight the anomaly period: $W_t = \sigma(\frac{t-a_s}{k_s}) + \sigma(\frac{a_e-t}{k_s}) - 1, t \in [t_s, t_e)$, where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the Sigmoid function and $k_s \in (0, +\infty)$ is a smooth factor. For this Sigmoid-based function, we have:

$$\lim_{k_s \to 0} W_t = \begin{cases} 0 & \text{if } t \notin [a_s, a_e) \\ 1 & \text{otherwise} \end{cases} \text{ and } \lim_{k_s \to \infty} W_t = 1. \quad (1)$$

If k_s approaches 0, the weighted correlation is equivalent to calculating correlation only on the anomaly period. If k_s approaches $+\infty$, it is equal to the naive Pearson correlation. We can adjust to what extend the anomaly period is emphasized by altering the smooth factor.

Scale-level. Scale-level score determines the impact of the template on $session_t$. For example, some affected templates may well correlate with $session_t$, while having executed little times. Such templates can hardly make the $session_t$ fluctuate significantly. The total active session of templates over the anomaly time period can well present their scale. To obtain a uniform value of scale that is consistent with the Pearson coefficient, we use min-max normalization to normalize the result into $[-1, 1]$. We define the scale-level score as $scale(Q) = 2 \cdot minmax_{Q \in \mathcal{Q}}(\sum_{t \in [a_s, a_e)} session_{Qt}) - 1$.

Scale-trend-level. Scale-trend-level score targets at SQLs that have high impacts on the active session when the anomaly phenomenon occurs, while having low impacts on the active session during other periods. For a SQL template Q, the scale-trend-level score is computed as the ratio of its active session over the total active session: $scale\_trend(Q) = corr(\frac{session_{Qt}}{session_t}, session_t)$. We calculate the correlation between $\frac{session_{Qt}}{session_t}$ and $session_t$, in order to make SQL templates with large scale-trend-level scores having high impacts on active session during the anomaly period.

Weighted Final Score. The above three score functions all have the range of $\mathbb{R}^N \to [-1, 1]$. Hence, we can fuse the scores by the weighted sum to measure how a template Q impacts the active session metric. Here, we use the weighted sum, which is simple but effective to control the weights of three different levels. For example, some templates may have a large stable traffic flow, which is also stable during the anomaly period, resulting in high scale-level scores. The trend-level score should filter these templates. Therefore, a large weight of trend-level score is required. We define $impact(Q) = \beta \cdot trend(Q) + scale\_trend(Q) + \alpha \cdot scale(Q)$, where $\alpha = corr(session_{Q_{max}t}, session_t)$, $Q_{max} = \arg\max_{Q \in \mathcal{Q}} scale(Q)$ and $\beta = -\alpha$ are weights to adjust the scale and trend

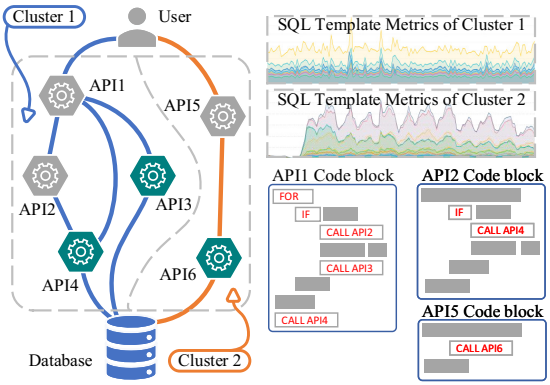


Fig. 4. An example of a microservice architecture accessing the cloud database in a complex business scenario. API1, API2, and API5 are Web back-end applications. API3, API4, and API6 encapsulate the interfaces for database access.

respectively. These two weights detect whether the largest Q by $session_{Qt}$ have the greatest impact on the active session, and adjust the proportion of Scale-level and Trend-level score accordingly. Finally, the templates with a high impact score denote the direct cause of anomaly phenomena.

VI. Root Cause SQL Identification Module

In this section we introduce the Root Cause SQL Identification Module of PinSQL. From the production practices, we use two key observations to derive whether a template is R-SQL: (i) this template needs to affect a H-SQL, or that it is H-SQL; (ii) this template is usually a new SQL or SQL that has different #execution trend from its history. Based on these two findings, we develop a cluster-based method to locate R-SQLs. To identify R-SQLs, we firstly cluster SQL templates according to their trend of #executed SQLs, based on the observation that SQLs with similar trends usually have similar business logic. Next, we filter the template clusters according to H-SQLs ranking. Finally, we verify whether the template is R-SQL with the historical trend of the templates.

SQL Template Clustering. Following the anomaly propagation chain, after locating H-SQLs, we need to find R-SQLs that may affect the H-SQLs. Specifically, for each template in H-SQLs, we need to find its corresponding R-SQLs. An intuitive method is to classify templates into different business logic because SQLs in the same business logic are most likely to affect each other. For example, two templates under the same business logic are more likely to access the same table, causing lock waits. The best way to distinguish different businesses is to analyze the App ID and code corresponding to the business logic. Unfortunately, this information belongs to user privacy for cloud databases and is not accessible. However, we have an alternative approach: cluster the templates with trends of #execution. This benefits from popular modern design patterns in back-end applications, as described below.

Modern implementation of business logic follows the microservice architecture [35], which helps back-end programmers reduce code coupling. Multiple microservices will call each other to form a DAG with a call relationship in a user's request. Therefore, the number of calls to all APIs in this DAG will maintain a relatively consistent trend. Take Fig. 4 as an example, in this scenario, the request trend of SQL statements between API3 and API4 is roughly similar, and it is unlikely that they and API6 have an obvious relationship. Therefore, we can use trends to classify SQL requests from API 1 to 4 into one category and to classify SQL requests from API 5 to 6 into another category. The #execution metric of templates is shown in the right half of the figure. The template time-series trend in each business cluster is relatively similar. Moreover, the back-end architecture designers often need to ensure the idempotence [36] of the interface, which means one business process will be accompanied by multiple templates being serial or parallel. The trend of #execution correlates naturally in these SQLs. This relationship is contributed by both the business logic and the back-end system architecture. We can cluster SQL templates by #execution.

Clustering with Trend of #execution. To cluster massive SQL data through these trend characteristics of SQL queries, we first calculate the pairwise Pearson correlation coefficient of time series data for all templates. Then we can use a threshold τ to get an adjacency matrix. We also add performance metrics to serve as auxiliary information for clustering SQLs. They are added as temporary nodes to form a denser graph, providing more accurate clusters. The edges of these temporary nodes are constructed similarly to SQL template nodes. These operations are general and could be applied to any other metric or logs by simply converting them to time series. Formally, $adj_{X,Y} = corr(metric(X), metric(Y)) > \tau$ $(\forall(X,Y) \in (\mathcal{Q} \cup \mathcal{M}) \times (\mathcal{Q} \cup \mathcal{M}))$, where $metric(X)$ indicates corresponding time-series data of X (e.g., $\#\text{execution}_X$ if X is a SQL template). We calculate the connected components of adj as the clustering result (denoted as D). Note that these temporary nodes will be filtered from the cluster results since they are not SQL templates.

Ranking Clusters for Filtering. After that, we have obtained clusters of multiple templates. We continue to find clusters that may contain R-SQLs. The templates of the same cluster belong to the same business and are more likely to have an influence relationship. We assume that if H-SQLs exists in a cluster, then it is very likely that R-SQLs is also in this cluster. By ranking the impact, we can already obtain the ranking of possible H-SQLs. We then use the largest impact value of all templates in a cluster to sort all clusters. For a cluster $c \in D$, we define the impact of the cluster as $impact(c) = \max_{Q \in c} impact(Q)$.

Cumulative Threshold. We can keep top-k clusters based on the impact of clusters. Nevertheless, we have to make sure to choose as many R-SQLs as possible while narrowing the range of choices. Because the instance $session$ anomaly may be caused by multiple H-SQLs with different trends, these H-SQLs with different trends will also be affected by different R-SQLs. Those templates are divided into different clusters. Directly selecting the cluster with the top-k impact or filtering based on a threshold of impact cannot cover these situations. In response to these situations, we designed an approach with a *cumulative threshold*. Specifically, we iteratively calculate a sum of $session_Q$ for every Q in tem-

plates selected and calculate the correlation with the active session until the threshold is reached. More specifically, we first sort the clustering result D by $impact(c)$ $c \in D$ in descending order, and then iterate K_c times, where K_c is a hyperparameter. In each iteration i, we sum up all the template sessions $S_i = \sum_{j \in [1,i]} \sum_{Q \in D_i} session_Q$, and check whether the correlation score $corr(S_i, session)$ is larger than or equals to threshold τ_c. Once the threshold is reached, we terminate the iteration and select templates of top-i clusters in D.

History Trend Verification After the above steps, we have got the possible R-SQLs set. The last stage is to use historical data to verify whether they are R-SQLs or not. For a performance anomaly, the number of executions of its R-SQLs will definitely have a sudden increase because it is difficult for stable traffic to cause anomalies. Such a sudden increase may not be noticeable, as the instance is flooded with queries in the anomaly period. For example, a newly appeared template may have a relatively low number of executions. Such templates may be covered by other templates with apparent growth in the trend. As a result, we need to look back at the history of each template to see whether the trend is actually increased. For a template that may be R-SQL, we use historical data to confirm whether it is R-SQL. We record the metric time-series of the template in $N_d \in \{1, 3, 7\}$ days ago compared with $[t_s, t_e)$ in each case. We perform anomaly detection over the four time-series data and only keep templates that satisfy these two rules: (i) Anomaly detected during the anomaly period. (ii) No anomaly was detected during the relative anomaly period of N_d days ago. This ensures that the number of executions of this template during the anomaly period has abruptly increased and has not been dramatically increased in the last few days. We apply Tukey's rule [37] to detect anomalies efficiently.

Root Cause SQLs Ranking. To obtain the final ranking of R-SQLs, we rank the remaining templates using the correlation between templates' number of execution and $session$.

VII. Repairing Module

To further improve the practical values of PinSQL, we develop a Repairing Module that recommend problem-solving actions on R-SQLs. We fuse multiple autonomous actions on R-SQLs based on different situations and user demands. Users can enable the automatic execution of suggested actions. If it is not enabled, these actions will not be executed. The main functionality of PinSQL is to pinpoint the problematic SQLs for the downstream repairing algorithms. Thus, we leave the space for user customization with a rule-based configuration. We also do not consider the dependencies of SQLs and only apply actions on the detected SQL templates. The repairing actions are treated as black boxes and could be separately executed on both SQL queries and the instance. In our current implementation, we provide three actions, including SQL Throttling, Query Optimization, and AutoScale, while other actions can be easily integrated into PinSQL. The default configuration is first to utilize SQL Throttling and then Query Optimization, while users can switch on/off actions for different types of anomalies (shown in Fig. 5). The provided actions are described below.

```
{
    "event_name": "SQLAvgExaminedRowsSuddenIncrease",
    "expr": "(mysql.cpu_usage.feature in [\"spike\",\"shift_up\"]
    or mysql.active_session.feature in [\"spike\", \"shift_up\"])
    and r_SQLs.examinedRows.feature in [\"spike\", \"shift_up\"]",
    "anomaly_status": "Warning",
    "suggestion": "SQL Optimization"
}
```

Fig. 5. Example configuration code of problem-solving actions on R-SQLs. This example shows how to configure query optimization operation on R-SQLs that #examined_rows sudden increase, when an anomaly of CPU usage metric is detected. Users can define the anomaly status to receive notifications via DingTalk or SMS. The anomaly types come from the Phenomenon Perception Layer, and the algorithm is adapted again for detecting the anomaly phenomenon of SQL template metrics.

SQL Throttling. When PinSQL detects R-SQLs, it will suggest throttling R-SQLs. Specifically, the PinSQL throttles R-SQLs by applying rate-limiting thresholds to these R-SQLs, where the rate-limiting threshold is configurable. Users can also customize the time duration of the throttling, the metric thresholds to throttle R-SQLs, and whether to kill R-SQLs.

Query Optimization. PinSQL automatically reports R-SQLs to query optimizer. By default, it is configured to execute only when the anomaly phenomenon detected by Phenomenon Perception Layer that is related to CPU/IO usage (e.g. [cpu_usage.spike]). Our optimizer combines a series of optimization techniques following previous studies, including automatic indexing [38], SQL query rewrite [39], and so on. We suggest DBAs to optimize the database as the optimizer suggests for R-SQLs.

Instance AutoScale. In many cases, increased SQL traffic is a phenomenon known in advance by the business department, where we should not apply throttling. If the business department has such a demand, we recommend that DBAs turn on AutoScale. We build the AutoScale action similar to AWS Auto Scaling [40]. It can automatically upgrade the performance configuration of the instance, such as adding read-only nodes or expanding the number of CPU cores.

VIII. Experimental Evaluation

In this section, we detail the experimental setting and evaluate the performance of our proposed system.

A. Experimental Setup

Evaluated Dataset. For evaluating the performance of PinSQL in real-world scenarios, we randomly sample a set of real-life anomaly cases collected from the internal databases of Alibaba online services, termed as ADAC. These services have two characteristics, i.e. (i) *high database loads*: contributed by the success of Alibaba's business in China and the world, millions of users around the world have been using these services, resulting in high database loads; and (ii) *complex business logic*: different groups develop diversified services in different application areas, such as online retail, corporate collaboration, and logistics, resulting in complex business logic. More specifically, ADAC contains 168 anomaly cases. These cases come from 36 unique DB instances with 15.9 cores and 87.9GiB Memory on average. ADAC records anomaly

time series for 1,653 minutes, during which 9.4 billion queries are executed. The queries are aggregated into 77,450 unique templates, with each anomaly case containing 3,357 templates on average. In ADAC, the ground truth of both R-SQLs and H-SQLs are two sets of templates, indicating all possible root causes and direct causes of anomalies. They are manually labeled by DBAs. We would like to emphasize that labeling R-SQLs in anomaly cases is non-trivial. It requires expert knowledge of the database to compare each SQL template's metric data and understand the underlying business logic of the corresponding database instances.

Evaluation metrics. We use Hits@k (H@k), $k \in \{1, 5\}$, Mean Reciprocal Rank (MRR), and running time (Time) as the evaluation metric for identifying R-SQLs and H-SQLs. Here, H@k denotes the proportion of correctly found templates in the top-k-ranks. MRR represents the average of the reciprocal ranks of the correctly found template, where reciprocal rank reports the mean rank of the correctly found template derived from all the templates (denoted as $|\mathcal{Q}|$). Formally, $\mathrm{MRR} = \frac{1}{|\mathcal{Q}|}\sum_{i=1}^{|\mathcal{Q}|}\frac{1}{\mathrm{rank}_i}$. The correctly found template is considered the first in the rank list that appears in the annotated set.

Competitors. We compare PinSQL with the following four competitors, each of which is derived from Top SQLs, a common practice adopted by various products. All the competitors output a rank of anomalous SQLs detected. We evaluate these methods by separately comparing the ground truth of R-SQLs and H-SQLs with the ranks produced by them. The competitors include (i) *Top SQLs of #execution* (Top-EN): The execution number is a vital metric for SQLs. The sudden increase of #execution may indicate a sudden change in business logic; (ii) *Top SQLs of total response time* (Top-RT): It is consistent with ranking the *average active session* [21] metric, the most important metric to present the instance load [23], [41] of each template during the anomaly period provided by Performance Insights of Cloud vendors; (iii) *Top SQLs of #examined_rows* (Top-ER): This metric is highly correlated with CPU usage and may indicate possible CPU anomaly; and (iv) *Top SQLs of all metrics* (Top-All): It tries to simulate the process that DBAs manually sort multiple metrics and then find out the most likely root cause from top pages. We use *Top-All* to denote the best results of the variants of Top SQLs (including Top-EN, Top-RT, and Top-ER).

Recall that *Autoregressive* methods and *classification-based* ones also can solve the root cause analysis problem. Nevertheless, in the current implementation, we skip the evaluation of the comparison between the proposed PinSQL and the above methods. The reasons why we neglect the comparison are as follows. We conduct experiments for several Autoregressive methods, i.e., cMLP [13], cLSTM [13], and SCGL [15]. Concretely, cMLP and cLSTM use neural network models, MLP and LSTM, respectively, to calculate the Granger causality. SCGL [15] uses deep convolution networks and low-rank approximation. However, they all face the problem of huge dependency function space and gradient explosion during the training process (e.g., as the scale of the SQL templates is large), thus failing to give any reasonable result. For the classification-based methods, since they are difficult to extend to the problem of finding R-SQLs, it is not suitable for PinSQL to compare these methods.

Implementation Details. All of the parameters are set to their default values. We set $\delta t_{start} = 30$ min to obtain sufficient data for diagnosing. For highlighting the anomaly period, we set the smooth factor $k_s = 30$. For clustering, we set the threshold $\tau = 0.8$. For ranking and filtering clusters, we set the threshold for the cluster number $K_c = 5$ and the cumulative threshold $\tau_c = 0.95$. Our study was implemented in Python 3.6 and tested on an Alibaba Cloud ECS server with an Intel Xeon E5-2682 CPU and 96GB memory.

B. Main Results

First, we evaluate the performance of PinSQL compared against four competitors in terms of both *effectiveness* and *efficiency*. Table I summarizes H@1, H@5, MRR, and Time results of PinSQL and its competitors.

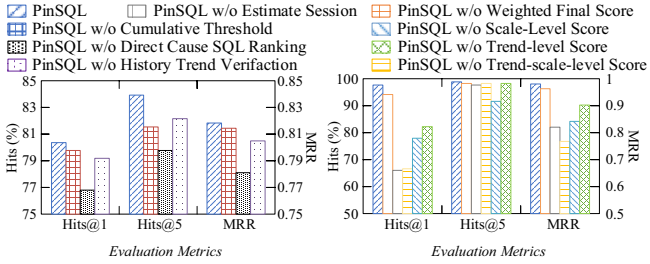
Effectiveness Evaluation. It is observed that PinSQL outperforms other competitors when identifying both R-SQLs and H-SQLs. Specifically, when identifying R-SQLs, PinSQL brings about 47% improvement in H@1 over the best baseline. This is because H-SQLs are different from the R-SQLs in many anomaly cases. In large-scale and complex business scenarios, most of the R-SQLs are not ranked top by the metrics. However, a latent relationship still exist between R-SQLs and the anomaly phenomenon through H-SQLs. Thus, we can find the root cause by looking for such a relationship. PinSQL is able to find the relationship by tracing back the anomaly propagation chain with consideration of business logic and SQL execution trends, resulting in significantly better results. When identifying H-SQLs, PinSQL gains $\sim$32% improvement in H@1 over the best baseline (i.e., Top-All). It is attributed to the following two reasons. First, PinSQL computes the individual active session, which enables possible H-SQLs to correlate with the instance $session$ directly. Computing individual active session of H-SQLs achieves better performance on correlation with H-SQLs than of solely using aggregated response time, as verified in Section VIII-F.

Second, PinSQL fuses multi-level information including trend-level information, scale-level information and scale-trend-level information of SQL templates. This further enhances the accuracy of PinSQL for identifying H-SQLs. In addition, Top-RT achieves the best performance among all the evaluated competitors that identify H-SQLs with a single metric. This demonstrates that the active_session metric of instance is highly correlated with SQL response time.

Efficiency Evaluation. We now turn our attention to the efficiency evaluation. As shown in Table I, the time consumption of Top-SQL-based methods is negligible. This is because Top SQL can get aggregated values from the stream processed data provided by Data Collection And Anomaly Detection Module. It only needs to sort an array that may contain several thousand elements. The time consumption of our method is 14.94s on average. It consists of the following parts: estimating the individual active sessions (8.01s), ranking the possible H-SQLs (0.47s), clustering and filtering (1.01s), and finally,

TABLE I
OVERALL RESULTS OF IDENTIFYING R-SQLS AND H-SQLS OF PINSQL AND ITS COMPETITORS (H@k ARE IN PERCENTAGE).

Method	R-SQLs				H-SQLs			
	H@1	H@5	MRR	Time	H@1	H@5	MRR	Time
Top-RT	31.0	56.0	0.40	2.73ms	64.3	97.0	0.75	2.73ms
Top-ER	13.7	47.6	0.28	2.27ms	44.0	67.3	0.52	2.27ms
Top-EN	6.5	6.5	0.08	2.26ms	3.0	10.7	0.08	2.26ms
Top-All	33.3	56.0	0.42	-	66.1	97.0	0.76	-
PinSQL	80.4	83.9	0.82	14.94s	97.6	98.8	0.98	8.48s



(a) R-SQLs (b) H-SQLs

Fig. 6. The result of ablation on identifying both R-SQLs and H-SQLs.

history trend verification (5.45s). Although the running time of PinSQL is slower than that of competitors, the running time is still significantly lower than the averaged anomaly time (∼540 seconds). Thus, it is able to diagnose R-SQLs in real-time.

C. Ablation Study

Second, we conduct ablation studies for identifying both R-SQLs and H-SQLs, with results plotted in Fig. 6.

Ablation on Identifying R-SQLs. By replacing the *Cumulative threshold* component with a fixed Top-1 cluster, the performance of PinSQL drops in H@1 (PinSQL vs. PinSQL w/o Cumulative threshold). This verifies that considering the combined effect of SQLs from multiple different businesses enables PinSQL to identify sufficient R-SQLs for anomaly cases. Furthermore, if R-SQLs are clustered separately with corresponding H-SQLs, our system with the *Cumulative threshold* can also be regarded as a remedy for the clustering algorithm. By replacing H-SQLs with Top-RT (the best baseline on identifying H-SQLs) for ranking the clustering results, the results drop by 3.6% in H@1 (PinSQL vs. PinSQL w/o Direct Cause SQL Ranking). It verifies that correctly identifying H-SQLs following the anomaly propagation chain is essential for identifying R-SQLs. By removing the *History Trend Verification*, the results drop by 1.2% in H@1 (PinSQL vs. PinSQL w/o History Verification). We have already ranked the filtered result by the correlation coefficient of #executions with instance active session.

Ablation on Identifying H-SQLs. By replacing variable parameters α and β with a constant value 1, the performance of PinSQL drops by 3.6% in H@1 (PinSQL vs. w/o PinSQL Weighted Final Score). This shows that considering the effect of templates with huge stable traffic flows is important for identifying H-SQLs. By replacing the individual active session with aggregated response time metric, the results drop by 31.5% in H@1 (PinSQL vs. PinSQL w/o Estimate Session). It verifies that the estimated value of the active session has

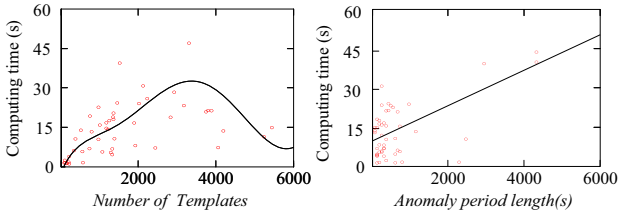


Fig. 7. The red dots indicate the computing time of PinSQL on anomaly cases. The black curve is obtained by using polynomial curve fitting algorithm to show the relationship between the number of SQL templates (or length of the anomaly) and the computing time of PinSQL. Note that, we randomly keep 50 red dots for a clear view of the results.

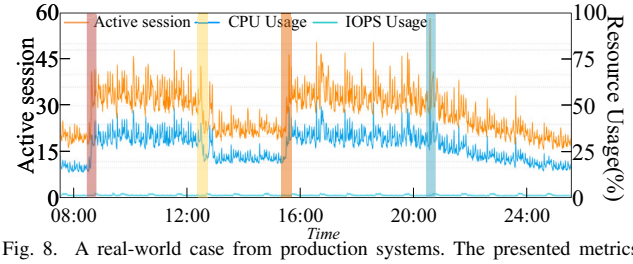


Fig. 8. A real-world case from production systems. The presented metrics include CPU usage, IOPS usage, and active session.

a great impact on the accuracy of correlation. By separately removing each level infusing (including PinSQL vs. PinSQL w/o Scale level, PinSQL vs. PinSQL w/o Trend level, and PinSQL vs. PinSQL w/o Trend-scale level), H@1 drops significantly in every case. This verifies that each level is necessary for locating H-SQLs. Among these levels, we also observe that, by removing each component, H@5 performs stably in general. The reason is that large-scale templates usually have a great impact on the active session.

D. Scalability Analysis

Next, we provide a scalability analysis of PinSQL. Specifically, we report the computing time of PinSQL by varying the number of SQL templates and length of the anomaly period, as shown in Fig. 7. The first observation is that the running time of slowest cases (i.e., the anomaly takes place longer than an hour) does not exceed 1 minute, proving that the running time of PinSQL is efficient for diagnosing anomalies. The second observation is that the running time is positively correlated with the length of the anomaly period, while it does not show a clear relationship with the number of SQL templates. This is because the running time of PinSQL is more sensitive to anomaly period length. Another possible reason is that the collected anomaly cases are not sufficient.

E. Case Study: Repairing Module

Then, we present a case study on the Repairing Module to further demonstrate the superiority of PinSQL for handling anomalies. We first describe a real-world case taken from the production environment to discuss the following two actions that heavily rely on the output R-SQLs of the identification module, i.e., *SQL Throttling* and *Query Optimization* . Then discuss the long-term impact of the Query Optimization action.

Real-world case. In this case, all the actions are made by a user (i.e., a system maintainer of a downstream application

using Alibaba Cloud Database) under the instruction of our customer service. We demonstrate the key performance metrics of the user's instance, as depicted in Fig. 8, where the anomaly is resolved using PinSQL. In the first place, the user does not subscribe to the function of PinSQL. It is worth noticing that in the DAS product of Alibaba Cloud, pinpointing and resolving R-SQLs with PinSQL are charged functions, while other functionalities, such as warning of anomalies, actions on a specified SQL, and Top-SQL-based algorithms, are free. When an anomaly occurs (red rectangle), the user receives a notice from DAS and thinks it will soon disappear. However, the metrics do not recover after several hours. Then the user manually applies SQL throttling on Top-1 SQL, which has reduced the anomaly phenomenon (yellow rectangle). However, the throttling action makes running the specific queries slow, thus sabotaging the downstream experience. Thus, the user then switches off SQL throttling(orange rectangle), and the anomaly phenomenon reappears. After a few hours, the user enables PinSQL, and it starts to analyze the root cause (blue rectangle). PinSQL then identifies the R-SQLs. PinSQL follow the default setting to execute both SQL throttling and Query Optimization. It recommends a SQL optimization action, but does not recommend SQL throttling since the metrics do not reach the default threshold. The user then approves the optimization and executes the optimization action, and the performance metrics are gently back to normal. PinSQL does not recommend SQL throttling since the metrics do not reach the default threshold. The first observation is that when the user switches off SQL throttling on Top-SQLs, the anomaly phenomenon reappears. This is because throttling Top-SQL does not solve the anomaly fundamentally. However, taking proper actions on R-SQLs can make the response time of the affected SQLs (that originally have slowed down due to R-SQLs) become normal, thereby fundamentally solving the anomaly case. Compared with Top SQLs, taking actions on R-SQLs is more practical in real-life scenarios. The second observation is that after throttling Top SQL, the values of metrics are still higher than their normal values. This is because multiple SQLs are usually affected by R-SQLs. SQLs with high total t_{res} rankings are among these affected SQLs. After throttling Top SQLs, other affected SQLs can still make the values metrics high. In addition, it is infeasible for us to perform SQL throttling on all of them. The third observation is that the slow SQL detector doesn't activate query optimization, which further verifies the superiority of PinSQL.

Long-term impact of Query Optimization. Since the Query Optimization action modifies the production environment, it would be necessary to further evaluate the long-term impact of this action on R-SQLs. We accumulated 141 Query optimization suggestions approved by DBAs in two months. We divide them into two categories, one is based on slow SQL detection as in previous studies [7], [9], [11], and the other category contains R-SQLs from PinSQL. Poor SQLs always occupy huge resources. After optimization, queries of these statements are expected to occupy fewer resources, resulting in smaller metrics. Therefore, whether a query optimization suggestion is useful is determined by how much the averaged metrics of SQL template have dropped (denoted as *gain* of optimization). We collect the averaged metric data, including t_{res} and #examined_rows of each query, corresponding to the SQL template 24 hours before and after optimization. We report the averaged gain based on the collected metrics data in Table II. As shown in the table, on average, the gain from R-SQLs can increase by 10% from optimizing slow SQLs to optimizing R-SQLs of performance anomalies. This is because slow SQLs may be due to a combination of reasons. Although slow SQL statements themselves may have room for optimization, they may also be slowed down by other SQLs [9], making less gain in query optimization. Finding R-SQLs can eliminate the possibility of slowing down due to the influence of other SQLs, thereby increasing the gain in optimizing statements.

TABLE II
STATISTICS OF AVERAGED GAINS ON EACH METRIC.

	#Optimized SQLs	t_{res} Gain	#examined_rows Gain
R-SQLs	85	92.44%	91.17%
Slow SQLs	56	82.59%	81.56%

TABLE III
CASE STUDY: ESTIMATED ACTIVE SESSION

	Pearson Correlation	MSE
Estimate By RT	0.54	324450.57
Estimate w/o buckets	0.92	8205.82
Estimate (K=10)	0.96	7617.44

F. Case Study: Individual Active Session

Last, we present a case study for the individual active session of SQL templates, a key metric for diagnosing performance anomalies [23]. The conventional approach uses built-in monitoring systems to obtain accurate individual active sessions. However, the database instances supporting large-scale cloud services come from online businesses. Changing the configuration of these databases will make their performance unstable, which may further cause severe damage to Alibaba's global business. As a result, it is unable to compare our individual active session calculation with Performance Schema directly. Recall that if each individual is correctly estimated, the sum of active sessions of all templates should be close to the active session of the database instance. As mentioned in Section IV-C, the proposed method can accurately estimate the individual active session of each SQL template according to the SQL query logs. Hence, we use the *estimated active session* for diagnosing performance anomalies alternatively. We utilize *Mean Squared Error* (MSE) and *Pearson correlation* to determine the error between the sum of estimated time-series and the real active session of the corresponding database instance. We conduct evaluations on three following methods: (i) *Estimate by RT*, which directly uses the response time metric of all SQL queries per second; (ii) *Estimate w/o buckets*, which utilizes our estimation method without splitting buckets; and (iii) *Estimate* $(K = 10)$, which employs our estimation method and splits each second into 10 buckets. As shown in Table III, estimating the active session of database instance via our approach brings about $1.7\times$ improvement in

TABLE IV
QPS AND QPS DECLINE RATE (DENOTED AS ↓QPS, IN PERCENTAGE) OF DIFFERENT CONFIGURATIONS.

Config	Read Only		Read Write		Write Only	
	QPS	↓QPS	QPS	↓QPS	QPS	↓QPS
normal	72,983	0.00	41,867	0.00	37,400	0.00
pfs	63,769	12.62	37,914	9.44	34,232	8.47
pfs+ins	65,485	10.27	34,477	17.65	34,406	8.01
pfs+con	64,964	10.99	34,740	17.02	33,305	10.95
pfs+con+ins	53,870	26.19	29,935	28.50	26,033	30.39

terms of correlation with the active session. In addition, by splitting each second into ten buckets, the accuracy of total active session estimation further improves. This is because it chooses the buckets by minimizing the error of individual active sessions. This estimation of the individual active session makes the impact of templates on instance active session becoming nearly linear, which guarantees high quality of followed up approaches in PinSQL.

Overhead of Built-in Monitoring Systems. We use MySQL built-in monitoring system (i.e., *Performance Schema*) to demonstrate the performance overhead of built-in monitoring systems. We design a stress test using a 4-core, 16GB cloud database instance with MySQL 8.0 kernel to do a 32-thread concurrent test. The database contains 20 tables with 10 million rows [42]. We record the QPS curve under the following configs: normal, pfs, pfs+con, pfs+ins, and pfs+con+ins, where normal indicates PERFORMANCE SCHEMA off, con indicates all consumers are turned on, and ins indicates all instrumentation is turned on. The QPS and QPS decline rate (QPS of current config vs. QPS of normal config) are recorded until the instance reaches its CPU bottleneck. We can observe that the performance of switching on pfs considerably drops from 8.01% to a maximum of 30.39%, varying on the configuration, as shown in Table IV. As a result, it is generally not recommended to open Performance Schema on large-scale cloud databases. Conversely, PinSQL is an external tool that analyzes logs outside the database instance, having little impact on the database performance.

IX. RELATED WORK

In this section, we review the work related to our problem.

Database Diagnostics Systems. many works have been focused on diagnosing DBMS, including tuning [43], optimizing [44] and workload management [45]. There are also self-driving DBMS with the functionality of forecasting workloads to optimize planning [46]. They have successfully improved the performance of DBMS by optimizing the system performance. However, these optimizing techniques focus on the overall system performance. In contrast, PinSQL aims to identify SQLs that cause or are highly related to performance anomalies, increasing the robustness of cloud DBMS.

Root Cause Analysis in Cloud Databases. Most recent studies on root cause analysis for cloud databases rely on classification algorithms. These classification-based approaches focus on classifying the root cause of anomalies by dividing the causes into a limited collection of types such as Poorly Written Query and *Workload Spike*. Specifically, DBSherlock utilizes ensemble causal models [6] to classify root causes. openGauss [7], [8] use LSTM to classify root causes. Furthermore, probabilistic graphical models are implemented in Explainit [47] for causal inference to analyze root causes; iSQUAD [9] focuses particularly on intermittent slow queries and adopts the Bayesian case model for classification. The ability of the above methods to locate root causes is limited in the number of distinct causes they divide. As a result, they can not locate specific R-SQLs in anomaly cases, which is essential in large-scale complex business scenarios.

Top SQLs. Many cloud vendors have provided database diagnosing products, such as AWS Performance Insight [2], Azure Intelligent Insight [10], Tencent DB Brain [11], and Huawei Database Admin Service [12]. The conventional approach for identifying R-SQLs is to sort the collected SQL metric data in various dimensions, allowing users to locate database problems themselves. However, it fails to locate root cause SQLs under complex business scenarios. As many complex queries exist in cloud databases, it is hard to tell which SQL template contributes more to the detected anomaly.

Autoregressive-based Methods. Finding the R-SQLs can also be seen as finding causality on multivariate time series data, which aims to find the relationships between performance metrics and SQL template metrics. Many Auto-regressive-based methods [13]–[15] have been proposed for causal inference on time series data. Specifically, Neural-GC [13] proposes multiple deep models, including MLP and recurrent neural networks based on Granger causality. Seq2graph [14] is an attention-based method for finding non-linear dependency. SCGL [15] utilizes ResNet and low-rank approximation for better model robustness and scalability in time and space. The above methods can identify linear and non-linear relationships between different variables. However, the function space for these methods to search is huge. Thus, these methods cannot locate R-SQLs by finding time-series dependency.

X. CONCLUSIONS

We present PinSQL, an autonomous diagnostic system with two main features (i.e., root cause analysis and repairing) for handling the problem of extracting and cracking root cause SQLs in cloud databases. Extensive experimental results demonstrate that PinSQL achieves an accuracy of 80% for its Top-1 returned potential cause (defined as Hits@1) on identifying R-SQLs. In the future, it is of interest to find specific queries (including system kernel bugs and host problems) that cause anomalies other than SQL templates and integrate more repairing actions into PinSQL. We also plan to utilize more performance metrics and explore the complex relationships among performance metrics when detecting R-SQLs.

XI. ACKNOWLEDGEMENTS

This work was supported by the NSFC under Grants No. (62025206, 61972338, and 62102351), Alibaba Group through Alibaba Innovation Research Program, and the Zhejiang Provincial Natural Science Foundation under Grant No. LR21F020005. Yunjun Gao is the corresponding author of the work.

REFERENCES

[1] Alibaba Cloud. Alibaba Cloud Databases. [Online]. Available: https://www.alibabacloud.com/product/databases

[2] Amazon EC. Amazon web services. [Online]. Available: http://aws.amazon.com/es/ec2/

[3] M. Copel, J. Soh, A. Puca, M. Manning, and D. Gollob, "Microsoft azure," *New York, NY, USA:: Apress*, 2015.

[4] S. P. T. Krishnan and J. L. U. Gonzalez, *Building your next big thing with google cloud platform: A guide for developers and enterprise architects*. Springer, 2015.

[5] Oracle. Oracle Performance: 80 Percent design + 20 percent hardware. [Online]. Available: https://logicalread.com/oracle-db-perf-80-percent-hw-dc01/#.YZUdBJOA5qs

[6] D. Y. Yoon, N. Niu, and B. Mozafari, "Dbsherlock: A performance diagnostic tool for transactional databases," in *SIGMOD*, 2016, pp. 1599–1614.

[7] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li, "opengauss: An autonomous database system," *PVLDB*, vol. 14, no. 12, pp. 3028–3041, 2021.

[8] X. Zhou, L. Jin, J. Sun, X. Zhao, X. Yu, S. Li, T. Wang, K. Li, and L. Liu, "Dbmind: A self-driving platform in opengauss," *PVLDB*, vol. 14, no. 12, pp. 2743–2746, 2021.

[9] M. Ma, Z. Yin, S. Zhang, S. Wang, C. Zheng, X. Jiang, H. Hu, C. Luo, Y. Li, N. Qiu, F. Li, C. Chen, and D. Pei, "Diagnosing root causes of intermittent slow queries in large-scale cloud databases," *PVLDB*, vol. 13, no. 8, pp. 1176–1189, 2020.

[10] Microsoft. Intelligent Insights using AI to monitor and troubleshoot database performance (preview). [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-sql/database/intelligent-insights-overview

[11] Tencent. Database autonomy service (das). [Online]. Available: https://cloud.tencent.com/product/dbbrain

[12] Huawei. Data admin service. [Online]. Available: https://www.huaweicloud.com/intl/en-us/product/das.html

[13] A. Tank, I. Covert, N. Foti, A. Shojaie, and E. Fox, "Neural granger causality," *arXiv preprint arXiv:1802.05842*, 2018.

[14] X. Dang, S. Y. Shah, and P. Zerfos, "seq2graph: Discovering dynamic non-linear dependencies from multivariate time series," in *IEEE BigData*, 2019, pp. 1774–1783.

[15] C. Xu, H. Huang, and S. Yoo, "Scalable causal graph learning through a deep neural network," in *CIKM*, 2019, pp. 1853–1862.

[16] MySQL 8.0 Reference Manual. Performance Schema Statement Digests and Sampling. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/performance-schema-statement-digests.html

[17] Q. T. Tran, K. Morfonios, and N. Polyzotis, "Oracle workload intelligence," in *SIGMOD*, 2015, pp. 1669–1681.

[18] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *SIGMOD*, 2018, pp. 631–645.

[19] Amazon Web Services. Amazon Aurora User Guide for Aurora. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-ug.pdf

[20] W. Cao, Y. Gao, B. Lin, X. Feng, Y. Xie, X. Lou, and P. Wang, "Tcprt: Instrument and diagnostic analysis system for service quality of cloud databases at massive scale in real-time," in *SIGMOD*, 2018, pp. 615–627.

[21] Oracle. Average Active Session. [Online]. Available: https://docs.oracle.com/cd/B16240_01/doc/doc.102/e16282/oracle_database_help/oracle_database_instance_throughput_avg_active_sessions.html

[22] Alibaba Cloud. Database autonomy service (das). [Online]. Available: https://www.alibabacloud.com/product/das

[23] Amazon Web Services. AWS Perfomance Insights FAQs. [Online]. Available: https://aws.amazon.com/rds/performance-insights/faqs/

[24] MySQL. Server status variables. [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en/server-status-variables.html

[25] W. Cao, X. Feng, B. Liang, T. Zhang, Y. Gao, Y. Zhang, and F. Li, "Logstore: A cloud-native and multi-tenant log database," in *SIGMOD*, 2021, pp. 2464–2476.

[26] B. Mozafari, C. Curino, A. Jindal, and S. Madden, "Performance and resource modeling in highly-concurrent oltp workloads," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 301–312.

[27] M. Theriault and W. Heney, *Oracle Security*. O'Reilly & Associates, Inc., 1998.

[28] A. N. Pettitt, "A non-parametric approach to the change-point problem," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 28, no. 2, pp. 126–135, 1979.

[29] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng, J. Chen, Z. Wang, and H. Qiao, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *WWW*, 2018, pp. 187–196.

[30] W. Chen, H. Xu, Z. Li, D. Pei, J. Chen, H. Qiao, Y. Feng, and Z. Wang, "Unsupervised anomaly detection for intricate kpis via adversarial training of VAE," in *INFOCOM*, 2019, pp. 1891–1899.

[31] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *USENIX*, 2021, pp. 443–457.

[32] Alibaba Cloud. Performance metrics of pinsql. [Online]. Available: https://www.alibabacloud.com/help/en/doc-detail/64901.htm

[33] Oracle. MySQL Performance Schema. [Online]. Available: https://dev.mysql.com/doc/mysql-perfschema-excerpt/8.0/en/performance-schema.html

[34] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[35] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[36] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[37] D. C. Hoaglin, B. Iglewicz, and J. W. Tukey, "Performance of some resistant rules for outlier labeling," *Journal of the American Statistical Association*, vol. 81, no. 396, pp. 991–999, 1986.

[38] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri, "Automatically indexing millions of databases in microsoft azure SQL database," in *SIGMOD*. ACM, 2019, pp. 666–679.

[39] Oracle. Basic query rewrite. [Online]. Available: https://docs.oracle.com/cd/B19306_01/server.102/b14223/qrbasic.htm

[40] Amazon Web Services. New aws auto scaling – unified scaling for your cloud applications. [Online]. Available: https://aws.amazon.com/blogs/aws/aws-auto-scaling-unified-scaling-for-your-cloud-applications/

[41] techgoeasy. What is DB time ,DB cpu Average Active sessions, Active session in oracle. [Online]. Available: https://techgoeasy.com/what-is-db-time-and-average-active/

[42] Alibaba Cloud. Best practices for setting the parameter performance_schema: Overview. [Online]. Available: https://www.alibabacloud.com/blog/598273

[43] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 1246–1257, 2009.

[44] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *Proc. VLDB Endow.*, vol. 4, no. 11, pp. 1111–1122, 2011.

[45] R. Marcus and O. Papaemmanouil, "Wisedb: A learning-based workload management advisor for cloud databases," *Proc. VLDB Endow.*, vol. 9, no. 10, pp. 780–791, 2016.

[46] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang, "Self-driving database management systems," in *CIDR*, 2017.

[47] V. Jeyakumar, O. Madani, A. Parandeh, A. Kulshreshtha, W. Zeng, and N. Yadav, "Explainit! - A declarative root-cause analysis engine for time series data," in *SIGMOD*, 2019, pp. 333–348.