DBPA: A Benchmark for Transactional Database Performance Anomalies

SHIYUE HUANG, Peking University, China
ZIWEI WANG, Peking University, China
XINYI ZHANG, Peking University, China
YAOFENG TU, ZTE Corporation, China
ZHONGLIANG LI, ZTE Corporation, China
BIN CUI, Peking University, China

Anomaly diagnosis is vital to the performance of online transaction processing (OLTP) systems. In the meanwhile, machine learning techniques can reason complex relationships beyond human abilities and perform well on such problems. However, they rely on a large number of training samples for anomalies, which are in serious shortage in both industry and academia due to the difficulty of collection. The problem raises the demand of a benchmark for anomaly reproduction and data collection.

In this paper, we propose DBPA, a benchmark for transactional database performance anomalies. Specifically, we identify nine common anomalies rooted in the diverse influence factors. For each anomaly, we carefully design a reproduction procedure, which consists with its root cause in real-world databases. With the reproduction procedures, users can easily generate a dataset in a new environment and extend new anomaly types. For compound anomalies, we provide a generation algorithm that allows users to generate compound anomalies data of any possible combinations with existing collected data. We also provide a large dataset of both normal and anomalous monitoring data collected from various environments, facilitating the training of machine learning models and the evaluation of new algorithms for anomaly diagnosis.

CCS Concepts: • **Information systems** → *Database performance evaluation*; **Autonomous database administration**.

Additional Key Words and Phrases: database performance, anomaly diagnosis, benchmark, dataset

ACM Reference Format:
Shiyue Huang, Ziwei Wang, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. 2023. DBPA: A Benchmark for Transactional Database Performance Anomalies. *Proc. ACM Manag. Data* 1, 1, Article 72 (May 2023), 26 pages. https://doi.org/10.1145/3588926

1 INTRODUCTION

Modern transactional databases are faced with complex problems of performance anomalies. With the growth of the size and complexity of the database, the queries may suffer from performance

Authors' addresses: Shiyue Huang, huangshiyue@pku.edu.cn, School of CS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University, Beijing, China; Ziwei Wang, wangziwei@stu.pku.edu.cn, School of CS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University, Beijing, China; Xinyi Zhang, zhang_xinyi@pku.edu.cn, School of CS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University, Beijing, China; Yaofeng Tu, tu.yaofeng@zte.com.cn, ZTE Corporation, Nanjing, China; Zhongliang Li, li.zhongliang@zte.com.cn, ZTE Corporation, Nanjing, China; Bin Cui, bin.cui@pku.edu.cn, School of CS & Key Lab of High Confidence Software Technologies (MOE), Institute of Computational Social Science, Peking University (Qingdao), Peking University, Beijing, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/5-ART72 $15.00
https://doi.org/10.1145/3588926

regression due to inappropriate indexes, vacuums in the data blocks, or bad transaction logic. The sudden increase in the arrival rate of the queries, especially for instant online services, can also lead to performance degradation. It is difficult to manually diagnose performance anomalies in modern transactional databases, and traditional algorithms might fail due to low precision and recall [53, 74].

The diagnosis of anomalies can be defined as anomaly detection and classification problems, where machine learning techniques can perform well [21, 35, 54, 69, 72–74]. In recent years, machine learning techniques have made significant advances in detection and classification problems, including tasks on images, natural language, and tabular data. Empirical studies [55] show that they can perform better than traditional algorithms in anomaly detection and classification. To diagnose database performance anomalies, the most commonly used data is the monitoring metrics collected from the database and the operating system [41, 43]. It is in the form of tabular data, which is suitable as the input of machine learning models.

However, there is a critical problem to apply machine learning to database performance anomaly diagnosis. The machine learning models require a large amount of data for training [24], which is deficient in both industry and academia. In industry, the performance anomalies have a low detected rate. It is difficult to detect and reason the performance degradation of certain queries within a complex workload [25]. Moreover, the monitoring metrics may not be recorded after a diagnosis procedure is finished. Database administrators (DBAs) usually only record the appearance, cause, diagnosis procedure, and fix procedure. Such data is not suitable for machine learning. In academia, there are some small datasets for the experiments in the research of diagnosis algorithms. Their sizes are small and the data lacks diversity in both environments and configurations, which limits the generalization abilities of the machine learning models.

This problem reflects the demand for a benchmark dataset for database performance anomalies. However, it is challenging to build a dataset due to the following issues.

Absence of determinate reproduction of database performance anomalies. A natural reproduction approach is to passively observe the performance during a long-time stress testing procedure [25]. The periods of the performance degradation are gathered as a dataset. A vital weakness of this approach is that it is difficult to accurately identify the root causes of the anomalies. The existing automatic diagnosis techniques are not reliable to work without manual intervention [23, 41, 66]. Note that the performance anomalies usually do not trigger warning logs, so their root causes are mainly identified by the monitoring metrics, which are unstable and complex. Thus, it requires experienced DBAs to spend much time and effort analyzing the data. This is unaffordable to construct a dataset with thousands of anomalous samples, which is a common scale for machine learning. To accurately identify the root causes of the anomalies, the appropriate approach is determinate reproduction that proactively reproduces each anomaly according to the root cause.

Limited diversity of scenarios for the same anomaly type. For one anomaly type, the corresponding monitoring data can be quite different because of the various influence factors, which compose the reproduction scenarios of the anomaly [44, 45]. Take the anomaly of missing indexes as an example, the scenario of this anomaly can be generally described as follows. First, the anomalous queries are operated in a certain *system environment* with some other concurrent queries as the *background workload*. Second, the table that suffer from missing indexes has a certain shape, which is determined by *the number of columns*, *the size of each column*, and *the number of rows*. Finally, there is a certain *concurrency degree* of the slow queries on that table, which is measured by the number of queries that concurrently execute. The dataset should contain multiple values for these dimensions so the machine learning algorithms can learn the common features

and generalize to other scenarios. However, the existing datasets [23, 41, 66] only contains limited scenarios, which essentially restricts the generalization ability of the applied algorithms.

Missing support for compound anomalies. It is common that different anomalies occur at the same time [66]. But there is no benchmark supporting the compound anomalies of various possible combinations. One reason is that it is unaffordable to reproduce all the possible compound anomalies because of the large number of combinations. An alternative approach is to allow the users to generate the data for compound anomalies with existing data, which is equally helpful as the samples for the diagnosis algorithm. However, the monitoring metrics of compound anomalies are not a simple combination of the values for single anomalies. A carefully designed algorithm is needed to generate the data of compound anomalies from single anomalies.

Motivated by the problems, we propose **DBPA**, a benchmark for transactional **D**ata**B**ase **P**erformance **A**nomalies, which provides both the anomaly reproduction procedures and a corresponding dataset with various scenarios. Users can employ DBPA for the following tasks.

(1) Train a machine learning model for anomaly diagnosis with DBPA dataset in a similar database environment.
(2) Reproduce the anomalies and collect enough data with DBPA benchmark for machine learning tasks in other various database environments.
(3) Evaluate the algorithms easily for database performance anomaly detection and diagnosis with DBPA dataset.

Based on our industrial diagnosis experiences and related research work [23, 66], we select nine common anomaly types rooted in the database environment, the workload amount, and the queries. To ensure **determinate reproductions**, we analyze the root cause of each anomaly type and design specific reproduction procedures. We also verify that the anomaly happens by checking the performance of specific queries. For **diversity**, we not only set up different system environments and background workloads, but also have different configurations for each anomaly type. To support **compound anomalies**, we propose a generation algorithm that can generate the data of compound anomalies through the data of single anomalies and the normal data. We select a few combinations of anomalies that cover all the supported types and construct a dataset for them through reproduction. The generation algorithm is trained on the collected combinations, and can be used to generate any combination of the supported anomalies

We summarize our contributions as follows.

- We propose the determinate reproduction procedures of common database performance anomalies, which allows the users to collect abundant data for anomaly diagnosis.
- We construct a large dataset of transactional database performance anomalies, which has diverse scenarios and supports compound anomalies.
- We evaluate common algorithms for anomaly detection and diagnosis with DBPA. With empirical studies, we prove that DBPA can improve the performance of anomaly diagnosis in transactional databases by enhancing the abilities of machine learning algorithms.

The rest of the paper is organized as follows. Section 2 presents some related work. Section 3 introduces the preliminaries of common anomaly types and requirements for a benchmark. Section 4 shows how we reproduce the selected common anomalies. Section 5 shows how we construct our dataset. Section 6 shows the validation experiments of DBPA. Section 7 shows the evaluation experiment of some common algorithms for anomaly detection and diagnosis. Section 8 summarizes this paper.

Table 1. Open Datasets of Database Anomaly.

Ref.	#Cases	Diversity	Anomaly Types
[66]	396	3 environments	10
[41]	319	Diverse services	10 iSQs
[23]	200	Diverse throughputs	10

2 RELATED WORK

Database Performance Anomaly Diagnosis. Some traditional algorithms are proposed to diagnose the performance anomalies in transactional databases, but machine learning is seldom used. DBSherlock [66] uses causal models to diagnose performance anomalies. The dataset for its experiment is small and lacks diversity. ISQUAD [41] focuses on intermittent slow queries and uses a clustering algorithm, TOPIC, to identify the anomaly types. Its dataset comes from the service workloads of Alibaba OLTP Database, and the number of anomalies is also small. Dundjerski et al. [14] use an end-to-end rule-based method to diagnose the anomalies in Azure SQL Databases [3]. Both the data and the diagnosis method are only for Azure SQL. AutoMonitor [23] uses a modified version of K-Nearest-Neighbor (KNN), a naive machine learning algorithm, for diagnosis. The dataset still has a small size and lacks diversity. Some other researchers use debug logs [18, 46] or time metrics [9, 26, 27] for diagnosis, but the collection of such data has a significant influence on database performance. We summarize the properties of the open datasets in Table 1. DBPA has a larger data size, more diversity in scenarios, and extensiveness to various anomaly types.

Benchmarks for Anomaly Detection. Some researchers [15, 20, 22, 33, 49] construct benchmarks for anomaly detection by transforming real-world data. Based on the datasets, they design certain evaluation metrics for different anomaly detection algorithms, such as point difficulty [15] and NAB Score [33]. Exathlon [22] serves as a benchmarking platform for both evaluations and data analysis. TSB-UAD [49] employs data augmentation and introduces novel measurements of dataset difficulty. ADBench [20] assesses comparison angles including the availability of supervision, the anomaly types, and the algorithm robustness. Our work differs from theirs in three aspects. First, we provide both the dataset and the approach to generate the data from scratch. Second, our benchmark contains different types of anomalies, so it can be used for both anomaly detection and diagnosis. Finally, we focus on the performance anomalies of transactional databases and design the reproduction procedures that consist with the root causes.

Database Tuning. The tuning techniques have been extensively studied in databases, including knob tuning [1, 13, 16, 17, 31, 34, 67, 68, 70, 71, 75], index selection [2, 30, 51, 60], workload balancing and resource management [42, 57, 58, 61, 62], etc. They help improve database performance as advisors for better configurations. Our work is orthogonal with the tuning techniques, whose target is not to *tune*, but to *identify* the type of the anomaly, *trigger* the tuning procedures and *indicate* which component (e.g., knob, index, etc.) to tune in complex modern databases. Even for a DBMS equipped with a tuning tool, DBPA can still help troubleshoot the system because we support various configurations and provide verification conditions for the occurrence of the anomalies.

3 PRELIMINARIES

3.1 Common Performance Anomalies

Database performance anomalies are presented as the increase of query latency or the decrease of transaction throughput. Their root causes correspond to the various influence factors of database performance, which helps categorize the common anomaly types for DBPA.

The performance of a transactional database system is mainly influenced by the database environment, the workload, and the design of the database management system (DBMS) [32, 74]. We consider a given DBMS design stable, so we focus on the environment and the workload, which can be separated into two aspects: the amount [61] and the queries [11, 29, 59]. So we explore the common anomalies caused by the three categories. Based on the diagnosis history from an industrial database service, our experiments, and other research work [23, 66]. We select the following anomaly types for DBPA.

Database Environment: We consider the database configurations and the resources as the main influence factors.

(1) *Small shared buffer.* It is a representative bad configuration [1] that causes database performance downgrade. The shared buffer works as the cache of the disk, which can significantly influence the database performance [34, 67, 70]. A small shared buffer is a common anomaly caused by mistaken adjustments. Some other bad configurations, such as small work memory, may not significantly affect the performance of most queries in a transactional database according to empirical studies [68].

(2) *I/O saturation due to other processes.* It is a representative of resource bottleneck and differs from the anomaly caused by the workload. The main system resources include I/O, CPU, memory, and network [37, 47]. In a hard-disk DBMS like PostgreSQL, CPU and memory are usually not the bottleneck resources [12], and our experiments also show that CPU and memory saturation does not lead to significant performance degradation. Network problems mainly occur in distributed systems [7, 39]. So we only select I/O bottleneck for reproduction.

Workload Amount: The amount of the workload can be measured as the number of transactions sent to the database per second. Large workload amount can cause pressure on the file system, the transaction system, and the resources [42].

(1) *Highly concurrent inserts.* The anomaly relates to the file system, where the expansion of data files becomes a bottleneck.

(2) *Highly concurrent commits.* The anomaly relates to the transaction system, where the wait events increase.

(3) *Heavy workloads.* Simply increasing the workload amount can lead to heavy resource consumption and degrades performance.

Query Triggered: These query-triggered anomalies only influence specific queries instead of the general workload. We select four typical anomalies related to indexes [30], vacuum, and locks [12].

(1) *Missing indexes.* The select queries on large tables may slow down due to index missing.

(2) *Too many indexes.* The performance of the insert, delete, and update queries may slow down because they need to update the indexes synchronously.

(3) *Vacuum.* The vacuums refer to the empty spaces of the deleted data in common disk-based databases. They can lead to unnecessary I/O consumption and typically cause performance degradation in PostgreSQL because it always appends new data to the end of the tables [12]. In MySQL, the new data is inserted into the vacuums, not causing performance anomalies [48].

(4) *Lock waits.* The queries with writes require the locks, which can increase the latency.

There are still some anomaly types that are not covered by the listed ones according to the database service environment, such as the poorly-written queries, which are related to the query design of specific database services and are difficult to generalize to various environments. These anomalies can still be categorized into the mentioned performance influence factors. Based on these factors, we allow the users of DBPA to extend new anomaly types as needed.

3.2 Requirements

Before presenting the design of DBPA, we discuss the requirements of a reliable benchmark for database performance anomalies as guidance. We consider both the challenge issues in Section 1 and practical demands [15, 33], and propose the following six requirements.

(1) *Consistency*. The reproduction procedures should consist with the root causes of the anomalies.
(2) *Correctness*. It is possible that the designed reproduce procedure does not cause the occurrence of the expected anomaly. The reproduction is correct only if the performance degradation appears.
(3) *Diversity*. The benchmark should involve as many influence factors, i.e., scenarios for the monitoring data as possible.
(4) *Support for compound anomalies*. The benchmark should support different combinations of anomalies.
(5) *Extensiveness*. Various anomaly types should be supported and the reproduction procedure should be able to generalize to new anomaly types.
(6) *Effectiveness*. The monitoring metrics that we collect should be effective to help performance anomaly diagnosis.

In the rest of the paper, we show how DBPA meets the requirements by describing the details of the anomaly reproduction and dataset construction, followed by validation experiments. Specifically, we carefully design the reproduction procedures and strive to reproduce the root causes for anomalies (*consistency*) with a verification of the performance degradation (*correctness*). And we set up different system environments, background workloads, and configurations for each anomaly type (*diversity*) and design a generation algorithm (*support for compound anomalies*). For *extensiveness*, we provide specific instructions for supporting new anomaly types. And finally, we validate the *effectiveness*, *diversity*, and *support for compound anomalies* of DBPA through experiments.

4 REPRODUCTION OF COMMON ANOMALIES

In this section, we first introduce the background workloads, then show the reproduction procedures of the common anomalies in three categories.

4.1 Background Workloads

Before reproducing the anomalies, we should design the background workloads as the simulation of a normal database service environment. The background workloads can also be used to generate normal data in contrast to the anomalies.

We use some OLTP benchmarks from OLTPBench [10] as background workloads. The selected benchmarks include TPC-C, TATP, Smallbank, and Voter. We limit the number of terminals to make the hard disk throughput less than 60 percent of the maximum. This ensures that the background workloads do not suffer from I/O saturation.

After designing the background workloads, we design the reproduction procedures of the anomaly types. Table 2 provides an overview of the anomaly categories, types, reproduction methods, and verification methods. The selected reproduction and verification methods are suggested by our industry partner based on the DB environments and the tolerance of performance degradation.

4.2 Database Environment Anomalies

To reproduce database environment anomalies, we set up an anomalous environment and run the background workload, which should be influenced by the environment and become anomalous. To verify the *correctness*, we compare the average throughput of the workload in the normal/anomalous environments.

Table 2. Selected Anomaly Types.

Category	Type	Reproduction	Verification
Database Environment	Small Shared Buffer	Modify the Knob	Throughput < 10% Normal
	I/O Saturation	Inject I/O Ops	Throughput < 10% Normal
Workload Amount	Highly Concurrent Inserts	Inject Queries	Latency > 1.5× Normal
	Highly Concurrent Commits	Inject Queries	Wait Events Detected
	Heavy Workloads	Inject Workloads	Latency > 1.5× Normal
Query Triggered	Missing Indexes	Inject Queries	Latency > 20× Normal
	Too Many Indexes	Inject Queries	Latency > 1.5× Normal
	Vacuum	Inject Queries	Latency > 1.5× Normal
	Lock Waits	Add Locks	Wait Events Detected

Small Shared Buffer. According to the experience of some DBAs from the industry, an appropriate size of the shared buffer is around 25 ~40 % of the available memory of the instances. To reproduce the anomaly, we set the knob 'shared_buffers' to a small value, which is decided by observing the corresponding throughput [65]. If the throughput is lower than the normal throughput by 10%, we consider the reproduction to be correct.

I/O Saturation Due to Other Processes. We use stress-ng [1], a tool to stress operating systems to inject I/O consumption. If the throughput of the background workload is lower than the normal throughput by 10%, we consider the reproduction to be correct.

4.3 Workload Amount Anomalies

The general reproduction procedure of workload amount anomalies is different from that of database environment anomalies. We continuously run the background workload, and inject a large amount of the specific workload. To verify the *correctness*, we construct a baseline that injects a small amount of the specific workload, which is still considered normal. Then we compare the average latency of the injected workload between the reproduction and the baseline. For certain anomaly types, we can also check the wait events.

Highly Concurrent Inserts. We create an empty table and inject insert queries to that table with high concurrency. To make a difference from highly concurrent commits, we commit only once for several inserts. We take single-thread inserts as the baseline, and check the latency of the injected inserts for *correctness*, which should be more than 1.5 times longer than the baseline.

Note that we only insert into one specific table. The column number and the column size of the table are part of the configuration parameters of this anomaly, which will be discussed in Section 5.2. The scenario of multiple inserted tables is considered as compound anomalies of the same type because of the various combinations of the table shapes, which will be discussed in Section 5.4.

Highly Concurrent Commits. We still create an empty table and inject insert queries, but we commit for every insert. To avoid the bottleneck of file expansion speed, which is the root cause of highly concurrent inserts, we set the data size of insertion to be small. We use the queries that commit once for several inserts as the baseline and valid the *correctness* by querying the active wait events for the WALWriteLock. In PostgreSQL, we execute *select * from pg_stat_activity where wait_event = 'WALWriteLock' and state <> 'idle';*. The result of the baseline should be empty but that of the reproduction should not.

[1] https://wiki.ubuntu.com/Kernel/Reference/stress-ng

Heavy Workloads. We inject the background workloads from OLTPBench with a higher concurrency degree. If the average latency of the injected workloads is more than 1.5 times longer than the baseline of the normal background workload, we consider the reproduction to be correct.

4.4 Query-triggered Anomalies

For query-triggered anomalies, the general reproduction procedure is similar to database environment anomalies. The only difference is that we need to construct the trigger for the anomalous queries at the beginning. To verify the *correctness*, we construct a baseline without the trigger for comparisons.

Missing Indexes. Missing indexes can lead to a full table scan on a large table, which is the main cause of the performance anomaly. So we first create a large table without the primary key index. Then we run the background workload, and inject some select queries that have filtering conditions on the primary key. As a baseline for the *correctness*, we build the primary key index and inject the same queries. If the average latency of the anomalous queries is more than 20 times longer than the baseline, we consider the reproduction to be correct. We choose a large number because a full table scan is significantly slower than an index scan on a large table.

Too Many Indexes. Building too many indexes can also cause a performance anomaly since the insert, delete, and update operations on indexed columns causes updates of the indexes [30]. To reproduce the anomaly, we first create a table of normal size, and build indexes on several columns. Then we run the background workload, and inject update queries. As the baseline, we remove all indexes except the primary key index, and inject the same queries. The *correctness* is validated if the average latency of the anomalous updates is more than 1.5 times longer than the baseline.

Vacuum. Vacuums refer to the space of the deleted rows in the tables. In PostgreSQL, new data is inserted at the end of the tables instead of the vacuums [12], so the space consumption can be much larger than the size of valid data. The performance of a scan on a table with too many vacuums degrades because of the unnecessary I/O consumption. To reproduce this anomaly, we first create a large table and delete a certain percentage of data at the beginning of the table, which ensures the existence of vacuums. Then we drop the primary key index if it exists so that the select query on this table will make a full table scan instead of an index scan. The reason for avoiding index scans is that the influence of vacuums is negligible for index scans. It differs from the reproduction of missing indexes by a smaller size of valid data. Next, we run the background workload and inject some select queries. To construct a baseline, we create a table with the same size of data but without vacuums. If the average latency of the anomalous queries is more than 1.5 times longer than the baseline, the *correctness* is verified.

Lock Waits. The reproduction of lock waits is slightly different. To trigger anomalous lock waits, we start a transaction with a full table update on a medium-sized table (i.e., around 10% of the database size) of the background workload. The transaction is not committed and we start the background workload. The updates on that table should wait for the lock [64]. We use the normal background workload as the baseline and the existence of active wait events validates the *correctness*.

4.5 Extension to New Anomaly Types

Users can extend our reproduction procedures to new anomaly types. For example, if a user wants to add an anomaly for a certain-form poorly-written queries, the general procedure of query-triggered anomalies should be followed. A table for that query should be created first. Then the background workload starts and the anomalous queries are injected. To validate the *correctness*, a baseline should be performed with the corresponding well-written query. If the average latency of the anomalous queries is more than certain times longer than the baseline, the *correctness* is ensured.

5 DATASET CONSTRUCTION

With the help of the reproduction procedures, we can construct a reliable dataset for anomaly diagnosis. In this section, we first introduce the collection of the monitoring metrics. Then we show how we construct the dataset with diversity in both the environments and the configurations. Next, we introduce how users can generate a dataset with DBPA in a new environment. Finally, we show how DBPA supports compound anomalies.

5.1 Monitoring Metrics

The data we collect should be helpful for performance anomaly diagnosis. The related work on database anomaly diagnosis takes either the monitoring metrics, the debug logs, or the query-level time metrics as the input of the diagnosis algorithm. The collection of the debug logs [18, 19, 46] can significantly influence the performance of the database system. To collect the time metrics [9, 26, 27], the database should write a query-level execution log, largely downgrading the database performance. So the most practical and commonly-used data for performance anomaly diagnosis is the monitoring metrics, which are also suitable for the machine learning algorithms that we want to apply.

All the monitoring objects and metrics we used are listed in Table 3. According to the related work [41, 43], the monitoring metrics for anomaly diagnosis should include both the operating system metrics and the database metrics. The operating metrics refer to the resources such as I/O, CPU, memory, and network, and other statistics such as the number of interrupts, locks, processes, and sockets. The database metrics are acquired by querying the database system, including the usage of the buffer, the database size and its changes, the connection status, the number of transactions, the status of the locks, and the configuration values. We adopt an open-source tool, Dool [2], to collect the operating system metrics and design some plugins for Dool to support the collection of database metrics. Each sample of the collected data consists of a series of the values of all the metrics at each timestamp, and the interval between two timestamps is five seconds.

5.2 Diversity

The diversity of data is beneficial to the generalization ability of machine learning models [40, 50]. Given test data from a new scenario, the machine learning model is more likely to make correct predictions if its training data is more diverse. To collect data with high diversity, we set up different scenarios for each anomaly type.

The scenario of an anomaly includes both the environment and the configurations. For the environments, we set up four background workloads as mentioned in Section 4.1 and four system environments as follows. To set up the system environments, we use a Ubuntu 16.04 server with 96 logical cores of CPU and 512 GB memory as a physical machine and use different dockers to simulate different system environments. Each docker has an operating system of Ubuntu 16.04 and a DBMS of PostgreSQL 12.9. We set up four dockers with different CPU and memory limitations, i.e., 32 / 64 cores of CPU and 128 GB / 256 GB memory. We adjust the database configurations of the shared buffers according to the available memory.

For the configurations, we use different parameters for each anomaly type. For database environment anomalies, we configure different values for the environmental factors. For workload anomalies, a new table usually needs to be created, which has the number and the size of the columns as parameters. If there are scans on that table, the number of rows should also be a parameter. For injected queries, the concurrency degree is a parameter. There are also other parameters for specific anomaly types, such as the number of indexes for the anomaly of too many indexes,

[2]https://github.com/scottchiefbaker/dool

Table 3. Monitoring Metrics.

Object	Metrics	Description
cpu/total	usr, sys, idl, wai, stl	CPU Usage
dsk/total	read, writ	Disk I/O Speed
net/total	recv, send	Network Speed
paging	in, out	Memory Pages
memory	used, free, buff, cach	Memory Usage
system	int, csw	Interrupts & Context Switches
procs	run, blk, new	Process Counts
load avg	1m, 5m, 15m	Avg Workload in Given Time
swap	used, free	Swap Usage
interrupts	interrupts	Interrupt Counts
io/total	read, writ	I/O Counts
async	#aio	Asynchronous I/O Counts
filesystem	files, inodes	File System
sysv ipc	msg, sem, shm	System V for IPC
file locks	pos, lck, rea, wri	File Locks
sockets	tot, tcp, udp, raw, frg	Socket Counts
tcp sockets	lis, act, syn, tim, clo	TCP Socket Counts
udp	lis, act	UDP Socket Counts
unix sockets	dgm, str, lis, act	Unix Socket Counts
vm	majpf, minpf, alloc, free	Virtual Memory Usage
advanced vm	stal, scanK, scanD, pgoru, astll	Advanced Virtual Memory
zones memory	d32F, d32H, normF, normH	Memory Zone Usage
dbsize	size, grow, insert, update, delete	Database Size
conn	conn, %con, act, LongQ, LongX, etc.	Database Connections
locks	Locks	Database Lock Counts
transactions	comm, roll	Database Transaction Counts
pg_buffer	clean, back, alloc, heapr, heaph, ratio	PostgreSQL Buffer Usage
pg_settings	shared_buffers, max_connections, work_mem, autovacuum_work_mem, autovacuum_max_workers, etc.	PostgreSQL Configurations
mysql_buffer	data, free, dirty, flushed, readr, reads, ratio	MySQL Buffer Usage
mysql_settings	innodb_buffer_pool_size, innodb_buffer_pool_instances, max_connections, etc.	MySQL Configurations

and the percentage of deleted data for the anomaly of vacuums. We do not consider the number of the tables as a parameter because there are too many combinations of the table shapes. Instead, we consider the scenario with multiple tables as compound anomalies, which will be discussed in Section 5.4.

We set various values for every parameter and construct the dataset. For each configuration, we repeat the reproduction procedure for specific times according to the number of configurations of the anomaly type. The data sizes are shown in Table 4. The interval between two timestamps is five

Table 4. Data Sizes of Single Anomalies.

Anomaly Type	Samples	Timestamps
Small Shared Buffer	64	768
I/O Saturation	240	2880
Concurrent Inserts	2304	27648
Concurrent Commits	2304	27648
Heavy Workloads	720	8640
Missing Indexes	1152	13824
Too Many Indexes	3072	36864
Vacuum	2304	27648
Lock Waits	120	1440
Normal	2880	34560

seconds. The duration of one sample is around one minute long, i.e., twelve timestamps, except for *small shared buffer* which is ten minutes long.

5.3 Benchmark Architecture and APIs

The architecture of DBPA consists of two parts: the reproduction framework and the evaluation framework.

The reproduction framework is shown in Figure 1. It serves the users that want to generate a dataset with DBPA in a new environment. The components are separated on two instances, one as the database server and the other as the client. On the database server, the DBMS, Dool, and stress-ng are installed. On the client, the OLTPBench is installed and the reproduction scripts of the anomalies are placed. For each anomaly type, there is a *Shell* script as the *reproduction controller*, which controls OLTPBench, Dool, stress-ng, and the *Python* scripts for *DB operations*. It records the execution logs.

We provide a *codegen* tool to generate the reproduction scripts from *XML* files with four main tags: *env*, *table*, *workload*, and *inject*. It serves the users in two ways. First, users can easily design the scenario combinations for each anomaly by modifying the *XML* tags. Second, users can extend DBPA to new anomaly types based on the *XML* templates.

In the evaluation framework, the data on the server and the logs on the client are processed into a dataset. Then the algorithms are applied to evaluate the metrics like accuracy, precision, recall, F1 score, etc. To support user-defined algorithms, we provide a basic interface with *init*, *train*, and *predict* functions. Both traditional and learning-based algorithms are supported.

5.4 Compound Anomalies

In real-world databases, it is common for several types of anomalies to occur at the same time, or for the same type of anomaly to occur on several tables. We consider them as compound anomalies that require to be supported by the benchmark. A challenge is that there are too many possible combinations of anomalies and it is unaffordable to conduct the reproduction procedure for all the combinations. Therefore, to support compound anomalies, we provide both a dataset and a generation algorithm that can learn compound anomalies from existing data.

Dataset. The dataset contains a few combinations and each of them involves two anomalies. For the validation experiments in Section 6.3, we design two groups of combinations. Each group contains three combinations, A + B, A + C, and B + C, where A, B, and C are three anomaly types.

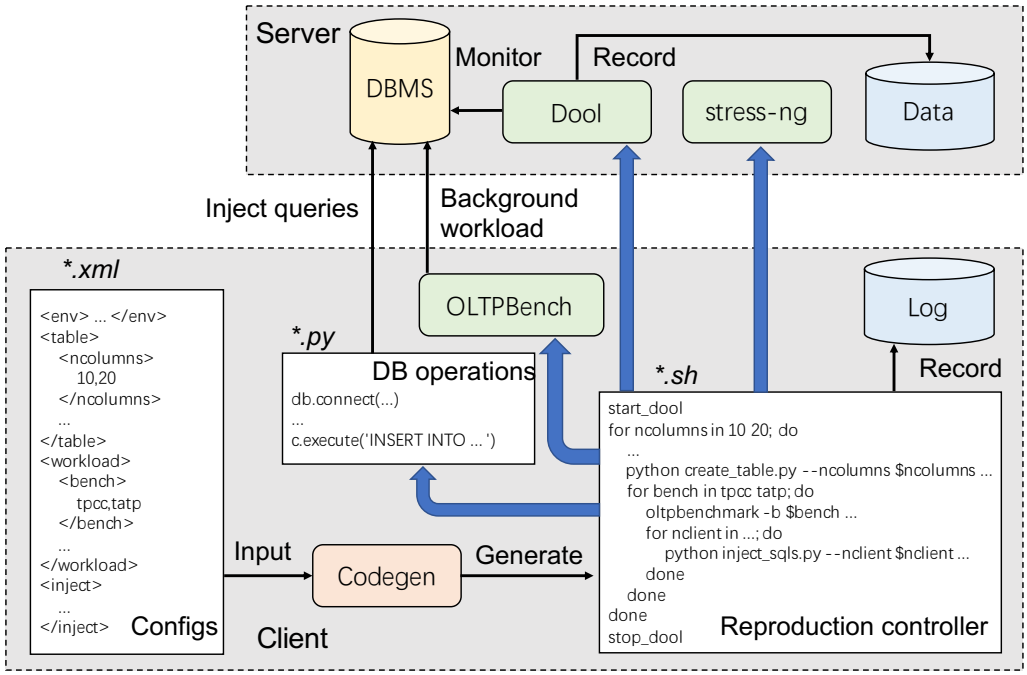

Fig. 1. Reproduction Framework of DBPA

The anomalies in the first group are from the same category, and those in the second are from different categories. In addition, we have another three combinations to make sure that the dataset covers all the supported anomalies. All the combinations and their data sizes are listed in Table 5, where the two groups are listed first.

Generation Algorithm. For other combinations that involve two or more anomalies, we provide a generation algorithm. Users can select appropriate combinations according to the application scenarios and generate the data with our algorithm. It takes two slices of anomalous data and a slice of normal data as the input and predicts a slice of data of the compound anomalies which means that the two input anomalies happened at the same time. The normal and anomalous data slices should be collected with the same system environment and background workload. To generate combinations that involve more than two anomalies, the user can simply input the data slices of compound anomalies.

There are some naive algorithms to combine the two slices of anomalous data, such as average, maximum, and minimum. These algorithms could have significant errors, especially for the metrics that have a significant difference between the normal and the anomalous. To improve the quality of the generated data, we propose to use learning-based regression models. The models take two anomalous data slices and one normal data slice as input and predict the value of data slices for the compound anomalies. We select *Random Forest* [4], *XGBoost* [8], and *LightGBM* [28] as the regression models, which generally have small errors in experiments.

Users can prepare the training data for the models by taking all the involved single anomalies and some normal data as features and taking the compound anomalies that cover the target anomaly types as labels. For instance, they can use the single anomalies of A, B, and C with some normal

Table 5. Data Sizes of Compound Anomalies.

Anomaly X	Anomaly Y	Minutes	MB
Missing Indexes	Lock Waits	192	11.72
Missing indexes	Too many indexes	512	5.67
Too many indexes	Lock Waits	192	12.30
Too many indexes	Heavy Workloads	512	13.41
Too many indexes	I/O Saturation	256	6.46
Heavy Workloads	I/O Saturation	256	6.28
Missing indexes	Concurrent Commits	768	18.20
Too many indexes	Vacuum	512	5.69
Small Shared Buffer	Concurrent Inserts	512	12.71

data as the features, and use the compound anomalies of A + B and A + C as the labels, to train a model to generate the data of B + C.

To ensure small errors, we do not predict multiple metrics with one model. There is a high cost of computation and storage if we fit models for all the metrics. To improve the efficiency of the generation process, we propose to combine the regression models with naive algorithms. For each monitoring metric, We use the anomaly detection module of *DBSherlock* [66] to check whether it has a significant difference between the normal and the anomalous. For those without a significant difference, we simply use the naive average values of the anomalies. For the rest, we employ the machine learning models. Specifically, we use *Random Forest* for anomalies of the same category and use *LightGBM* for those of different categories, because they generally have smaller errors.

6 VALIDATION EXPERIMENTS

Our designed anomalies reproduction procedures ensure that DBPA meets the requirements of *consistency*, *correctness*, and *extensiveness*. The rest of the requirements are related to the quality of our dataset. In this section, we show the results of the validation experiments on these requirements, i.e., the *effectiveness* of monitoring metrics, the *diversity* of scenarios, and the *support for compound anomalies*.

6.1 Validation on Effectiveness

In this part, we check the effectiveness of the monitoring metrics by verifying the differences between the data of each anomaly type and the normal data. First, there should be a significant difference between the normal and the anomalous. Second, such differences should be explicable according to the anomaly types. Finally, different anomaly types should have different important metric patterns that distinguish the anomalies from each other.

For the experiments, we first select the important metrics based on the *XGBoost* [8] models. We fit one model for each anomaly type. The model is trained for one-vs-rest binary classification to distinguish between the specific anomaly type and others (including the normal). We employ *XGBoost* as the model, which can generate the importance weight of each input feature, i.e., each monitoring metric. Specifically, the reduction of Gini indexes on the split points of the tree models in *XGBoost* is used to generate the weights, and we can find out the important metrics based on the weights. We do not use the anomaly detection algorithms in the related work of database anomaly diagnosis like *DBSherlock* [66], because they only focus on the differences between the normal and

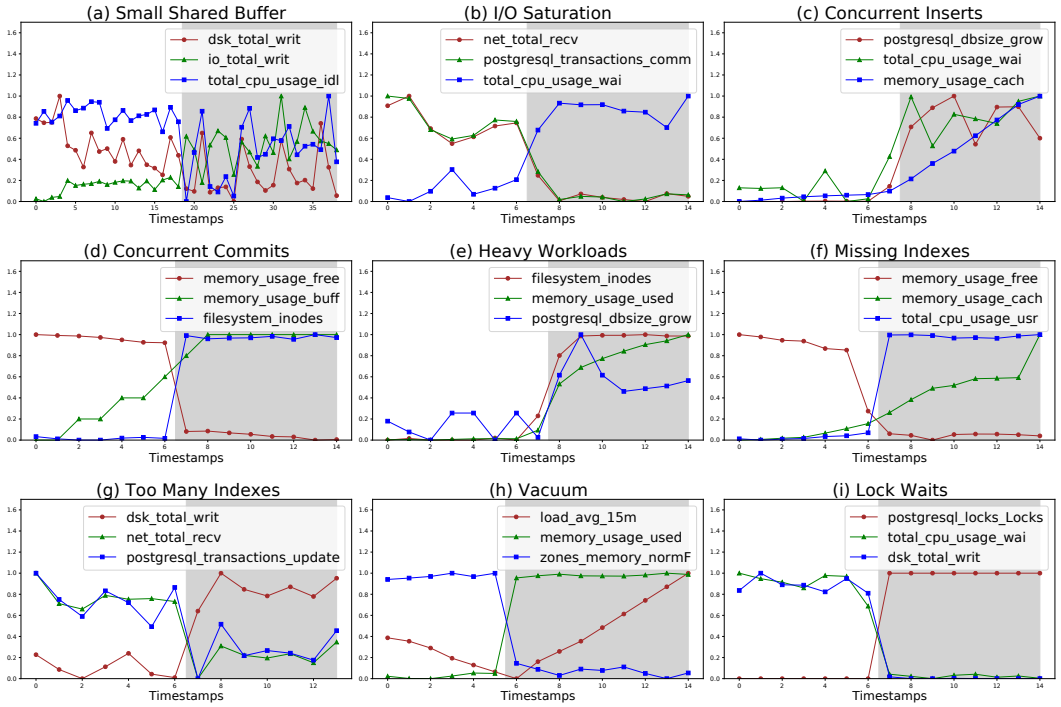

Fig. 2. Top-3 Metrics for Each Anomaly Type.

the anomalous, and omit the differences between different anomaly types. The one-vs-rest *XGBoost* models can meet both of the requirements.

We show the plot of the top-3 important metrics in Figure 2 for each anomaly type. The x-axis represents the timestamps and the y-axis represents the metrics normalized to the range 0-1. The junction of the white and the gray areas indicates the timestamp that we start to reproduce the anomaly, and the grey background indicates the region of anomaly. We show the results and explanations of the differences as follows.

Database Environment Anomalies. The results of *small shared buffer* are shown in Figure 2 (a). The top-3 metrics are the data size of disk writes (dsk_total_writ), the count of I/O writes (io_total_writ), and the CPU idle time (total_cpu_usage_idl). All the metrics become more unstable with a small shared buffer. The count of I/O writes generally increases because the count of swap-outs increases for the shared buffer. The data size of disk writes does not significantly increase because the average throughput of transactions is lower than normal.

For *I/O saturation*, the top-3 metrics are the network receive speed (net_total_recv), the rate of committed transactions (postgresql_transactions_comm), and the CPU time that waits for I/O (total_cpu_usage_wai). There are decreases in both 'net_total_recv' and 'postgresql_transactions_comm' because the throughput is lower than normal, and 'total_cpu_usage_wai' increases because of the I/O bottleneck.

Workload Amount Anomalies. As shown in Figures 2 (c), (d), and (e), all the anomalies of this category consume more resources due to the increase in the workload amount. The significant metrics include the growth rate of database size (postgresql_dbsize_grow), the CPU time of I/O waits (total_cpu_usage_wai), the memory metrics (memory_usage_cache for disk cache,

memory_usage_free for free memory, and memory_usage_buffer for buffer cache), and the inode number in the file system (filesystem_inodes).

For *highly concurrent inserts*, the expansion of the data file is the bottleneck, so 'total_cpu_usage_wai' increases, and 'filesystem_inodes' is not significant. For *highly concurrent commits*, many transactions of insertions are blocked, so 'postgresql_dbsize_grow' is not significant. For *heavy workloads*, both 'filesystem_inodes' and 'postgresql_dbsize_grow' are significant.

Query-triggered Anomalies. Both *missing indexes* and *vacuum* have full table scans that increase the consumption of resources as shown in Figures 2 (f) and (h). For *missing indexes*, the free memory (memory_usage_cache) decreases, while the usage of disk cache (memory_usage_cache) and the CPU time in the user mode (total_cpu_usage_usr) increase. For *vacuum*, the average system workload in 15 minutes (load_avg_15m) and the used memory (memory_usage_used) increases, while the normal free memory zones (zones_memory_normF) decrease. Note that 'total_cpu_usage_usr' is not a significant metric for *vacuum* because computations do not operate on the vacuum of the data blocks.

For *Too many indexes* and *Lock waits*, the index modifications and the waits for locks slow down the queries, so the resource consumption is lower as shown in Figures 2 (g) and (i). The number of update transactions (postgresql_transactions_update) is significant for *Too many indexes* and the number of locks (postgresql_locks_Locks) is significant for *Lock waits*, which consists with their root causes.

The results of the experiments show significant differences between anomaly types, which is explicable according to our analysis and validate the effectiveness of the monitoring metrics in DBPA.

6.2 Validation on Diversity

We have two groups of diversity validation experiments. One is to check whether the DBPA dataset has significant diversity. The other is to validate whether the diverse dataset is beneficial to the generalization abilities of the machine learning models for anomaly diagnosis. We focus on the background workloads and system environments for diversity validation, and we check the accuracy of diagnosis as the indicator.

First, we take the four background workloads for validation, where we mix the data of different system environments and uniformly split the training/test sets. We first train a multi-class classification model of *XGBoost* with the data from each workload and test its accuracy on every other workload. As a baseline, we also test the accuracy where we use the same workload for training and test. The training set contains 60 percent of the data and the test set contains the rest 40 percent. Then we train a model with the data from three scenarios and test the rest. The results are shown in Table 6. If the model is trained on one workload and tested on another, the accuracy is low, which validates the diversity of the data. If the model is trained on three workloads, the accuracy (column 'other' in Table 6) is much higher, which shows the benefits of the diversity to the generalization abilities of the models.

Then we focus on the four system environments, where we mix the data of different background workloads. The results are also shown in Table 6. We use the number of CPU cores and the memory size in GB as the identifiers of each environment. The accuracy is generally much lower than in the previous experiment, which indicates that the difference in the data is more significant for different system environments. The accuracy improvement due to the diversity of the training data is also more significant, which validates the benefits on generalization.

Table 6. Diversity Validation with Different Environments.

Test Workloads	Training TPC-C	TATP	Voter	Smallbank	Others
TPC-C	0.9499	0.8780	0.7718	0.7908	0.8876
TATP	0.8091	0.9458	0.8307	0.8312	0.9580
Voter	0.7925	0.8597	0.9502	0.8609	0.9434
Smallbank	0.7353	0.8208	0.8351	0.9500	0.911
Systems	32_128	32_256	64_128	64_256	Others
32_128	0.9879	0.1723	0.0554	0.1658	0.5123
32_256	0.2289	0.9809	0.2069	0.1905	0.7000
64_128	0.2913	0.4294	0.9943	0.4498	0.6089
64_256	0.2590	0.1557	0.3934	0.9822	0.6714

6.3 Validation on Compound Anomalies

In this part, we validate our generation algorithm for compound anomalies. First, we evaluate the prediction accuracy of our generation algorithm by comparing the similarity between the generated data and real data. Then we check the performance of diagnosis on both of the data.

As mentioned in Section 5.4, we have two groups of compound anomalies for the experiments. In the first group, we have *missing indexes*, *lock waits*, and *too many indexes*, which all belong to the query-triggered anomalies. We select *too many indexes + lock waits* for test, and train the machine learning models for the generation with the other two combinations. In the second group, we have *too many indexes*, *heavy workloads*, and *I/O saturation*, which belong to different anomaly categories. We select *too many indexes + heavy workloads* for test and use the rest for training.

To compare the similarity between the generated data and real data, we select the top-10 important metrics based on *DBSherlock* as representatives. These metrics have a significant difference between the normal and the anomalous, and they are supposed to be generated with machine learning models. We select *Random Forest* (RF) [4], *XGBoost* (XGB) [8], and *LightGBM* (LGBM) [28] as the machine learning models because they are popular tree-based ensemble models that are light-weighted and can be trained fast. We normalize the data and use mean square error (MSE) as the evaluation metric.

As shown in Tables 7 and 8, machine learning algorithms generally have low errors. For anomalies of the same category, *Random Forest* generally has a better performance. For those of different categories, *LightGBM* generally performs better. Based on this observation, we select the machine learning models for compound anomaly generation.

Then we verify the diagnosis performances on the generated data. We train *Decision Tree* classifiers based on the data with single anomalies and predict the categories of the generated compound anomalies and the real compound anomalies. We use precision, recall, and F1 as the metrics for the predictions. Precision refers to the ratio of true positive samples to the positive samples. Recall refers to the ratio of detected samples to the anomalous samples. F1 Score refers to the harmonic mean of precision and recall, which indicates the general performance. Similar results between the generated and the real data indicate a good generation performance.

The results are shown in Table 9. We observe that, on the diagnosis task, there is no significant difference between the generalized data and the real data, which validates the performance of our generation algorithm for compound anomalies.

Table 7. MSE of Compound Anomalies (Same Category).

Rank	Avg	Max	Min	RF	XGB	LGBM
1	1.6285	1.0894	3.4105	**1.0725**	1.6364	1.2711
2	3.1168	6.7090	2.7783	2.1952	3.8541	**2.1309**
3	0.6470	1.6380	0.7032	**0.0468**	0.0701	0.0645
4	0.6964	0.6568	1.6933	**0.0542**	0.0646	0.0563
5	1.1217	2.3334	**0.7316**	5.2703	44.0261	4.2365
6	1.8804	3.2446	1.3738	0.5557	**0.4802**	0.6490
7	0.9023	0.4746	3.9131	**0.0010**	0.0025	0.0027
8	0.0439	**0.0333**	0.1222	0.4325	0.4491	0.4304
9	1.0175	3.6749	**0.1494**	1.1214	1.0698	1.1196
10	1.4475	**0.3476**	4.8141	2.0628	1.1948	2.4158

Table 8. MSE of Compound Anomalies (Different Categories).

Rank	Avg	Max	Min	RF	XGB	LGBM
1	2.1442	2.9378	2.4112	2.0799	2.1418	**2.0231**
2	1.6362	2.7933	1.5770	1.3757	1.4091	**1.3370**
3	1.0522	2.2178	0.6786	0.0065	**0.0054**	0.0055
4	0.9560	0.6409	2.0743	0.0045	0.0021	**0.0019**
5	2.1493	4.1918	1.6205	1.6201	1.6306	**1.5903**
6	1.9014	2.0937	2.8961	**1.8216**	1.9533	1.9092
7	1.3060	0.2265	4.9390	**0.0070**	0.0072	0.0072
8	**2.5597**	3.7246	4.3570	4.2481	4.2516	4.2576
9	2.6503	2.9907	3.2142	2.1655	2.2320	**2.0205**
10	3.6089	4.6234	3.4771	**3.0750**	3.1590	3.1085

Table 9. Diagnosis with Generated Compound Anomalies.

Category	Test Type	P	R	F1
Same	Generated	0.8214	0.4792	0.6053
	Real	0.8384	0.5000	0.6264
Different	Generated	1.0000	0.4375	0.6087
	Real	0.9447	0.5176	0.6688

7 EVALUATION EXPERIMENTS

Based on our datasets, we evaluate some common algorithms to show the functionality of DBPA. The evaluation experiments include anomaly detection, diagnosis of single anomalies, diagnosis of compound anomalies, and evaluation on MySQL databases.

7.1 Anomaly Detection

For anomaly detection, we evaluate common unsupervised machine learning algorithms. These algorithms aim at distinguishing between normal and abnormal data instead of recognizing the

Table 10. Evaluation on Anomaly Detection.

Train %	40%			60%			80%		
	P	R	F1	P	R	F1	P	R	F1
IF	0.8074	0.5077	0.6234	0.8101	0.6065	0.6934	0.8046	0.5324	0.6404
OC-SVM	0.6654	0.9999	**0.7990**	0.6614	0.9998	**0.7961**	0.6686	1.0000	**0.8013**
LOF	0.7972	0.7529	0.7744	0.7968	0.7634	0.7797	0.7873	0.7585	0.7726
SVDD	0.6533	0.9441	0.7721	0.6489	0.9449	0.7693	0.6569	0.9515	0.7772

types of the anomalies. They are trained on normal data and tested on both normal and anomalous data. They are different from the diagnosis algorithms, which recognize the anomaly types and are trained on both normal and anomalous data.

We refer to the anomaly detection benchmark proposed by Emmott et al. [15], and select *Isolation Forest (IF)* [36], *One Class SVM (OC-SVM)* [56], *Local Outlier Factor (LOF)* [5], and *SVDD* [63] for evaluation. We do not evaluate the algorithms for database anomalies in the related work [23, 41, 66], because they only focus on every single monitoring metric. Instead, we evaluate the ability to distinguish the normal and anomalous data slices that contain all the metrics.

The input of the algorithm is the data slices from the samples in DBPA dataset. Each slice is a sequence of monitoring metrics for 10 timestamps, slightly shorter than the length of the anomalous samples, which is 12 timestamps. The output of the algorithm is a boolean value indicating whether the data is anomalous. We use precision, recall, and F1 Score as evaluation metrics, which are introduced in Section 6.3. We evaluate the anomaly detection algorithms with different settings of the training set ratio.

The results are shown in Table 10. The influence of the training set ratio is not significant. *One Class SVM* has the best overall performance with the highest F1 scores. *Isolation Forest* has a low recall, indicating a high possibility of omissions. *One Class SVM* and *SVDD* have low precision, indicating a high possibility of false alarms. *Local Outlier Factor* has medium precision and recall.

7.2 Diagnosis of Single Anomalies

We consider the diagnosis of single anomalies as a classification task, so we select some common machine-learning-based classification models [40], including *Logistic Regression* (*LR*), *Multilayer Perception*, *Decision Tree* (*DT*), *Random Forest* (*RF*), *XGBoost* (*XGB*), and *LightGBM* (*LGBM*). We do not use deep learning models with complex neural networks because it is difficult to select appropriate hyper-parameters. We also evaluate some diagnosis algorithms for database anomalies. We select *AutoMonitor* (*AutoM*) [23] and *DBSherlock* [66] in the related work for the experiments because they support arbitrary anomaly types. We do not evaluate *ISQUAD* [41] because it is designed for intermittent slow queries which have both spikes and level shifts [6, 52] in the monitoring metrics. We mainly focus on continuous performance regressions instead of intermittent ones, and we do not have spikes in the monitoring metrics for most of the anomalies.

AutoMonitor optimizes the K-Nearest-Neighbor (KNN) algorithm with a modified distance between the data slices, where each monitoring metric has a weight related to the anomaly types. In comparison to *AutoMonitor*, we also have the naive *KNN* for evaluation. Note that the performance of *AutoMonitor* and *KNN* can be strongly influenced by data imbalance. So we apply different weights to different anomaly types according to the data size. The diagnosis algorithm of *DBSherlock* requires a long time to build causal models for all the data, which is unaffordable for common users.

Table 11. Evaluation on Diagnosis Accuracy.

Ratio	40%	60%	80%
LR	0.8697	0.8769	0.8744
MLP	0.8801	0.8944	0.8987
DT	0.8874	0.9054	0.9088
RF	0.9441	0.9639	0.9767
XGB	**0.9733**	**0.9864**	**0.9897**
LGBM	0.9651	0.9829	0.9876
AutoM	0.7503	0.7714	0.8112
KNN	0.7404	0.7814	0.8112

So we do not evaluate *DBSherlock* in the full-data experiments, but have another experiment with different dataset sizes.

We evaluate the accuracy of classification with different settings of the training set ratio. The results are shown in Table 11. The training set ratio does not have a significant influence except for *AutoMonitor* and *KNN*. The tree-based ensemble models (*RF*, *XGB*, and *LGBM*) generally perform better than other algorithms. *AutoMonitor* generally performs better than naive *KNN*, but still worse than the machine learning models that we select.

We also present the diagnosis performance of each anomaly type with 60 percent training data. We use precision, recall, and F1 as the metrics, and the results are shown in Table 12. The relatively high scores indicate the effectiveness of the monitoring metrics. Some of the anomalies are naturally easy to recognize, but some are not. For the database environment anomalies and the workload amount anomalies, all the machine learning models have high precision, recall, and F1 scores. *AutoMonitor* generally performs worse than these models, but better than the naive *KNN*. For the query-triggered anomalies except for *lock waits*, the diagnosis performance is generally worse. *XGBoost* and *LightGBM* have the best performance, followed by *Random Forest*. The nearest-neighbor-based algorithms still generally perform worse than the machine learning models we select except for *Logistic Regression*. For *lock waits*, *KNN* fails because the metrics for locks are not significant among the various metrics. *AutoMonitor* performs better than *KNN* because it has higher weights for the important metrics. The other algorithms have better abilities to capture the feature of the metrics for locks, so they have high precision, recall, and F1 scores.

We have another evaluation experiment for *DBSherlock* with small datasets, because it needs to build one causal model for each piece of data and the time is unaffordable with the full dataset. We uniformly select 40, 80, and 120 pieces of data for each anomaly type. We use 70 percent of the data to construct the models for the algorithms, and the rest 30 percent for the accuracy test. We compare the accuracy and model construction time of *DBSherlock* (*DBS*) with *XGBoost* (*XGB*) and *LightGBM* (*LGBM*), which generally have a better performance in the previous experiment.

The results are shown in Table 13. With a heuristic diagnosis algorithm, *DBSherlock* has lower accuracy but significantly higher time cost than *XGBoost* and *LightGBM*. Besides, all the algorithms perform better with a larger dataset, and the improvements in *XGBoost* and *LightGBM* are more significant. This indicates that the machine learning algorithms need a large dataset to achieve a good performance.

For further discrimination between different machine learning algorithms, we test their generalization ability with the data from different scenarios for training and test. Specifically, we take 3 background workloads (TPC-C, TATP, and Voter) and 3 system environments (32_128, 32_256,

Table 12. Evaluation on Diagnosis of Each Anomaly Type.

	Small Shared Buffer			I/O Saturation			Concurrent Inserts		
	P	R	F1	P	R	F1	P	R	F1
LR	0.998	1.000	**0.999**	1.000	0.984	0.992	0.997	0.996	0.997
MLP	0.997	1.000	**0.999**	0.990	0.995	0.992	0.997	0.996	0.997
DT	0.994	0.998	0.996	0.945	1.000	0.972	0.999	0.990	0.995
RF	0.998	1.000	**0.999**	1.000	1.000	**1.000**	1.000	0.999	**1.000**
XGB	0.998	0.998	0.998	0.995	0.995	0.995	1.000	0.999	0.999
LGBM	0.996	1.000	0.998	1.000	0.995	0.997	1.000	1.000	**1.000**
AutoM	1.000	0.962	0.980	0.762	1.000	0.865	0.951	0.821	0.881
KNN	1.000	0.769	0.870	0.727	0.250	0.372	0.863	0.879	0.871
	Concurrent Commits			Heavy Workload			Missing Indexes		
	P	R	F1	P	R	F1	P	R	F1
LR	1.000	1.000	**1.000**	0.993	0.990	0.992	0.480	0.242	0.322
MLP	0.999	0.999	0.999	1.000	0.992	0.996	0.576	0.595	0.585
DT	0.998	0.999	0.998	0.984	0.988	0.986	0.890	0.325	0.476
RF	1.000	1.000	**1.000**	1.000	1.000	**1.000**	0.988	0.775	0.869
XGB	1.000	1.000	**1.000**	1.000	1.000	**1.000**	0.984	0.974	**0.979**
LGBM	1.000	1.000	**1.000**	1.000	0.998	0.999	0.985	0.954	0.970
AutoM	0.849	0.915	0.881	0.883	0.914	0.898	0.303	0.429	0.355
KNN	0.893	0.870	0.881	0.907	0.672	0.772	0.347	0.169	0.227
	Too Many Indexes			Vacuum			Lock Waits		
	P	R	F1	P	R	F1	P	R	F1
LR	0.772	0.871	0.819	0.698	0.869	0.774	1.000	1.000	**1.000**
MLP	0.777	0.899	0.834	0.793	0.784	0.788	1.000	1.000	**1.000**
DT	0.828	0.861	0.844	0.742	0.981	0.845	0.989	1.000	0.995
RF	0.865	0.991	0.924	0.899	0.995	0.944	1.000	1.000	**1.000**
XGB	0.933	0.980	**0.956**	0.987	0.992	**0.989**	1.000	1.000	**1.000**
LGBM	0.915	0.979	0.946	0.978	0.993	0.985	1.000	1.000	**1.000**
AutoM	0.891	0.880	0.886	0.645	0.544	0.590	1.000	0.556	0.714
KNN	0.827	0.920	0.871	0.639	0.818	0.717	0.000	0.000	0.000

Table 13. Evaluation on Different Dataset Sizes.

Size	40		80		120	
	Acc	Time (s)	Acc	Time (s)	Acc	Time (s)
XGB	0.811	2.0	0.822	2.6	0.889	2.7
LGBM	0.833	1.2	0.852	2.6	0.894	3.1
DBS	0.593	10307.8	0.616	20820.8	0.654	31080.0

and 64_128) as introduced in Section 6.2, 9 scenarios in total, for training. We take the scenario with Smallbank and 64_256 for test. The results of accuracy are shown in Table 14. There are more significant varieties in the results than in the previous experiments. *LGBM* has the best accuracy,

Table 14. Evaluation on Different Scenarios.

	LR	MLP	DT	RF	XGB	LGBM
Acc	0.7023	0.6134	0.7453	0.8402	0.8122	**0.8641**

Table 15. Case Study of Anomaly Diagnosis.

Case #	LGBM	AutoM		KNN	
	SHAP Top-3	DistT	DistF	DistT	DistF
1	0.2626	0.0780	0.0620	0.6431	0.4511
2	0.2609	0.1185	0.0963	0.9211	0.9143
3	0.2818	0.0823	0.0674	0.6433	0.5576
4	0.2640	0.0241	0.0237	0.1890	0.1732
5	0.2707	0.2089	0.2046	2.0391	1.7151

followed by *RF* and *XGB*. *MLP* performs the worst, indicating lower abilities of generalization for this task.

To explain why the machine learning models generally have higher accuracy than traditional algorithms, we analyze some cases where *AutoM/KNN* misclassify but *LGBM* does not. In Table 15, we show 5 cases from the anomalies where *LGBM* has substantially higher accuracy than *AutoM/KNN*. In cases #1 and #2, the correct label is *Missing Indexes* but misclassified by *AutoM* and *KNN* as *Vacuum*. In cases #3 and #4, the correct label is *Vacuum* but misclassified as *Missing Indexes*. In case #5, the correct label is *Lock Waits* but misclassified as *I/O Saturation*.

For *LGBM*, its tree nodes split based on the discriminative features. We compute the normalized SHAP importance [38] of each metric, which measures its marginal contribution to the model. The result shows that the top-3 metrics in *LGBM* make more than 1/4 contributions among the 110 metrics. For *AutoM/KNN*, various metrics are treated with equal importance. We show the distance from each case point to its closest training samples with the correct label (DistT) and misclassified label (DistF). The result shows only slight differences between the distances, indicating that the important metrics captured by *LGBM* are neglected by *AutoM* and *KNN*.

We have the following conclusions from the experiments. First, *XGBoost* and *LightGBM*, two machine learning models, generally perform better than *DBSherlock*, *AutoMonitor*, and the other machine learning algorithms that we select. Second, our benchmark supports both machine learning and heuristic algorithms. Third, the difficulty of diagnosis each anomaly type is different, which is also related to the diagnosis algorithm. Finally, the machine learning algorithms need a large dataset to achieve a good performance. This proves that DBPA can help to improve the performance of anomaly diagnosis in transactional databases by making machine learning available for this task.

7.3 Diagnosis of Compound Anomalies

To evaluate different algorithms for compound anomalies diagnosis, we employ a one-vs-rest model for each single anomaly type. The models are trained with single anomalies and normal data, and tested with compound anomalies and normal data. We only evaluate the selected machine learning algorithms, because *AutoMonitor* and *KNN* do not support one-vs-rest for compound anomalies and *DBSherlock* has low time efficiency for the full dataset.

Table 16. Evaluation on Compound Anomalies Diagnosis.

	Same Category			Different Categories		
Model	P	R	F1	P	R	F1
LR	1.0000	0.5262	0.6896	0.9642	0.4468	0.6106
MLP	0.9978	0.5017	0.6677	0.8195	0.5322	0.6454
DT	0.9670	0.6049	0.7442	0.8379	0.5400	**0.6568**
RF	0.9792	0.6032	**0.7465**	1.0000	0.3501	0.5186
XGB	0.8711	0.6032	0.7128	0.8605	0.1265	0.2205
LGBM	0.8762	0.5887	0.7043	1.0000	0.1265	0.2245

Table 17. Evaluation with MySQL-based Dataset.

Test Env.	32_128	32_256	64_128	64_256
LR	0.8100	0.8862	0.8563	0.9264
MLP	0.7590	0.7467	0.8888	0.8872
DT	0.9032	0.7667	0.8018	**0.9804**
RF	0.9176	**0.9712**	0.9027	0.9331
XGB	0.9583	0.9150	**0.9552**	0.8512
LGBM	**0.9670**	0.8713	0.9444	0.9258
AutoM	0.5118	0.3628	0.5133	0.5767
KNN	0.6386	0.4867	0.5472	0.3378

We have two groups of compound anomalies, one belongs to the same category and the other belongs to different categories, which has the same settings as Section 6.3. The evaluation metrics are the precision, recall, and F1 score of the model for each anomaly type.

As shown in Table 16, the diagnosis of compound anomalies generally has high precision but low recall, which indicates that most positive samples are correct, but there are many omissions. For the compound anomalies of the same category, there is not a significant difference between the performance of different machine learning algorithms. *Decision Tee* and *Random Forest* generally perform better, and *MLP* performs the worst. For the compound anomalies of different categories, *Decision Tree* and *MLP* generally perform better. The tree-based ensemble models (*RF*, *XGB*, and *LGBM*) perform worse, indicating that they may not be suitable for this task.

7.4 Evaluation on MySQL

The previous experiments are based on the open-source PostgreSQL databases. To illustrate the generality of DBPA, we extend DBPA to another popular open-source DBMS, MySQL, for the evaluation of diagnosis algorithms. The generation of the dataset follows the same approach as introduced in Section 4, except that we do not include the anomaly type *Vacuum* and the reason is mentioned in Section 3.1.

We take an experiment that is similar to Section 6.2. We use the data from three system environments for training and the rest one for test. We test the classification accuracy of the common diagnosis algorithms introduced in Section 7.2.

The results are shown in Table 17, where the headers show the system environment of test data and the values shows the classification accuracy. The accuracy is generally high for the machine learning models but still distinguishable. Thus, we prove that: *(1)* different anomalies are

distinguishable in the MySQL-based dataset; *(2)* the evaluation of common diagnosis algorithms is supported.

8 CONCLUSION

In this work, we propose DBPA, a benchmark for the common performance anomalies in transactional database systems. It can help to apply machine learning techniques to the diagnosis of database performance anomalies. To ease the burden of benchmarking different anomaly detection algorithms, we first need to generate the anomaly data, which is the main effort of our work. There is no open-source dataset available and it is difficult to collect such data from real-life DBs. To the best of our knowledge, this is the first work to propose specific reproduction procedures to generate data for database performance anomalies. In DBPA, we select nine common anomaly types categorized by the influence factors of the database performance. We propose a general reproduction procedure for each anomaly category, and the specific procedure for each anomaly type. We describe the dataset construction procedure and propose a generation algorithm for compound anomalies. DBPA meets the requirements of consistency, correctness, diversity, support for compound anomalies, extensiveness, and effectiveness. With DBPA, users can easily extend new anomaly types, generate a dataset in new environments, or evaluate different algorithms for anomaly detection and diagnosis. Empirical studies show that the tree-based ensemble machine learning models perform better than the heuristic algorithms for database anomaly diagnosis, which indicates that our work can help to improve the diagnosis performance in modern transactional database systems.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (NSFC) (No. 61832001, U22B2037) and ZTE-PKU joint program (HC-CN-20220614004). Bin Cui is the corresponding author.

REFERENCES

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. ACM, 1009–1024.

[2] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2016. Regularized Cost-Model Oblivious Database Tuning with Reinforcement Learning. *Trans. Large Scale Data Knowl. Centered Syst.* 28 (2016), 96–132.

[3] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. 2011. Adapting microsoft SQL server for cloud computing. In *ICDE*. IEEE Computer Society, 1255–1263.

[4] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32.

[5] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. 2000. LOF: Identifying Density-Based Local Outliers. In *SIGMOD*. ACM, 93–104.

[6] Wei Cao, Yusong Gao, Bingchen Lin, Xiaojie Feng, Yu Xie, Xiao Lou, and Peng Wang. 2018. TcpRT: Instrument and Diagnostic Analysis System for Service Quality of Cloud Databases at Massive Scale in Real-time. In *SIGMOD*. ACM, 615–627.

[7] Pengfei Chen, Yong Qi, Pengfei Zheng, and Di Hou. 2014. CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *INFOCOM*. IEEE, 1887–1895.

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *KDD*. ACM, 785–794.

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

[10] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[11] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. 2020. SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns. In *SIGMOD*. ACM, 2331–2345.

[12] Korry Douglas and Susan Douglas. 2003. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases.* SAMS publishing.

[13] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257.
[14] Dejan Dundjerski and Milo Tomasevic. 2020. Automatic database troubleshooting of Azure SQL Databases. *IEEE Transactions on Cloud Computing* (2020).
[15] Andrew F Emmott, Shubhomoy Das, Thomas Dietterich, Alan Fern, and Weng-Keen Wong. 2013. Systematic construction of anomaly detection benchmarks from real data. In *SIGKDD workshop on outlier detection and description*. ACM, 16–21.
[16] Ayat Fekry, Lucian Carata, Thomas F. J.-M. Pasquier, Andrew Rice, and Andy Hopper. 2020. To Tune or Not to Tune?: In Search of Optimal Configurations for Data Analytics. In *KDD*. ACM, 2494–2504.
[17] Jia-Ke Ge, Yanfeng Chai, and Yunpeng Chai. 2021. WATuning: A Workload-Aware Tuning System with Attention-Based Deep Reinforcement Learning. *J. Comput. Sci. Technol.* 36, 4 (2021), 741–761.
[18] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, and Amit Levi. 2020. Sentinel: Universal Analysis and Insight for Data Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2720–2733.
[19] Jing Han, Tong Jia, Yifan Wu, Chuanjia Hou, and Ying Li. 2021. Feedback-Aware Anomaly Detection Through Logs for Large-Scale Software Systems. *ZTE Communications* 19, 3 (2021), 88–94.
[20] Songqiao Han, Xiyang Hu, Hailiang Huang, Mingqi Jiang, and Yue Zhao. 2022. ADBench: Anomaly Detection Benchmark. In *NeurIPS*. 32142–32159.
[21] Shiyue Huang, Yanzhao Qin, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. 2023. Survey on performance optimization for database systems. *Sci. China Inf. Sci.* 66, 2 (2023).
[22] Vincent Jacob, Fei Song, Arnaud Stiegler, Bijan Rad, Yanlei Diao, and Nesime Tatbul. 2021. A Demonstration of the Exathlon Benchmarking Platform for Explainable Anomaly Detection. *Proc. VLDB Endow.* 14, 12 (2021), 2827–2830.
[23] Lianyuan Jin and Guoliang Li. 2021. AI-based Database Performance Diagnosis. *Journal of Software* 32, 3 (2021), 845–858.
[24] Michael I Jordan and Tom M Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349, 6245 (2015), 255–260.
[25] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woon-Hak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proc. VLDB Endow.* 13, 1 (2019), 57–70.
[26] Prajakta Kalmegh, Shivnath Babu, and Sudeepa Roy. 2017. Analyzing Query Performance and Attributing Blame for Contentions in a Cluster Computing Framework. *CoRR* abs/1708.08435 (2017).
[27] Prajakta Kalmegh, Shivnath Babu, and Sudeepa Roy. 2019. iQCAR: inter-Query Contention Analyzer for Data Analytics Frameworks. In *SIGMOD*. ACM, 918–935.
[28] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *NIPS*. 3146–3154.
[29] Poonyanuch Khumnin and Twittie Senivongse. 2017. SQL antipatterns detection and database refactoring process. In *SNPD*. IEEE Computer Society, 199–205.
[30] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.
[31] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *SIGMOD Conference*. ACM, 1667–1683.
[32] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Sci. Eng.* 6, 1 (2021), 86–101.
[33] Alexander Lavin and Subutai Ahmad. 2015. Evaluating Real-Time Anomaly Detection Algorithms - The Numenta Anomaly Benchmark. In *ICMLA*. IEEE, 38–44.
[34] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.
[35] Zhichao Li, Li Tian, and Xuefeng Yan. 2022. A novel deep quality-supervised regularized autoencoder model for quality-relevant fault detection. *Sci. China Inf. Sci.* 65, 5 (2022), 1–3.
[36] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation Forest. In *ICDM*. IEEE Computer Society, 413–422.
[37] Ping Liu, Shenglin Zhang, Yongqian Sun, Yuan Meng, Jiahai Yang, and Dan Pei. 2020. FluxInfer: Automatic Diagnosis of Performance Anomaly for Online Database System. In *IPCCC*. IEEE, 1–8.
[38] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS*. 4765–4774.
[39] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD*. ACM, 2530–2542.
[40] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *SIGMOD*. ACM,

1248–1261.

[41] 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. 2020. Diagnosing Root Causes of Intermittent Slow Queries in Large-Scale Cloud Databases. *Proc. VLDB Endow.* 13, 8 (2020), 1176–1189.

[42] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning. In *SIGMOD*. ACM, 1253–1267.

[43] Jeffrey C. Mogul and John Wilkes. 2019. Nines are Not Enough: Meaningful Metrics for Clouds. In *HotOS*. ACM, 136–141.

[44] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*. ACM, 301–312.

[45] Barzan Mozafari, Carlo Curino, and Samuel Madden. 2013. DBSeer: Resource and Performance Prediction for Building a Next Generation Database Cloud. In *CIDR*. www.cidrdb.org.

[46] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *NSDI*. USENIX Association, 353–366.

[47] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. 2005. Continuous resource monitoring for self-predicting DBMS. In *MASCOTS*. IEEE Computer Society, 239–248.

[48] Xiaolong Pan, Weiming Wu, and Yonghao Gu. 2011. Study and optimization based on MySQL storage engine. In *Advances in Multimedia, Software Engineering and Computing Vol. 2*. Springer, 185–189.

[49] John Paparrizos, Yuhao Kang, Paul Boniol, Ruey S. Tsay, Themis Palpanas, and Michael J. Franklin. 2022. TSB-UAD: An End-to-End Benchmark Suite for Univariate Time-Series Anomaly Detection. *Proc. VLDB Endow.* 15, 8 (2022), 1697–1711.

[50] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*. www.cidrdb.org.

[51] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *ICDE*. IEEE, 600–611.

[52] Anthony N Pettitt. 1979. A non-parametric approach to the change-point problem. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 28, 2 (1979), 126–135.

[53] Robert B. Ross, George Amvrosiadis, Philip H. Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Robert W. Robey, Dana Robinson, Bradley W. Settlemyer, Galen M. Shipman, Shane Snyder, Jérome Soumagne, and Qing Zheng. 2020. Mochi: Composing Data Services for High-Performance Computing Environments. *J. Comput. Sci. Technol.* 35, 1 (2020), 121–144.

[54] Durgesh Samariya and Jiangang Ma. 2022. A New Dimensionality-Unbiased Score for Efficient and Effective Outlying Aspect Mining. *Data Sci. Eng.* 7, 2 (2022), 120–135.

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

[56] Bernhard Schölkopf, John C. Platt, John Shawe-Taylor, Alexander J. Smola, and Robert C. Williamson. 2001. Estimating the Support of a High-Dimensional Distribution. *Neural Comput.* 13, 7 (2001), 1443–1471.

[57] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proc. VLDB Endow.* 7, 12 (2014), 1035–1046.

[58] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10, 4 (2016), 445–456.

[59] Shudi Shao, Zhengyi Qiu, Xiao Yu, Wei Yang, Guoliang Jin, Tao Xie, and Xintao Wu. 2020. Database-Access Performance Antipatterns in Database-Backed Web Applications. In *ICSME*. IEEE, 58–69.

[60] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *CoRR* abs/1801.05643 (2018).

[61] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Jose Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *SIGMOD*. ACM, 205–219.

[62] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[63] David M. J. Tax and Robert P. W. Duin. 2004. Support Vector Data Description. *Mach. Learn.* 54, 1 (2004), 45–66.

[64] Alexander Thomasian. 1994. On a More Realistic Lock Contention Model and Its Analysis. In *ICDE*. IEEE Computer Society, 2–9.

[65] Wenhu Tian, Patrick Martin, and Wendy Powley. 2003. Techniques for automatically sizing multiple buffer pools in DB2. In *CASCON*. IBM, 294–302.
[66] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *SIGMOD*. ACM, 1599–1614.
[67] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD*. ACM, 415–432.
[68] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 15, 9 (2022), 1808–1821.
[69] Xin Zhang, Siyuan Lu, Shui-Hua Wang, Xiang Yu, Su-Jing Wang, Lun Yao, Yi Pan, and Yu-Dong Zhang. 2022. Diagnosis of COVID-19 Pneumonia via a Novel Deep Learning Architecture. *J. Comput. Sci. Technol.* 37, 2 (2022), 330–343.
[70] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD*. ACM, 2102–2114.
[71] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. In *SIGMOD Conference*. ACM, 631–645.
[72] Zipiao Zhao, Yongli Zhao, Boyuan Yan, and Dajiang Wang. 2022. Auxiliary Fault Location on Commercial Equipment Based on Supervised Machine Learning. *ZTE Communications* 20, S1 (2022), 7–15.
[73] Yuanhong Zhong, Xia Chen, Jinyang Jiang, and Fan Ren. 2022. Reverse erasure guided spatio-temporal autoencoder with compact feature representation for video anomaly detection. *Sci. China Inf. Sci.* 65, 9 (2022), 1–3.
[74] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2022. Database Meets Artificial Intelligence: A Survey. *IEEE Trans. Knowl. Data Eng.* 34, 3 (2022), 1096–1116.
[75] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*. ACM, 338–350.

Received July 2022; revised October 2022; accepted November 2022