Hulk: Exploring Data-Sensitive Performance Anomalies in DBMSs via Data-Driven Analysis

ZHIYONG WU, Tsinghua University, China
JIE LIANG*, Beihang University, China
JINGZHOU FU, Tsinghua University, China
MINGZHE WANG, Tsinghua University, China
YU JIANG*, Tsinghua University, China

Performance is crucial for database management systems (DBMSs), and they are always designed to handle ever-changing workloads efficiently. However, the complexity of the cost-based optimizer (CBO) and its interactions can introduce implementation errors, leading to data-sensitive performance anomalies. These anomalies may cause significant performance degradation compared to the expected design under certain datasets. To diagnose performance issues, DBMS developers often rely on intuitions or compare execution times to a baseline DBMS. These approaches overlook the impact of datasets on performance. As a result, only a subset of performance issues is identified and resolved.

In this paper, we propose Hulk to automatically explore these data-sensitive performance anomalies via data-driven analysis. The key idea is to identify performance anomalies as the dataset evolves. Specifically, Hulk estimates a reasonable response time range for each data volume to pinpoint performance cliffs. Then performance cliffs are checked for deviations from expected performance by finding a reasonable plan that aligns with performance expectations. We evaluate Hulk on six widely-used DBMSs, namely MySQL, MariaDB, Percona, TiDB, PostgreSQL, and AntDB. Hulk totally reports 135 anomalies, with 129 have been confirmed as new bugs, including 14 CVEs. Among them, 94 are data-sensitive performance anomalies.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Information systems → Query optimization**; **Structured Query Language**;

Additional Key Words and Phrases: Database Management Systems; Performance Testing; Fuzzing

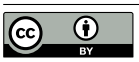
ACM Reference Format:
Zhiyong Wu, Jie Liang, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2025. Hulk: Exploring Data-Sensitive Performance Anomalies in DBMSs via Data-Driven Analysis. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA096 (July 2025), 22 pages. https://doi.org/10.1145/3728973

1 Introduction

Database management systems (DBMSs) are the infrastructure for efficient management of data [1, 43]. DBMSs are constantly striving for better performance, which is reflected in the time it takes to fetch, analyze, and update data [13, 29]. Generally, data-intensive applications often operate under shifting workloads, where data may change or accumulate over time [19]. This requires DBMSs to be resilient to data changes to maintain smooth operation.

*Jie Liang and Yu Jiang are the corresponding authors.

Authors' Contact Information: Zhiyong Wu, Tsinghua University, KLISS, BNRist, School of Software, Beijing, China, wzy19990306@gmail.com; Jie Liang, Beihang University, Beijing, China, liangjie.mailbox.cn@gmail.com; Jingzhou Fu, Tsinghua University, KLISS, BNRist, School of Software, Beijing, China, fuboat@outlook.com; Mingzhe Wang, Tsinghua University, KLISS, BNRist, School of Software, Beijing, China, wmzhere@gmail.com; Yu Jiang, Tsinghua University, KLISS, BNRist, School of Software, Beijing, China, jiangyu198964@126.com.



ACM 2994-970X/2025/7-ARTISSTA096
https://doi.org/10.1145/3728973

Unfortunately, due to the complexity in implementing an optimizer and the system scale, a DBMS may not always perform as expected under certain data conditions. To achieve the efficient processing of diverse datasets, a DBMS typically uses the optimizer [26, 29]. It transforms a query into a plan, which outlines a sequence of execution steps to access data[41]. Most current DBMSs use a cost-based optimizer (CBO) [33]. For each SQL query, it first generates amounts of alternative plans, then estimates the cost of each plan and selects the optimal one for execution (e.g., whose time cost is the lowest). A CBO typically consists of three components: cardinality estimation (CE), cost model (CM), and plan enumeration (PE). The components rely on many complex algorithms in their functioning, which are closely linked to the dataset. Specifically, CE uses statistics of data to calculate the cardinalities for each operation in the plan. CM maps the current state of the DBMS and estimated cardinalities to the cost of executing a plan. PE selects the optimal plan with the lowest cost for execution in the DBMS.

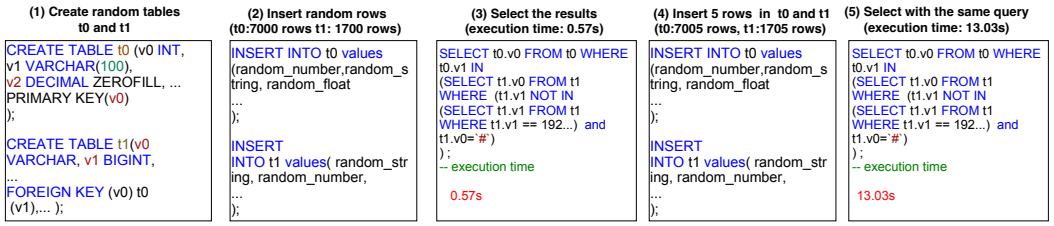


Fig. 1. A data-sensitive performance anomaly found in Percona. When executing one query with 7000 rows in `t0` and 1700 rows in `t1`, Percona takes about 0.57s. However, after inserting 5 rows in table `t0` and `t1`, respectively, Percona takes about 13.03s to execute the same query. It causes 2185% performance degradation.

The optimization process of a DBMS is greatly affected by the data stored in it, making its performance more susceptible to changes in data volumes. Any implementation errors may cause performance to fluctuate abnormally when the dataset changes. For example, when the dataset changes, even small estimation errors in CE will enlarge along with other factors such as data distribution, and propagate into CM and PE [54]. Consequently, these errors may lead to *performance cliffs* (i.e., a significant performance degradation such as a sudden increase in response time). Even with the most popular DBMSs like MySQL, response times can fluctuate significantly with small changes in the dataset[40]. These performance anomalies arise from the creation of highly expensive plans, which we refer to as **data-sensitive performance anomalies**. Figure 1 illustrates a data-sensitive performance anomaly in Percona, which results in a 21-fold performance degradation with only inserting 5 rows in two tables, respectively. The process unfolds in five steps. First, it creates two tables and inserts some random rows in step ① and step ②, table `t0` contains 7000 rows, and table `t1` contains 1700 rows. Then, Percona executes the query with the inserted rows and takes 0.57 seconds to return the results in ③. However, after inserting 5 rows in table t0 and t1 in step ④, Percona takes about 13.03s to execute the same query in step ⑤. This issue arises due to the implementation errors in the PE, which does not choose the optimal plan to execute in step ⑤.

Data-sensitive performance anomalies can lead to severe damage. They can cause significant performance degradation compared to the expected design. These anomalies not only degrade user experience through frustrating delays and unreliability in system interactions but also impede productivity by decelerating data processing and analysis workflows, ultimately leading to economic losses. For example, e-commerce platforms are a significant application scenario for DBMSs, typically processing over 10,000 transactions per second to ensure sales volume and revenue during shopping festivals (e.g., Black Friday [15]). However, the data-sensitive performance anomalies may inflate response latency by more than an order of magnitude, which may cause the entire platform system to be paralyzed or even out of service, potentially leading to significant economic losses.

However, conventional techniques test DBMS performance with fixed datasets, potentially ignoring data-sensitive performance anomalies. Specifically, conventional methods (e.g., system validation tests) compare the response time for predefined queries and data with empirical values as the baseline. Latest fuzzers like Amoeba [37] and APOLLO [30], also compare the response time of a query either with the time for a semantically equivalent query or in a previous version of the target DBMS with the fixed data. Consequently, they can hardly detect data-sensitive performance anomalies. Additionally, *CERT*[1] [11] tests cardinality estimation to find performance anomalies. However, it still cannot detect data-sensitive performance anomalies due to the fixed dataset.

To efficiently detect data-sensitive performance anomalies caused by optimizers, the intuition is to execute queries with a changed dataset and check whether the DBMS generates a significantly costly plan. However, it is challenging to determine whether the generated plan has low performance for each combination of data and query because it requires accurately estimating cardinality and building the cost model for different datasets. It can be a costly and complex process, which is almost equivalent to implementing a perfect optimizer [33].

In this paper, we propose a data-driven analysis approach to find data-sensitive performance anomalies in DBMS. It identifies performance anomalies by *sampling the historical performance of the tested DBMS alongside a comparable DBMS to estimate a threshold range, without necessarily aiming for an ideal DBMS*. The approach has two stages: ① First, it synthesizes data-sensitive queries from metadata derived from randomly generated tables. After that, it executes queries in the target DBMS and a comparable DBMS with increased data volume. For each volume, it estimates the range of reasonable response times to capture *performance cliffs*. If the actual response time falls outside this range, a performance cliff is detected and a potential data-sensitive anomaly is identified. ② Second, the performance cliff detected in the first stage is validated as a performance anomaly by checking if there is a significant difference in the uniform cost of plans and if the comparable DBMS can supply a better plan. The availability of a better plan confirms the performance cliff as a data-sensitive performance anomaly.

We implemented the approach in Hulk and evaluated it on six widely-used DBMSs: MySQL, MariaDB, Percona, TiDB, AntDB, and PostgreSQL. Hulk reports a total of 135 anomalies, with 129 anomalies have been confirmed as new bugs, including 14 CVEs assigned. Among them, 94 are confirmed as data-sensitive performance anomalies, and 35 are confirmed as crash bugs. In addition, we compare Hulk with the state-of-the-art DBMS validation tools in industry, including both DBMS performance testing tool APOLLO [30] and SQLancer^{CERT} [11], as well as DBMS fuzzing tools Squirrel [60]. The 24-hour result shows that Hulk covered 17%, 21%, and 6% more branches, and found 50, 52, and 58 more bugs than APOLLO, SQLancer ^{CERT}, and Squirrel, respectively. Particularly, Hulk found 37 and 35 more performance anomalies than APOLLO and SQLancer ^{CERT} in 24 hours. In summary, our paper makes the following contributions:

- We find the data sensitive performance anomalies of DBMSs are common and can do great damage to modern data-intensive applications, but the state-of-the-art testing techniques pay little attention to the problem.
- We propose data-driven analysis to detect data sensitive performance anomalies, consisting of data-sensitive query synthesis and sampling-based performance estimation. It first estimates a range of reasonable response times for growth data volume to capture the *performance cliff*, and then validates the performance anomaly by plan cost comparison.
- We implement our approach in Hulk and detect 135 anomalies in six widely-used DBMSs. Among them, 129 are confirmed as new bugs, including 14 CVEs assigned. Among them, 94 data-sensitive performance anomalies and 35 are crash bugs.

[1]It is implemented in SQLancer and will be referred to as SQLancer^{CERT} hereafter.

2 Data-Sensitive Performance Anomaly

Basic Concepts. *Database Management Systems (DBMS)* are software systems designed to store, retrieve, query, and manage data. *Structured Query Language (SQL)* is a declarative query language that mediates information exchange between users and the DBMS. Specifically, users describe their demands with it in a query. A *clause* is a built-in function that processes the database table and gives intermediate results. A *query* is a request to retrieve or update information (e.g., adding or removing data) from a database, which may contain one or several clauses. DBMSs transform the query into a *plan*, which is a sequence of operations that the DBMS needs to follow to execute. *Response time* can be used to measure the performance of a DBMS, which is the time interval between when the DBMS receives a query and when the DBMS returns the result of the query.

Definition and Severity. A *data-sensitive performance anomaly* refers to the performance dramatic fluctuation of a SQL query when there are only small changes in the amount of data. DBMSs are designed to access and manage huge amounts of data. They are the infrastructure of data-intensive applications, and their performance directly affects the overall efficiency of these applications. As the scenarios of these applications are enriched and the frequency of use increases, they tend to accumulate a large amount of data. To keep them running smoothly despite the growth in data volume, a fundamental requirement of DBMS design is to maintain stable performance as the dataset changes. Unfortunately, due to the complexity of the optimizer, *data-sensitive performance anomalies may occur when the data set changes*. These anomalies typically manifest as sudden and unpredictable changes in the DBMS's performance, especially as the volume of data escalates.

Data-sensitive anomalies pose serious risks as they present significant obstacles to maintaining efficient and reliable DBMSs. First, they can cause significant performance degradation compared to the expected design. The example in Figure 1 that the performance anomaly in Percona resulted in 2185% performance degradation with 5 rows increase. The performance degradation leads to slower data processing, potential system failures, and a worse experience for users. Second, they are widely prevalent across various DBMSs. As databases grow in size and complexity, the variety of data and the complexity of queries that operate on this data also increase. This complexity makes it challenging to predict how different data types and structures will interact with the database's query-processing algorithms. Thirdly, detecting these anomalies can be particularly difficult. They often only appear under certain conditions, such as specific data sizes or types that might not be covered by standard testing procedures. This makes them hard to spot and fix.

Root Cause. Data-sensitive performance anomalies mainly arise from implementation errors due to the complexity of the optimizer and its interactions with other components. Almost all current DBMSs adopt CBO for performance improvement [33]. It transforms a query into a minimal-cost execution plan by estimating the cost (i.e., cardinality) of the operation and using complicated strategies to find a join order. CBO normally has three components, namely CE, CM, and PE [34]. During the optimization, CE uses statistics of data to estimate the number of tuples (i.e., cardinalities) for each basic SQL operation. CM is a complex function that maps the current DBMS state and estimated cardinalities to the plan cost. PE calculates the cost of each execution plan based on CE and CM and selects the optimal plan with the lowest cost for query execution.

The data-sensitive performance anomalies may be caused by errors in any component of CBO and its interactions. First, the accuracy of CE is affected by the dataset and the complexity of the query. As the volume grows, CE will approximate the distribution of data. Besides, a multi-join query may be related to multiple tables. The correlations between columns of different tables increase the difficulty in estimating the cardinality of a join operator. For a complex query with multiple clauses, the estimated deviation propagates and amplifies from the leaves of the plan to the root. Second, CM will also be affected by the data and the state of the database. Any error in CE will also propagate to CM. Finally, finding the optimal solution has proven to be NP-complete [25]. When the volume of data grows, PE sometimes has to adopt heuristic algorithms (e.g., the genetic algorithm

in PostgreSQL) to sacrifice accuracy to satisfy a reasonable optimization time. The operations of the chosen plan will be sent to the executor for execution.

Besides, CBO is a pivotal element within the DBMS, intricately connected with multiple system components to ensure optimal query execution. By collaborating with the query parser, statistics, metadata, and the execution engine, CBO tailors plans to the database's current state and workload characteristics. Therefore, the optimization process is highly influenced by factors such as data distribution, column correlation, join relationships, and search principles. Any error in the process can lead to unreasonable plans, reflected as data-sensitive performance anomalies.

Motivating Example. Figure 2 shows a real data-sensitive performance anomaly found in MySQL, which can cause severe performance degradation. In the beginning, the database has three tables, namely `Staff`, `Lawyer`, and `Salary`. They have 1000, 50000, and 1000 records, respectively. When executing the query "SELECT * FROM Staff, Lawyer, Salary WHERE Staff.v0=Lawyer.v0 and Lawyer.v2=Salary.v2", MySQL takes about 1.75 seconds. However, when inserting 10 rows into table `Staff` (the green region), MySQL takes about 67.43 seconds. **With a 0.1% increase in the amount of data, the response time increases by 3753%.** After we reported this anomaly to the developers and got their confirmation, they expressed surprise: "*We never expected that such a small change in data volume could cause such serious performance issues.*"

To investigate the cause of the significant performance degradation, we extract the execution plan of this query with `EXPLAIN ANALYZE` commands in MySQL. As Figure 2 shows, when executing the query with original data, MySQL first employs the hash join between table `Staff` and `Lawyer` with the condition "`Staff`.v0 = `Lawyer`.v1". The size of the hash join is 1000*50000, and the results will be filtered into a temp table with 1000 records. Then the second hash join will be performed on the temp table and table `Salary`. The size of the second hash join is 1000*1000. Consequently, the total number of rows scanned is 1000*50000+1000*1000 = 60,000,000.

After inserting 10 rows into table `Staff`, MySQL changes the plan. It first employs a hash join between table `Staff` and `Salary` with no filter condition. The operation produces a temp table with size 1010*1000. And then MySQL calculates hash join between the temp table and `Salary`. The size of the second hash join is 1010*1000*50000. Therefore, the total number of rows scanned is 1010*1000 + 1010*1000*50000 = 50,501,010,000, which is 842 times more than the original one. Consequently, MySQL takes 38 times longer after the rows are inserted. As a comparison, we also

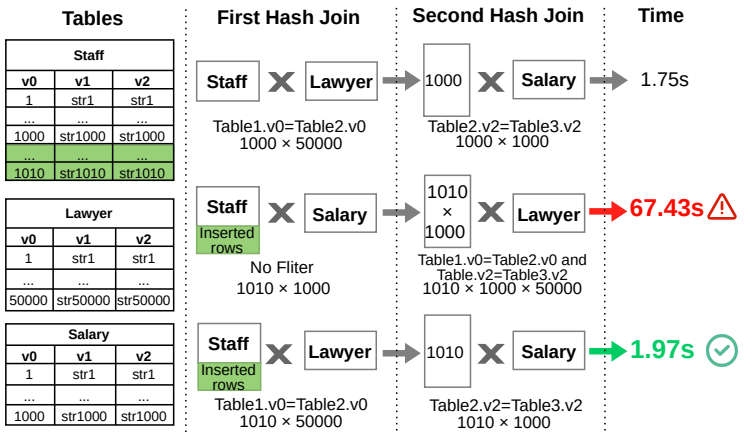


Fig. 2. A data-sensitive performance anomaly in MySQL. When executing one query with 1000 records in `Staff`, MySQL takes about 1.75s. However, when inserting 10 rows into `Staff`, MySQL takes about 67.43s. With a **0.1%** increase in the amount of data, the response time increases by **3753%**.

feed the same query to MariaDB. It always executes the first plan, regardless of whether the 10 rows are inserted. As a result, it spends 1.97s after inserting the rows.

The root cause of the anomaly is that when the number of records in the 'Staff' table increases to 1010, MySQL's CE incorrectly estimates that the temporary result of 'Staff' joining 'Lawyer' will exceed the temporary join_buffer_size, due to an error in the estimation algorithm. Therefore, it first performs a hash join on two small tables, which causes performance degradation. As the example shows, these problems are related to changes in the amount of data. They can be very harmful, as small changes in the amount of data can cause serious performance damage. However, current DBMS performance tests still lack the means to detect them.

Challenges. The ideal way to identify data-sensitive performance anomalies in the CBO of DBMS is to execute queries with changed data and check whether the DBMS generates a significantly costly plan. We can consider a query to be anomalous if the optimizer selected a poor plan and its actual performance diverges significantly from the calculated value. *However, determining whether each plan is optimal for each combination of data and query is challenging.* Calculating the desired performance requires estimating cardinality and building the cost model for different data volumes. It can be a costly and complex process, which is almost equivalent to implementing a perfect DBMS. These may result in significant challenges. To avoid these difficulties, a possible approach is using a comparable DBMS to run the query and compare its response time with the time of the tested DBMS. However, *directly comparing the response time will result in a large number of false positives due to differences in DBMS states, design preferences, and environments.* Additionally, it can be difficult to define what constitutes a "significant" divergence.

Basic Idea of Hulk. The basic idea of Hulk is to first find the performance cliff by comparing the response time trends with the various datasets in two comparable DBMSs, and subsequently finding a better query plan within one DBMS to demonstrate that the plan chosen by the other is indeed costly. Comparable DBMSs are DBMSs that are homologous and have similar syntax, system design, and performance goals. For example, MariaDB and MySQL are such a pair, with similar system design and performance goals, and have a similar trend of response time for queries to the data size. The detailed method for finding a comparable DBMS will be discussed in Section 4.

However, we still can not directly compare the execution time of the query in two comparable DBMSs to identify the performance anomaly, due to the underlying performance differences that arise from different diverse optimizer implementations, live tuning capabilities, buffer cache state, and other factors. Instead, Hulk compares the response time increasing trend between two DBMSs to identify performance cliff. *Since the comparable DBMSs strive to find the best execution plan, and when the performance of one DBMS increases steadily while the other suddenly skyrockets, it always indicates the presence of a performance cliff.* Subsequently, Hulk will check if comparable DBMSs can offer a better execution plan, to verify whether it has generated an unreasonable plan.

3 Design of Hulk

Figure 3 illustrates Hulk's approach step by step. In Step ①, Hulk creates hundreds of tables in a new database. The tables are populated with records of different magnitude orders (1 to 1000 random distribution). Specifically, to trigger complex behaviors of the DBMS, the tables contain different features (such as index and foreign key). In Step ②, Hulk randomly selects several tables from the database and collects the metadata of the selected tables to synthesize a *data-sensitive query*. The query contains multiple clauses which are sensitive to the data in the selected tables. In Step ③, Hulk randomly selects a table from the selected tables used in the last step and inserts various amounts of rows to change the data volume. In Step ④, Hulk sends the data-sensitive query to comparable DBMSs for execution. Step ③ to ④ are repeatedly performed for sampling until the amount of data volume grows to a predefined value. In Step ⑤, Hulk analyzes the trends of increasing response time as the data volume grows. In the meantime, it estimates the

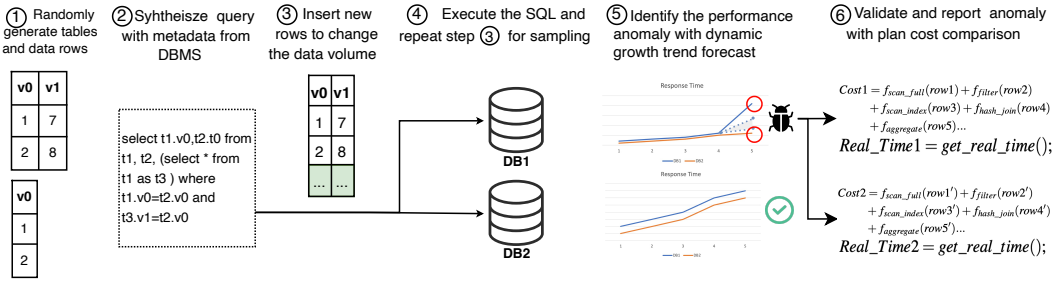


Fig. 3. Overview of Hulk. Hulk first randomly generates tables and initial data rows. Then it synthesizes the data-sensitive queries with metadata from DBMS. The query generated will be executed several times by two comparable DBMSs with growth data volumes (Steps 1–4, Section 3.1). During execution, Hulk identifies the potential performance anomalies by sampling the response time for each data volume and estimates the threshold for the next one. Finally, Hulk calculates and compares the plan cost of two DBMSs to validate the anomaly. Anomalies that pass the validation will be reported with the plan-level cause (Steps 5–6, Section 3.2).

threshold of response time for the same data volume in each DBMS. If the response time exceeds the estimated threshold, Hulk detects a **performance cliff** and considers it as a potential data-sensitive **performance anomaly**. In Step ⑥, Hulk first reproduces and analyzes the execution procedure to calculate the plan costs. Then, Hulk compares the two plan costs of the real execution plans. Once the cause of the performance cliff is due to the differences in the execution plan, then Hulk will report this potential anomaly. For the next iteration, Hulk will go back to Step ②.

3.1 Data-Sensitive Query Synthesis

3.1.1 Definition. To better describe the data-sensitive method, we give the following definitions:

Data-sensitive clause: a clause whose processing strategy is dependent on the characteristics of the data. For example, consider the JOIN clause in "SELECT * from A JOIN B". The specific JOIN strategy (e.g., hash join, nested loop join, merge join) is dependent on the data in tables A and B. Therefore, the JOIN clause is a data-sensitive clause.

Data-sensitive query: a SQL statement that contains data-sensitive clauses. The execution plan of the query may be influenced by the stored data in the DBMS. Given the SQL query in Figure 2 as an example, the `SELECT` statement contains a `WHERE` clause. When more data is inserted into table `Staff`, the query execution plan changes because MySQL optimizer estimates that table `Staff` joins `Salary` with few results. With more data-sensitive clauses (e.g., "`ORDER BY c0`", "`LIKE`", "`LEFT JOIN`", "`WHERE`"), the query optimizer is more susceptible to changes in the data when building the plan. In other words, the data sensitivity of the query is higher when its **clause number** is higher. With a low clause number, data-sensitive performance anomalies may be hard to find. However, a too-high clause number affects the efficiency of the testing since it takes more time to execute longer queries. How to find a suitable clause number will be discussed in Section 5.5 empirically.

Homomorphic data: the data with the same schema. Homomorphism ensures that all the characteristics of the data (such as foreign key relationships) are consistent except for the data volume. Homomorphic data is used for cross-validation of performance issues between two databases.

3.1.2 Query Synthesis Design. Figure 4 shows the design for generating semantically correct data-sensitive queries and homomorphic data for different DBMSs shown in steps ① to ④. It contains three main components, namely Homomorphic Data Augmentor, Metadata Synchronizer, and Data-Sensitive Query Synthesizer. Metadata is the intermediate for data and query generation. It records the schema of tables, columns, and indexes that exist in DBMSs. Metadata Synchronizer collects metadata and synchronizes it to other components when receiving requests.

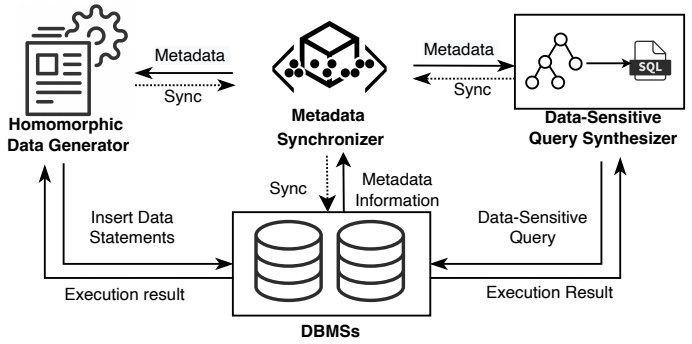


Fig. 4. The flow of query synthesis and data augmentation. Hulk uses the Metadata Synchronizer to obtain the most up-to-date metadata information of DBMSs. Homomorphic Data Augmentor and Data-Sensitive Query Synthesizer synchronize the metadata to alternatively generate queries and data.

Data Augmentation. To provide the raw material for data-related clauses and continuously generate data to influence the execution plan of the data-sensitive query, Hulk is designed with *Homomorphic Data Augmentor*. The component creates an amount of initial complex tables and continuously inserts lots of data during the performance testing.

Data augmentation has three steps: (1) Table generation. Hulk first creates amounts of tables randomly, with a random number (distributed in 10-1000) and type of columns. (2) Constraint augmentation. Index and foreign keys are two major constraints added. Hulk will iteratively scan all tables to randomly add these constraints. To ensure the correctness of foreign keys, Hulk will get all the information (e.g., the data type of each column) of each table and randomly choose a same-data-type column to construct the foreign key constraints. (3) Data insertion. Hulk periodically inserts data into existing tables while adhering to the constraints. To ensure complexity, Hulk has fully modeled the SQL grammar, enabling the random selection of various data type objects. The semantic correctness of the SQL statements that insert the data is ensured by the synchronized metadata information from DBMS. In each data insertion loop, with the metadata information, it inserts semantic-correct data (e.g., the data type should match in the target table) into the database. It will continue this process until the requested amount of data has been fully inserted.

Query Synthesis. After generating amounts of initial tables and data records in the database, Hulk then synthesizes data-sensitive queries that contain amounts of data-related clauses with *Data-Sensitive Query Synthesizer*, to trigger more optimizer behaviors during the DBMS testing.

Algorithm 1 shows the detailed steps to synthesize data-sensitive queries with chosen tables based on metadata information. Hulk first selects several tables from databases and randomly constructs an AST for a query. The AST has data nodes that indicate data characteristics (e.g., `table name`) and structure nodes that indicate the syntactic structure (e.g., `SELECT` and `FROM`). The AST first generates data nodes and structure nodes randomly with the average value of the **clause number** and the number of chosen tables (Lines 2-3). Then, it scans each node in the AST with depth-first traversal. For each first traversal node, if it is the predecessor of a data node, Hulk will instantiate the sub-data tree containing data nodes with the function `instantiateSubNodes` (Lines 5-6). Specifically, to instantiate the successor nodes of `n0`, Hulk first checks the types of each successor node (i.e., `m0`) of `n0`. If `m0` is a structure node, it does not need to be populated with data (Line 11). If `m0` is a data node, Hulk first finds the associated data nodes of `m0`, and then instantiates these data nodes with the table and column metadata (Lines 12-14). If `m0` is the predecessor of subclause nodes, Hulk will recursively call `InstantiateSubNodes` to instantiate the subclause nodes (Lines 15-16). After populating all data nodes, Hulk converts the AST to a SQL statement.

Algorithm 1: Data-sensitive query synthesis

```
Input  : T: Tables that are used to generate queries,
         M: Metadata information of the tables,
         C: average data-sensitive clause number
Output : Data-sensitive SQL query statement s0
1  begin
2      Tables ← chooseTables (T);
3      QueryTree ← randomSynthsizeAST (Tables.len, C);
4      foreach n0 ∈ dfsTreeNodes(QueryTree) do
5          if isPreDataNode(n0) then
6              instantiateSubNodes(n0, QueryTree);
7      s0 ← convertToStatement(QueryTree);
8      return s0;
9  Function instantiateSubNodes(n0, QueryTree):
10     foreach m0 ∈ successorNodes(n0, QueryTree) do
11         if m0.type = structNodes then continue ;
12         else if m0.type = dataNodes then
13             sub_nodes = getSubDataNodes(m0);
14             instantiateNodes(sub_node, t0, cols);
15         else if m0.type = SubClauseNodes then
16             instantiateSubNodes(m0, QueryTree);
17 End Function
```

3.2 Performance Anomaly Analysis

This module consists of performance cliff identification and performance anomaly validation.

Performance Cliff Identification. In the continuously repeating Step ③ to ④, Hulk samples the response time of the two comparable DBMSs being tested under increasing data volumes. By comparing the samples on two comparable DBMSs, based on one of them, Hulk estimates a reasonable response time range for each newly increased data volume on the comparable DBMS. Any test case outside this range is considered a *performance cliff*.

Algorithm 2 shows the detailed procedures to identify the data-sensitive performance anomalies with sampling-based response time estimation. Suppose A_i represents the response time with different data volumes for DBMS α, and B_i represents the response time for DBMS β, Hulk first samples the Euclidean distance d_i (i=0...n-1) between two points on two curves with the same horizontal coordinates. Then, Hulk estimates the reasonable response time region of the current data volume with sampled Euclidean distance d_i (i=0...n-1) and response time A_i and B_i (i=0...n-1). As Section 2 shows, the growth curves of the response time of comparable DBMS are similar to the data volume growth, otherwise, it may be a *performance cliff* of DBMS. According to the central limit theorem [32, 39, 48], the Euclidean distances d_i between two points on two curves should conform to Gaussian distribution, as the number of samples for data volume increases. Specifically, the central limit theorem (CLT) states that if a series of variables in a distribution is identical and independent, the variables will tend to follow a normal distribution between the sample mean. The Euclidean distance between response times of the same data can be considered as identically and independently distributed (IID) samples.

(1) *Identity*: the response times of comparable DBMSs should demonstrate a consistently increasing trend, indicating that the Euclidean distance d_i between response times for the

same dataset should have comparable values, thereby fulfilling the criteria for identity in the Euclidean distance d_i satisfying the identity.

(2) *Independence*: the comparable DBMSs have similar system designs and run in the same environment, thus the operating system and the hardware should have a similar influence on them. Therefore, with different data sets, the Euclidean distance d_i between the response time of comparable DBMSs for the same data volume should be approximately independent.

Algorithm 2: Estimate response time with sampling

```
Input : A_i(i = 0, 1...n): the sequence of response
        times for DBMS α ,
        B_i(i = 0, 1...n): the sequence of response
        times for DBMS β
1 begin
2     Distance ← vector();
3     foreach B_i ∈ B_1...B_n do
4         d_i = A_i - B_i;
5         push(Distance, d_i);
6         μ_i = means(Distance);
7         σ_i = standardDeviation(Distance);
8         E'_{bi+1} ∈ estimateValue(A_{i+1}, μ_i, σ_i);
9         if B_{i+1} ∉ E'_{bi} then reportAnomaly() ;
```

Therefore, the sampled variables can be considered as identically and independently, and should be approximately Gauss Distribution. Based on the central limit theorem, the estimated thresholds for normal Euclidean distance values are 95.4% and 99.7% [20] for two and three standard deviations of the expected sampled data, respectively. We prefer to use two standard deviations to avoid missing potential performance anomalies and the excess (i.e., anomalies that are wrongly indicated) will be filtered through the later anomaly validation process with **plan cost comparison**. Therefore, if the d_i does not fall within two standard deviations of the expected sampled Euclidean distances float, it could be a performance outlier with an accuracy of 95.4%. Following the above analysis, Hulk first calculates the mean μ and the standard deviation σ of sequence d_i (i=1...n-1). Then, Hulk get the valid estimation region E'_{b_n} with the standard deviation (σ), the mean (μ), and the response time of corresponding data volume in the other DBMS (A_n):

$$E'_{b_n} \in [A_n + \mu_{n-1} - 2 * \sigma_{n-1}, A_n + \mu_{n-1} + 2 * \sigma_{n-1}]$$

If the response time B_n of DBMS β does not in the region of E'_{b_n}, Hulk detects a performance cliff and thinks it is a potential data-sensitive performance anomaly in DBMS α or DBMS β.

Figure 5 shows an example of how Hulk identified the data-sensitive performance cliff(potential performance anomaly) in DBMS. In iteration n, the Euclidean distance $d_i(i = 1...n-1)$ is first calculated. Based on $d_i(i = 1...n-1)$, Hulk calculates the valid region (i.e., E'_{b_n}) of the response time for DBMS β, which is represented as the shaded region in the figure. When the response time B_n falls outside the expected performance range E'_{b_n}, Hulk thinks it a potential data-sensitive performance anomaly exists in DBMS β or DBMS α.

Performance Anomaly Validation. Based on the sampling-based dynamic threshold estimation, Hulk can discover data-sensitive performance cliffs effectively. However, it is still very difficult to report these performance cliffs to DBMSs vendors for confirmation. The main reason is that some of the performance cliffs may not be performance bugs in DBMSs. As Section 2 describes,

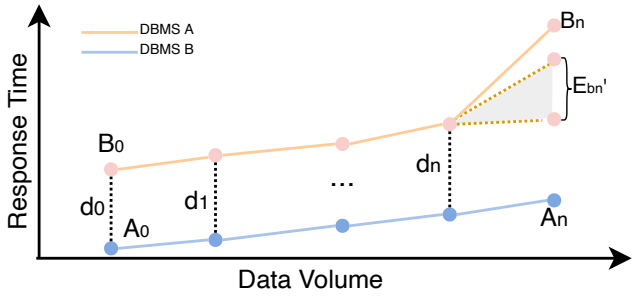


Fig. 5. An example shows the process of identifying data-sensitive performance anomalies. The curves depict the trend of the response time of DBMS α and β as the volume of data increases. d_i represents the Euclidean distance between the sampling points on the two curves. The shaded area indicates the reasonable response time range. B_n is beyond this range, it signifies a performance cliff. Therefore, DBMS β is considered to have a potential data-sensitive performance anomaly.

different DBMSs strive to find the least costly plan for each query [54], if the *performance cliff* is caused by the generation of significantly expensive plans, Hulk reports it as an anomaly.

Hulk automatically validates the cliffs as anomalies with the plan cost comparison. It checks if there is a significant difference in the uniform cost of plans and if the comparable DBMS can supply a better plan. The availability of a better plan confirms the existence of performance issue. Hulk first extracts the PoC (Proof-of-Concept) by reducing the test cases to statements relevant to the anomaly. Then Hulk automatically cleans the databases and re-sent the PoC and dataset to two DBMSs for validation. Specifically, Hulk adds keywords (e.g., "Explain Analyze" for MySQL) on the PoC to get the execution plans from both DBMS. Next Hulk calculates the uniform "plan cost" to eliminate the impact of the operating environment. If the comparison of plan costs reveals a problem, Hulk will save the PoC, data set, and execution plan in the two DBMSs to form a report. Or else, the PoC and data set will be removed.

Plan cost calculation. A basic idea for validating is to extract and compare the plan optimized from the query by two DBMSs. Directly comparing plans of two DBMSs for exact consistency does not indicate that one of them has problems, as different DBMSs may have different design preferences. Besides, we found that there was a discrepancy between the planned estimated data volume and the actual data volume produced in each operation of the plan. For example, the CE of MySQL's CBO is biased to assume that when two tables are hashed together, only 10% of the rows satisfy the filtering conditions. This may deviate significantly from the actual execution results. Therefore, it is difficult to make a correct judgment by directly using the costs estimated in the plan. To address the challenge, Hulk first extracts the real execution process of the plan, and then uses the real data volume to calculate a **plan cost** time with the uniform cost model. The unit of "plan cost" is the cost associated with each operation performed by the query execution engine. Each operation within a query plan, such as table scans, joins, or aggregations, is associated with a specific cost. The specific cost of each operation consists of I/O cost and CPU tuple cost, followed by almost all major DBMS (PostgreSQL, MySQL, etc.). For example, PostgreSQL calculates the cost of the "scan table operator" with *"cost = (disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost)"*.

If the difference in "plan cost" exhibits a **decision margin**, then it implies that there might be an optimization problem of the target DBMSs. The *decision margin* is critical for performance anomaly detection. A small decision margin will cause a lot of false positives, but a too large decision margin will also miss many true anomalies. We will give the specific decision margin in Section 4. In general, a plan indicates the operation to be executed, and its execution time is proportional to the number of times this operation is executed. For i_{th} operation $operation_i$, suppose num_i represents

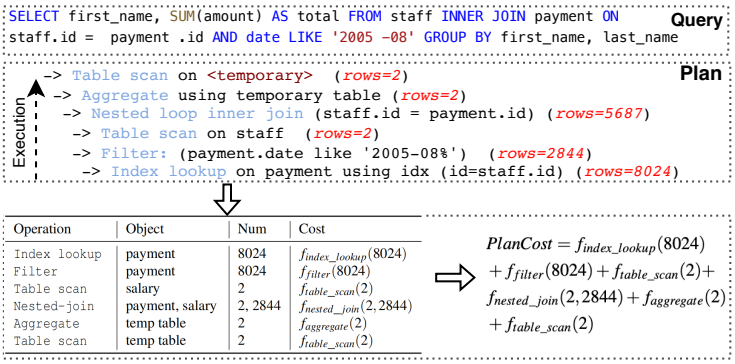


Fig. 6. An example to calculate the *"plan cost"*. Hulk obtains the execution plan for the query, extracts the number of rows for each operation, and then calculates the plan cost.

the number of $operation_i$ executed, the time cost to finish $operation_i$ is $f_{operation_i}(num_i)$. The plan cost is calculated by the following formula:

$$plan\_cost = \sum_{i=1}^{m} f_{operation_i}(num_i)$$

For example, Figure 6 shows the process to calculate the *"plan cost"* of a query. The top part of the figure shows the plan to execute the query in MySQL. The operations performed by the plan are described in order from bottom to top. The value of rows indicates the number of rows generated by the plan. The table in Figure 6 shows the operation and its cost. With the predefined cost function and the formula, the "plan cost" of the SQL statements can be calculated as: $f_{index\_lookup}(8024) + f_{filter}(8024) + f_{table\_scan}(2) + f_{nested\_join}(2, 2844) + f_{aggregate}(2) + f_{table\_scan}(2)$.

Plan cost comparison and anomaly report. After calculating the "plan cost" for the final data-sensitive query, Hulk then compares the "plan cost" of execution on both DBMSs. If the difference in the cost for two DBMSs exhibits a significantly large, then we consider the anomaly to be caused by the logic of CBO, and therefore it needs to be further reported and repaired. For one anomaly, the report consists of the query (i.e., the `SELECT` statement), dataset (i.e., the volume and schema reflected in the `CREATE` and `INSERT` statements), reasonable response time region, the problematic plan generated by the target DBMS, and a plan with a reasonable response time.

4 Implementation

This section explains other implementation details, which we consider significant for Hulk.

Finding Comparable DBMS. The critical step of Hulk in practice is to find a comparable DBMS against which the DBMS under test can be evaluated. As mentioned in Section 2, the comparable DBMS should have similar syntax, system design, and performance objectives. First, we can select comparable DBMSs by identifying those that have branched from the same origin. For example, MySQL and MariaDB are derived from the MySQL branch [3]. We observe that almost all commonly used DBMSs in the industry can find their homologous DBMS that are derived from the same source branch. They can be used as references to each other for evaluating query plans generated by the optimizer. Table 1 presents the part of comaprale DBMSs for PostgreSQL, MySQL, and SQLite families collected from the DBMS Ranking Website [28].

In situations where it is challenging to find DBMSs from the same branch, Hulk can opt for different versions of a single DBMS as the comparable DBMS. For example, we have utilized Hulk to test TiDB 7.1.0-7.1.5 and discovered 16 performance bugs.

Table 1. Part of the category and comparable DBMSs

Category	Comparable DBMS
MySQL Series	MySQL, MariaDB, PolarDB, Percona
PostgreSQL Series	PostgreSQL, AntDB, Citus, GaussDB
SQLite Series	SQLite, Comdb2, SQLCipher

Decision Margin Setting. As Section 3.2 mentions, the decision margin plays a vital role in performance anomaly validation. Hulk set the decision margin based on the statistics and communication with the DBMS developers. In practice, when the decision margin is 1.5, Hulk totally reports anomalies with 27% false positives. However, when the decision margin is 2, the false positive rate reduces to 6%. Hulk finally set the decision margin with 2 in implementation.

SQL Grammar Scope. To improve the ease of adaptation to new DBMSs, we implement the abstract syntax tree (AST) model for synthesizing data-sensitive queries based on the SQL-2003 standard [27]. However, different DBMSs still have unique features in terms of data types and functions. Currently, Hulk supports all the data types and functions for major-popular series of DBMSs, including MySQL series [18], PostgreSQL series [6], SQLite series [7]. Hulk can generate semantically correct SQL statements by switching the data types and functions of the AST model.

Effort of Adaptation. The effort of adapting Hulk to a new DBMS could be little. We just need to write metadata information query statements and customize the data types and functions for the DBMS under test. Generally, a metadata information query for a DBMS has no more than ten lines of SQL statements. Since Hulk has supported the data types and functions for popular DBMSs, the basic data types and functions can be directly used for new DBMSs such as INT, VARCHAR, STRING, DATE, TIMESTAMP, TIME, and FLOAT. Consequently, only a few dialect-related data types and functions need to be implemented for a new DBMS.

5 Evaluation

We evaluate Hulk in terms of its ability to discover the data-sensitive performance anomalies, as well as its efficiency in generating the data-sensitive queries and analyzing the performance anomalies. Our evaluation aims to answer the following questions:

- **RQ1:** Can Hulk discover the data-sensitive performance anomalies?
- **RQ2:** How does Hulk perform compared to other related DBMS testing techniques?
- **RQ3:** How effective of the workloads generated by Hulk?
- **RQ4:** How important is the data-sensitivity of queries in finding performance anomalies?

5.1 Evaluation Setup

Tested DBMSs. To evaluate the generality and efficiency of Hulk, we select six popular open-source DBMSs, namely MySQL [4, 55], MariaDB [2, 12], Percona [16], TiDB [24, 42], AntDB [8, 9], and PostgreSQL [5, 38], which are widely used in industry.

Compared Techniques. We tried our best to compare Hulk with open-source DBMS performance testing tools. Since only APOLLO [30] and SQLancer ^{CERT} [11] are open-source tools, as a remedy, we still compared Hulk with other state-of-the-art SQL fuzzer (Squirrel [60]) to evaluate the capability to explore the state space on DBMSs.

Basic Setup. We perform the experiments on a machine running 64-bit Ubuntu 20.04 with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 488 GiB of main memory. All DBMS tested are run in docker containers and can be downloaded directly from their website. For quantitative comparisons, we run the docker containers for each DBMS experiment(including DBMS server

and Hulk) with 5 CPU cores and 32 GiB of main memory. To detect real-world data-sensitive performance anomalies, we perform testing of Hulk on all six DBMSs for two weeks continuously.

5.2 Performance Anomaly Detection

Overall Result. Hulk has found a total of 129 previously unknown bugs, confirmed by the developer, on six well-tested DBMSs. Among them, 94 anomalies are data-sensitive performance anomalies, and 35 anomalies cause the DBMS crash.

Table 2. Hulk discovered 129 bugs in six DBMSs, including 94 data-sensitive performance anomalies.

DBMS	Bug Type	Bug Status and Number	Position
MySQL	Performance	confirmed(26),fixed (10)	regexp(2), dd(2), cache(4), info_schema(4), performance_schema(2), join_optimizer(4),storage(5), range_optimizer(3)
MySQL	Crash	confirmed(11), fixed(8)	gis(1), libmysqld(1), join_optimizer(2), storage(6), scripts(1),
MariaDB	Performance	confirmed(29), fixed(7)	sql(17), storage(5), libmysqld(1), strings(3),
MariaDB	Crash	confirmed(13), fixed(9)	storage(6), mysys(1), scripts(1), sql(5)
Percona	Performance	confirmed(11), fixed(6)	cache(1), dd(1), info_schema(2), join_optimizer(3), locks(1), range_optimizer(2), memeory(1)
Percona	Crash	confirmed(3), fixed(1)	storage(1), strings(1), binlog(1)
TiDB	Performance	confirmed (16), fixed(4)	lock(2), server(3), planner(5), statistic(2), meta(4)
TiDB	Crash	confirmed(4), fixed(1)	meta(1), server(1), store(2),
AntDB	Performance	confirmed(4), fixed(0)	rewrite(1), optimizer(3)
AntDB	Crash	confirmed(2), fixed(2)	rewrite(1), storage(1)
PostgreSQL	Performance	confirmed(8), fixed(2)	statistics(1), optimizer(1), rewrite(2)
PostgreSQL	Crash	confirmed(2), fixed(2)	storage(1), item(1)
Total		**129 bugs, 14 CVEs,**	**94 performance bugs and 35 crash bugs**

Statistics. Table 2 shows the statistics of data-sensitive performance anomalies reported by Hulk for each DBMS. Specifically, Hulk detected a total of 94 data-sensitive anomalies, including 26, 29, 11, 16, 4, and 8 performance anomalies in MySQL, MariaDB, Percona, TiDB, AntDB, and PostgreSQL, respectively. Among them, 29 performance anomalies have been fixed. The results reflect that data-sensitive performance is prevalent in these popular DBMSs. With data-sensitive query synthesis and sampling-based performance estimation, Hulk is able to detect them. Note that in addition to performance anomalies, Hulk also finds 35 previously unknown crash bugs in the tested DBMS, 23 of them have been fixed, including 14 that have been assigned CVE IDs. They are founded because Hulk generates a large number of tables, data, and data-sensitive queries to simulate various complex scenarios of the DBMS. Thus Hulk covers deep behaviors of the DBMSs and triggers these crashes.

Table 3. Number of data-sensitive performance bugs related to the CE, CM, and PE Component.

Component	MySQL	MariaDB	Percona	TiDB	AntDB	PostgreSQL
CE related	13	13	4	6	2	3
CM related	3	4	1	2	1	2
PE related	10	12	6	8	1	3

Classification. Table 3 shows the classification of the 94 confirmed data-sensitive performance anomalies, organized by the components where errors occur. We can see that 41, 13, and 40 performance anomalies are due to the implementation errors in CE, CM, PE, and others, respectively.

For instance, the motivation example in Section 2 shows a bug caused by the implementation error in CE. Specifically, when the number of records in the `Staff` table increases to 1010, MySQL incorrectly estimates that the temporary result of `Staff` joining `Lawyer` will exceed the temporary `join_buffer_size`. Therefore, it first performs a hash join on two small tables, which causes a

3753% increase in the response time to a 0.1% increase in the amount of data. Other works can not detect these anomalies because they can only be triggered by increasing data volume. Additionally, approaches like APOLLO and Amoeba both lack the processes. Particularly, the crash bugs are mainly represented as memory safety errors such as stack overflow, which are caused by the incorrect use of memory operations. They are detected because the data-sensitive queries help Hulk trigger behaviors of the CBO that are never covered by other tools.

Table 4. Number of performance degradation at each grade

DBMS	1x-10x	11x-20x	21x-30x	≥ 30x
MySQL	11	8	4	3
MariaDB	15	8	3	3
Percona	3	4	3	1
TiDB	7	6	1	2
AntDB	2	1	1	0
PostgreSQL	4	3	1	0
Total	42	30	13	9

Impact for Performance Anomalies. Table 4 summarizes the impact of the discovered data-sensitive performance anomalies. It shows the number of performance degradation at different grades. We can see that the performance anomalies detected by Hulk cause serious impacts on the performance of DBMS. Specifically, compared to the estimated response time, 42, 30, 13, and 9 anomalies exhibit speed decreases of 1x-10x, 11x-20x, 21x-30x, and over 30x, respectively. According to the developers' responses to the reported anomalies, they indicated that in real-world application scenarios. Besides, 21 of the detected bugs have been hidden in the DBMSs for over 5 years. In particular, one of the detected bugs was imported 9 years ago, as the paper was written. These performance anomalies may cause serious financial losses.

Anomaly Case Study. We present a performance anomaly caused by the incorrect use of the temporary table in optimization. The performance anomaly has serious implications and can result in performance degradation of about 100 times.

Schema and Data of the Anomaly. Figure 7 shows the `CREATE` statements to create two tables related to the performance anomalies. Table `t0` contains two columns `c0` and `c1`, where `c0` is the primary key and `c1` is created with an index constraint. Table `t1` contains three columns `c0`, `c1`, and `c2`, where `c0` is the primary key. `c1` and `c2` are also created with index constraints. The bottom half of the figure shows the statements that insert 600 random rows into `t0` as well as the statements to incrementally insert random data into `t1`.

Create Tables:

```
CREATE TABLE `t0` (
`c0` int(11) AUTO_INCREMENT,
`c1` varchar(25) DEFAULT
NULL,
 PRIMARY KEY (`c0`)
);
CREATE INDEX i0 ON t0(c1);
```

```
CREATE TABLE `t1` (
`c0` int(11) AUTO_INCREMENT,
`c1` int(11) DEFAULT NULL,
`c2` datetime DEFAULT NULL,
 PRIMARY KEY (`c0`)
) ;
CREATE INDEX i1 ON t1(c1);
CREATE INDEX i2 ON t1(c2);
```

Insert Records:

```
INSERT INTO t0(c1) values (random_string); -- 600 rows
INSERT INTO t1(c1, c2) values (random_int,random_date);
--1000-55000 rows
```

Fig. 7. The `CREATE` statements and `INSERT` statements to trigger a performance anomaly in MariaDB. Table `t0` contains two columns, namely `c0` and `c1`. Table `t1` contains three columns, namely `c0`, `c1`, and `c2`.

Data-Sensitive Query to Trigger The Anomaly. Figure 8 shows the data-sensitive query and corresponding execution plan in MariaDB and MySQL for different data volumes. The query looks for rows with the same primary key in `t0` and `t1`, and then joins them in descending order. In particular, table `t0` is generated using the `SELECT` subquery. When the amount of data volume in table `t1` does not exceed 300,000, MariaDB uses the "Using filesort" method to fully scan the table `t1`. MariaDB takes 0.104 seconds to execute the query, which is similar to MySQL. However, when the amount of data volume increases to 550,000, MariaDB creates a temporary table in addition to the "Using filesort" method in `Plan2`. Additional operations may introduce significant overhead to fully scan table `t1` with `plan2`. This causes MariaDB take 9.874 seconds to execute `Plan2`, while MySQL takes only 0.124 seconds to execute `Plan1` on the same amount of data volume.

Root Cause. The response time delay is caused by changes in the execution plan, not the environment. Thus we thought this was a performance anomaly and reported it to the developers of MariaDB. They analyzed it and found that the performance anomaly was introduced to MariaDB in version 10.6. The root cause of the bug lies in a coding mistake within the CE component of MariaDB's CBO, specifically found in the optimization logic for temporary tables. Other methods may have difficulty detecting data-sensitive performance anomalies, which are only triggered when a certain amount of data is reached.

The SQL Query:

```
SELECT c2,c1 FROM t1 JOIN (SELECT c0,c1 FROM t0) AS t0 WHERE t0.c0=t1.c0 ORDER BY c2 DESC;
```

Execution Plan with 300000 rows in t0. -- 0.104s (mariadb) Plan1

id	select_type	...	type	posssible_keys	key	...	rows	...	Extra
1	SIMPLE	...	ALL	PRIMARY	NULL	...	300000	...	Using filesort
1	SIMPLE	...	eq_ref	PRIMARY	PRIMARY	...	...	...	Usingindex

Execution Plan with 550000 rows in t0. -- 9.874s(mariadb) Plan2

id	select_type	...	type	posssible_keys	key	...	rows	...	Extra
1	SIMPLE	...	ALL	PRIMARY	NULL	...	550000	...	Using temporary, Using filesort
1	SIMPLE	...	eq_ref	PRIMARY	PRIMARY	...	...	...	Usingindex

Fig. 8. The query to trigger a performance anomaly in MariaDB. The execution plan with 300,000 rows takes 0.104s. When the volume of data grows to 500,000 rows, the changed plan takes 9.874 seconds, while MySQL takes only 0.124 seconds to execute the query under the same data volume.

5.3 Comparison With Other Techniques

We run Hulk, APOLLO, SQLancer ^{CERT}, and Squirrel on six selected DBMSs for 24 hours. Table 5 shows the number of confirmed performance anomalies, crashes, and covered branches.

From the table, we can see that Hulk discovers more performance anomalies than APOLLO and SQLancer ^{CERT}. Specifically, compared to APOLLO and SQLancer ^{CERT}, Hulk totally detects 37 and 35 more confirmed performance anomalies. Notice that the 2 anomalies found by APOLLO are not among the 42 bugs found by Hulk. In other words, these two bugs are regression performance anomalies but may not be sensitive to the volume of data. The results suggest that data-sensitive performance anomalies are more likely to arise in DBMSs than regression performance anomalies, but they are hard to find without Hulk's approach. Specifically, Hulk generates amounts of data-sensitive queries containing complex clauses. As the volume of data increases, these data-sensitive queries can trigger deep code in the DBMS optimizer. In addition, Hulk estimates reasonable performance from the sampled historical data to identify anomalies. Consequently, Hulk detects more performance anomalies compared to APOLLO.

Moreover, we also find that Hulk performs well in detecting crash bugs. Specifically, Hulk detects 10 more crash bugs when compared to Squirrel. This can be explained by the design of data-sensitive query synthesis. Hulk generates amounts of complex queries that contain 40

Table 5. Number of confirmed performance anomalies and crashes, and covered branches on six DBMSs

	Performance Bugs	Crashes	Branches
APOLLO	5	6	281,056
SQLancer	7	3	273,467
Squirrel	0	9	312,713
Hulk	42	19	331,909

data-sensitive clauses on average (which will be discussed in Section 5.5). These data-sensitive queries trigger more complex behaviors of CBO in DBMSs with the growing data volume. It could be seen from the branches covered by each tool in 24 hours. The third column in Table 5 shows the number of branches covered for APOLLO, SQLancer ^{CERT}, Squirrel, and Hulk. We can see that Hulk covers 17%, 21%, and 6% more branches than APOLLO, SQLancer ^{CERT}, and Squirrel, respectively. The improvement of branches proves that the data-sensitive query generated by Hulk can trigger more behaviors of DBMSs compared to other tools. Therefore, Hulk detects more crash bugs compared to other tools.

5.4 Analysis of Hulk's Workloads

To investigate the effectiveness of the data and queries generated by Hulk, we statistics the data volume of the triggered performance anomalies and randomly select 10,000 generated queries during testing to analyze the clause categories of the queries.

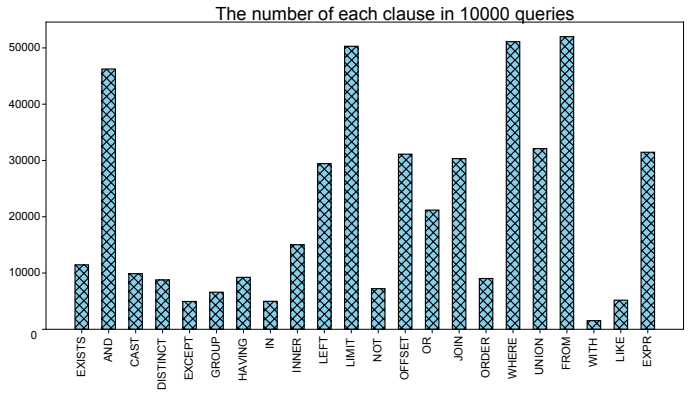


Fig. 9. Clause number of each type in generated queries.

Clauses Category. We extract clauses of each query and analyze the category of the clauses based on the SQL standard [51]. Figure 9 shows the number of each type of clause contained in all analyzed queries. We can see that the data-sensitive SQL queries generated by Hulk cover all the clause types of MySQL (e.g. `EXISTS`, `CAST`, and `EXCEPT`). Specifically, 10,000 data-sensitive queries contain 21 kinds of clauses and each kind of clause is covered 20,909 times on average. It shows that the queries generated by Hulk have complex combinations and deep clause nesting. These combined and nested clauses make the execution of plans corresponding to these queries vary significantly when the volume of data changes.

Data Volume. We collect the test cases when the data-sensitive performance anomaly happened, and analyze the data volume of these test cases generated by Hulk. Table 6 shows the proportion of different-grade data volumes when the performance degradation happened. We can see that the data volume of tables has an influence in triggering performance anomalies. Most performance anomalies

Table 6. The proportion of data volume for performance bugs.

Data Volume	1-1000	1000-10000	10000-10000	>100000
Proportion	5%	23%	20%	52%

require certain data volumes to be triggered. Specifically, only 20% performance anomalies happen with fewer than 1,000 records in the table. About 52% and 23% of the performance anomalies are triggered with 1,000-10,000, and 10,000-100,000 records in the table, respectively. We also find that excessive data volume is also not always conducive to detecting performance anomalies. Specifically, only 5% of performance anomalies are triggered when the data volume is over 100,000 records.

5.5 Importance of Data-Sensitivity

Section 3.1 highlights that a SQL query's data sensitivity increases as the number of data-sensitive clauses it contains increases. To investigate the impact of data sensitivity for detecting performance anomalies, we conducted an experiment that involved running Hulk on 5 different DBMSs with data-sensitive query synthesis settings that varied average **clause number** (5, 10, 20, 30, 40, 50, 100). To accommodate the majority of queries generated by popular DBMS fuzzers like Squirrel and SQLancer, we set the minimum average clause number for evaluation to 5. We then analyzed the number of performance bugs, crashes, and branches found by Hulk across different settings.

Table 7 presents the number of bugs detected by Hulk when using data-sensitive query synthesis settings with varying clause numbers. The results demonstrate that the data-sensitivity of a query has a significant impact on Hulk's ability to detect anomalies. As the number of clauses in a query increases up to a certain point (i.e., when the clause number is less than 50), we observe an increase in the number of performance bugs and crash bugs detected by Hulk. This is because a query with more data-sensitive clauses can cover more DBMS states, which is necessary for finding anomalies. Additionally, the fourth column of Table 7 shows the number of covered branches, which also increases as the clause number increases. This indicates that Hulk can find more bugs when the data-sensitivity of the query is higher.

Table 7. The number of bugs found by Hulk on various clause numbers in six DBMSs for 24 hours.

Clause Number	5	10	20	30	40	50	100
Performance Bugs	9	19	27	35	42	39	32
Crashes	3	7	11	17	22	21	18
Branches	181,412	191,321	197,230	203,321	211,991	210,879	199,164

However, we also found that increasing the number of clauses in a SQL query does not always lead to better anomaly detection. Specifically, when the clause number reached 100, Hulk detected fewer bugs compared to when the clause number was 40. This is because DBMSs take more time to execute longer queries, which can reduce the efficiency of Hulk. For example, Hulk tested a total of 1,054k SQL queries when the clause number was 40, but only tested 813k queries when the clause number was 100. As a result, Hulk covered fewer branches and detected fewer performance bugs and crash bugs when the clause number was 100.

Overall, these results indicate that data sensitivity in queries is important for Hulk to detect performance anomalies, which adequately answers **RQ4**. Based on experimental results, there is an optimal range for the number of clauses that maximizes Hulk's efficiency in detecting errors. Consequently, we set the average number of clauses to 40 and generate queries with the number of clauses normally distributed around this value in practice. This setting helps to balance the trade-off between the efficiency of Hulk and the ability to detect a wide range of anomalies.

6 Discussion

Realism of Hulk's Query and Workloads. Using the queries with more clauses can increase the possibility of triggering data-sensitive performance anomalies. A query with more data-sensitive clauses can cover more DBMS states, which helps find these anomalies. Nevertheless, these anomalies may also occur with fewer clauses and real-life workloads. Among the 94 bugs detected by Hulk, 58 cases could be reproduced using queries comprising fewer than 10 clauses and typical workloads after reducing the test cases. These queries are all confirmed by DBMS developers, and they thought these queries may exist in real workload and cause serious performance degradation.

The statistics also show that the 74 queries that triggered the anomalies contain no more than 20 clauses. For example, the SQL query "SELECT c2,c1 FROM t1 JOIN (SELECT c0, c1 FROM t0) AS t0 WHERE t0.c0=t1.c0 ORDER BY c2 DESC;" triggers a performance anomaly in MariaDB and only has 7 clauses. As a comparison, TPC-H is a test suite that consists of common queries, and 23% of them contain over 30 clauses. Moreover, although some SQL queries have over 30 clauses, they are still being confirmed by the developers.

Generability of Hulk. Hulk detects the data-sensitive performance anomalies by estimating the threshold of response time for specific data volume based on the response time trends comparison. In our implementation, Hulk uses differential analysis because it is difficult to predict the trend of query response time as data volume increases due to the diversity of DBMS states and the complexity of queries. We use comparable DBMSs as the target and referenced DBMS in the differential analysis. For example, we performed MySQL-MariaDB, MySQL-Percona, MySQL-TiDB, MariaDB-Percona, AntDB-PostgreSQL testing combinations in our experiments (Section 5.2) because they have similar optimizer design and derived from the same source branch. In practice, most DBMSs have their homologous DBMS for differential analysis. Even without homologous DBMSs, Hulk can also be applied to the different versions of a single DBMS. For example, we used Hulk to test TiDB 7.1.0-7.1.5 and also detected 16 data-sensitive performance bugs.

Influence of Optimizer Parameters. DBMS Optimizer parameters can greatly affect performance and may also cause critical performance issues. In our experiment, we use default optimizer parameters of all DBMSs to ensure a fair comparison and detect many data-sensitive performance anomalies. However, finding the performances related to optimizer parameters is still important. Our paper focuses on the data-sensitive performance anomalies and provides an oracle to find them by analyzing query plans to data variations. No matter whether it is a change in data volume or a change in optimizer parameters that causes performance anomalies, the plan-guided oracle could be used to determine both of them. Moreover, since the two types of anomalies are orthogonal, we can even change both the data and parameters to find more bugs for a given workload in the future.

7 Related Work

DBMS Fuzzing. Fuzzing is one of the most popular testing techniques and it is natural to adapt fuzzing to the testing of DBMSs. Generally, current DBMS fuzzers are used to detect crash bugs, memory safety bugs with AddressSanitizer [50], or logic bugs with the test oracle [36, 47]. SQLsmith [49] generates queries based on predefined rules in target DBMSs to detect crash bugs. It adapts very well to PostgreSQL and finds many bugs in it. However, it will require additional human efforts when testing other DBMSs. SQLancer [47] detects logic bugs of DBMSs by generating queries to fetch an existing row from databases. If the DBMS fails to fetch that, the likely cause is a logic bug. Its following works [45, 46] also utilize the similar idea by generating semantically equivalent queries and comparing the results. Squirrel [60], Ratel [52], Unicorn [57], and Griffin [21] introduce coverage-feedback into the query mutation to cover more branches, and they use the AddressSanitizer [50] to detect the memory safety bugs. Lego [35] proposes sequence-oriented fuzzing to improve the code coverage by combining different SQL Type Sequences of the statements.

Different from them, Hulk aims at data-sensitive performance anomalies. First, Hulk continually increases the data volume to trigger these performance anomalies. Second, Hulk not only generates queries based on rules and metadata to ensure semantic correctness but also uses them to equip queries with more data-sensitive clauses. Finally, Hulk utilizes sampling-based performance estimation to identify data-sensitive performance anomalies.

Performance Testing. The traditional approach to detecting performance anomalies is using predefined test suites (i.e. workloads) and checking whether the result exceeds the baseline obtained from experience or empirical experiments [44, 56, 58, 59]. Specifically, APOLLO [30] detects performance regression bugs by differential testing. It uses the response time in the older version of the target DBMS as a baseline. Amoeba [37] sets the baseline as the execution of equivalent queries on the target DBMSs. *CERT* [11] identifies performance issues stemming from unexpectedly estimated cardinalities, which indicate the projected number of rows returned by a query. It transforms a query into a more constrained form, where the estimated cardinality should ideally be no greater than that of the original query. Any violation suggests a potential performance concern.

Differently, Hulk focuses on performance anomalies related to the optimizer and variation of data volume. It changes the data volume on each query and checks whether the growth in data causes abnormal behaviors in the target comparable DBMSs. After identifying the performance cliffs, it validates the performance anomaly with the plan cost comparison.

Data Generation in DBMS. Generating a sufficient number of diverse data is a prerequisite for evaluating the performance of the DBMS. Conventional works generate data with varying distributions and intra- and inter-table correlations [17, 22, 23]. They focus on generating large databases as a benchmark and performing analyses and testing on them. Later works begin to consider more the relationship between query and data. QAGen [14] proposes query-aware data generation. Specifically, given the schema and a query, QAGen extends symbolic execution to generate data which guarantees that the query can get the desired query results. ADUSA [31] also uses a query-aware approach but targets generating a large set of small databases for exhaustively testing a DBMS. Domino [10] introduces a method for automatically generating data to detect schema faults that violate integrity constraints. *Touchstone* [53] achieves a query-aware parallel data generation that bounds usage to memory to improve the performance.

Hulk differs from these works in generating both data and query. It first generates initial data based on the schema. Then, it utilizes the schema to synthesize queries that contain entities that are sensitive to data and find data-sensitive performance anomalies. For each query, Hulk increases the amount of data volume and checks the response time. By aligning data and queries more closely, Hulk could find data-sensitive performance anomalies that might be ignored by other methods.

8 Conclusion

We propose Hulk to detect data-sensitive performance anomalies. We find that various data distributions are important for performance testing, however, current works usually test DBMSs in a fixed data volume, which might ignore the anomalies associated with the data volume. Consequently, we design a data-driven analysis approach to address the test oracle problem. Hulk reports 135 anomalies in six widely-used DBMSs. Among them, 129 have been confirmed.

Data Availability

The artifact of Hulk is available at https://anonymous.4open.science/r/Hulk-C0B0.

Acknowledgments

We thank the shepherd and reviewers for their valuable comments. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002, U2441238), and CCF-Aliyun Fund Program (No. 2024007).

References

[1] [n. d.]. databases. https://en.wikipedia.org/wiki/Database. Accessed: April 24, 2025.
[2] [n. d.]. MariaDB. https://mariadb.org/. Accessed: April 24, 2025.
[3] [n. d.]. MariaDB GitHub. https://github.com/MariaDB/server. Accessed: 2023-10-05.
[4] [n. d.]. MySQL. https://www.mysql.com/. Accessed: April 24, 2025.
[5] [n. d.]. PostgreSQL. https://www.postgresql.org/. Accessed: April 24, 2025.
[6] [n. d.]. PostgreSQL derived databases. https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases. Accessed: April 24, 2025.
[7] [n. d.]. SQLite and SQLite alternatives - databases for the Mobile and IoT edge. https://objectbox.io/sqlite-alternatives/. Accessed: April 24, 2025.
[8] AISWare. [n. d.]. AntDB Github. https://github.com/ADBSQL. Accessed: April 24, 2025.
[9] AISWare. [n. d.]. AntDB Website. http://www.antdb.co//. Accessed: April 24, 2025.
[10] Abdullah Alsharif, Gregory M Kapfhammer, and Phil McMinn. 2018. DOMINO: Fast and effective test data generation for relational database schemas. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 12–22.
[11] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[12] Daniel Bartholomew. 2014. *MariaDB cookbook*. Packt Publishing Ltd.
[13] Youssef Bassil. 2012. A comparative study on the performance of the Top DBMS systems. *arXiv preprint arXiv:1205.2889* (2012).
[14] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 341–352.
[15] BlackFriday. [n. d.]. Get Access to Black Friday Prices All Year. https://blackfriday.com/. Accessed: April 24, 2025.
[16] Avatar Magnus Blåudd. [n. d.]. Percona Website. https://www.percona.com/. Accessed: April 24, 2025.
[17] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible database generators. In *Proceedings of the 31st international conference on Very large data bases*. 1097–1107.
[18] Oļegs Čapligins and Andrejs Ermuiža. [n. d.]. MySQL Database Management System Forks Comparison and Usage. ([n. d.]).
[19] CL Philip Chen and Chun-Yang Zhang. 2014. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information sciences* 275 (2014), 314–347.
[20] Raj Chhikara. 1988. *The inverse Gaussian distribution: theory: methodology, and applications*. Vol. 95. CRC Press.
[21] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *Conference on Automated Software Engineering (ASE'22)*.
[22] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. 243–252.
[23] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and realistic data generation. In *Proceedings of the 32nd international conference on Very large data bases*. 1243–1246.
[24] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[25] Toshihide Ibaraki and Tiko Kameda. 1984. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)* 9, 3 (1984), 482–502.
[26] Yannis E Ioannidis. 1996. Query optimization. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 121–123.
[27] ISO/IEC. [n. d.]. ISO/IEC 9075-1:2003. https://www.iso.org/standard/34132.html. Accessed: April 24, 2025.
[28] Solid IT. [n. d.]. DB-Engines Ranking. https://db-engines.com/en/ranking. Accessed: April 24, 2025.
[29] Matthias Jarke and Jurgen Koch. 1984. Query optimization in database systems. *ACM Computing surveys (CsUR)* 16, 2 (1984), 111–152.
[30] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2020. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems (to appear). In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*. Tokyo, Japan.
[31] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O Laleye, and Sarfraz Khurshid. 2008. Query-aware test generation using a relational constraint solver. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 238–247.
[32] Sang Gyu Kwak and Jong Hae Kim. 2017. Central limit theorem: the cornerstone of modern statistics. *Korean journal of anesthesiology* 70, 2 (2017), 144–156.
[33] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering* 6 (2021), 86–101.
[34] Jonathan Lewis and Thomas Kyte. 2006. *Cost-based Oracle fundamentals*. Springer.

[35] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. [n. d.]. Sequence-Oriented DBMS Fuzzing. In *2023 IEEE International Conference on Data Engineering (ICDE)*. IEEE.
[36] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS bugs via configuration-based equivalent transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
[37] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. (2022).
[38] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.
[39] Douglas C Montgomery and George C Runger. 2010. *Applied statistics and probability for engineers*. John wiley & sons.
[40] Oracle. [n. d.]. MySQL Bug List. https://bugs.mysql.com/. Accessed:April 24, 2025.
[41] Oracle. [n. d.]. MySQL Explain Manual. https://dev.mysql.com/doc/refman/8.0/en/using-explain.html. Accessed: April 24, 2025.
[42] PingCAP. [n. d.]. TiDB. https://github.com/pingcap/tidb. Accessed: April 24, 2025.
[43] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. 2003. *Database management systems*. Vol. 3. McGraw-Hill New York.
[44] Kim-Thomas Rehmann, Changyun Seo, Dongwon Hwang, Binh Than Truong, Alexander Boehm, and Dong Hun Lee. 2016. Performance monitoring in sap hana's continuous integration process. *ACM SIGMETRICS Performance Evaluation Review* 43, 4 (2016), 43–52.
[45] M Rigger and Z Su. [n. d.]. Finding Bugs in Database Systems via Query Partitioning. PACMPL 4 (OOPSLA)(Nov 2020).
[46] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.
[47] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation OSDI 20)*. 667–682.
[48] Murray Rosenblatt. 1956. A central limit theorem and a strong mixing condition. *Proceedings of the national Academy of Sciences* 42, 1 (1956), 43–47.
[49] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2018. SQLsmith: a random SQL query generator. https://github.com/anse1/sqlsmith
[50] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany
[51] Rick F Van Der Lans. 1989. *The SQL standard: a complete guide reference*. Prentice Hall International (UK) Ltd.
[52] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 328–337.
[53] Qingshuai Wang, Yuming Li, Rong Zhang, Ke Shu, Zhenjie Zhang, and Aoying Zhou. 2022. A Scalable Query-Aware Enormous Database Generator for Database Evaluation. *IEEE Transactions on Knowledge and Data Engineering* (2022).
[54] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, et al. 2020. Tempura: A general cost based optimizer framework for incremental data processing (extended version). *arXiv preprint arXiv:2009.13631* (2020).
[55] Michael Widenius, David Axmark, and Kaj Arno. 2002. *MySQL reference manual: documentation from the source*. " O'Reilly Media, Inc.".
[56] Zhiyong Wu, Jie Liang, Jingzhou Fu, Mingzhe Wang, and Yu Jiang. 2024. PUPPY: Finding Performance Degradation Bugs in DBMSs via Limited-Optimization Plan Construction. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 560–571.
[57] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. 2022. Unicorn: Detect Runtime Errors in Time-Series Databases With Hybrid Input Synthesis. In *Symposium on Software Testing and Analysis (ISSTA'22)*.
[58] Khaled Yagoub, Peter Belknap, Benoit Dageville, Karl Dias, Shantanu Joshi, and Hailing Yu. 2008. Oracle's SQL Performance Analyzer. *IEEE Data Eng. Bull.* 31, 1 (2008), 51–58.
[59] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D Viglas, and Allison Lee. 2018. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.
[60] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing Database Management Systems with Language Validity and Coverage Feedback. In *The ACM Conference on Computer and Communications Security (CCS), 2020*.

Received 2024-10-31; accepted 2025-03-31