# THANOS: DBMS Bug Detection via Storage Engine Rotation Based Differential Testing

Ying Fu*, Zhiyong Wu†, Yuanliang Zhang*, Jie Liang†, Jingzhou Fu†, Yu Jiang†✉
Shanshan Li*✉, Xiangke Liao*
* National University of Defense Technology, China
† KLISS, BNRist, School of Software, Tsinghua University, China

***Abstract*—Differential testing is a prevalent strategy for establishing test oracles in automated DBMS testing. However, meticulously selecting equivalent DBMSs with diverse implementations and compatible input syntax requires huge manual efforts. In this paper, we propose THANOS, a framework that finds DBMS bugs via storage engine rotation based differential testing. Our key insight is that a DBMS with different storage engines must provide consistent basic storage functionalities. Therefore, it's feasible to construct equivalent DBMSs based on storage engine rotation, ensuring that the same SQL test cases to these equivalent DBMSs yield consistent results. The framework involves four main steps: 1) select the appropriate storage engines; 2) extract equivalence information among the selected storage engines; 3) synthesize feature-orient test cases that ensure the DBMS equivalence; and 4) send test cases to the DBMSs with selected storage engines and compare the results.**

**We evaluate THANOS on three widely used and extensively tested DBMSs, namely MySQL, MariaDB, and Percona against state-of-the-art fuzzers SQLancer, SQLsmith, and SQUIRREL. THANOS outperforms them on branch coverage by 24%–116%, and also finds many bugs missed by other fuzzers. More importantly, the vendors have confirmed 32 previously unknown bugs found by THANOS, with 29 verified as *Critical*.**

***Index Terms*—DBMS Testing, Differential Testing, Storage Engine**

## I. INTRODUCTION

Database Management Systems (DBMSs) provide a structured framework for efficient and secure data handling in contemporary computing. They enable the storage, retrieval, and management of vast amounts of data, enhancing accessibility and integrity [1, 2]. Testing DBMSs is essential to guarantee reliability, performance, and security, directly influencing the accuracy and consistency of data management in vital business and technological operations [3].

Establishing test oracles capable of discerning the correct behavior of a DBMS for a given input is a primary problem in automated DBMS testing [4, 5]. Differential testing has emerged as a prevalent strategy in this context [6, 7, 8], involving the comparison of outputs from equivalent DBMS implementations with compatible input syntax to detect inconsistencies. Within this approach, there are two primary schemes: version-based comparison and vendor-based comparison. In *version-based comparison*, identical SQL queries are executed across different DBMS versions, as done in APOLLO [9] to detect performance regressions in MySQL. However, this approach may overlook real issues due to minor version variability. In *vendor-based comparison*, equivalent SQL statements are executed on various DBMSs, like the logic testing of SQLite with different SQLs on PostgreSQL, MySQL, Microsoft SQL Server, and Oracle [6, 8, 10]. However, adapting this scheme to more complex systems poses challenges due to their diverse features, indexing strategies, and data storage mechanisms. Ideally, a differential testing scheme should blend diverse DBMS implementations with similar input syntax to uncover hidden issues and reduce SQL generation costs. Unfortunately, existing methods struggle to achieve both goals simultaneously.

Recognizing these limitations, this paper introduces a novel differential testing scheme for DBMS bug detection. This scheme blends diversity in implementation with similarity in input syntax by rotating storage engines, the critical component in DBMS that serves as interfaces between the query planner and the underlying storage infrastructure. In essence, by equipping the DBMS with various storage engines, we create equivalent instances of the DBMS to establish test oracles for detecting bugs. Most production DBMSs support multiple storage engines [11, 12, 13], each tailored to specific capabilities (e.g., full-text search) and performance characteristics (e.g., column-based storage). These storage engines introduce complexity in indexing data, ensuring atomicity, and defining physical layouts. Moreover, different storage engines contribute to variations in execution plan optimization for SQL queries, enhancing diversity in DBMS logic. Since the storage engine is invoked after SQL parsing, semantic analysis, and query planning, it also ensures consistency in input syntax, reducing the cost of tool development.

The main challenge in rotating storage engine lies in *ensuring the generated test cases are equivalent across various storage engines while comprehensively testing the full spectrum of the storage engine's functionalities.* First, the generated test cases involving the unsupported features of the rotated storage engines are not entirely equivalent. For instance, in MySQL, *InnoDB* supports full-text indexes while *Memory* does not [14], which indicates that test cases involving the creation of full-text indexes are not entirely equivalent for these two engines. Consequently, the functionalities involved in the test case must be aligned with the features of the storage

✉Shanshan Li and Yu Jiang are the corresponding authors.

engine. Randomly selecting storage engine-related features may result in inequivalence, while selecting irrelevant features may prove ineffective. Second, it is crucial for the generated test cases to stimulate a broad spectrum of storage engine features, ensuring the comprehensive testing of the DBMS. It entails the deliberate selection of test scenarios that not only cover the basics but also delve into the myriad functionalities offered by different storage engines. This inclusivity ensures that the generated cases test a DBMS in diverse ways, examining its ability to handle multiple storage-related operations, data structures, and optimization mechanisms.

We take four main steps to address the challenge. Firstly, we select the appropriate storage engines as the foundation of equivalent DBMS construction with the guidance of covered features, which helps cover more storage engine features of DBMSs; secondly, we extract equivalence information among the selected storage engines to construct the equivalent DBMS; thirdly, we synthesize the feature-orient test cases with the equivalence information and metadata guidance, which not only promise the semantic correctness but also explore the full spectrum of storage engine features; and finally, we send the test cases to the DBMSs with selected storage engines and compare the results for bug detection.

To demonstrate the effectiveness of our approach, we implement a generic DBMS testing framework called THANOS and apply THANOS on three well-tested DBMSs: MySQL, MariaDB, and Percona. THANOS has discovered 32 new bugs confirmed by the corresponding vendors, including 11 bugs in MySQL, 17 bugs in MariaDB, and 4 bugs in Percona, respectively. Among these bugs, 29 bugs were verified as *Critical*. To assess the effectiveness of THANOS, we compare THANOS against contemporary state-of-the-art DBMS testing methods, namely SQLancer, SQLsmith, and SQUIRREL. THANOS covers 115.95%, 45.23%, 23.72% more branches, and finds 14, 13, and 11 more bugs in 24 hours on three DBMSs than SQLancer, SQLsmith, and SQUIRREL, respectively. Meanwhile, we also show the effectiveness of the feature-oriented test case synthesis.

In conclusion, our paper makes the following contributions:

- We propose a novel differential testing approach to complement existing methods of defining DBMS test oracle, which constructs equivalent DBMSs by rotating the storage engine components, serving as a reference for validating execution results.
- We implement THANOS, a DBMS testing framework that synthesizes test cases based on the features of the selected storage engine to ensure the equivalence of the tested DBMS instances. Any inconsistent execution results will be considered potential anomalies.
- We evaluate THANOS on three widely used and extensively tested DBMSs against other state-of-the-art techniques. The results show that THANOS outperforms others and 32 bugs are detected. It also found more basic blocks and bugs than other techniques.

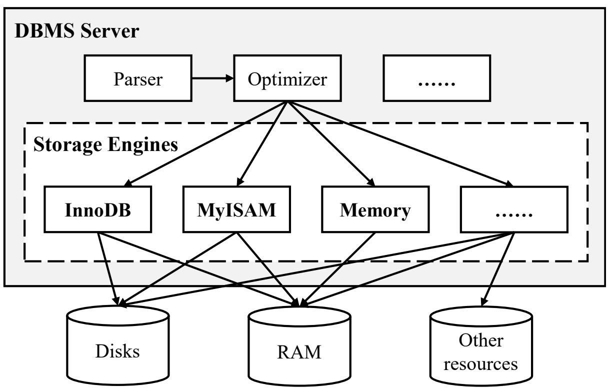


Fig. 1: Overview of MySQL Storage Engine Architecture.

## II. BACKGROUND AND MOTIVATION

**DBMS Storage Engines.** The storage engine in a DBMS plays a pivotal role as the foundational component responsible for managing data storage, retrieval, and manipulation within the database. It acts as an intermediary between the DBMS and the physical data storage, ensuring efficient data processing and access. Figure 1 illustrates the pluggable storage engine architecture within the MySQL server, one of the most widely used DBMSs. It underscores that MySQL supports various storage engines such as *InnoDB*, *MyISAM*, and *Memory*. These engines manage disk, memory, and other physical resources, providing essential storage support for higher-level modules. The storage engine component possesses several key characteristics: *1) Diversity:* Modern DBMSs typically offer support for multiple storage engines, each with unique features and capabilities tailored to specific use cases. *2) Significance:* As shown in Figure 1, the storage engines are intricately linked with many foundational components, such as *Optimizer* and *Parser*, thereby having a decisive impact on the overall performance of the database. *3) Scalability:* According to the architecture shown in Figure 1, the storage engine component is highly scalable, allowing for the seamless integration of new storage engines as plugins. This adaptability ensures that the system can meet emerging technological challenges and business needs efficiently.

**Storage Engine Features.** Storage engine features refer to the specific functionalities and attributes provided by the storage engines within a DBMS. These features are designed to optimize various aspects of database operations, ensuring efficient data management, retrieval, and manipulation. Formally, these features can be defined as a set $F$ that includes data indexing, transaction support, data partitioning, and others:

$$F = \{f_1, f_2, f_3, \dots, f_n\}$$

where each $f_i$ represents a distinct feature provided by the storage engine. For instance: $f_1$: Data Indexing, $f_2$: Transaction Support, $f_3$: Data Partitioning, $f_n$: Other features. In this paper, our primary focus is on those features that can be utilized and demonstrated through SQL statements. We follow the official DBMS documentation and classify storage

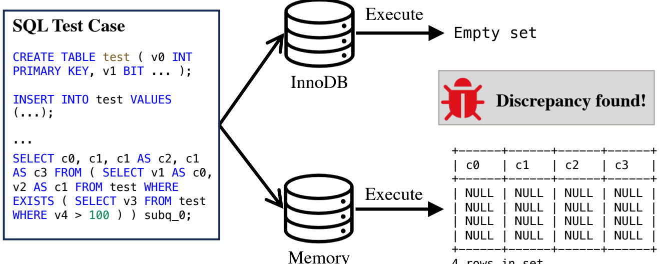


Fig. 2: A correctness bug found by THANOS in MySQL.

engine features into 10 major categories. The detailed process of storage engine features collection can be referenced in Section III-F. Due to the extensive length of the complete feature list, the detailed list of specific features can be found on THANOS's website [15].

**Basic Idea of THANOS.** The key insight of THANOS is to construct equivalent DBMSs by leveraging the storage engines, serving as a reference for validating execution results. By equipping the DBMS with different storage engines, we construct equivalent tested DBMS instances to build a test oracle for bug detection. Specifically, instead of spending considerable time finding DBMSs for differential comparison, THANOS opts to construct equivalents by utilizing the storage engines of the DBMS itself. When equivalent DBMS instances produce different results while executing the same test case, it indicates potential bugs in the DBMS's implementation.

Figure 2 illustrates a correctness bug in MySQL identified by THANOS utilizing the method of constructing storage engine-oriented equivalent DBMSs[1]. This bug exposes an inconsistency in handling NULL values between the InnoDB and Memory storage engines. Due to the extensive utilization of these two storage engines and MySQL's capability for real-time engine switching for database tables, the identified disparity may pose a considerable risk, potentially leading to severe disruptions in business logic and financial losses. One developer commented, "*we will definitely verify this bug as an S2 (serious) bug*". As shown in Figure 2, when the same test cases generated by THANOS are executed separately by both MySQL engines, the *InnoDB* returns an 'empty set', whereas the *Memory* returns a result set with four rows and four columns of NULL values. Theoretically, these two engines should return the same results. This inconsistency in the results indicates errors within the MySQL storage engines' implementation. In this case, both storage engines successfully execute the given test cases, generating query results without any crashes or error messages. Traditional DBMS fuzzing methods, such as SQUIRREL, would be unable to detect this issue. Similarly, metamorphic testing tools like SQLancer, which rely on specially crafted rules still prove ineffective in detecting this particular problem. As a result, this bug is difficult for other DBMS testing methods to detect within the same time frame.

[1]https://bugs.mysql.com/bug.php?id=112913

## III. DESIGN OF THANOS

Figure 3 illustrates the approach overview of THANOS, utilizing the storage engine as a key point to build equivalent DBMSs, thus defining test oracle for automated DBMS bug detection. THANOS consists of four main steps: In Step ①, THANOS selects appropriate storage engines to serve as the foundation for constructing equivalent DBMSs. In Step ②, THANOS extracts equivalence information among the selected storage engines from Step ①. This information includes the features supported by each storage engine, covering both common and complementary features. Then, in Step ③, THANOS synthesizes feature-oriented test cases using the extracted equivalence information. These test cases are designed to be semantically relevant to the DBMSs with the selected storage engines. In Step ④, THANOS executes the test cases on the DBMSs with the selected storage engines and analyzes the outcomes across different engines. Inconsistencies in the results indicate potential bugs, which THANOS can then detect.

The following text will provide definitions and detailed explanations for each step.

### A. Definitions

**DBMS Equivalence.** DBMS equivalence refers to the condition where different DBMS instances, despite variations in their implementation or configurations, exhibit identical functional behaviors. Equivalent DBMSs respond consistently to the same semantic queries and operations.

Formally, given a SQL statement $s$ and a DBMS $D$, we denote the result of executing $s$ on $D$ as $D(s)$. A test case in DBMS testing comprises a series of SQL statements. Two DBMSs are considered equivalent with respect to these test cases if the execution results of the SQL statements are consistent across both systems. Specifically, for any pair of corresponding semantic SQL statements ($s$ and $s'$) from two test cases ($T$ and $T'$), two DBMSs $D$ and $D'$ are equivalent with respect to $T$ and $T'$ if and only if:

$$\mathrm{D} \equiv \mathrm{D'} \Leftrightarrow \forall s \in T, \forall s' \in T', D(s) = D'(s')$$

In this paper, our focus is on DBMS equivalence related to storage engines. We consider a DBMS that supports a variety of optional storage engines. Let $Engines = \{e_1, e_2, \ldots, e_n\}$ represent the set of these available storage engines, with $\mathrm{D}_{e_i}$ defined as the instantiation of the DBMS equipped with storage engine $e_i$. Additionally, let $F = \{f_1, f_2, \ldots, f_m\}$ denote the set of distinct features inherent to each storage engine. Define $T = \{t_1, t_2, \ldots, t_k\}$ as the collection of test cases for the DBMS, where each test case $t_i$ includes SQL statements that invoke a combination of features from $F$.

Given two distinct storage engines $e_i, e_j$ within $Engines$, and a test case $t$ from $F_{equivalent}$ — the subset of test cases that include features common to both $e_i$ and $e_j$ — we postulate that if $\mathrm{D}_{e_i}$ and $\mathrm{D}_{e_j}$ exhibit identical results when executing any given test case $t$ within $F_{equivalent}$, these two DBMS

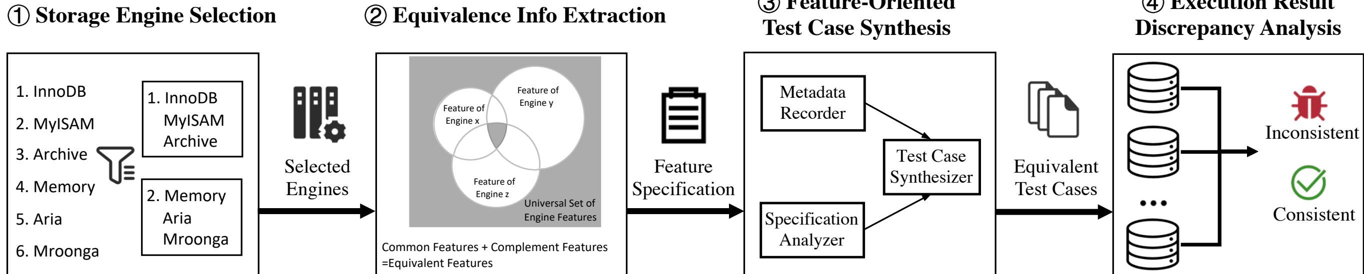


Fig. 3: Approach Overview of THANOS. In Step 1, THANOS selects storage engines forming the basis for constructing equivalent DBMSs. Step 2 involves extracting equivalence information among the chosen storage engines, comprising shared features supported by all selected engines. Moving to Step 3, THANOS synthesizes feature-oriented test cases, guaranteeing the DBMSs' equivalence. Finally, in Step 4, THANOS dispatches the test cases to the equivalent DBMSs, comparing the execution results to detect the bugs.

instances are deemed equivalent. This equivalence can be formally expressed as:

$$\mathrm{D}_{e_i} \equiv \mathrm{D}_{e_j} \Leftrightarrow \forall t \in F_{equivalent}, \mathrm{D}_{e_i}(t) = \mathrm{D}_{e_j}(t)$$

### B. Feature-Guided Storage Engine Selection

The primary strategy of THANOS entails outfitting the tested DBMS with various storage engines and generating the equivalent test cases based on these engine features. THANOS first selects the storage engines as the foundation and extracts the equivalent information to construct the equivalent DBMSs during the testing preparation phase. Some of these storage engine combinations have more functionalities and code branches than other combinations and may also have more potential vulnerabilities. Testing each combination equally could result in missing the chance to uncover potential vulnerabilities. For example, if we select two storage engines in MariaDB to construct the equivalent DBMS, 28 storage engine combinations would be generated. *InnoDB* and *Aria* support over 70 equivalent features, but the *Archive* and *Memory* only support 40 equivalent features. In practice, 70% of the real bugs we detected were in InnoDB-related storage engine combinations. Therefore, selecting the appropriate storage engines as the basis for constructing the equivalent DBMS during the testing preparation phase is very important.

We design the feature-guided storage engine selection, utilizing covered features and code branches as guidance to choose storage engine combinations. Algorithm 1 shows the detailed procedures for selecting storage engine combinations during DBMS testing. THANOS first generates all storage engine combinations and assigns an equal *weight value* to each combination (Lines 1-6). Throughout the DBMS testing phase, THANOS continually selects storage engine combinations to construct equivalent DBMS instances for testing. If a combination has a higher weight value, it is more likely to be chosen for testing (Lines 7-8). Using the selected combinations, THANOS extracts equivalent information and generates a specific number of test cases, then dispatches them to equivalent DBMSs for execution. If THANOS discovers new features or identifies new bugs during the testing of

**Algorithm 1:** Feature-Guided Storage Engine Combination Selection

```
Input : S: Set of the support storage engines
1  begin
2      C ← generateCombinations (S);
3      foreach c0 ∈ C do
4          f ← supportFeatures (c0);
5          initializeWeight (c0,f);
6      while true do
7          c ← chooseCombinationWithWeight (C);
8          if testCombination(c) then
9              addWeight(c);
10 Function testCombination(c0):
11     B ← currentBranches;
12     N ← currentBugNumber;
13     while canBeTested(c0) do
14         executionWithTestcase(c0, B, N);
15     if findNewbranches(B, N) or
        findNewBugs(B, N) then
16         return true
17 End Function
```

selected combinations, it increases the weight value of those combinations (Lines 11-17).

### C. Extract Equivalence Information

After selecting the storage engine combinations, THANOS then needs to generate the test cases for the DBMSs with the selected storage engines to perform testing. However, given the differing features and default configurations of various storage engines, mere test case generation cannot ensure equivalence among DBMSs employing different storage engines, potentially resulting in less coverage of storage engine-related features. To ensure consistency between DBMSs with selected different storage engines, THANOS extracts equivalence information from the selected storage engines to manage feature differences, using these shared features to maintain DBMS equivalence across various storage engines.

TABLE I: Illustrative List of Equivalent Features.

| | Data type | | | | Index type | | | Health-check | | Partition | Encryption | Transaction | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | INT | BIT | BLOB | ... | B-tree | Hash | ... | Check | Repair | | | | |
| InnoDB | ✓ | ✓ | ✓ | | ✓ | × | | ✓ | ✓ | ✓ | ✓ | ✓ | |
| MyISAM | ✓ | ✓ | ✓ | | ✓ | × | | ✓ | ✓ | ✓ | ✓ | × | |
| Archive | ✓ | × | ✓ | | × | × | | ✓ | ✓ | × | ✓ | × | |
| Mroonga | ✓ | ✓ | ✓ | | ✓ | × | | ✓ | ✓ | ✓ | ✓ | × | |

The equivalence information extracted by THANOS, denoted as $F_{equivalent}$, includes features commonly supported by the selected storage engines, as well as those not supported by any of them(complement features). To better understand the computation of $F_{equivalent}$, consider $U$ as the universal set of all features across storage engines, $k$ represents the number of selected storage engines. The formula for $F_{equivalent}$ is thus defined as:

$$F_{equivalent} = \bigcap_{i=1}^{k} f_i \ \cup \ \left( U - \bigcup_{i=1}^{k} f_i \right)$$

This formula calculates $F_{equivalent}$ as the union of two sets: the intersection of feature sets ($f_i$) of all $k$ selected engines, representing the common features to all engines, and the complement features indicates the features that are absent in all selected engines. In other words, the equivalence information includes features that are either shared by all the engines or missing in all of them, thus forming a basis for evaluating DBMS behavior equipped with different storage engines.

Table I presents the part of extracted equivalence information of four storage engines: *InnoDB*, *MyISAM*, *Archive* and *Mroonga* by THANOS as an example. The extracted equivalence information by THANOS contains the support status for various features of the selected storage engines, including data types, index types, partitioning, transactions, encryption, and so on. Due to space constraints, Table I does not encompass all features related to these storage engines. A comprehensive list of features and their support across different engines is available on our website [15]. THANOS automatically extracts the equivalence information, $F_{equivalent}$, based on the formula previously mentioned. This information corresponds to the features highlighted with a gray background in the table. For instance, all these engines support data encryption and none supports using hash indexes. These equivalent features will be used to synthesize test cases for equivalent DBMSs.

*D. Feature-Oriented Test Case Synthesis*

Using the extracted equivalence information from selected storage engines, THANOS systematically generates feature-oriented test cases. These test cases activate and schedule various storage engine features within the scope of equivalence information. This method ensures THANOS's ability to establish equivalence across DBMS instances using different storage engines while also conducting a thorough DBMS assessment by leveraging their unique features. Algorithm 2 illustrates the feature-oriented test case synthesis approach. To enhance comprehension, Figure 4 provides a synthesis process example. We'll offer a detailed explanation of the algorithm, using Figure 4 to highlight key concepts and steps.

**Algorithm 2:** Feature-Oriented Test Case Synthesis

```
Input  : E: The chosen storage engines
         F_equivalent: The equivalent features
Output : T: Test cases for chosen storage engines
1  begin
2      testcase ←init ();
3      Tables, testcase ←createTables (F_equivalent);
4      foreach t ∈ Tables do
5          t.index, testcase ←createIndex (t,
             F_equivalent);
6      Metadata ← getMetadata (Tables);
7      tables ← randomChooseTable (Tables);
8      foreach t ∈ tables do
9          t, Metadata ← changeTable(t,F_equivalent) ;
10         testcase ←generateQuery (Metadata);
11     testcase ←addStorageRelatedStmts (testcase,
         F_equivalent);
12     T ← convertForChosenEngines(testcase, E);
13     return T ;
```

THANOS takes the chosen storage engines and equivalent features as input, corresponding to the results of step ① and step ② of THANOS, respectively. The upper-left corner of Figure 4 shows the equivalent feature specification derived from Table I. It consists of standard SQL features and Non-standard SQL features. The standard SQL features represent the features that can be expressed as keywords or clauses on standard SQL statements. The Non-standard SQL features represent storage engine features that are expressed through non-standard SQL statements, such as health check statements (`CHECK TABLE`, `REPAIR TABLE`).

THANOS first synthesizes the data definition language (DDL) to construct the storage foundations, such as `CREATE TABLE` statements, `CREATE INDEX` statements, and so on, which involve managing physical storage (Lines 1-5). Then, THANOS synthesizes the data modification language (DML) as well as data query language (DQL) to modify and query the data stored in storage engines (Lines 6-10). Executing storage-related statements, such as data modification statements, in areas with frequent data changes will cause difficulty in generating semantic-correct SQL statements. To address these challenges, we have developed a metadata pool. This pool serves to preserve essential metadata, encompassing details about tables, columns, indexes, and data types within the DBMSs. With the metadata, THANOS can synthesize the semantic-correct test cases after complex operations such as

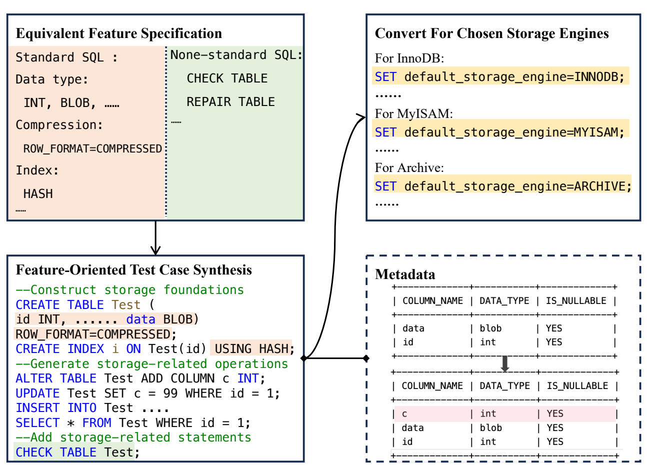


Fig. 4: An example of test case synthesis based on the equivalent features from selected storage engines.

data insertion and table structure modification. While the previous steps are related to standard SQL features of storage engines, there are some engine features expressed in non-standard SQL statements. When $F_{equivalent}$ includes these statements, THANOS incorporates them into the test case (Line 11). THANOS constraints the test cases with features in $F_{equivalent}$ by pushing the equivalent feature element into the SQL statement skeleton. For example, following the operation in Line 3, THANOS first synthesizes the "CREATE TABLE Test (node, node, node);" skeleton, then it replaces the node with the data type in $F_{equivalent}$(e.g. id INT). Lines 5 and 9 have the same usage for equivalent features as Line 3.

The bottom-left part of Figure 4 displays a simplified synthesized test case. Firstly, THANOS generates some `CREATE TABLE` and `CREATE INDEX` statements to construct the storage foundations. Secondly, THANOS maintains the metadata and synthesizes DML and DQL including the `ALTER TABLE` statement, `UPDATE` statement, `INSERT` statement, and `SELECT` statement. Thirdly, THANOS synthesizes the functionality-related statements. In the test case, lines highlighted with a background color correspond to specific storage engine features. For instance, including `ROW_FORMAT=COMPRESSED` indicates the implementation of row compression during table creation. In the lower right corner, it is shown that THANOS continuously records the metadata of tables, allowing for accurate data modeling. Finally, THANOS converts the test cases into versions executable by the specified selected storage engines (Line 12), which corresponds to the upper right part of Figure 4. THANOS specifies the storage engine for executing the test cases by setting the configuration parameter of `default_storage_engine`.

### *E. Discrepancy Analysis of Execution Results*

THANOS detects correctness bugs in the DBMS by distributing test cases generated in Section III-D for execution and analyzing result discrepancies. THANOS focuses on three types of DBMS information discrepancies during analysis. The first two types are compared immediately after each SQL statement execution, while the third type is compared only after completing the entire test case.

**Discrepancies in Query Results.** When test cases contain query statements (e.g., `SELECT`), the DBMS generates query results. THANOS then compares these results, reporting any discrepancies as potential correctness bugs. It is important to note that the order of data in query results may vary across DBMSs. THANOS accounts for this by ignoring the order during comparison.

**Discrepancies in Error Message.** Error messages are a crucial aspect as well. If equivalent DBMS instances produce inconsistent warnings or errors, THANOS will report these discrepancies. Furthermore, if the content of error messages differs between the various DBMSs, THANOS will also record these inconsistencies.

**Discrepancies in Database Final States.** After executing the test case's statements, THANOS compares the final state of databases between tested DBMS instances, such as verifying table consistency. If there is any inconsistency, THANOS reports the discrepancies in the final state of the databases.

When discrepancies are found, THANOS employs a widely-used deduplication method. Firstly, it minimizes test cases to reveal bugs. Secondly, THANOS logs each test case with its configuration and output data. If both match previous records, the test case is marked as a duplicate. Also, if the tested DBMS crashes, we log the bug's call stack for further deduplication and comparison.

### *F. Implementation*

We follow the official DBMS documentation [11, 12, 13] and classify storage engine features into 10 major categories: Data type, Index type, Data Integrity, Partition, Encryption, Compression, Transaction, Health check, Cache, and Update statistics. Secondly, for each engine, we count the number of features in each category. Each category typically includes around 2 to 40 specific features. Thirdly, we define the smallest unit of the counted feature as "feature–storage engine" (e.g., BTree-MyISAM), representing one feature supported by one storage engine. The detailed feature count for each DBMS can be found on THANOS's website [15].

The fundamental elements of THANOS include the selection of appropriate storage engines (detailed in Algorithm 1) and the generation of test cases ensuring their equivalence based on the shared features of the chosen engines (as outlined in Algorithm 2). THANOS implements the abstract syntax tree (AST) model for synthesizing feature-oriented test cases following the SQL-2003 standard [16], leveraging BISON 3.3.2 [17] and FLEX [18] to generate parser and lexer files. To prevent interference between DBMSs employing different storage engines, we utilize the Open Database Connectivity (ODBC) client interface. Through unified interface and scheduling, THANOS can establish direct connections to each equivalent DBMS.

## IV. Evaluation

In this section, we evaluate the effectiveness of detecting bugs with the test oracle constructed by THANOS. Our evaluation aims to answer the following research questions:

- **RQ1:** Can THANOS find DBMSs' bugs?
- **RQ2:** How does THANOS perform compared to other DBMS testing techniques?
- **RQ3:** How effective is the feature-oriented test case synthesis algorithm?

### A. Evaluation Setup

**Tested DBMSs and Compared Techniques.** We evaluated THANOS on three widely used DBMSs, namely MySQL [19], MariaDB [20], and Percona [21]. All three tested DBMSs support multiple storage engines and have been extensively tested by existing works. To evaluate the effectiveness of THANOS, we compared THANOS with the state-of-art DBMS test tools, SQUIRREL [22, 23], SQLancer [24], and SQLsmith [25, 26]. SQLsmith construct the AST model to generate amounts of queries for detecting crash bugs. SQLancer generates SQL queries and detects the logic bugs of DBMSs. SQUIRREL uses the coverage to guide the SQL generation for detecting the crash bugs of DBMSs.

**Experiment Environment.** We perform the evaluation on a machine running 64-bit Ubuntu 20.04, each DBMS testing instance runs in a docker container with 5 CPU cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 40 GiB of main memory. The three tested DBMSs are compiled by AFL++ [27] for collecting coverage and feedback.

### B. DBMS Bug Detection

THANOS successfully detected 32 previously unknown bugs in three well-tested DBMSs within three weeks. All 32 bugs were reported to and confirmed by developers, with no false positives. Of these, 29 were verified as *Critical*. Table II displays the distribution of bugs found by THANOS across various DBMS components and the severity of each bug.

Note that, different DBMSs have different severity levels. We use *Critical* to denote *Critical* and *Serious* in MySQL, *Blocker* and *Critical* in MariaDB, *Urgent* and *High* in Percona.

From Table II, we also observe that the bugs detected by THANOS span 12 components. Apart from identifying bugs related to *Storage Engines*, it also uncovers issues in other DBMS critical components such as the *Optimizer* and *Parser*. The storage engine acts as a bridge between the DBMS's physical storage and its key functionalities, closely linking to the core components of the DBMS. THANOS aims to expose vulnerabilities in the DBMS through storage engine rotations and extensive testing. THANOS tested the storage engine features, and while the DBMS executed these features, it also triggered critical business logic in other core components, thereby uncovering hidden bugs and vulnerabilities. The following are analyses of three bugs found by THANOS, each corresponding to one of the three types of discrepancies detected and analyzed by THANOS.

TABLE II: Statistics of bugs detected by THANOS.

| DBMS (Reported/Confirmed) | Component | Bug Severity and Number |
|---|---|---|
| MySQL (11/11) | Optimizer | Critical (2) |
| | DDL | Moderate (2) |
| | Options | Critical (1) |
| | Storage Engines | Critical (4), Moderate (2) |
| MariaDB (17/17) | Optimizer | Critical (7) |
| | Storage Engines | Critical (5) |
| | Window | Critical (1) |
| | Handler | Critical (1) |
| | Parser | Critical (2) |
| | Update | Critical (1) |
| Percona (4/4) | Optimizer | Critical (2) |
| | Storage Engines | Critical (1), Moderate (1) |
| **Total** | 12 components | 32 confirmed |

Listing 1: Inconsistent query results among InnoDB, Memory and MyISAM.

```
CREATE TABLE test ( v1 DECIMAL, v2 DATE, v3 FLOAT
    );
INSERT INTO test VALUES (......);
WITH cte AS ( SELECT v1 AS c0, v3 AS c2 FROM test
    WHERE EXISTS ( SELECT v1 FROM test WHERE v3 >
    50 ) ) SELECT c0, c2 FROM cte WHERE EXISTS (
    SELECT c0 FROM cte WHERE c2 < 100 );

-- Result of INNODB -- Result of MEMORY and MYISAM
-- Empty set            -- +------+------+
                        -- | c0 | c1 |
                        -- +------+------+
                        -- | NULL | NULL |
                        -- | NULL | NULL |
                        -- +------+------+
                        -- 2 rows in set
```

**Case 1: Discrepancies in Query Results.** Listing 1 demonstrates a bug we reported to MySQL[2], which was detected by THANOS due to discrepancies in query results. The Listing 1 includes a simplified SQL test case that triggered the issue. When executing the same test case using MySQL's *InnoDB*, *MyISAM*, and Memory storage engines, the *InnoDB* returns set, while the *MyISAM* and Memory return a result set with two rows and two columns of NULL values. This issue stems from varying implementations of NULL across different storage engines. The presence of inconsistent storage engine implementations poses significant security risks, as users may employ commands such as `ALTER TABLE` to change the storage engine of existing tables. Once the inconsistency is triggered, serious business logic issues may arise, potentially leading to financial losses. We have reported this inconsistency issue, sparking extensive discussions among developers, and efforts are underway to resolve and fix the problem.

Existing DBMS fuzz testing methods, such as SQUIRREL, typically focus on detecting issues like crashes. Or they require adherence to specific rules, as seen in SQLancer where the NOREC test oracle is optimizer-related, and TLP necessitates a pivot row. Identifying this issue, which involves comparing

[2]https://bugs.mysql.com/bug.php?id=112917

execution results across multiple engines, is challenging using current DBMS testing methods.

Listing 2: Inconsistent error message between CSV and ARCHIVE.

```
-- ENGINE = CSV
CREATE TABLE test (v0 INT NOT NULL, v1 CHAR(5)
    NOT NULL);
CREATE INDEX index0 ON test (v0) USING BTREE;
-- ERROR 1069 (42000): Too many keys specified;
    max 0 keys allowed

-- ENGINE = ARCHIVE
CREATE TABLE test (v0 INT NOT NULL, v1 CHAR(5)
    NOT NULL);
CREATE INDEX index0 ON test (v0) USING BTREE;
-- ERROR 1030 (HY000): Got error -1 - 'Unknown
    error -1' from storage engine
```

**Case 2: Discrepancies in Error Message.** Listing 2 demonstrates a bug we reported to MySQL[3], which was detected by THANOS due to discrepancies in error messages. The listing contains simplified SQL test cases that trigger the identified issue. Both the *CSV* storage engine and the *Archive* storage engine in MySQL do not support the creation of B-tree indexes. When the *Archive* storage engine executes a statement to create a B-tree index, it throws an "unknown error," while the *CSV* storage engine provides error messages with user-friendly prompts. Within the same DBMS, error messages originating from similar issues should maintain user-friendly and consistent information prompts. The developers have acknowledged the issue and have stated their intention to address and fix it.

THANOS has also identified several similar issues through error message prompts, such as inconsistent error codes for analogous problems. Although these issues have minimal impact on security, they are crucial for the reliability and robustness of the DBMS. Current DBMS testing tools are unable to assist developers in identifying such problems, whereas THANOS, through its testing, can automate the detection of these issues.

Listing 3: Discrepancies in database final state with Mroonga storage engine, which has been hidden for almost a decade.

```
SET default_storage_engine=Mroonga;
CREATE TABLE test(pk INT AUTO_INCREMENT, a INT, b
    INT, c INT, d INT, PRIMARY KEY (pk), KEY (a));
INSERT INTO test(a, b) VALUES (0,100),(200,2000);
INSERT INTO test(c, d) VALUES (0,100),(200,2000);
CREATE TRIGGER tr1 AFTER UPDATE ON test FOR EACH
    ROW SET @a= 100;
UPDATE test SET b = 3 WHERE a = 0;
-- Server crash
```

**Case 3: Discrepancies in Database Final States.** Listing 3 demonstrates a bug we reported to MariaDB[4], which has been hidden over 10 years and was detected by THANOS due to discrepancies in *database final state*. When executing the test case in Listing 3 using the *Mroonga* storage engine, it triggers a server crash. However, when executed on *InnoDB* or other storage engines, the query results can be obtained normally. This test case first creates a table `test` with 5 columns and then inserts data into the table. After the data insertion, THANOS creates a trigger `tr1` on the updated data region of table `test`. If the data stored in table `test` are updated, the trigger `tr1` will automatically reset the value of column `a` in table `test`. Then THANOS executes the `UPDATE` statement to update the data in table `test` and finally triggers the crash of MariaDB with *Mroonga* storage engine.

Based on the call stack information, we identified the issue originating from a problem in the *Mroonga* storage engine when handling the auto_increment feature, and this part of the code has not been updated for 10 years. In other words, THANOS discovered an issue in the *Mroonga* storage engine that had been hidden for nearly a decade. Existing DBMS testing methods struggle to identify this problem as they perform DBMS testing only with the default storage engine and do not change the storage engines during the test period. THANOS can detect the bug with the storage engine-orient equivalent DBMS construction. First, with the feature-guided storage engine combination selection, THANOS can select the *Mroonga* and *InnoDB* combination and then extract the equivalence information. Then, with the feature-orient test case synthesis algorithm, the synthesized test cases contain amounts of intensive data storage-related operations, which could be more likely to trigger issues related to DBMS storage engines.

### C. Comparison with Existing Techniques

To assess the effectiveness of THANOS, we conducted a comparative study that pitted THANOS against contemporary state-of-the-art DBMS testing methods, namely SQLancer, SQLsmith, and SQUIRREL. Each DBMS underwent a 24-hour testing period using these tools, and we documented

TABLE III: Number of bugs detected by THANOS, SQLancer, SQLsmith and SQUIRREL on 3 DBMSs in 24 hours.

| DBMS | SQLancer | SQLsmith | SQUIRREL | THANOS |
|---|---|---|---|---|
| MySQL | 0 | 1 | 1 | 6 |
| MariaDB | 0 | 0 | 1 | 5 |
| Percona | 0 | 0 | 1 | 3 |
| Total | 0 | 1 | 3 | 14 |
| Increment | 14 ↑ | 13 ↑ | 11 ↑ | – |

TABLE IV: Number of branches covered by THANOS, SQLancer, SQLsmith, and SQUIRREL on 3 DBMSs in 24 hours.

| DBMS | SQLancer | SQLsmith | SQUIRREL | THANOS |
|---|---|---|---|---|
| MySQL | 59,242 | 93,742 | 109,323 | 120,156 |
| MariaDB | 60,293 | 88,923 | 100,920 | 132,532 |
| Percona | 63,829 | 89,987 | 109,823 | 143,293 |
| Total | 183,364 | 272,652 | 320,066 | 395,981 |
| Increment | 115.95% ↑ | 45.23% ↑ | 23.72% ↑ | – |

[3] https://bugs.mysql.com/bug.php?id=113300
[4] https://jira.mariadb.org/browse/MDEV-32488

TABLE V: The number of triggered features and feature-related statements ratios in test cases generated by THANOS- and THANOS in 24 hours.

| DBMS | | Number of Triggered Features | | | Feature-Related Statements Ratios | | |
|---|---|---|---|---|---|---|---|
| Name | Features | THANOS- | THANOS | Increment | THANOS- | THANOS | Increment |
| **MySQL** | 289 | 70 | 257 | 187↑ | 67.18% | 83.32% | 16.14%↑ |
| **MariaDB** | 492 | 161 | 434 | 273↑ | 63.11% | 85.59% | 22.48%↑ |
| **Percona** | 359 | 92 | 306 | 214↑ | 65.15% | 82.75% | 17.60%↑ |

the number of bugs they uncovered. All identified bugs were reported to the developers, and the confirmed bug count was used as the final result. To ensure a fair comparison, we collected the generated test cases from each testing method and dry-ran the test cases to collect the uniform branch coverage.

THANOS outperforms other DBMS testing methods in detecting bugs and covering new branches. Table III shows the number of bugs detected by each method in 24 hours. THANOS detected a total of 14 bugs. From the table, we can see that THANOS finds more bugs than other state-of-the-art DBMS testing methods. Specifically, THANOS found 14, 13, and 11 more bugs than SQLancer, SQLsmith, and SQUIRREL, respectively. Note that all 14 bugs discovered by THANOS during the 24-hour experiment were subsequently confirmed as previously unknown bugs. Table IV shows the number of branches covered by each DBMS testing method in 24-hour experiments. From the table, we can see that THANOS outperformed other testing methods in terms of branch coverage. Specifically, THANOS totally covered 115.95%, 45.23%, and 23.72% more branches than SQLancer, SQLsmith, and SQUIRREL, respectively.

The main reason for THANOS's improvement in bug detection and branch coverage stems from its adept utilization of the DBMS storage engine components for testing purposes. On one hand, THANOS systematically tests various storage engines during the evaluation of each DBMS, whereas conventional DBMS testing methods typically focus on the default storage engine. Consequently, THANOS conspicuously manages to test a more extensive portion of DBMS code. Figure 5 illustrates the branch coverage results after a 24-hour test on the storage engine components, showcasing that THANOS achieves an average coverage of approximately 60% more branches compared to other testing methods. As mentioned earlier, the storage engine components serve as the core link between the underlying storage and the database system. By thoroughly testing the features of storage engines, comprehensive evaluation, and testing of the overall features of the DBMS system become possible. Therefore, from a holistic perspective, the improvement in coverage is quite significant for the DBMS.

On the other hand, THANOS aims to construct the equivalent DBMSs and thoroughly test its features. In comparison to existing tools, it can test a broader spectrum of DBMS functionalities. Specifically, the NoREC test oracle in SQLancer detects logical errors in the DBMS optimizer by constructing equivalent optimized and unoptimized queries based on optimizer rules. However, this method only covers SQL syntax that complies with optimizer rules. In contrast, THANOS lacks such limitations, as the storage engine, being the underlying software of the DBMS, essentially encompasses all the functionalities of the DBMS. Meanwhile, THANOS performs a comprehensive functional modeling of the DBMS storage engine components, enabling thorough testing of the DBMS when generating test cases. Even during the generation of test cases, THANOS incorporates numerous non-standard SQL statements related to storage engine features. In comparison, tools like SQLancer and SQLsmith use standard SQL syntax to build test cases, resulting in THANOS covering more branches and detecting more bugs.

### D. *Effectiveness of Feature-Oriented Test Case Synthesis*

To evaluate the effectiveness of our feature-oriented test case synthesis algorithm, we developed a comparison tool, THANOS-, which generates test cases randomly without a focus on the storage engines' features. We conducted a 24-hour experiment running both THANOS- and THANOS on MySQL, MariaDB, and Percona. During this period, we collected data on the number of features triggered and the proportion of feature-related statements produced by each tool for comparative analysis.

Table V presents the number of features activated and the proportion of statements in the test case with features for both tools across three DBMSs (MySQL, MariaDB, and Percona). The data indicates that THANOS generates more

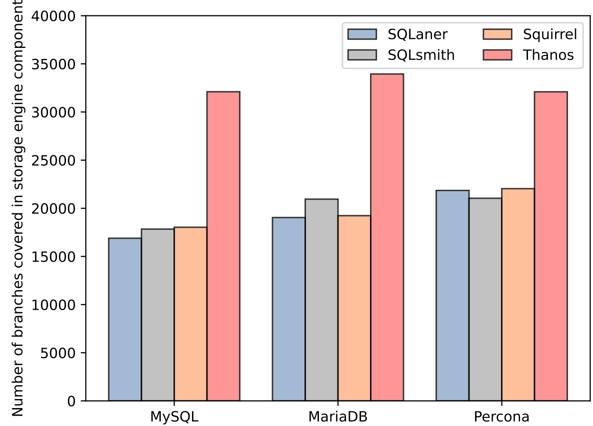


Fig. 5: Number of branches covered by THANOS, SQLancer, SQLsmith, and SQUIRREL in the *Storage Engine* component of each DBMS in 24 hours.

feature-related test cases than THANOS-, which benefits from its feature-oriented test case synthesis approach. In detail, THANOS activated 187, 273, and 214 more features than THANOS- in MySQL, MariaDB, and Percona, respectively. This is primarily because the feature-oriented synthesis significantly enhances the proportion of test cases that include storage engine features, with increases of 16.14%, 22.48%, and 17.60% in each DBMS, respectively. These outcomes align with our expectations, as the test case synthesis algorithm was specifically designed to trigger a broader range of storage engine features and behaviors within the DBMS. In contrast, random test case synthesis is less effective in this regard, often failing to adequately explore and activate diverse storage engine features.

## V. DISCUSSION

THANOS is adaptable to DBMSs supporting multiple storage engines. THANOS provides a template for feature lists [15] with 8 commonly used storage engines for adaptation, allowing for efficient organization based on existing lists. To improve the effectiveness of one specific storage engine, one could also gather other supported features from the DBMS's official documentation and add them to the feature list. The cost of the adaptation is acceptable since the manual effort is a one-time task, and the template can be reused. For example, it takes about 6 hours for a master to collect features of MySQL.

For DBMSs with a single storage engine, testing can still be conducted using an approach that involves constructing the equivalent DBMSs. For instance, we can perform differential testing using storage engines from different versions of DBMS, reusing the feature-oriented test case generation method. Alternatively, we can build equivalent DBMSs by replacing other components of the DBMS or modifying configurations related to the DBMS storage engine. In the future, we plan to expand our capabilities to support single-storage-engine configurations.

THANOS conducts DBMS bug detection through differential testing. Table II displays the distribution of bugs discovered by THANOS, indicating that approximately 40% (13/32) of the bugs are attributed to the storage engine. The remaining bugs occurred in core components closely associated with the storage engine, such as the Optimizer and Parser. These bugs are identified as a result of storage engine features triggering critical business logic during test case execution.

## VI. RELATED WORK

**Differential Testing.** DBMS differential testing involves executing identical SQL statements across different DBMSs to compare behaviors and outputs, aiming to identify inconsistencies, performance issues, and potential bugs. This method highlights how each system handles the same queries differently. It is categorized into version-based and vendor-based comparison schemes. In version-based comparison, identical SQL queries are executed across different versions of the same DBMS. For example, APOLLO [9] focuses on detecting performance bugs by comparing SQL execution speed across different DBMS versions. Vendor-based comparison entails running equivalent SQL statements on various DBMSs. For instance, RAGS [6] validates SQL outputs by comparing results from multiple DBMS vendors.

THANOS employs a specialized form of differential testing to evaluate DBMSs. It equips a DBMS with various storage engines and constructs equivalent test cases, thereby creating equivalent DBMSs. In doing so, THANOS combines diverse DBMS implementations with similar input syntax to uncover hidden issues and minimize the cost of SQL generation.

**Metamorphic Testing.** Metamorphic testing in DBMS involves transforming SQL queries and verifying if the resulting output changes align with expected behavior [28, 29, 30, 31]. It relies on metamorphic relations, logical links between input and output, for test case design, with transformations including sorting, filtering, or aggregating data. Developers use this to ensure correct and robust database functions, maintaining consistent accuracy despite input variations, crucial for revealing hidden bugs and bolstering DBMS reliability. Methods like Amoeba [32] detect performance bugs by comparing response times of semantically equivalent query pairs. SQLancer proposes constructing functionally equivalent queries to test one DBMS [30, 31, 33]. TxCheck [34] detects transactional bugs of DBMSs through graph-based oracle construction. DQE [35] verifies consistency of rows fetched by SQL statements designed to access the same rows.

Compared to metamorphic testing, our approach focuses more on changing the DBMS itself, specifically by altering the storage engine. Metamorphic testing relies on manually defining metamorphic relations, whereas our method automatically generates equivalent DBMSs, currently applicable for correctness testing. This approach can also be extended to other types of test oracles, such as authorization testing and compliance testing.

**DBMS Fuzzing.** Fuzz testing continuously generates and executes SQL test cases on a DBMS, monitoring system responses. DBMS fuzzers [22, 25, 33, 36, 37, 38, 39, 40, 41, 42], automate this process, focusing on creating complex SQL queries to uncover memory safety issues. They are categorized into generation-based and mutation-based approaches. Generation-based fuzzing generates large SQL queries using predefined models. SQLsmith [25] is a leading generation-based DBMS fuzzer that creates semantically correct SQL queries using an AST model, primarily limited to `SELECT` statements due to grammar constraints. Mutation-based fuzzers modify existing queries. Squirrel [22] enhances syntactic accuracy by aligning with SQL grammar and utilizing an Intermediate Representation (IR), facilitating type-based mutations while maintaining syntax correctness. LEGO [36] generates diverse SQL statement sequences by analyzing various types and relationships, while Griffin [37] introduces metadata graphs for SQL statements, offering a grammar-independent method to mutate SQL test cases.

To the best of our knowledge, THANOS is the first framework to utilize storage engine rotation based differential testing. Inspired by existing works, THANOS employs feature-

oriented test case synthesis techniques to assure equivalence among DBMSs and thereby offers a noteworthy extension to existing methodologies.

## VII. Conclusion

In this paper, we propose THANOS, a framework for discovering bugs in DBMSs via storage engine rotation based differential testing. By equipping the DBMS with different storage engines and generating test cases that trigger common features, THANOS constructs equivalent DBMS instances. Bug detection is achieved by comparing the execution results of these equivalent DBMSs. Experimental results indicate that THANOS outperforms existing testing methods by covering more branches and discovering more bugs, thereby demonstrating its effectiveness. Moreover, THANOS has discovered 32 new bugs confirmed by the corresponding vendors, including 11 bugs in MySQL, 17 bugs in MariaDB, and 4 bugs in Percona, with 29 verified as *Critical*.

## Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This research was funded by NSFC No. 62272473, the Science and Technology Innovation Program of Hunan Province (No. 2023RC1001) and NSFC No. 62202474.

## References

[1] Wikipedia, "Dbms," https://https://en.wikipedia.org/wiki/Database, 2023.

[2] M. Stonebraker, S. Madden, and P. Dubey, "Intel" big data" science and technology center vision and execution plan," *ACM SIGMOD Record*, vol. 42, no. 1, pp. 44–49, 2013.

[3] Wikipedia, "Database security," https://en.wikipedia.org/wiki/Database_security, 2023.

[4] W. E. Howden, "Theoretical and empirical studies of program testing," *IEEE Transactions on Software Engineering*, no. 4, pp. 293–298, 1978.

[5] Wikipedia, "Test oracle," https://en.wikipedia.org/wiki/Test_oracle, 2023.

[6] D. R. Slutz, "Massive stochastic testing of sql," in *VLDB*, vol. 98. Citeseer, 1998, pp. 618–622.

[7] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[8] Z. Cui, W. Dou, Q. Dai, J. Song, W. Wang, J. Wei, and D. Ye, "Differentially testing database transactions for fun and profit," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[9] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "Apollo: Automatic detection and diagnosis of performance regressions in database systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 1, pp. 57–70, 2019.

[10] Y. Zheng, W. Dou, Y. Wang, Z. Qin, L. Tang, Y. Gao, D. Wang, W. Wang, and J. Wei, "Finding bugs in gremlin-based graph database systems via randomized differential testing," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 302–313.

[11] MySQL, "Mysql storage engines," https://dev.mysql.com/doc/refman/8.0/en/storage-engines.html, accessed: August 16, 2024.

[12] MariaDB, "Mariadb storage engines," https://mariadb.com/kb/en/storage-engines/, accessed: August 16, 2024.

[13] Percona, "Percona storage engines," https://docs.percona.com/percona-server/8.0/glossary.html?h=storage+engine#storage-engine/, accessed: August 16, 2024.

[14] MySQL, "Mysql full-text index," https://dev.mysql.com/doc/refman/8.0/en/create-index.html/, 2023.

[15] "Thanos website," https://github.com/Thanos2024/Thanos, accessed: August 16, 2024.

[16] ISO/IEC, "Iso/iec 9075-1:2003," https://www.iso.org/standard/34132.html, accessed: August 16, 2024.

[17] "Bison," https://www.gnu.org/software/bison/, accessed: August 16, 2024.

[18] "Flex, the fast lexical analyzer generator," https://github.com/westes/flex, accessed: August 16, 2024.

[19] "Mysql," https://www.mysql.com/, accessed: August 16, 2024.

[20] "Mariadb," https://mariadb.org/, accessed: August 16, 2024.

[21] "Percona," https://www.percona.com/, accessed: August 16, 2024.

[22] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "Squirrel: Testing database management systems with language validity and coverage feedback," in *The ACM Conference on Computer and Communications Security (CCS), 2020*, 2020.

[23] C. chen, "Squirrel website," https://github.com/s3team/Squirrel, accessed: August 16, 2024.

[24] M. Rigger, "Sqlancer website," https://github.com/sqlancer/sqlancer, accessed: August 16, 2024.

[25] A. Seltenreich, B. Tang, and S. Mullender, "Sqlsmith: a random sql query generator," 2018. [Online]. Available: https://github.com/anse1/sqlsmith

[26] "Sqlsmith description," https://github.com/anse1/sqlsmith#description, accessed: August 16, 2024.

[27] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," 2020.

[28] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.

[29] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2013.

[30] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proceedings of the ACM on*

*Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.
[31] Rigger, Manuel and Su, Zhendong, "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*, 2020, pp. 1140–1152.
[32] X. Liu, Q. Zhou, J. Arulraj, and A. Orso, "Automatic detection of performance bugs in database systems using equivalent queries," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 225–236.
[33] M. Rigger and Z. Su, "Testing database engines via pivoted query synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 667–682.
[34] Z.-M. Jiang, S. Liu, M. Rigger, and Z. Su, "Detecting transactional bugs in database engines via Graph-Based oracle construction," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 397–417. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/jiang
[35] J. Song, W. Dou, Z. Cui, Q. Dai, W. Wang, J. Wei, H. Zhong, and T. Huang, "Testing database systems via differential query execution," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.
[36] J. Liang, Y. Chen, Z. Wu, J. Fu, M. Wang, Y. Jiang, X. Huang, T. Chen, J. Wang, and J. Li, "Sequence-oriented dbms fuzzing," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 668–681.
[37] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-free dbms fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
[38] M. Wang, Z. Wu, X. Xu, J. Liang, C. Zhou, H. Zhang, and Y. Jiang, "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, 2021, pp. 328–337.
[39] Z. Wu, J. Liang, M. Wang, C. Zhou, and Y. Jiang, "Unicorn: detect runtime errors in time-series databases with hybrid input synthesis," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 251–262.
[40] Y. Liang, S. Liu, and H. Hu, "Detecting Logical Bugs of DBMS with Coverage-based Guidance," in *Proceedings of the 31st USENIX Security Symposium (USENIX 2022)*, Boston, MA, aug 2022.
[41] J. Fu, J. Liang, Z. Wu, and Y. Jiang, "Sedar: Obtaining high-quality seeds for dbms fuzzing via cross-dbms sql transfer," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.
[42] J. Liang, Z. Wu, J. Fu, M. Wang, C. Sun, and Y. Jiang, "Mozi: Discovering dbms bugs via configuration-based equivalent transformation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.