DBdoctor: A Fine-grained and Non-intrusive Performance Diagnosis Platform for Databases

Xinyue Shi†, Quanqi Xin‡, Zhengjin Wang†, Xinyi Zhang†, Haoqiong Bian†, Wei Lu†
Qiyu Zhuang†, Shuang Liu†, Jikuan Zhang‡, Xiang Zheng‡, Yunpeng Chai†, Xiaoyong Du†
† *Renmin University of China* ‡ *Juhaokan Technology, Hisense*
†{xinyueshi, zhengjin.wang, xinyizhang.info, bianhq, lu-wei, qyzhuang, shuang.liu, ypchai, duyong}@ruc.edu.cn
‡{xinquanqi, zhangjikuan, zhengxiang2}@hisense.com

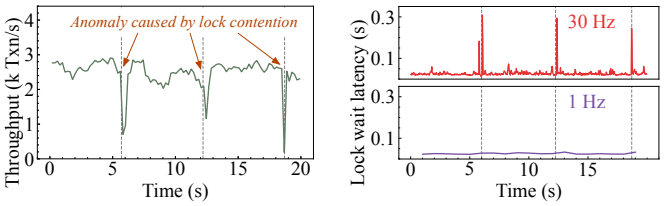
***Abstract*—Database performance anomalies are prevalent in large-scale deployments and often result in substantial business losses, making online diagnosis and resolution of anomalies indispensable in production environments. Existing anomaly diagnosis frameworks rely on sample-based external monitoring tools, which suffer from two critical limitations: (1) an unfavorable granularity–overhead trade-off, where low-frequency sampling misses transient anomalies while high-frequency sampling introduces significant overhead, and (2) black-box inference due to a lack of internal database context. To address these limitations, we propose** DBdoctor**, a non-intrusive performance diagnosis platform applicable across different database engines.** DBdoctor **proposes a novel event-based metric collection framework that leverages eBPF to collect fine-grained both external and internal database metrics with low overhead. These metrics are modeled as SQL temporal resource metrics and dependency graphs to enable white-box anomaly diagnosis, supporting precise root cause identification.** DBdoctor **has been deployed in large enterprises such as China Unicom and Hisense to manage thousands of database instances. Experimental evaluations using popular benchmarks and real-world workloads demonstrate that** DBdoctor **achieves higher diagnosis accuracy than existing approaches, with comparable or lower performance overhead.**

***Index Terms*—database monitoring, performance anomaly diagnosis, eBPF**

I. Introduction

Database performance anomalies are common and often inevitable in large-scale deployments [29], [42], arising from unpredictable workload patterns, resource contention, inefficient SQL statements, or hardware failures. Such anomalies can cause serious availability issues, including SLA violations and service outages, making effective anomaly diagnosis a critical capability for modern database systems [16].

In production environments, the diagnosis process is typically conducted online and involves three phases: (1) *Metric collection* that gathers predefined metrics related to performance anomalies, such as CPU utilization and lock wait time; (2) *Root cause identification* that analyzes the collected metrics to identified anomalies and their root causes (e.g., missing indexes), either manually by DBAs or automatically using machine learning or LLM-based methods; and (3) *Anomaly resolution* that suggests and applies targeted optimizations based on the identified causes to mitigate the performance anomalies. Of the three phases, metric collection provides inputs for the subsequent phases and fundamentally impacts the efficiency and accuracy of diagnosis. Existing monitoring tools, either general-purpose (e.g., Prometheus [12], perf [10]) or DBMS-native (e.g., pg_stat_statements [11] for PostgreSQL), typically adopt a *sample-based external* monitoring approach [10]–[12] where predefined metrics are collected at fixed intervals (e.g., every 1 s or 1 min). However, such a monitoring approach introduces two major limitations.



(a) TPS degradation anomalies (b) Sampled lock-wait metrics

Fig. 1: Anomalies and effects of various sampling frequencies.

L1. Unfavorable granularity–overhead trade-off. Existing monitoring approaches adopt a *sample-based monitoring paradigm*, periodically collecting system metrics (e.g., CPU utilization) from the predefined OS or database interfaces. While this approach is simple and general, its sampling frequency directly trades off accuracy against overhead. To avoid runtime interference, production systems typically adopt relatively low sampling frequencies (around 1 Hz or lower), which inevitably risks missing critical but short-lived root cause events. To illustrate, we use the YCSB [14] benchmark on MySQL to simulate a high-concurrency scenario and periodically inject a heavy range-scan query, simulating a user browsing request. As shown in Figure 1a, this query temporarily monopolizes locks and blocks other concurrent transactions for about 150 ms, causing transaction throughput drops. As shown in Figure 1b, 30 Hz monitoring successfully captures the lock-contention spikes but incurs considerable runtime interference, leading to an additional 8.72% throughput degradation. In contrast, 1 Hz monitoring introduces negligible overhead (only 3.02% throughput loss) but entirely misses the contention spikes. Prior studies show that even a 100 ms increase in web response latency can reduce sales by 1% [2] and decrease conversion rates by 7% [1]. Persistently overlooking such transient anomalies leaves many performance issues undiagnosed. In complex production environments, their effects can cascade across components, leading

to significant business losses. *This motivates the need for a fine-grained, low-overhead monitoring framework capable of capturing transient but critical root-cause events in real time.*

L2. Black-box inference due to lack of internal context. Existing diagnosis methods operate primarily on the system's external outputs and lack visibility into the internal execution mechanisms of databases. Some of them [18], [31] perform diagnosis at the *system level*, relying on coarse-grained metrics like CPU utilization or overall transaction throughput. Others [23], [38] attempt to diagnose at the *query level*, obtaining per-query summaries such as execution time or resource consumption through database-specific plugins or system tables, tightly coupled to particular database implementations. However, all existing methods provide only external and aggregated statistics, failing to capture a query's *full execution lifecycle* or its *interactions with other concurrent queries*. Both aspects are crucial for accurate root cause analysis: understanding the execution lifecycle reveals where performance bottlenecks arise in the query pipeline (e.g., identifying which actions of which queries contribute most to anomalous CPU spikes), whereas modeling inter-query dependencies uncovers how anomalies propagate through lock waits. The inability of existing methods to capture these aspects stems from the lack of *internal observability*—the capability to correlate higher-level SQL activities with low-level runtime events (e.g., resource consumption, lock operations). As a result, they can only observe system symptoms but cannot causally link them to query behaviors, thereby treating the database as a black box and failing to accurately localize root causes. *This motivates the need for a monitoring framework that provides internal observability and bridges query-level behaviors and function-level runtime events to enable white-box diagnosis.*

To address these limitations, we propose DBdoctor, a novel diagnosis platform for DBMS empowered by event-based, fine-grained, and internal metric collection with low overhead. Instead of relying on fixed sampling frequencies, DBdoctor attaches probes to specific database functions and triggers metric collection upon predefined events, ensuring lightweight and lossless monitoring. To achieve this, we adopt the eBPF technology, which allows predefined monitoring probes to be non-intrusively (without modifying database source code or restarting database instances) and dynamically attached to user or kernel functions, to collect database metrics efficiently.

Although eBPF enables non-intrusive and low-overhead metric collection by bypassing the user space and executing efficiently in the kernel, applying it for database monitoring presents two major challenges. First, balancing monitoring granularity and overhead. Capturing database events and transient anomalous metrics requires probing database functions and kernel functions to collect real-time messages. However, monitoring all functions in real time, even with eBPF, can incur excessive overhead. Second, transforming raw eBPF messages into SQL-level semantics efficiently. Metrics collected by eBPF are inherently function-level and lack database context, making it difficult to associate them with SQL queries. Under heavy workloads, the large volume of database events and resource consumption triggers massive eBPF message generation. Efficiently managing these kernel-space messages and constructing SQL-level metrics from the raw eBPF data is highly challenging. To overcome these challenges, DBdoctor introduces two core components:

(1) *Adaptive event-based metric collection framework.* We propose an event-based, fine-grained metric collection paradigm to resolve the granularity-overhead trade-off inherent in existing sampling-based approaches. Instead of periodically sampling external metrics, DBdoctor attaches eBPF probes to critical database and kernel functions, collecting fine-grained internal metrics per function invocation. To reduce metric collection overhead, we present an offline probe set identification approach to only inject probes into a high-impact set of functions necessary for effective anomaly diagnosis. Moreover, we design an online anomaly-aware probing mechanism that prioritizes the full capture of anomalous events while efficiently aggregating non-anomalous metric data of continuous events into a single message, ensuring both lossless capture of anomaly events and minimal runtime overhead.

(2) *White-box anomaly diagnosis component.* We provide internal database observability for anomaly diagnosis by modeling the collected metric messages into SQL temporal resource metrics and SQL dependency graphs. With both external and internal fine-grained metric data, DBdoctor performs white-box anomaly diagnosis, identifying precise root causes and performing effective resolutions.

We summarize our contributions as follows.

- We propose an adaptive event-based metric collection framework that leverages eBPF to collect fine-grained database metrics with low overhead. It is non-intrusive and does not depend on a specific DBMS implementation.
- We propose a white-box anomaly diagnosis approach that captures the function-level database internal context to construct SQL temporal resource metrics and dependency graphs, enabling precise diagnosis.
- We implement these approaches in a production diagnosis platform, namely DBdoctor, which has been deployed in over 500 enterprises, including China Unicom, BMW, and Hisense, to monitor and diagnose dozens to thousands of database instances, heterogeneous or not.
- We evaluate DBdoctor's diagnosis performance using popular benchmarks and real-world workloads in popular databases, including MySQL and PostgreSQL. The experiments prove that DBdoctor achieves higher root cause identification accuracy than existing approaches with comparable or lower performance overhead.

II. Preliminary

A. Database Performance Anomalies

A database performance anomaly is any unexpected deviation or degradation in a database system's key performance metrics, such as response time or throughput. These anomalies can disrupt applications, degrade user experience, and even lead to service outages. Here we list three representative

database performance anomalies. (1) Slow Query. A specific query or set of queries that executes significantly longer than normal. (2) Abnormal Resource Utilization. The database server's core resources, including CPU, memory, I/O, and network, show an abnormal spike or high usage. This often indicates a systemic bottleneck, even if individual queries are not particularly slow. (3) Database Unresponsive. The database system completely stops responding to new requests, often caused by deadlock storms or complete resource exhaustion.

Upon recognizing a performance anomaly, the primary tasks for a DBA are to perform root cause analysis and anomaly resolution. The causes of these issues vary, stemming from a single problem or a combination of multiple factors. Below are some of the most common root causes of database performance anomalies. (1) SQL and Application Issues, such as suboptimal SQL, overlarge transactions, or poor connection management. These issues often arise from a lack of developer awareness of database best practices. (2) Locking Issues, including lock contention and deadlocks, where multiple transactions compete for accessing the same data units. (3) Resource Bottlenecks, i.e., a lack of sufficient resources on one or more resource dimensions. These bottlenecks can manifest as an overloaded CPU, slow I/O operations, or a lack of available memory to handle the workload efficiently. (4) Configuration and Architectural Problems, such as misconfigured database parameters or poorly designed indexes, can create systemic weaknesses. These issues may not be immediately obvious, but can lead to progressive performance degradation.

B. eBPF Technology

eBPF (extended Berkeley Packet Filter) is a kernel-native technology that allows user-defined programs to be safely executed within the OS kernel without kernel modifications [40]. An eBPF program is typically written in a restricted C-like language, compiled into eBPF bytecode via LLVM or CLang, and loaded into the kernel via dedicated system calls (e.g., bpf() syscall). Before setting up for execution, the kernel performs static verification to ensure safety properties such as memory safety and bounded loops. Once verified, the bytecode is just-in-time (JIT) compiled into native instructions and attached to specific hook points, such as system calls, kernel functions (kprobes), user functions (uprobes), or tracepoints. The program is then triggered whenever the corresponding kernel or user-space event occurs, enabling fine-grained, event-driven monitoring and control of kernel and application behaviors, while ensuring system stability and security.

eBPF communication mechanism. eBPF provides several mechanisms for exposing kernel-collected information to user programs. (1) Perf Buffer. The perf buffer is a per-CPU circular buffer that enables efficient data exchange between eBPF programs and the user space. (2) Ring buffer. It is a multi-producer, single-consumer circular buffer enabling communications between kernel and user space, ensuring strict preservation of message ordering. (3) eBPF Maps. It can store various data structures, designed for storing and sharing data between different eBPF programs and between eBPF programs and user-space applications. (4) Trace Pipe. For debugging or prototyping, eBPF programs can write messages into the *trace_pipe* file under *debugfs*.

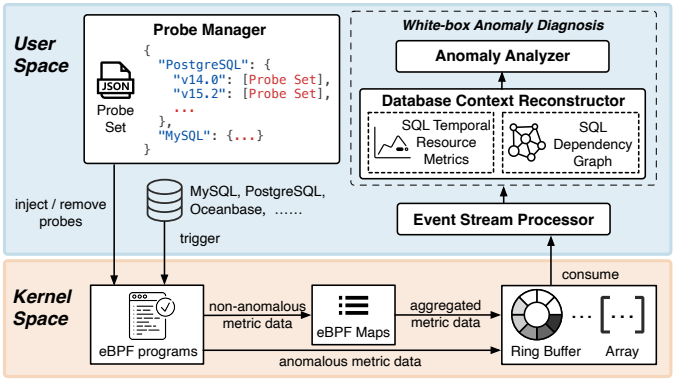


Fig. 2: Overview of DBdoctor

III. System Overview

DBdoctor works as an anomaly diagnosis and resolution platform for databases. It follows a non-intrusive and vendor-agnostic diagram, making it universally applicable across different database engines. This is critical for production environments as large organizations often have heterogeneous databases for various applications. DBdoctor can serve as a unified and standardized platform for diagnosing different database systems, avoiding the management and operational complexity of heterogeneous ecosystems.

DBdoctor applies eBPF to track database events and resource consumption in real-time and models database context as SQL temporal resource metrics and SQL dependency graphs. Thus, it can accurately diagnose the root causes of anomalies and provide actionable resolutions. As shown in Figure 2, the platform consists of four core components. (1) *Probe Manager* maintains the optimal eBPF probe set for each database. It dynamically injects or removes function probes on demand, ensuring fine-grained metric collection at runtime with low overhead. (2) *Event Stream Processor* efficiently reads the messages generated by the probes from kernel space into user space, and forwards these messages into a Kafka message queue. (3) *Database Context Reconstructor* consumes the raw eBPF messages and reconstructs them into SQL temporal resource metrics and SQL dependency graphs. (4) *Anomaly Analyzer* identifies performance anomalies, analyzes root causes, and provides concrete solutions to address anomalies. We illustrate DBdoctor's workflow of four phases.

Adaptive probing. This phase aims to balance the monitoring precision and overhead. In this phase, before monitoring a target database, the *Probe Manager* performs offline probe set selection that uses a utility-aware probe selection method to determine an optimal function set for injection (§IV-A). Specifically, each database function is assigned a utility quantifying its diagnostic benefit for anomaly detection if probed. Functions with higher utilities or assigned a higher weight by domain experts are preferentially selected, subject to a constraint that the total monitoring overhead is below a preferred level. The *Probe Manager* stores the optimal

probe set for each database version in a JSON configuration file, enabling efficient reuse across subsequent monitoring of the same or similar DBMSs. By default, only functions in this optimal set of probes are injected. At runtime, when a fine-grained analysis of specific metrics is required, *Probe Manager* supports on-demand probe injection, enabling additional probes to be dynamically attached to relevant functions without interrupting or blocking database execution.

Anomaly-aware message generation. In this phase, the injected eBPF probes are triggered to capture database internal contexts whenever their associated database functions are invoked. DBdoctor employs an anomaly-aware message generation strategy that adaptively controls the frequency of eBPF message emission (§IV-B). Upon activation, each probe executes a predefined callback function that collects metric data such as thread identifier, query identifier, transaction identifier, resource usage, and lock states. The collected data is buffered in eBPF maps and then aggregated into eBPF messages at an adaptive frequency. The messages are written into the ring buffer and eBPF Map data structures, such as arrays. Specifically, the kprobes injected in kernel functions generate resource consumption messages, whereas the uprobes injected in the database functions generate database event messages. This entire process occurs in kernel space without requiring user-space intervention. On the user side, the *Event Stream Processor* continuously consumes messages from the ring buffer and eBPF Map data structures, transfers them into user space, and forwards them into a Kafka message queue [21] for subsequent processing.

Database internal context modeling. In this phase, the *Database Context Reconstructor* consumes messages from the Kafka message queue, modeling the database internal context as SQL temporal resource metrics (§V-A) and a temporal SQL dependency graph (§V-B). The raw messages are first indexed by their SQL identifier and then categorized by resource dimension, including CPU, I/O, memory, network, and lock counts. For each SQL, messages are converted as a temporal resource metric sequence embedded with query execution semantics, reflecting the resource consumption during the query's execution lifecycle. These per-query metrics are further aggregated by SQL template, where all executions of the same template contribute to a unified template-level resource metric. Meanwhile, the raw lock event messages are consumed to construct a temporal SQL dependency graph, which records the evolving lock relationships among concurrently executing queries. Both SQL temporal resource metrics and SQL dependency graphs are stored in a database for subsequent anomaly analysis.

White-box anomaly diagnosis. In this phase, *Anomaly Analyzer* performs automated detection, white-box diagnosis, and targeted optimization. Anomalies are detected when either system-wide metrics or the metric patterns of individual SQL templates deviate significantly from normal behavior (§V-C1). Once an anomaly is detected, the system extracts the temporal resource metrics of all SQL templates and the SQL dependency graph within the anomaly time window. The *Anomaly Analyzer* identifies root causes and root-cause SQLs using a combination of feature-based anomaly detection and correlation analysis (§V-C2). After that, it proposes resolution strategies tailored to the identified root causes.

IV. Metric Probing and Collection

To address the challenges of granularity-overhead trade-off in sample-based metric collection approaches, we propose an adaptive event-based metric collection framework in DBdoctor that employs eBPF probes to capture fine-grained metrics related to anomalies with low performance overhead. In this section, we introduce how this framework identifies the optimal function set to probe (§IV-A), how the probes are dynamically attached to database instances to collect fine-grained metrics for diagnosis with low overhead (§IV-B).

Message Types. In DBdoctor, we package metric data into eBPF messages and collect two types of messages: (1) resource usage messages that are collected through the kprobes injected into kernel functions. These messages capture the real-time consumption of CPU, memory, I/O, and network resources; (2) database event messages that are collected through uprobes injected into database functions. These messages capture the SQL actions and background database events. For example, a *lock_wait* event represents that a SQL is waiting for locks held by another SQL, and a *log_flush* event represents that the database is flushing logs to persistent storage. Both message types embed correlation identifiers (e.g., SQL ID, process ID, thread ID, or file descriptor), allowing the observed resource usage to be linked to the corresponding database events. For example, the I/O usage messages collected from kernel functions can be correlated with SQL actions via the corresponding file descriptors.

A. Offline Probe Set Identification

Probing all relevant kernel and database functions can achieve comprehensive and fine-grained monitoring. However, this introduces significant performance overhead even with eBPF.Therefore, we propose a probe set identification mechanism to select the most effective set of functions to probe.

The collection of resource usage messages relies on inserting kprobes into critical kernel functions. For instance, probes on *vfs_read()* and *vfs_write()* capture I/O costs, and probes on *tcp_sendmsg()* and *tcp_recvmsg()* monitor network consumption. These probes are indispensable, as they provide the foundation for resource usage collection. Fortunately, the number of such kernel functions is relatively small, hence the probing overhead is negligible.

In contrast, database event messages are gathered by inserting uprobes into the database's internal functions. Since probing every function would incur prohibitive overhead, we design a utility-aware probe-selection method to determine the optimal function set $\mathcal{F}_{opt}$ for database event monitoring, thereby minimizing probing cost while preserving diagnostic coverage. The selection process consists of two steps: base function set construction and probe set refinement.

Base Function Set Construction. We first pre-select candidate functions from the symbol table of the DBMS kernel [1] to construct a minimal but complete function set $\mathcal{F}_{base}$ that ensures full observability for SQL temporal resource metric construction and dependency graph generation. We categorize database events into three types: (1) resource consumption events, $\mathcal{F}_{res}$, which capture SQL actions that consume CPU, I/O, memory, and network resources; (2) SQL dependency events, $\mathcal{F}_{dep}$, which capture query blocking relationships such as lock acquisition and release; (3) background events, $\mathcal{F}_{back}$, which capture resource-consuming background activities such as flushing or purging logs. By combining expert knowledge and source code analysis, we identify the basic database functions of these types, forming the base set $\mathcal{F}_{base}$:

$$\mathcal{F}_{base} = \mathcal{F}_{res} \cup \mathcal{F}_{dep} \cup \mathcal{F}_{back} \tag{1}$$

To minimize monitoring overhead, we exclude low-level functions (e.g., *memcpy()*) that lack query execution semantics and functions (e.g., *ReadRecord()*) that are excessively invoked during SQL execution and covered by higher-level functions.

Probe Set Refinement. Then we refine $\mathcal{F}_{base}$ into an optimal probe set $\mathcal{F}_{opt}$ through quantitative evaluation of each function's *diagnostic utility*. The diagnostic utility $\mathcal{U}(f)$ measures how important it is to inject a probe into the function f for performance anomaly diagnosis. As defined in Eq.2, it combines two components: (1) the anomaly utility $\mathcal{U}_{ano}(f)$, reflecting the function's sensitivity to performance anomalies, and (2) the expert prior weight $\mathcal{W}(f)$, representing the domain knowledge on its diagnostic importance.

$$\mathcal{U}(f) = max(\ \mathcal{U}_{ano}(f),\ \mathcal{W}(f)\ ) \tag{2}$$

The anomaly utility for each function is calculated as the difference in resource consumption between the normal and anomalous states. Specifically, we collect a set of real-world performance anomaly cases $\mathcal{M}$ and conduct controlled experiments under two conditions: (1) a normal baseline state S_{norm} and (2) an anomalous state S_m where anomaly case $m \in \mathcal{M}$ is injected. For each resource dimension $res \in \mathcal{RS}$ (e.g., CPU, I/O, memory) and function f, we denote the resource consumption of f in system state S as $T_{res}(f, S)$, where $\mathcal{RS}$ denotes the set of resource dimensions considered. The corresponding resource increment is defined as $\Delta T_{res}(f, S_m) = T_{res}(f, S_m) - T_{res}(f, S_{norm})$. The anomaly utility for function f is then calculated as the aggregated normalized increment across all resources and anomaly cases, as defined in Eq.3.

$$\mathcal{U}_{ano}(f) = \frac{1}{\mathcal{M}} \sum_{m \in \mathcal{M}} \sum_{res} \frac{\Delta T_{res}(f, S_m)}{\sum_{f' \in \mathcal{F}_{base}} \Delta T_{res}(f', S_m)} \tag{3}$$

The expert prior weight $\mathcal{W}(f)$ of each function in $\mathcal{F}_{base}$ is assigned by DBAs based on domain knowledge about each function's role in anomaly detection and database context collection. First, functions that play critical roles in diagnosing representative anomalies are assigned higher weights. For instance, *LockAcquire()*, *LockRelease()*, and *ExecSeqScan()* are known to be highly influential in diagnosing lock contentions and scan inefficiencies. Second, functions necessary for collecting correlation identifiers, which link the resource usage to the corresponding database events, are also assigned higher weights. For example, *mysql_execute_command()* is required to collect the correlations between SQL statements and query identifiers. In contrast, other functions that are not on the core path of query execution or are less likely to performance bottlenecks receive lower weights. These assigned weights fall within the same range as $\mathcal{U}_{ano}$, i.e., $[0, |\mathcal{RS}|]$.

After that, all functions in $\mathcal{F}_{base}$ are ranked in descending order according to the diagnosis utility $\mathcal{U}(f)$. We employ a binary search approach to determine the largest K such that the transaction throughput degradation caused by probing the top-K functions is acceptable (e.g., $\leq 3\%$). Specifically, we measure the throughput degradation through comprehensive stress testing across various scenarios. The final selected top-K functions form the optimal probe set $\mathcal{F}_{opt}$.

B. Online Anomaly-Aware Probing

To further improve the efficiency of metric collection (i.e., comprehensively cover anomaly-relevant functions with lower overhead), we design an online anomaly-aware probing mechanism that dynamically attaches the probe sets to database instances and adapts the frequency of message generation.

Probe Set Repository. The probe set identified in §IV-A is materialized as a standard-format JSON file. We have a probe set repository that maintains a mapping between the probe set files and the compatible DBMS distributions, similar to an App store. Thus, a probe set can be built once and used everywhere. We can reuse probe sets for identical or similar versions of the same DBMS, or even for different DBMSs forked from the same code base. In addition to the optimal probe set, we also maintain some optional probe sets that can be attached on demand for major DBMSs. We encourage DBMS providers to publish their own probe sets in the repository or to work with us to build and publish probe sets. This is beneficial to providers, customers, and DBdoctor.

Probe Attachment. We leverage the hot-pluggable nature of eBPF probes to support dynamic injection and removal of probes without halting the database. This capability enables runtime balancing between probing overhead and probing granularity. For resource usage monitoring, we inject all required kprobes into relevant system calls. For database event monitoring, however, only the functions in the optimal probe set $\mathcal{F}_{opt}$ are instrumented with uprobes by default. During execution, the system periodically evaluates the runtime overhead of probing. If the observed overhead exceeds a predefined threshold, probes with the smallest utility $\mathcal{U}(f)$ are removed adaptively. Conversely, if a DBA requires more fine-grained insights into specific resources (e.g., CPU usage), the framework can dynamically add probes from the optional probe sets for that metric.

[1]Even for closed-source DBMSs, the provider can usually provide the symbol table to customers along with the DBMS distribution package.

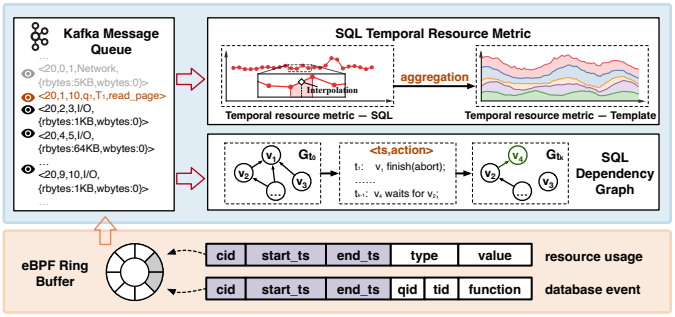


Fig. 3: Database context reconstruction

eBPF Message Format. The metrics collected by probes are packed into eBPF messages. As shown at the bottom in Figure 3, both the resource usage messages and database event messages embed three fixed-length fields, including the correlation identifiers (*cid*), start timestamp (*start_ts*), and end timestamp (*end_ts*). These fields enable DBdoctor to associate resource usage messages with specific database event messages. For example, in MySQL, a SQL statement's disk I/O operation can be linked to the corresponding *vfs_read()* probe through the file descriptor [8], which uniquely identifies the accessed file object, and thread_id, which is related to the SQL statement. Combined with temporal alignment via timestamps, this design accurately attributes system-level I/O costs to individual SQL statements. In addition to these three fields, resource usage messages include a *type* field that specifies the consumed resource dimension (e.g., I/O, CPU) and a *value* field that records the consumed resource volume. Database event messages include the query identifier (*qid*), transaction identifier (*tid*), function signature (*function*), and other essential database contexts.

Adaptive Message Generation. The generation of messages follows an anomaly-aware adaptive strategy. Naively generating a message on every function invocation would incur prohibitive overhead. Hence, we embed anomaly-aware logic directly within the eBPF programs to adaptively generate messages when necessary.

When collecting resource usage messages, for each resource dimension, we define a reporting interval and a threshold for spikes. During each probe invocation, the program first checks whether the observed resource consumption in this invocation is an anomalous behavior (e.g., resource usage exceeds the threshold). If an anomaly is detected, current resource usage is immediately constructed into an individual message and submitted to the ring buffer. Otherwise, the resource usage is accumulated with previous values stored in the eBPF Maps. Once the reporting interval elapses, the accumulated resource usage is packaged into a single message and submitted to the eBPF ring buffer or eBPF Maps. After submission, the user space is notified to consume messages for further analysis.

The generation of database event messages follows a similar aggregation strategy. Instead of generating a message for every function invocation, multiple homogeneous events within the same SQL execution are coalesced. For example, repeated *lock_acquire* events for a single SQL query are aggregated and pushed to user space only upon SQL completion or a transaction commit. However, if a lock wait time exceeds a predefined threshold, an immediate message is generated and pushed to the user space to highlight the anomaly. This design balances the trade-off between low runtime overhead and the timely detection of critical anomalies.

Message Consumption. In the user space, a daemon continuously polls the eBPF ring buffer, retrieving messages and forwarding them into a Kafka message queue. The adoption of Kafka absorbs message bursts and prevents excessive consumption of system resources. The queued messages are then consumed by the *Database Context Reconstructor*, which performs the higher-level SQL context reconstruction (§V).

V. Anomaly Diagnosis

The fine-grained event-based metric data enables precise and efficient anomaly diagnosis and resolution. However, it remains a significant challenge to identify the anomalies and their root causes from the massive raw messages. In this section, we present a solution that first reconstructs raw eBPF messages into two higher-level temporal database contexts: the *SQL-level temporal resource metric* (§V-A) and the *temporal SQL dependency graph* (§V-B). These contexts represent the real-time resource metrics across the query lifecycle and the inter-query dependencies. On this foundation, the solution then performs white-box anomaly diagnosis to identify anomalies and their root causes, enabling efficient and effective anomaly resolution (§V-C).

A. SQL-Level Temporal Resource Metric.

Since SQL statements with the same template are highly likely to exhibit similar resource consumption patterns and behavioral trends, we construct a temporal resource metric for each template. Formally, for each SQL template $\mathcal{Q}$, we maintain a temporal resource metric as Eq.4.

$$RM_{res}(\mathcal{Q}) = \{(t_i, v_{res}(t_i), \mathcal{A}_i) | t_i \in \mathcal{TS}\}, res \in \mathcal{RS} \quad (4)$$

where $\mathcal{RS}$ denotes the collection of monitored resource dimensions (e.g., CPU, I/O, network, lock count, etc.), $\mathcal{TS}$ denotes discrete time points, and $v_{res}(t_i)$ represents the consumption of resource res by template $\mathcal{Q}$ at t_i, and $\mathcal{A}_i$ represents the SQL actions that contribute to the resource consumption at t_i, which is the subset of predefined action set $\mathcal{A}$ that includes key actions in the query lifecycle.

Temporal Resource Metric Construction. As shown in the upper part of Figure 3, DBdoctor continuously consumes messages from the Kafka message queue and performs a multi-stage aggregation process to construct the temporal resource metric for each SQL template.

First, resource usage messages are matched with their corresponding database event messages based on temporal and identifier correlations. A resource usage message $Msg_r =< cid, start\_ts_0, end\_ts_0, ... >$ is mapped to a database event message $Msg_e =< cid, start\_ts_1, end\_ts_1, ... >$ if they share the same correlation identifier cid and the resource consumption interval is temporally contained within the

database event's execution interval ($[start\_ts_0, end\_ts_0] \subseteq [start\_ts_1, end\_ts_1]$). Notably, the correlation identifier alone is insufficient for unique attribution, since different queries may reuse the same identifier over time. Therefore, the temporal containment condition is required for precise correlation.

Second, temporal resource metrics are constructed for each SQL statement. Database event messages and the corresponding resource usage messages are aggregated for each SQL statement. The cumulative resource consumption values within each interval $[start\_ts, end\_ts]$ are converted into per-unit-time consumption rates $v_{res}(t_m) = \frac{value}{end\_ts - start\_ts}$, where t_m represents the midpoint timestamp within the interval. Each consumption value is then assigned a SQL action label $act \in \mathcal{A}$ based on the function name in the corresponding database event message (e.g., function *ExecSeqScan()* is labeled as *scan*). The temporal resource metric for each SQL statement q is thus defined as Eq.5.

$$RM_{res}(q) = \{(t_i, v_{res}(t_i), act_i) | t_i \in \mathcal{TS}\}, res \in \mathcal{RS} \quad (5)$$

Third, the statement temporal resource metrics are aggregated into template temporal resource metrics. For each SQL template, we construct a unified temporal sequence $\mathcal{TS}(\mathcal{Q}) = \bigcup_{q \in \mathcal{Q}} \mathcal{TS}(q)$ by merging the timestamp sequences of all SQL statements of the same template. For each timestamp in this unified sequence, the per-unit resource consumption of the template is calculated by aggregating (e.g., sum, average, max, etc.) the resource usage values of corresponding SQLs. In the case where a timestamp in the unified sequence falls within a query's execution interval but no consumption value is recorded for that query at that timestamp, the missing value is imputed using linear interpolation based on neighboring points. Moreover, the action tags of SQL statements are aggregated as the action set of the template.

B. Temporal SQL Dependency Graph.

To provide observability of inter-query dependencies, we construct the temporal SQL dependency graph (TSDG) from lock-related database event messages. Let $G_t = (V_t, E_t)$ denote the dependency graph at time t. Each vertex $v \in V_t$ represents an active SQL, and each directed edge $(v_i \to v_j) \in E_t$ indicates that SQL v_j is blocked, waiting for locks currently held by v_i. The directed edges E_t are dynamically derived from lock-related events including *lock_acquire*, *lock_wait*, *lock_release*. An edge $(v_i \to v_j)$ is inserted into E_t when v_i acquires a lock and v_j subsequently issues a *lock_wait* event on the same lock. The edge is removed once v_i releases the lock, as indicated by a *lock_release* event.

We also correlate SQL statements to transactions using the transaction ID in the event messages, and group the SQL of the same transaction in the TSDG. Thus, TSDG can reflect lock dependencies not only between SQL statements but also between transactions. In addition, in some DBMSs such as MySQL, the kernel may add additional locks for transactions. Such lock events not triggered by SQL are attached to the transaction group in TSDG.

As shown in the upper part of Figure 3, to reduce storage cost while ensuring graph reconstruction efficiency, we adopt a hybrid snapshot-and-delta storage strategy for TSDG. Rather than continuously persisting the full graph state, we periodically store a complete snapshot of the dependency graph $G_{snapshot_i}$. The intermediate changes between snapshots are captured as incremental updates $\Delta G_t = (\Delta V_t, \Delta E_t)$, where ΔV_t and ΔE_t denote newly added or removed nodes and edges relative to G_{t-1}. To reconstruct the dependency graph G_t at any time point t, we first retrieve the most recent preceding snapshot $G_{snapshot_i}$ where $snapshot_i \leq t$ and then sequentially apply all delta updates recorded between $snapshot_i$ and t. This snapshot-delta sequence forms the complete TSDG structure over a time horizon $\mathcal{TS}$: $\{G_{snapshot_i}, \Delta G_{t_i}, \Delta G_{t_{i+1}}, ...\}$.

C. Anomaly Identification and Resolution

Based on the temporal database contexts, we introduce how DBdoctor identifies anomalies (§V-C1) and their root causes (§V-C2), and performs anomaly resolution (§V-C3).

1) Anomaly Identification: DBdoctor identifies anomalies under two conditions: (1) system-level anomaly triggered when the consumption of a resource (e.g., CPU, memory, or I/O bandwidth) or the overall system throughput degradation exceeds a predefined threshold; and (2) SQL-level anomaly triggered when a query's temporal resource metric or latency significantly deviates from its historical behavior. Such SQL-level deviations are captured using time-series analysis techniques, including statistical outlier detection [13], [27] and change-point detection [15], [28]. Upon satisfying either condition, DBdoctor initiates the diagnosis pipeline.

2) Root Cause Identification.: To achieve accurate root cause identification, DBdoctor follows a two-step procedure. We first identify the root-cause SQLs (RC-SQLs) of the anomaly, leveraging the SQL temporal resource metrics and dependency graph. Then, we feed both internal and external database metrics into an inference engine to identify the accurate root causes of the anomaly.

RC-SQL Identification. Since multiple anomalies may coexist, DBdoctor performs independent RC-SQL identification for each anomaly. For an anomaly detected within the interval $[t_s, t_e]$, DBdoctor retrieves the SQL dependency graph and the SQL temporal resource metrics over an expanded interval $[t_s - \delta, t_e]$, as RC-SQLs triggering the anomaly may start before the SQL anomaly becomes observable.

For lock-related anomalies, such as long lock waits or deadlocks, DBdoctor identifies RC-SQLs through a structure analysis of the TSDG. For deadlocks, the DBMS kernel generally rolls back at least one transaction in the deadlock cycle. Such rollback events are captured and reflected on the TSDG. SQL with deadlock rollbacks are identified as the RC-SQLs. Whereas for long lock waits, users can set a configurable timeout (e.g., 1s). Then, waiting on a lock longer than the timeout is considered a long lock wait anomaly. We traverse the TSDG from the SQL with long lock waits to find the RC-SQLs that acquired the corresponding locks. If the

lock does not belong to any SQL in TSDG, the corresponding transaction is considered the root cause.

For resource-related anomalies, such as CPU or I/O anomalies, DBdoctor identifies RC-SQLs through correlation analysis of the system-level temporal metrics and the SQL template temporal metrics of the anomalous resources. The SQL statements belonging to the template with the highest Spearman's Rank correlation coefficient [32] are designated as RC-SQLs, as their execution most closely aligns with the anomalous resource fluctuations.

Root Cause Inference. After RC-SQL identification, DBdoctor infers the root cause through an inference engine. It first constructs a multi-dimensional *contextual feature vector* F_C, which serves as the input to the inference engine. This vector integrates three essential categories of information: (1) SQL action features. This includes the temporal resource metrics $RM_{res}(\mathcal{Q})$ of RC-SQLs during the anomalous interval. Specifically, the action tags embedded in the temporal resource metrics are also embedded into the feature vector, providing database contexts for the diagnosis; (2) SQL structure features. This includes the SQL statements and their execution plans. (3) Database metrics. This includes the database settings, database-level statistics, etc. Given F_C, the inference engine evaluates a hypothesis set $\mathcal{H} = \{h_1, h_2, ..., h_k\}$, where each h_i represents a known database root cause type, such as *Missing Index*, *Deadlocks*. For each hypothesis h_i, the inference engine computes a causal score $S(h_i|F_C)$ using expert-defined rules derived from domain knowledge and historical anomaly data. Hypotheses whose score $S(h_i)$ exceeds a predefined threshold are designated as root causes.

It is worth noting that the fine-grained, context-aware metrics collected by DBdoctor can be seamlessly integrated with other root cause diagnosis approaches, such as machine learning–based, deep learning–based, or LLM–driven methods, for better accuracy and interpretability.

3) Anomaly Resolution: DBdoctor employs a two-tiered performance anomaly resolution framework that integrates instant anomaly resolution and long-term optimization to ensure both rapid recovery and sustained system stability.

Instant Resolution. When an anomaly is detected and RC-SQLs are identified, DBdoctor immediately mitigates performance degradation through adaptive workload control. Specifically, it either terminates sessions associated with the RC-SQLs to promptly reduce resource contention or applies SQL throttling to limit the concurrency of RC-SQLs.

Long-Term Resolution. To prevent recurrent anomalies, DBdoctor employs targeted optimization strategies, such as SQL rewriting [22], parameter tuning [34], and index recommendation [36], based on the identified root causes. We combine rule-based resolution and LLM-based resolution in DBdoctor. We also sandbox the database optimizer and key execution logic to quickly audit candidate solutions (e.g., adding an index) before executing them, confirming their effectiveness (e.g., whether the optimizer selects the added index). Furthermore, we build a knowledge base of anomalies and their corresponding effective solutions, enabling us to use RAG to improve the accuracy of LLM-based resolution. Due to space limitations, we do not discuss the details of resolution.

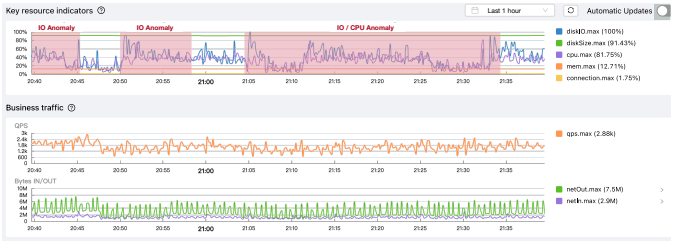


Fig. 4: Resource indicator and anomaly warning

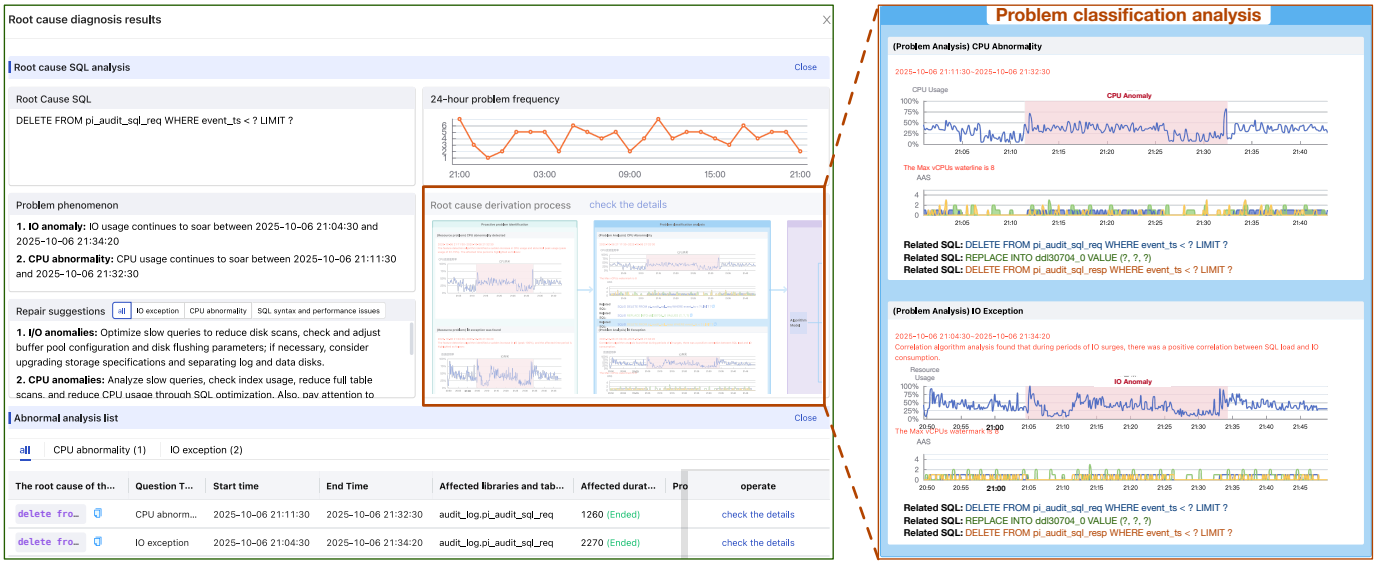
VI. Case Study

In this section, we demonstrate the visualization and diagnosis workflow of DBdoctor through a representative example. The workflow includes three stages: real-time resource monitoring and anomaly detection, root cause identification, and anomaly diagnosis and optimization guidance.

Resource monitoring and anomaly detection. Figure 4 presents DBdoctor's real-time resource dashboard, which continuously visualizes both system-level resource metrics (e.g., CPU and memory) and business traffic indicators (e.g., query throughput). By monitoring the metrics collected by probes, DBdoctor detects anomalies and annotates them with the anomalous type and time range, as shown by the pink-brushed regions in Figure 4. Once an anomaly is detected, DBdoctor automatically triggers the root cause identification process.

Root cause analysis. As presented in Figure 5a, DBdoctor performs both RC-SQL identification and root cause inference. For each identified RC-SQL, DBdoctor presents the identification process in the *Root cause derivation process* module.In this example, DBdoctor performs correlation analysis of CPU and I/O anomalies. Figure 5b shows the process of correlation analysis. Taking the CPU anomaly as an example, DBdoctor first extracts active SQL templates within the anomalous interval (labeled as Related SQLs) and then performs a correlation analysis between the CPU temporal resource metrics of these SQL templates and the system-level CPU metrics. In this case, the template `DELETE FROM pi_audit_sql_req WHERE event_ts < ? LIMIT ?` is recognized as the RC-SQL for both CPU and I/O anomalies. Subsequent analysis attributes the root causes to *Missing Index* and *Misconfigured Parameters*.

Anomaly diagnosis and optimization guidance. As shown in the *Repair Suggestions* module in Figure 5a, DBdoctor recommends optimizing buffer pool and disk flush parameters to mitigate the Misconfigured Parameters issue and creating an index to address the Missing Index issue, thereby reducing I/O and CPU overhead. Detailed recommendations are also provided in the *Parameter Tuning* and *Index Recommendation* panels. Specifically, the system suggests adjusting the innodb_buffer_pool_size parameter from 1024MB to 11520 MB and innodb_max_dirty_pages_pct_lwm parameter to 41, and creating an index on event_ts on table pi_audit_sql_req. Moreover, DBdoctor recommends avoiding non-deterministic DML patterns that rely on LIMIT, suggesting indexed predicates or deterministic ranges for safe deletions.



(a) Root Cause diagnosis results

(b) Root Cause derivation process

Fig. 5: Root cause SQL identification

VII. Evaluation

In this section, we compare DBdoctor to state-of-the-art diagnosis solutions and validate three critical aspects through experiments: (1) DBdoctor's effectiveness in anomaly diagnosis, including identification of root causes and the associated root-cause SQL statements (§VII-B), (2) DBdoctor's monitoring overhead compared to existing monitoring tools (§VII-C), and (3) the generality of DBdoctor's metric collection approach across other diagnosis methods (§VII-E).

A. Experimental Setup

Both DBdoctor and databases are deployed on a server running Linux v4.18 with 8 CPU cores (16 threads), 32 GB DRAM, and 500 GB SSD storage. We utilize the official bpftool [4] to manage the eBPF programs.

1) Default configuration: We conduct experiments using two popular databases: MySQL 8.0.22 [9] and PostgreSQL 15.2 [17]. By default, we set the isolation level to serializable.

2) Baselines: In our experiments, we compare DBdoctor against the following baselines. For root cause SQL (RC-SQL) identification, we compare DBdoctor against **PinSQL** [23], a state-of-the-art solution that identifies RC-SQLs using the active session metric (i.e., the number of active sessions related to the query at a given timestamp). For root case diagnosis, we evaluate DBdoctor against the machine learning-based approach **DBPA** [18] (including OC-SVM [25] and SVDD [26]) and the LLM-driven approach **D-Bot** [38] (deepseek-v3 without fine-tuning). All baseline approaches rely on sample-based metric collection, and we set their sampling frequency to 1 Hz, the highest frequency adopted in these works.

3) Benchmarks: We conducted experiments on public benchmarks and datasets collected in production environments.
(1) TPC-C. We use 100 warehouses by default. Following previous works [35], [43], we exclude *think time* and user data errors anomaly cases.
(2) TPC-H. We set the scale factor to 10 (i.e., 10GB dataset). We conduct experiments on all 22 TPC-H queries.
(3) Sysbench is a flexible, open-source benchmark widely used to evaluate OLTP performance [20]. We use the OLTP read–write workload (*oltp_read_write*) with the canonical sbtest schema to represent a mix of point reads, index lookups, and short updates. For our experiments, we provisioned 16 sbtest tables, each containing 1 million rows, for a total dataset of approximately 3.2 GB. Workload generation was performed from a separate machine using 64 concurrent client threads.
(4) Real-world workload. We use the real-world workload over a period of time in production environment to evaluate the diagnosis performance. The workload includes 203 anomaly cases caused by a single root cause and 315 anomaly cases caused by the interaction of multiple root causes. Notably, Table I presents nine representative categories of root causes included in the workload.

4) Evaluation Metrics: Following prior work [23], we evaluate RC-SQL identification using *HitRate* and Mean Reciprocal Rank (*MRR*). For each anomaly case within the evaluation, the per-case *HitRate* can be calculated by $\frac{|P_i \cap G_i|}{|G_i|}$, where G_i represents the ground-truth RC-SQL set and P_i represents the predicted set. The overall *HitRate* is the average across all cases. *MRR* measures ranking quality, calculated by $MRR = \frac{1}{|\mathcal{Q}|}\sum_{i=1}^{|\mathcal{Q}|}\frac{1}{rank_i}$, where $rank_i$ represents the position of the first correctly predicted RC-SQL within the ground-truth set. For root cause identification, we report *Precision*, *Recall*, and *F1-score*, computed in a multi-label setting.

B. Anomaly Diagnosis Accuracy

In this experiment, we evaluate DBdoctor's effectiveness for anomaly diagnosis using PostgreSQL. We evaluate the identification accuracy of RC-SQLs in §VII-B1 and evaluate the identification accuracy of root causes in §VII-B2.

TABLE I: Representative Root Causes in the real-world workload

Root Cause	Description
Long transactions	Long-running transactions caused by inefficient business logic.
Uncommitted transactions	Transactions suspended without commit or rollback (such as waiting for third-party API responses).
Missing indexes	Lack of indexes on large tables result in slow full table scans.
Redundant indexes	Unused indexes result in increased update overhead.
Lock contention	Intensive concurrent access on the same item, leading to long lock waits.
Deadlocks	Cyclic lock dependencies between transactions can cause the database to become unresponsive.
Excessive data scan	Wide query ranges or data skew can lead to large scan volumes.
Misconfigured parameters	Inappropriate parameters can lead to poor performance.
Poorly written SQL	Poorly written SQLs, such as `SELECT *, poor joins`, lead to bad performance.

TABLE II: Accuracy of RC-SQL diagnosis

Method	Single Cause Anomaly		Multi-Cause Anomaly	
	HitRate	MRR	HitRate	MRR
PinSQL	0.343	0.458	0.271	0.313
DBdoctor	0.914	0.944	0.863	0.904

1) Root cause SQL identification: We evaluate RC-SQL diagnosis accuracy for both single- and multi-cause anomalies and compare DBdoctor with PinSQL; the results are shown in Table II. DBdoctor substantially outperforms PinSQL across both categories. In single-cause scenarios, DBdoctor yields improvements of 57.1 percentage points in *HitRate* and 48.6 in *MRR* relative to PinSQL. The advantage is more obvious for multi-cause anomalies, where DBdoctor gains the performance improvements on *HitRate* of 59.2 percentage points and improvements on *MRR* of 59.1 percentage points.

These performance improvements primarily benefit from two aspects. First, DBdoctor provides SQL temporal resource metrics and dependency graphs, capturing detailed CPU, I/O, memory, network, and lock consumption for each SQL template. By directly correlating anomalous system-level resource trends with per-template SQL resource usage and tracking SQL dependencies, DBdoctor can precisely identify the RC-SQLs responsible for the anomaly. In contrast, PinSQL identifies RC-SQLs indirectly by analyzing the number of active sessions per SQL template to infer which queries are affected by RC-SQLs, then clustering SQL templates by business logics and analyzing their historical behavior to infer the RC-SQLs. This approach may lead to inaccuracies in some conditions. For instance, if the queries affected by RC-SQLs are aborted immediately by traffic control rather than blocked, there will be no observable increase in active sessions. Similarly, when the RC-SQL has complex business logic dependencies, PinSQL's identification of RC-SQLs may also be inaccurate. Second, DBdoctor captures fine-grained anomalous events and reports them via the anomaly-aware message generation strategy in eBPF programs, enabling precise detection of short-lived performance spikes. In contrast, PinSQL reports metric data at 1-minute or 1-second intervals, failing to capture anomalies caused by spike events.

2) Root cause identification: Then, we evaluate the accuracy of identifying root cause types. We compare DBdoctor with three representative diagnosis baselines: OC-SVM, SVDD, and D-Bot. All baseline methods use an open-source tool Dool [6] to collect operating system metrics and integrate database plugins, such as pg_stat_statements [11], to collect database metrics. All metrics are collected at 1-second intervals (the minimum sampling interval supported by the operating system). The results are shown in Table III.

TABLE III: Accuracy of root cause diagnosis

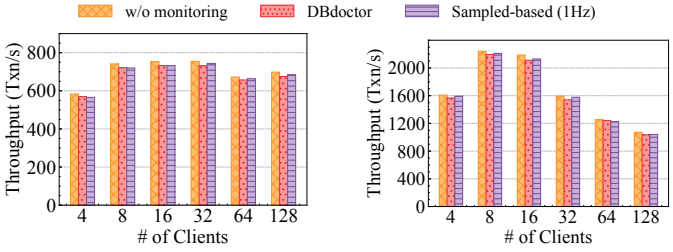
Method	Single Cause Anomaly			Multi-Cause Anomaly		
	Precision	Recall	F1-score	Precision	Recall	F1-score
OC-SVM	0.688	1	0.803	0.268	0.193	0.225
SVDD	0.683	1	0.799	0.421	0.372	0.395
LLM	0.717	1	0.879	0.478	0.402	0.437
DBdoctor	0.943	1	0.971	0.938	0.911	0.924

DBdoctor outperforms all baselines in both single-cause anomalies and multi-cause anomalies. For single-cause anomalies, DBdoctor achieves a precision of 0.943, perfect recall and an F1 score of 0.971, representing an improvement of 9.1 percentage points over the best-performing baseline. For multi-cause anomalies, DBdoctor remains highly accurate while the identification accuracy of baseline approaches degrades significantly. Specifically, DBdoctor attains an F1-score of 0.924, exceeding the best-performing baseline by more than 50%. These improvements are primarily due to DBdoctor 's white-box diagnosis capability.

In single-cause scenarios, baseline models perform poorly on anomalies caused by *Long transactions*, *Uncommitted transactions*, and *Redundant indexes*. The accurate identification of these root causes requires detailed temporal SQL contexts. For example, redundant indexes introduce hidden write amplification by index maintenance costs and they are rarely used by SQL statements. DBdoctor collects the detailed metric consumption and corresponding SQL actions, making the identification accurate. However, baseline approaches only observe overall database resource metrics or aggregated metrics (such as total I/O consumption) of SQL statements, making them hard to identify these root causes.

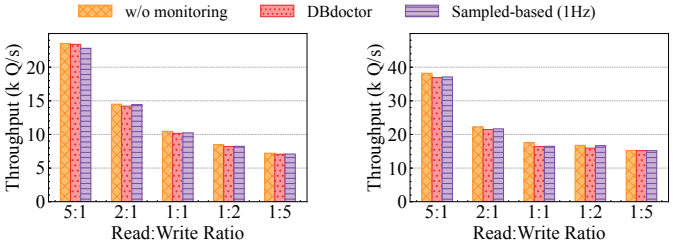
The advantage of DBdoctor is more pronounced in multi-cause scenarios, where multiple anomalies co-occur and interact. For example, an uncommitted transaction and an update without an index jointly cause contention. Baseline methods can only detect lock contention, but they fail to identify uncommitted transactions and missing index simultaneously due to the lack of fine-grained internal database observability.

Summary. DBdoctor provides fine-grained metrics and performs white-box diagnosis, thus achieving high accuracy on both RC-SQL and root cause identification.



(a) MySQL (b) PostgreSQL

Fig. 6: Impact of concurrency - TPC-C



(a) MySQL (b) PostgreSQL

Fig. 7: Impact of write operation percentage - Sysbench

C. Diagnosis Overhead

In this experiment, we evaluate the overhead of metric collection strategies of DBdoctor against the sample-based monitoring used in baseline approaches in both MySQL and PostgreSQL. Since workload characteristic may influence probe invocation frequencies, we perform evaluations using various benchmarks.

Impact of concurrency. We use TPC-C benchmark to evaluate the impact of concurrency by varying the number of clients. As shown in Figure 6, we observe an up-down pattern as the number of terminals increases. The throughput increases as the concurrency increases and reaches saturation. As the number of terminals increases, CPU contention becomes more severe, leading to performance degradation. Compared to normal execution, the throughput of DBdoctor declines 3.9% and 3.5% on PostgreSQL and MySQL, respectively. Moreover, DBdoctor achieves comparable performance against the sample-based collection (the gap is less than 1%).

Impact of write ratio. As shown in Figure 7, we use Sysbench to evaluate the performance overhead with different write operation percentages. The performance decreases as the write operation percentage increases for all methods. The average throughput of DBdoctor decreases by 2.1% on MySQL and 3.8% on PostgreSQL versus normal execution. It remains comparable to the sample-based collection strategy.

Overhead on TPC-H. We evaluate the performance overhead on analytical workloads with all queries on TPC-H. As shown in Figure 8, we report latencies on a log scale due to the large variance in query plans. On average, DBdoctor increases only about 3.9% for MySQL and 3.8% for PostgreSQL.

Furthermore, as shown in Table IV, we list the memory and CPU usage of DBdoctor using TPC-C in MySQL. DBdoctor introduces the acceptable performance overhead. Specifically, in terms of CPU utilization, DBdoctor incurs extra CPU

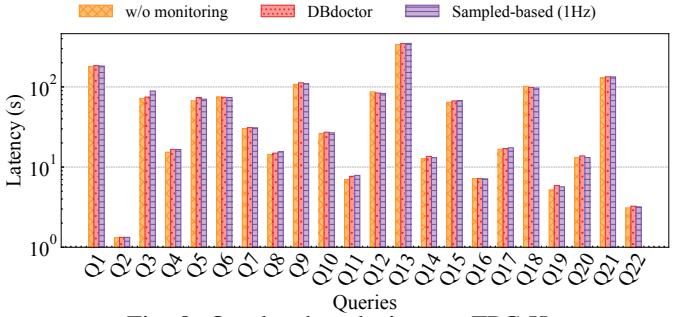


Fig. 8: Overhead analysis over TPC-H

TABLE IV: Memory and CPU usage over TPC-C

terminals		4	8	16	32	64	128
Avg CPU (%)	w/o monitoring	23.57	38.23	40.23	35.79	36.18	41.31
	DBdoctor	30.78	47.25	49.12	41.55	41.09	47.81
Avg Mem (MB)	w/o monitoring	415.83	459.41	476.96	932.52	1189.85	1465.89
	w DBdoctor	419.44	494.91	503.25	1076.53	1222.81	1462.62

utilization of approximately 4.9 to 9.02 percentage points. This overhead primarily stems from its eBPF probes capturing and construction for the SQL temporal resource metric and SQL dependency graph. For memory overhead, DBdoctor exhibits even smaller impact. Under low concurrency scenarios (e.g., 4 terminals), memory consumption increases by only about 3.61 MB. Even at peak load with 32 terminals, the total memory is 144.01 MB, a relative increase of under 15.4%. Its overhead is primarily driven by storage demands for collected metrics (such as time-series metric buffers), remaining within an acceptable range.

Summary. DBdoctor incurs negligible performance overhead, comparable to or less than existing sample-based monitoring approaches, making it suitable for deployment in production.

D. Evaluation on Probe Set Identification

In this experiment, we evaluate the effect of injecting and removing eBPF probes using PostgreSQL. Specifically, we compare the root-cause identification accuracy of DBdoctor with two baselines DBdoctor (-) and DBdoctor (+). DBdoctor inject all probes in the optimal probe set $\mathcal{F}_{opt}$, DBdoctor (-) removes part of the lock-related probes and DBdoctor (+) injects more probes with high utility $\mathcal{U}(f)$. As shown in Table.V, although DBdoctor (-) achieves lower performance degradation, it performs less identification accuracy. Specifically, it fails to construct the complete SQL dependency graph, resulting in inaccurate identification of the root cause of lock-related anomalies. In contrast, DBdoctor (+) achieves insignificant improvement in accuracy while introducing higher overhead compared to DBdoctor.

Summary. DBdoctor can select the optimal probe sets that achieve high diagnosis accuracy with low runtime overhead.

E. Evaluation on Generality

In this experiment, we evaluate the generality of our adaptive event-based metric collection framework using PostgreSQL. We use our collected metrics to replace the metrics in the LLM-based diagnosis approach D-Bot; the results are

TABLE V: Effect of probe set coverage

Method	Accuracy			QPS degradation(%)
	Precision	Recall	F1-score	
DBdoctor (-)	0.854	0.895	0.874	3.22
DBdoctor	0.941	0.950	0.946	4.81
DBdoctor (+)	0.944	0.954	0.944	7.49

TABLE VI: Accuracy of root cause diagnosis

Method	Single Cause Anomaly			Multi-Cause Anomaly		
	Precision	Recall	F1-score	Precision	Recall	F1-score
LLM	0.717	1	0.879	0.478	0.402	0.437
LLM(DBdoctor)	0.879	1	0.932	0.806	0.614	0.697

shown in Table VI. The LLM's diagnostic accuracy significantly increases with the fine-grained metrics provided by DBdoctor (LLM(DBdoctor)). For single-cause anomalies, precision achieves an improvement of 16.2 percentage points, and false positives are greatly reduced. For complex multi-cause anomalies, the benefit is even greater, where the precision of LLM is improved by 32.8 percentage points and the recall improves by 21.2 percentage points. Although LLM(DBdoctor) performs slightly worse than DBdoctor alone, mainly due to LLM hallucination and difficulty focusing on long, metric-rich prompts, the results still confirm that DBdoctor 's fine-grained metrics substantially enhance the effectiveness of LLM-based diagnosis methods.

Summary. With fine-grained internal database metrics collected in DBdoctor, the root cause identification accuracy of LLM-based diagnosis approaches can be improved, demonstrating the generality of our fine-grained event-based metric collection framework.

VIII. Related Work

Database performance diagnosis. This is the process of identifying the root causes behind performance anomalies. Automated diagnosing approaches [23], [24], [31], [38], [42], are proposed to improve the diagnosis accuracy and reduce human effort. For example, iSQUAD [24] leverages machine learning to diagnose the root causes of intermittent slow queries in cloud databases. DBPecker [31] proposes a graph-based approach to model and diagnose compound anomalies in distributed relational database systems. D-Bot [38] uses LLM to automatically acquire knowledge from documentation and generate diagnostic reports with solutions. PinSQL [23] identifies root cause SQLs that are truly responsible for the performance anomalies using session metrics and application logics. However, these approaches rely on existing sample-based monitoring tools which fail to capture fine-grained metrics and provide database internal context observability necessary for accurate diagnosis. DBdoctor proposes an event-driven metric collection framework to capture fine-grained database internal metrics. Moreover, we model the metric data into SQL temporal resource metrics and dependency graph to enable white-box anomaly diagnosis. Notably, our metric collection and modeling methods can complement existing diagnostic approaches to improve diagnosis accuracy further.

Database performance optimization. Existing works primarily focus on SQL rewriting [22], [37], index recommendation [36], knob tuning [19], [34], plan hints [30], [33], etc. Our work is orthogonal to these works. Rather than proposing a new optimization algorithm, DBdoctor is designed to identify both the root-cause SQL statements and the underlying root causes of the performance anomalies. Once identified, DBdoctor may apply existing automatic optimization approaches to recommend solutions to resolve the anomalies.

eBPF for databases. The eBPF technology has been widely used in many applications, including network load balancing and observability [5], runtime security monitoring [7], general-purpose system observability [3], and distributed protocols [39]. A few studies have introduced eBPF into database systems. For example, Dint [41] leverages eBPF to accelerate the performance of distributed transactions. In contrast to those approaches, DBdoctor focuses on leveraging eBPF to monitor database events and context, aggregating these events into SQL-level metrics for database performance diagnosis.

IX. Discussion

DBdoctor uses fine-grained event-based metric collection and modeling to comprehensively capture anomaly events and reconstruct database contexts for anomaly diagnosis and resolution with low overhead. Through successful and wide commercial deployments, DBdoctor proves this to be a viable approach that fundamentally improves the efficiency and accuracy of online performance anomaly diagnosis for databases. And the framework of DBdoctor is extensible for applying new diagnosis and resolution methods.

Identifying the eBPF probe sets requires the participation of DBAs and kernel experts. This is a one-time job for a DBMS and DBMS providers are cooperating with us in building probe sets, because customers are willing to choose DBMSs with better diagnostic tools. Moreover, the benefits are not limited to database anomaly diagnosis. Our framework actually builds an ecosystem that solidifies database kernel knowledge and accumulates critical database runtime data. The probe set repository, fine-grained event messages, and verified-effective root causes and resolutions provide a vast amount of valuable data to address the major problem of lacking data in intelligent database design and tuning based on ML and LLMs.

X. Conclusion

In this paper, we present DBdoctor, a non-intrusive performance diagnosis platform applicable to different databases. DBdoctor proposes an adaptive event-based metric collection framework, collecting fine-grained internal and external database metrics with low overhead. The collected metrics are modeled as SQL temporal resource metrics and dependency graphs, providing internal observability of query executions and enabling white-box anomaly diagnosis. Extensive evaluations on both popular benchmarks and real-world workloads demonstrate that DBdoctor outperforms other diagnosis approaches in root cause identification accuracy with comparable or less overhead.

REFERENCES

[1] Akamai state of online retail performance spring 2017. https://www.scribd.com/document/451425146/akamai-state-of-online-retail-performance-spring-2017, 2025.

[2] Amazon study: Every 100ms in added page load time cost 1% in revenue. https://www.conductor.com/academy/page-speed-resources/faq/amazon-page-speed-study/, 2025.

[3] Bcc - tools for bpf-based linux io analysis, networking, monitoring, and more. https://github.com/iovisor/bcc, 2025.

[4] bpftool. https://github.com/libbpf/bpftool.git, 2025.

[5] Cilium - cloud native, ebpf-based networking, observability, security. https://cilium.io/, 2025.

[6] Dool, linux cli tool providing real-time system resource monitoring. https://github.com/scottchiefbaker/dool.git, 2025.

[7] Falco - detect security threats in real time. https://falco.org/, 2025.

[8] File descriptor. https://en.wikipedia.org/wiki/File_descriptor, 2025.

[9] mysql-server. https://github.com/mysql/mysql-server.git, 2025.

[10] perf: Linux profiling with performance counters. https://perfwiki.github.io/main/, 2025.

[11] pg_stat_statements, track statistics of sql planning and execution. https://www.postgresql.org/docs/current/pgstatstatements.html, 2025.

[12] Prometheus - monitoring system & time series database. https://prometheus.io/, 2025.

[13] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: identifying density-based local outliers. In SIGMOD Conference, pages 93–104. ACM, 2000.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010, pages 143–154. ACM, 2010.

[15] Joshua Corneck, Edward A. K. Cohen, James S. Martin, and Francesco Sanna Passino. Online bayesian changepoint detection for network poisson processes with community structure. Stat. Comput., 35(3):75, 2025.

[16] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. In Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings, pages 84–94. www.cidrdb.org, 2005.

[17] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. Proc. VLDB Endow., 8(11):1190–1201, 2015.

[18] Shiyue Huang, Ziwei Wang, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. DBPA: A benchmark for transactional database performance anomalies. Proc. ACM Manag. Data, 1(1):72:1–72:26, 2023.

[19] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Tieying Zhang, Jianjun Chen, Hong Chen, and Cuiping Li. E2etune: End-to-end knob tuning via fine-tuned generative language model. CoRR, abs/2404.11581, 2024.

[20] Alexey Kopytov. Sysbench: a system performance benchmark. http://sysbench. sourceforge. net/, 2004.

[21] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB, volume 11, pages 1–7. Athens, Greece, 2011.

[22] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. LLM-R2: A large language model enhanced rule-based rewrite system for boosting query efficiency. Proc. VLDB Endow., 18(1):53–65, 2024.

[23] Xiaoze Liu, Zheng Yin, Chao Zhao, Congcong Ge, Lu Chen, Yunjun Gao, Dimeng Li, Ziting Wang, Gaozhong Liang, Jian Tan, and Feifei Li. Pinsql: Pinpoint root cause sqls to resolve performance issues in cloud databases. In ICDE, pages 2549–2561. IEEE, 2022.

[24] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. Diagnosing root causes of intermittent slow queries in large-scale cloud databases. Proc. VLDB Endow., 13(8):1176–1189, 2020.

[25] Carla Sauvanaud, Mohamed Kaâniche, Karama Kanoun, Kahina Lazri, and Guthemberg Silvestre. Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned. J. Syst. Softw., 139:84–106, 2018.

[26] Carla Sauvanaud, Mohamed Kaâniche, Karama Kanoun, Kahina Lazri, and Guthemberg Silvestre. Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned. J. Syst. Softw., 139:84–106, 2018.

[27] Xiuyao Song, Mingxi Wu, Christopher M. Jermaine, and Sanjay Ranka. Statistical change detection for multi-dimensional data. In KDD, pages 667–676. ACM, 2007.

[28] Charles Truong, Laurent Oudre, and Nicolas Vayatis. Selective review of offline change point detection methods. Signal Process., 167, 2020.

[29] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023, pages 452–468. ACM, 2023.

[30] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Fastgres: Making learned query optimizer hinting effective. Proc. VLDB Endow., 16(11):3310–3322, 2023.

[31] Qingliu Wu, Qingfeng Xiang, Yingxia Shao, Qiyao Luo, and Quanqing Xu. Dbpecker: A graph-based compound anomaly diagnosis system for distributed rdbmss. Proc. VLDB Endow., 18(12):5383–5386, 2025.

[32] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In SOSP, pages 117–132. ACM, 2009.

[33] Xianghong Xu, Zhibing Zhao, Tieying Zhang, Rong Kang, Luming Sun, and Jianjun Chen. COOOL: A learning-to-rank approach for SQL hint recommendations. In VLDB Workshops, volume 3462 of CEUR Workshop Proceedings. CEUR-WS.org, 2023.

[34] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. A demonstration of the ottertune automatic database management system tuning service. Proc. VLDB Endow., 11(12):1910–1913, 2018.

[35] Zhanhao Zhao, Hongyao Zhao, Qiyu Zhuang, Wei Lu, Haixiang Li, Meihui Zhang, Anqun Pan, and Xiaoyong Du. Efficiently supporting multi-level serializability in decentralized database systems. IEEE Transactions on Knowledge and Data Engineering, 35(12):12618–12633, 2023.

[36] Wei Zhou, Chen Lin, Xuanhe Zhou, and Guoliang Li. Breaking it down: An in-depth study of index advisors. Proc. VLDB Endow., 17(10):2405–2418, 2024.

[37] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. A learned query rewrite system using monte carlo tree search. Proc. VLDB Endow., 15(1):46–58, 2021.

[38] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. D-bot: Database diagnosis system using large language models. Proc. VLDB Endow., 17(10):2514–2527, 2024.

[39] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with ebpf. In NSDI, pages 1391–1407. USENIX Association, 2023.

[40] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: fast in-kernel distributed transactions with ebpf. In NSDI, pages 401–417. USENIX Association, 2024.

[41] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: fast in-kernel distributed transactions with ebpf. In NSDI, pages 401–417. USENIX Association, 2024.

[42] Xuhang Zhu, Xiu Tang, Sai Wu, Jichen Li, Haobo Wang, Chang Yao, Quanqing Xu, and Gang Chen. Cola: Model collaboration for log-based anomaly detection. Proc. VLDB Endow., 18(11):3979–3987, 2025.

[43] Qiyu Zhuang, Xinyue Shi, Shuang Liu, Wei Lu, Zhanhao Zhao, Yuxing Chen, Tong Li, Anqun Pan, and Xiaoyong Du. Geotp: Latency-aware geo-distributed transaction processing in database middlewares. In ICDE, pages 433–445. IEEE, 2025.