Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction

Jiansen Song*
Institute of Software at CAS, China
songjiansen20@otcaix.iscas.ac.cn

Wensheng Dou†‡
Institute of Software at CAS, China
wsdou@otcaix.iscas.ac.cn

Yu Gao
Institute of Software at CAS, China
gaoyu15@otcaix.iscas.ac.cn

Ziyu Cui
Institute of Software at CAS, China
cuiziyu20@otcaix.iscas.ac.cn

Yingying Zheng
Institute of Software at CAS, China
zhengyingying14@otcaix.iscas.ac.cn

Dong Wang
Institute of Software at CAS, China
wangdong18@otcaix.iscas.ac.cn

Wei Wang†
Institute of Software at CAS, China
wangwei@otcaix.iscas.ac.cn

Jun Wei†
Institute of Software at CAS, China
wj@otcaix.iscas.ac.cn

Tao Huang‡
Institute of Software at CAS, China
tao@otcaix.iscas.ac.cn

ABSTRACT

Database Management Systems (DBMSs) are widely used to efficiently store and retrieve data. DBMSs usually support various metadata, e.g., integrity constraints for ensuring data integrity and indexes for locating data. DBMSs can further utilize these metadata to optimize query evaluation. However, incorrect metadata-related optimizations can introduce metadata-related logic bugs, which can cause a DBMS to return an incorrect query result for a given query.

In this paper, we propose a general and effective testing approach, *Raw database construction* (Radar), to detect metadata-related logic bugs in DBMSs. Given a database *db* containing some metadata, Radar first constructs a raw database *rawDb*, which wipes out the metadata in *db* and contains the same data as *db*. Since *db* and *rawDb* have the same data, they should return the same query result for a given query. Any inconsistency in their returned query results indicates a metadata-related logic bug. To effectively detect metadata-related logic bugs, we further propose a metadata-oriented testing optimization strategy to focus on testing previously unseen metadata, thus detecting more metadata-related logic bugs quickly. We implement and evaluate Radar on five widely-used DBMSs, and have detected 42 bugs, of which 38 have been confirmed as new bugs and 16 have been fixed by DBMS developers.

PVLDB Reference Format:
Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. PVLDB, 17(8): 1884 - 1897, 2024.
doi:10.14778/3659437.3659445

*All authors are affiliated with Key Lab of System Software at CAS, State Key Lab of Computer Science at Institute of Software at CAS, and University of CAS, Beijing. CAS is the abbreviation of Chinese Academy of Sciences.
†Affiliated with Nanjing Institute of Software Technology, University of CAS, Nanjing.
‡Wensheng Dou and Tao Huang are the corresponding authors.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659445

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tcse-iscas/radar.

1 INTRODUCTION

Database Management Systems (DBMSs) are fundamental to data-intensive applications that demand for efficient data storage and retrieval. Among various kinds of DBMSs, relational DBMSs, e.g., MySQL [17], PostgreSQL [19], and SQLite [22], are among the most popular DBMSs [8]. Relational DBMSs are built on the relational data model [35], and adopt Structured Query Language (SQL) [33] as their standard query language. In this work, we mainly focus on relational DBMSs. Unless otherwise stated, we simply call relational DBMSs as DBMSs.

In DBMSs, database metadata describes the organization of data in a database, including the structure of the database (e.g., data types and integrity constraints in the database schema), indexes, and storage configurations for the database. DBMSs can utilize these metadata to ensure data integrity in their managed databases. For example, a column $c1$ with the `TINYINT` data type in a table can only store numeric values between -128 and 127. If the `NOT NULL` constraint is applied on column $c1$, then `NULL` values cannot be stored in column $c1$. If the `UNIQUE` constraint is applied on column $c1$, then column $c1$ is not allowed to store duplicate numeric values.

DBMSs can also utilize database metadata (e.g., integrity constraints and indexes) to optimize query evaluation [32, 41, 47, 64]. For example, if the `NOT NULL` constraint is applied on column $c1$ in table $t1$, when evaluating the query $SELECT\ c1\ FROM\ t1\ WHERE\ ISNULL(c1)$, a DBMS can directly return `FALSE` for the predicate $ISNULL(c1)$ rather than evaluating this predicate on every record of column $c1$ in table $t1$ [12]. If an index is applied on column $c1$ in table $t1$, when evaluating the query $SELECT\ c1\ FROM\ t1\ WHERE\ c1 > 10$, a DBMS can quickly find the records that match the condition $c1 > 10$, and avoid unnecessary data accesses [3].

However, such complex metadata-related query optimizations can potentially introduce metadata-related logic bugs, i.e., *metaBugs* for short. *metaBugs* can cause a DBMS to silently compute an incorrect result for a given query, and can easily go unnoticed by DBMS developers. *metaBugs* in DBMSs are also difficult to detect

```
1.  -- Database db
2.  CREATE TABLE t1(c1 INT NOT NULL);
3.  INSERT INTO t1 VALUES (0);

4.  -- Database rawDb
5.  CREATE TABLE t1(c1 INT);
6.  INSERT INTO t1 VALUES (0);

7.  -- Query Q
8.  SELECT CAST(IFNULL(c1,'0') AS DATE) FROM t1;
9.  -- {0000-00-00} in database db ✘
10. -- {NULL} in database rawDb ✔

11. -- A simplified code patch for IFNULL function
12. if c1 has Constraint 'NOT NULL' then
13. - return c1
14. + c1' ← TypeCast(c1,'0')   -- Change 0 to '0'
15. + return c1'
16. else
17.   implement IFNULL correctly
```

Listing 1: TiDB#41734. The IFNULL function incorrectly handles the NOT NULL constraint in TiDB.

automatically, since we lack effective strategies to specifically test metadata-related optimizations, as well as effective test oracles to judge whether the metadata-related optimization in a DBMS behaves correctly for a given query.

Listing 1 shows a real-world *metaBug* TiDB#41374 detected by our approach in the widely-used DBMS TiDB [46]. The database *db* contains a table $t1$, which has a column $c1$ with the INT data type and the NOT NULL constraint (Line 1–3). Due to the incorrect optimization in the IFNULL function shown in Line 13 (the IFNULL function directly returns the value of column $c1$, i.e., an INT value 0), an incorrect value $0000 - 00 - 00$ is returned (Line 9). In the correct implementation (Line 14–15), the IFNULL function needs to cast the INT value 0 into '0', resulting in a correct query result NULL (Line 10) [9]. We report this bug to TiDB developers, who classified it as a *Major* bug and quickly fixed it.

Recently, researchers have proposed many approaches to detect logic bugs in DBMSs [34, 43, 61–63, 65, 66, 68]. RAGS [65] executes queries on multiple DBMSs, and identifies differences among their query results. However, differential testing cannot detect bugs that occur in all DBMSs and cannot test specific features of individual DBMSs. DQE [66], NoREC [61], and TLP [62] construct queries that are equivalent to original queries, and identify differences among their execution results. However, such query transformations cannot effectively disable the buggy metadata-related optimizations, and thus miss *metaBugs*. MutaSQL [34] can detect logic bugs related to indexes by adding indexes to tables and observing changes in the query results. However, MutaSQL cannot support most database metadata, e.g., NOT NULL, GENERATED, and FOREIGN KEY. Therefore, existing approaches cannot effectively detect *metaBugs* in DBMSs, and may miss many real-world *metaBugs*.

Given a database *db* with some metadata (e.g., integrity constraints and indexes), DBMSs can utilize these metadata to evaluate a query Q in an optimized way. If we remove these metadata from database *db*, DBMSs have to evaluate the same query Q in the corresponding unoptimized way. We observe that these two kinds of query evaluations should return the same query result for query Q. Listing 1 shows such an example. If the NOT NULL constraint is applied on column $c1$ in table $t1$, TiDB evaluates query Q in an optimized way (Line 12–15). If we remove the NOT NULL constraint (Line 5), TiDB evaluates Q in an unoptimized way (Line 16–17). If any inconsistency occurs between the optimized and unoptimized query evaluations, a *metaBug* is revealed.

Based on the above observation, we propose *Raw database construction* (Radar), a general testing approach to detect *metaBugs* in DBMSs. Specifically, we first randomly generate a database *db* that contains some metadata. Then, we construct a raw database *rawDb*, which has the same data with *db*, but does not contain the metadata in *db*. Given a query Q, we execute it on *db* and *rawDb*, respectively, and then compare their returned query results. Any inconsistency in their returned query results indicates a *metaBug* in the target DBMS. As shown in Listing 1, the database *db* and its corresponding raw database *rawDb* return different query results for query Q (Line 8–10), thus we can detect this *metaBug*. To improve testing efficiency and avoid testing databases with similar metadata, we further propose a metadata-oriented testing optimization strategy. So, we can continue testing databases with previously unseen metadata, and detect more unique *metaBugs* quickly.

To evaluate the effectiveness of Radar, we implement and evaluate it on five widely-used DBMSs, i.e., MySQL [17], SQLite [22], MariaDB [14], CockroachDB [5], and TiDB [24]. In total, we have detected 42 bugs among these DBMSs, of which 38 have been confirmed as unique and previously unknown bugs, and 16 bugs have been fixed by DBMS developers. Our experimental results also show that the metadata-oriented testing optimization strategy can help to test databases with diverse metadata faster and discover more unique bugs. We further compare Radar with four state-of-the-art DBMS testing approaches, i.e., DQE [66], NoREC [61], TLP [62], and MutaSQL [34] in their bug detection capabilities. At most 13 out of the confirmed *metaBugs* can be detected by these approaches. The DBMS developers greatly appreciate our approach, e.g., TiDB developers want to integrate Radar to their internal testing process [11]. We believe that the generality of Radar can greatly help improve the reliability of DBMSs.

In summary, we make the following contributions.

- We propose Radar, a general and effective testing approach to detect metadata-related logic bugs in DBMSs. We solve the test oracle problem by comparing the query results on databases that contain the same data but with different metadata.
- We propose a metadata-oriented testing optimization strategy to improve Radar's testing efficiency, which can test databases with diverse metadata and discover unique bugs quickly.
- We implement Radar and evaluate it on five widely-used DBMSs. We have found 42 bugs in these DBMSs, of which 38 have been confirmed as unique and new bugs.

2 PRELIMINARIES

We first introduce Database Management Systems (DBMSs) and our target DBMSs (Section 2.1). Then we explain database metadata (Section 2.2), and metadata-related query optimizations (Section 2.3) in our target DBMSs.

2.1 Database Management Systems and SQL

Database Management Systems (DBMSs) provide efficient data storage and retrieval for many business-critical applications. Specifically, relational DBMSs (e.g., MySQL [17], PostgreSQL [19], and

Table 1: Target DBMSs

DBMS	DB-Engines Ranking	GitHub Stars	Type
MySQL	2	9.6K	Traditional
SQLite	9	4.7K	Embedded
MariaDB	13	5k	Traditional
CockroachDB	64	28K	NewSQL
TiDB	88	35K	NewSQL

SQLite [22]) are built on the relational data model [35] that organizes data into tables composed of columns and rows. Relational DBMSs are among the most widely-used DBMSs.

Relational DBMSs commonly support SQL as their standard query language. According to the functionalities of SQL statements, they can be roughly divided into four categories: (1) statements for creating and modifying database metadata, e.g., `CREATE TABLE`, `ALTER TABLE` and `CREATE INDEX`; (2) statements for retrieving database metadata, e.g., `SHOW CREATE TABLE`; (3) statements for modifying data, e.g., `INSERT`, `UPDATE`, `DELETE` and `TRUNCATE`; (4) statements for retrieving data, e.g., `SELECT`.

In this work, we select five popular relational DBMSs (shown in Table 1), including two traditional DBMSs (MySQL [17] and MariaDB [14]), one embedded DBMS (SQLite [22]) and two NewSQL DBMSs (CockroachDB [5] and TiDB [24]). MySQL, MariaDB, and SQLite are among the most popular DBMSs according to the DB-Engines Ranking [8]. CockroachDB and TiDB are among the most popular DBMSs according to the GitHub Database Topic [6].

2.2 Database Metadata

DBMSs use database metadata to describe the organization and structure of data in their managed databases. Figure 1 shows a formal description for database metadata in our target DBMSs. A database consists of one or more tables. Each table consists of a table name (i.e., $tName$), one or more columns (i.e., *column*), some optional table constraints (i.e., *tConstraint*) and table configurations (i.e., $tConfig$). Each column consists of a column name (i.e., $cName$), a data type (i.e., $type$), and some optional column constraints (i.e., *cConstraint*). Note that all the above database metadata are user-configurable metadata. We do not consider system-level metadata as database metadata in our work, e.g., Write-Ahead Logging (WAL) logs [48] and Manifest files [13], which are typically maintained by the underlying DBMS.

Data types (i.e., $type$) define the type of values that can be stored in a column. DBMSs support various data types, e.g., `INT`, `VARCHAR`, `BLOB`, and `BOOLEAN`. Note that every column is required to have a data type in MySQL, MariaDB, CockroachDB and TiDB, except SQLite. When no type is specified on a column, SQLite uses `BLOB` as its type affinity.

Column constraints (i.e., *cConstraint*) are integrity constraints that are directly attached to a specific column, and are used to limit the values stored in the column. Our target DBMSs support three kinds of column constraints, i.e., `NOT NULL`, `DEFAULT`, and `GENERATED`. The `NOT NULL` constraint enforces a column not to store `NULL` values. The `DEFAULT` constraint specifies the default value of a column, e.g., $DEFAULT\ 0$. The `GENERATED` constraint specifies how values are automatically generated for a column through a specified

$$
\begin{aligned}
database &:= < table+ > \\
table &:= < tName, column+, tConstraint*, index*, tConfig* > \\
column &:= < cName, type, cConstraint* > \\
type &:= INT \mid VARCHAR \mid BOOLEAN \mid BLOB \mid DATE \mid \\
&\quad FLOAT \mid DECIMAL \mid \ldots \\
cConstraint &:= NOT\ NULL \mid DEFAULT\ const \mid GENERATED(exp) \\
tConstraint &:= [PRIMARY\ KEY \mid UNIQUE](cName+) \mid \\
&\quad CHECK(exp) \mid FOREIGN\ KEY(srcCol, targetCol) \\
index &:= [UNIQUE\ INDEX \mid INDEX](cName+) \\
tConfig &:= ENGINE \mid PARTITIONING\ BY\ HASH(cName) \mid \\
&\quad CHARACTER\ SET \mid COLLATE \mid \ldots \\
exp &:= const \mid cName \mid FUN(exp) \mid exp\ [+ \mid - \mid \times \mid \div]\ exp \mid \\
&\quad exp\ [OR \mid XOR \mid AND \mid]\ exp \mid NOT\ exp \mid \ldots
\end{aligned}
$$

Figure 1: A formal description for database metadata. $term+$ **(e.g.,** $table+$**) denotes one or more terms, and** $term*$ **(e.g.,** $tConstraint*$**) denotes zero or more terms.**

expression. For example, the value of a column $c1$ with the `INT` data type and the $GENERATED(c2 + c3)$ constraint is automatically obtained by calculating the sum of values in columns $c2$ and $c3$.

Table constraints (i.e., *tConstraint*) are integrity constraints that can be applied to one or more columns. We focus on four types of table constraints in our paper.

- The `PRIMARY KEY` constraint uniquely identifies each record in a table. `PRIMARY KEY` must contain unique values, and usually cannot contain `NULL` values. A table can have at most one `PRIMARY KEY`. The `PRIMARY KEY` can be applied on one or more columns. For example, the $PRIMARY\ KEY(c1, c2)$ represents that the combination of columns $c1$ and $c2$ is a unique row identifier.
- The `UNIQUE` constraint ensures that all values are unique. A table can have zero or more `UNIQUE` constraints. A `UNIQUE` constraint can be applied on one or more columns, and these columns can contain `NULL` values. For example, the $UNIQUE(c1, c2)$ constraint requires that the combination of values in columns $c1$ and $c2$ is unique.
- The `CHECK` constraint is used to limit the values that can be stored in the table through an expression. The `CHECK` constraint can be applied on one or more columns. For example, the $CHECK(c1 + c2 > 0)$ constraint specifies that the sum of columns $c1$ and $c2$ should be greater than 0.
- The `FOREIGN KEY` constraint is used to define a relationship between two columns in different tables. For example, the $FOREIGN\ KEY\ t1.c1\ REFERENCES\ t2(c2)$ constraint enforces the values of source column $c1$ in table $t1$ to be the subset of the values of target column $c2$ in table $t2$.

Index (i.e., *index*) is an auxiliary data structure that optimizes data querying on a table without having to search each row in the table. An `INDEX` can be created using one or more columns of a table, and may be declared as `UNIQUE INDEX`, which both creates an index and applies the `UNIQUE` constraint on the specified columns.

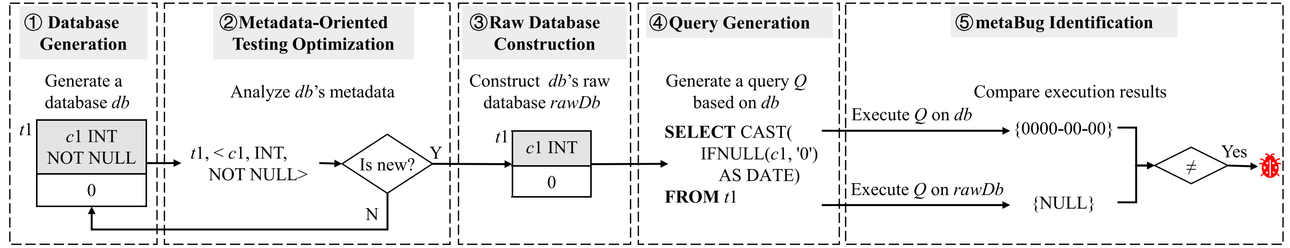

Figure 2: The architecture of Radar.

Table configurations (i.e., $tConfig$) are used to customize various aspects of the table's performance and storage characteristics. For example, we can use the `ENGINE` configuration in MySQL to specify the storage engine used for the table. We can specify the character set and collection by the `CHARACTER SET` and `COLLATE` configurations, respectively. We can use the `PARTITIONING` configuration in TiDB to distribute data physically in a table.

2.3 Metadata-Related Query Optimization

According to how a database metadata affects a query Q's execution result on a database, we classify database metadata into two categories, i.e., *compulsory* database metadata and *optional* database metadata. Compulsory database metadata must be required to execute a query Q, including table names, column names, data types and partial table configurations (e.g., `ENGINE` and `CHARACTER SET`). Optional database metadata cannot affect the execution result of the query Q, including column constraints, table constraints, indexes, and partial table configurations e.g., `PARTITIONING`.

DBMSs utilize optional database metadata to optimize query evaluation [32, 41, 47, 64]. For example, for a column that has the `NOT NULL` constraint, a DBMS can directly return the evaluation results of some functions and operators (e.g., the `IFNULL` function and the `IS NULL` operator) instead of evaluating the functions or operators on each record in the table. Moreover, a DBMS can utilize indexes to quickly match those records that satisfy a predicate. For example, a hash index on some columns can help MySQL perform efficient lookups for equality comparisons [16] on these columns.

3 APPROACH

We propose `Radar` to effectively detect *metaBug*s in DBMSs. The core idea of `Radar` is as follows. Given a database *db* with some optional database metadata (e.g., integrity constraints and indexes), we can wipe out these optional database metadata, and construct a raw database *rawDb*. Since *rawDb* does not contain optional database metadata, metadata-related optimizations in DBMSs will be disabled when evaluating a query Q on *rawDb*. Since *db* and *rawDb* contain the same data, evaluating query Q on *db* and *rawDb* should return the same query result. Any inconsistency in their returned query results indicates a *metaBug*.

3.1 `Radar`'s Architecture

Figure 2 shows the architecture of `Radar`, which consists of five components. The database generation component (①) generates valid databases, and the query generation component (④) utilizes the generated databases to generate valid queries. `Radar` utilizes a syntax-guided generation approach to avoid generating syntactically or semantically incorrect SQL statements. The raw database construction component (③) synthesizes SQL statements to create a raw database *rawDb* for a given database *db*. These two databases serve as a cross-reference oracle to validate the query executions in the *metaBug* identification component (⑤). To improve the testing efficiency, `Radar` utilizes the metadata-oriented testing optimization component (②) that filters out databases with similar metadata.

Algorithm 1: The workflow of `Radar`

Input: *maxQueries* (The number of generated queries)

1 **repeat**
2 $\quad db \leftarrow generateDatabase()$
3 $\quad$ **if** $isNewMetadata(db)$ **then**
4 $\quad\quad rawDb \leftarrow constructRawDatabase(db)$
5 $\quad\quad$ **for** $i \leftarrow 1$ **to** $maxQueries$ **do**
6 $\quad\quad\quad Q \leftarrow generateQuery(db)$
7 $\quad\quad\quad executeAndValidate(Q, db, rawDb)$
8 **until** *a fixed time budget*

Algorithm 1 shows the workflow of `Radar`. We explain the workflow by using the example in Figure 2. We first randomly generate a database *db*, e.g., a database contains a table $t1$, which has a column $c1$ with the `INT` data type and the `NOT NULL` constraint (Line 2). We further obtain and analyze *db*'s metadata to check whether we need to test *db* (Line 3). If *db*'s metadata is new (i.e., has not been previously tested), we will test *db* (Line 4-7). Otherwise, we will regenerate a new database (Line 2). For an interesting *db* with new database metadata, we construct its corresponding raw database *rawDb* by wiping out the optional database metadata in *db* (e.g, the `NOT NULL` constraint in column $c1$) (Line 4). We then generate a random query Q based on *db* (Line 6). We execute Q on *db* and *rawDb* respectively, compare the returned query results on *db* and *rawDb*, and report any inconsistency as a *metaBug* (Line 7). We continue testing *db* (Line 6-7) until we have generated a configurable number (i.e., *maxQueries*) of queries on *db*. We set *maxQueries* as $5,000$ in our experiments. We repeat the above testing process (Line 2-7) until a fixed time budget is exhausted.

3.2 Database and Query Generation

We present the database and query generation in `Radar` as follows.

Algorithm 2: Table generation

```
   Input: maxColumns (The maximum number of columns)
1  columnNum ← random(1, maxColumns)
2  stmt ← 'CREATE TABLE'
3  stmt ← stmt + ' ' + randomName() // e.g., t1
4  stmt ← stmt + '('
5  for i ← 1 to columnNum do
6      stmt ← stmt + ' ' + randomName() // e.g., c1
7      stmt ← stmt + ' ' + randomType() // e.g., INT
8      if randomBoolean() then
9          stmt ← stmt + ' NOT NULL'
10     if randomBoolean() then
11         stmt ← stmt + ' DEFAULT ' + randomConstant()
12     if randomBoolean() then
13         stmt ← stmt + ' GENERATED ALWAYS AS (' +
             randomExpression() + ')'
14     if i ≠ columnNum then
15         stmt ← stmt + ','
16 appendRandomTableConstraints(stmt) // e.g., UNIQUE(c1)
17 stmt ← stmt + ')'
18 appendRandomTableConfigurations(stmt) // e.g.,
     ENGINE=InnoDB
19 return stmt
```

3.2.1 *Database Generation.* We generate a database in the target DBMS by constructing and executing statements that are used to create a database schema and populate data. Specifically, we first create a database named *db* by executing the *CREATE DATABASE db* statement. We then create at most *maxTables* tables by generating and executing `CREATE TABLE` statements on *db* according to the database metadata in Figure 1 for the target DBMS.

Algorithm 2 illustrates the process of constructing `CREATE TABLE` statements in `Radar`. Specifically, we generate a table with a random name (Line 3) and a random number (at most *maxColumns*) of columns (Line 5-15). Each column has a random name (Line 6), a random data type (e.g., `INT` and `DOUBLE`) (Line 7) and some random column constraints (e.g., `NOT NULL`, `DEFAULT` and `GENERATED`) (Line 8-13). We further randomly add some table constraints (i.e., `CHECK`, `PRIMARY KEY` and `UNIQUE`) on some columns (Line 16), e.g., $UNIQUE(c1)$. If the target DBMS supports table configurations (e.g., MySQL and TiDB), we also randomly add some table configurations with proper values (Line 18), e.g., ENGINE=InnoDB. During table generation, we record the generated columns and table constraints to avoid violating semantic constraints in `CREATE TABLE` statements. For example, if we have already generated a `PRIMARY KEY`, we do not generate another one because a table can have at most one `PRIMARY KEY`.

For the generated tables, we randomly build some indexes by executing at most *maxIndexes* `CREATE INDEX` statements. When the generated database contains more than one table, we randomly add `FOREIGN KEY` constraints by executing at most *maxForeignKeys* `ALTER TABLE` statements. We finally populate random data into each table by executing at most *maxRows* `INSERT` statements. We follow a similar statement generation process when generating `CREATE INDEX`, `ALTER TABLE`, and `INSERT` statements.

SELECT [fields] FROM [table source] WHERE [predicate]

↓ ① Randomly generate a table source

SELECT [fields] FROM $t1$ LEFT JOIN $t2$ WHERE [predicate]

↓ ② Randomly generate fields based on the table source

SELECT $t1.c1$, $t2.c2$ FROM $t1$ LEFT JOIN $t2$ WHERE [predicate]

③ Randomly generate an expression based on the table source ↓

SELECT $t1.c1$, $t2.c2$ FROM $t1$ LEFT JOIN $t2$ WHERE $t1.c1$>1 AND $t2.c2$<3

Figure 3: An illustrative example for query generation.

`Radar` supports generating databases with various metadata, including various data types, column constraints, table constraints, indexes, and table configurations. The SQL features used in database generation can be either standard or dialectal.

Note that *maxTables*, *maxColumns*, *maxRows*, *maxIndexes* and *maxForeignKeys* are all configurable. We set them as 3, 3, 30, 5 and 3 in our experiments, respectively. As shown in Figure 2, we can generate a database *db* that contains a table $t1$ with a column $c1$ and one row with value 0. The column $c1$ has the `INT` data type and the `NOT NULL` constraint.

3.2.2 *Query Generation.* We generate queries based on the syntax of `SELECT` statements and the database metadata in *db*. The generated queries should be able to execute on both the generated database *db* and the raw database *rawDb* (Section 3.3), and produce deterministic query results on these two databases.

Specifically, we randomly generate a select-from-where query, and then randomly generate other optional clauses. Figure 3 illustrates an example for query generation. We first randomly generate a table source based on the generated database metadata. The table source may contain one table or multiple tables that are connected with a join operator, e.g., $t1$ *LEFT JOIN* $t2$ (step ①). We then randomly generate the select fields that contain some columns based on the table source, e.g., $t1.c1, t2.c2$ (step ②). We finally randomly generate an expression based on the columns in the table source to form the predicate, e.g., $t1.c1 > 1$ *AND* $t2.c2 < 3$ (step ③). Similarly, we can generate other optional clauses, e.g., `GROUP BY`, `ORDER BY` and `LIMIT`, and append these optional clauses to the generated select-from-where query.

Query generation in `Radar` supports almost all key features (e.g., joins, sub queries, and complex predicates) as specified by the SQL standard, and tailored features (e.g., high_priority and straight_join in MySQL) adopted by the target DBMS.

But, `Radar` cannot support some functions that can make it ineffective. First, `Radar` cannot support queries that contain the `DEFAULT` function and index hints. For example, the $DEFAULT(c1)$ function requires that the column $c1$ should have the `DEFAULT` constraint, and the index hint $@\{FORCE_INDEX = i0\}$ requires the existence of index $i0$. But, *rawDb* does not contain `DEFAULT` constraints and indexes, thus the above queries cannot be executed on *rawDb*. Second, `Radar` cannot support queries with database-related functions that return information specific to databases, e.g., the `CURRENT_DATABASE` function that returns the name of current database will return different names for *db* and *rawDb*. Third, `Radar` cannot support queries with non-deterministic functions, e.g., the `RAND` function that returns different values in different executions.

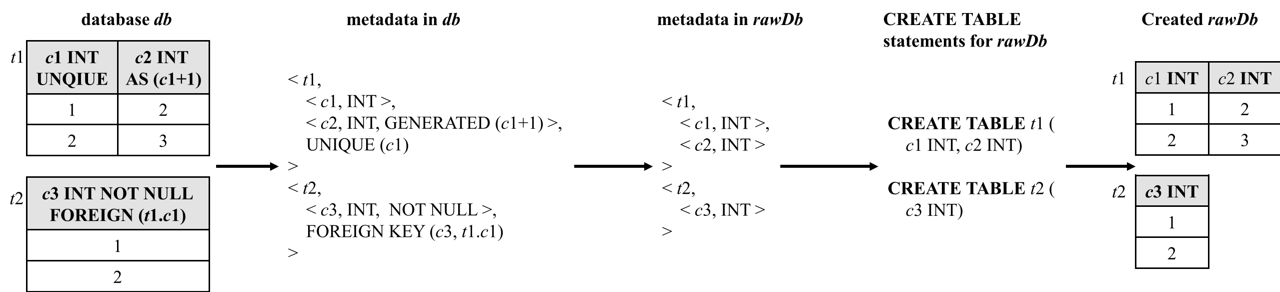

Figure 4: An example for constructing a raw database $rawDb$ for a given database db.

$$database := < table+ >$$
$$table := < tName, column+ >$$
$$column := < cName, type >$$
$$type := INT \mid VARCHAR \mid BOOLEAN \mid BLOB \mid DATE \mid FLOAT \mid DECIMAL \mid \ldots$$
$$tConfig := ENGINE \mid CHARACTER\ SET \mid COLLATE \mid \ldots$$

Figure 5: A formal description for database metadata in raw databases.

3.3 Raw Database Construction

After generating a database *db*, we construct its corresponding raw database *rawDb* by wiping out the optional database metadata in *db*. In the rest of this section, we first present a formal description for database metadata in raw databases in our target DBMSs (Section 3.3.1), and then explain how we construct a raw database *rawDb* for a given database *db* (Section 3.3.2).

3.3.1 Database Metadata in Raw Databases. A raw database can only contain compulsory database metadata, and can not contain any optional database metadata. Figure 5 shows a formal description for database metadata in raw databases in our target DBMSs. A raw database consists of one or more tables. Each table consists of a table name (i.e., *tName*), and one or more columns. Each column consists of a column name (i.e., *cName*) and a data type (i.e., *type*). Note that a raw database may contain some table configurations (e.g., `ENGINE` in MySQL and TiDB), because such table configurations are compulsory for these DBMSs.

3.3.2 Constructing Raw Databases. For a given database *db*, we first create an empty database as its corresponding raw database *rawDb* by executing the `CREATE DATABASE` statement. Then we analyze *db*'s database metadata, and construct *rawDb*'s database metadata according to Figure 5. Finally, we create the corresponding tables in *rawDb* based on *rawDb*'s metadata, and copy table data in *db* into *rawDb*.

Construct database metadata in *rawDb*. We construct database metadata in *rawDb* by first obtaining all the database metadata in *db* and then extracting compulsory database metadata in *db*. DBMSs usually provide some SQL statements to obtain the metadata in a database, e.g., the *SHOW CREATE TABLE* statement in MySQL, CockroachDB, and TiDB, and the *PRAGMA TABLE_INFO* statement in SQLite. We can obtain database metadata in *db* by directly applying such statements on *db* and formalizing the returned results of these statements in the manner shown in Figure 1. Then we construct the database metadata in *rawDb* by removing all the optional metadata in *db*.

Create tables and copy data to *rawDb*. Based on the metadata in *rawDb*, i.e., the table name, column names and data types in each table, we first build the corresponding `CREATE TABLE` statements, and execute these `CREATE TABLE` statements on *rawDb*. Then we copy the data in *db*'s tables into the corresponding tables in *rawDb*. Different DBMSs support different ways to copy table data from *db* to *rawDb*. We encounter two situations for our target DBMSs.

- Some DBMSs, e.g., MySQL, MariaDB, CockroachDB, and TiDB, support cross-database references. For these DBMSs, we can directly utilize the `INSERT INTO SELECT` statement to copy data across databases. For example, we can execute the *INSERT INTO rawDb.t1 SELECT * FROM db.t1* statement to copy the data of table *t1* in *db* into table *t1* in *rawDb*.
- Some DBMSs, e.g., SQLite, do not support cross-database references. For these DBMSs, we first clone *db* as *rawDb* (e.g., copying *db*'s database file in SQLite) and rename all the tables in *rawDb* with different table names. For example, we rename table *t1* as *t1new* in *rawDb*. We then create tables based on the extracted metadata in *rawDb*. These newly created tables do not contain any optional metadata and have the same table names, column names and data types as those in *db*. Then, we can utilize the `INSERT INTO SELECT` statement to copy data from *t1new* to *t1* in *rawDb*. Finally, we delete all the renamed tables (e.g., *t1new*) in *rawDb*.

Figure 4 shows an example about how to construct a raw database *rawDb* for a given database *db*. The database *db* contains two tables *t1* and *t2*. Table *t1* contains a column *c1* with the `INT` data type, and a column *c2* with the `INT` data type and the $GENERATED(c1{+}1)$ constraint. Table *t1* also contains a $UNIQUE(c1)$ constraint on column *c1*. Table *t2* contains a column *c3* with the `INT` data type and the `NOT NULL` constraint. Table *t2* also contains a $FOREIGN\ KEY(c3, t1.c1)$ constraint, indicating that column *c3* refers to column *c1* in table *t1*. We first obtain the metadata in *db* (shown in the second part in Figure 4), and extract the compulsory metadata in *db* to construct the metadata in *rawDb* (shown in third part in Figure 4). We then build the `CREATE TABLE` statements based on *rawDb*'s metadata, i.e., table names *t1* and *t2*, column names *c1*, *c2*, and *c3*, and data type `INT`. We build two `CREATE TABLE` statements,

Algorithm 3: *metaBug* identification

Input: Q (The generated query), db (The generated database), $rawDb$ (The corresponding raw database)

1 **Function** executeAndValidate(Q, db, $rawDb$) **do**
2 $\quad R_{db}, E_{db} \leftarrow executeQuery(db, Q)$
3 $\quad R_{rawDb}, E_{rawDb} \leftarrow executeQuery(rawDb, Q)$
4 $\quad$ **if** $E_{db} \neq E_{rawDb} \mid\mid R_{db} \neq R_{rawDb}$ **then**
5 $\quad\quad reportMetaBug(Q, db, rawDb)$

i.e., $CREATE\ TABLE\ t1\ (c1\ INT,\ c2\ INT)$ and $CREATE\ TABLE\ t2\ (c3\ INT)$ (shown in the fourth part in Figure 4), and apply these two statements on $rawDb$. Finally, we copy data in db's tables into the corresponding tables in $rawDb$, and obtain $rawDb$ shown in the right part in Figure 4. Note that $rawDb$ does not contain the optional database metadata in db, i.e., $UNIQUE(c1)$, $GENERATED(c1 + 1)$ and $FOREIGN\ KEY(c3, t1.c1)$.

3.4 *metaBug* Identification

For a query Q generated in Section 3.2.2, evaluating Q on a database db and its corresponding raw database $rawDb$ should return the same query result.

We apply Algorithm 3 to identify *metaBug*s. We first execute Q on database db and $rawDb$ to obtain their returned results, including the query results R_{db} and R_{rawDb}, and error messages E_{db} and E_{rawDb} (Line 2-3). We then compare the query results and error messages separately to identify *metaBug*s. If db and $rawDb$ return different query results or different error messages, a potential *metaBug* is revealed in the target DBMS (Line 4-5). Note that we compare the query results by ignoring the order of the items in their result sets.

3.5 Metadata-Oriented Testing Optimization

Random database generation (Section 3.2.1) can generate many databases with the same or similar database metadata. Testing on these similar databases usually triggers duplicate *metaBug*s, and can hardly reveal new *metaBug*s. Therefore, we identify the databases with similar database metadata, and avoid testing these databases to improve testing efficiency and find more unique *metaBug*s quickly.

For a database db generated in Section 3.2.1, table names (i.e., $tName$) and column names (i.e., $cName$) cannot reflect its structure. That said, we can only change db's table names and column names to create a different database that share the same structure as db. Therefore, we remove db's table names and column names, and replace a reference to column *col* with *col*'s metadata. In this way, we construct an abstract database metadata without table names and column names. If db's abstract database metadata has not been previously recorded (Section 3.5.2), we continue our testing on db. Otherwise, we will discard db and regenerate a new database.

3.5.1 Extracting Abstract Database Metadata. For a given database db, we remove the table names and column names from db's metadata. For column references in the GENERATED constraint and table constraints, we replace them by their data types (i.e., *type*) and column constraints (i.e., *cConstraint*). We further ignore the constants (i.e., *const*), functions (i.e., *FUN*) and operators (e.g., + and −) in the

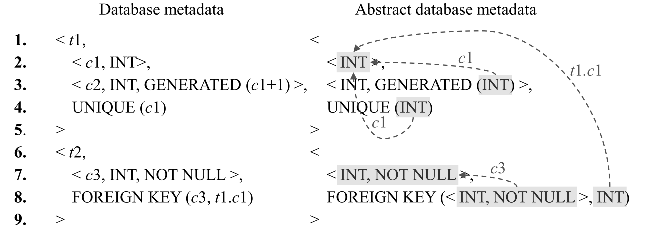

Figure 6: An example for extracting abstract metadata. The dashed lines denote how to replace column names with their corresponding metadata.

$expr$ of the $GENERATED(expr)$ and $CHECK(expr)$ constraints to simplify the abstract database metadata.

Figure 6 shows an example for extracting the abstract database metadata of db. In db's database metadata, for table $t1$, we remove its table name $t1$ (Line 1). For column $c1$ in table $t1$, we remove its column name $c1$, and remain its data type INT (Line 2). For column $c2$, we first remove its column name $c2$, and then replace the expression $c1+1$ in the $GENERATED(c1+1)$ constraint by $c1$'s corresponding data type INT, which is shown in the gray dashed line labelled by $c1$, and finally obtain $< INT, GENERATED(INT) >$ (Line 3). Similarly, for the $UNIQUE(c1)$ constraint, we replace $c1$ by its corresponding data type INT, and obtain $UNIQUE(INT)$ (Line 4). For table $t2$, we remove its table name $t2$ (Line 6). For column $c3$, we remove its column name $c3$, and obtain $< INT, NOT\ NULL >$ (Line 7). For the $FOREIGN\ KEY(c3, t1.c1)$ constraint, we replace the source column reference $c3$ and the target column reference $c1$ in table $t1$ by their corresponding metadata (i.e., $< INT, NOT\ NULL >$ and INT), respectively, which is shown in the gray dashed line labelled by $c3$ and $t1.c1$, and obtain $FOREIGN\ KEY(< INT, NOT\ NULL >, INT)$ (Line 8). Note that the abstract metadata of db does not contain the table names $t1$, $t2$, and column names $c1$, $c2$, and $c3$.

Note that when a DBMS (e.g., MySQL and TiDB) consists of compulsory table configurations, we analyze them as follows. If a table configuration (i.e., $tConfig$) is applied on columns, we replace the column names (i.e., $cName$) by their corresponding data types (i.e., *type*) and column constraints (i.e., *cConstraint*). Otherwise, we do not handle them and store their textual values. For example, if table $t2$ has the $PARTITIONING\ BY\ \ HASH(c3)$ configuration, we replace column $c3$ by $< INT, NOT\ NULL >$, and get $PARTITIONING\ BY\ HASH(< INT, NOT\ NULL >)$. If a table has the $Engine = InnoDB$ configuration, we store its textual value.

3.5.2 Choosing Interesting Databases. We maintain a set *unique Metadata* to record all tested abstract database metadata. After extracting the abstract database metadata in db, we compare db's abstract database metadata with the recorded abstract database metadata in *uniqueMetadata* to check whether db's abstract database metadata has been previously tested. If the abstract metadata has not been previously tested, we choose to test db and store db's abstract metadata into *uniqueMetadata*.

When comparing two abstract database metadata, we only validate whether two abstract database metadata contain the same elements regardless of the order of their elements, since the order is generally irrelevant to metadata-related query optimizations.

Table 2: Unique bugs detected by Radar

DBMS	Total	Bug Status				Bug Categories	
		Confirmed	Fixed	Duplicate	False Positive	*metaBug*	Non-*metaBug*
MySQL	4	3	0	1	0	3	0
SQLite	4	2	2	0	2	2	0
MariaDB	1	1	0	0	0	1	0
CockroachDB	2	1	1	0	1	0	1
TiDB	31	31	13	0	0	24	7
Total	**42**	**38**	**16**	**1**	**3**	**30**	**8**

4 EVALUATION

We implement Radar on five DBMSs, i.e., MySQL, SQLite, MariaDB, CockroachDB, and TiDB. Radar mainly consists of three parts, i.e., database and query generation, raw database construction and metadata-oriented testing optimization. Database and query generation are implemented based on SQLancer [21] in Java. We implement the general logic of our raw database construction and metadata-oriented testing optimization with 179 LOC. To test our target DBMSs, we implement the specific testing logic with 261, 192, 161, 243, 431 LOC for MySQL, SQLite, MariaDB, CockroachDB, and TiDB, respectively.

We evaluate the effectiveness of Radar by answering the following three research questions:

- **RQ1**. What *metaBug*s can Radar detect in real-world DBMSs?
- **RQ2**. How effective is the metadata-oriented testing optimization in Radar?
- **RQ3**. How many bugs detected by Radar can be found by existing approaches?

4.1 Experimental Setup

Target DBMSs. We evaluate Radar on five widely-used DBMSs. We discuss their detailed information in Section 2.1. We test these DBMSs with their latest release versions when we started this work, i.e., MySQL 8.0.32, SQLite 3.41.0, MariaDB 11.0.3, CockroachDB 22.2.5, and TiDB 6.6.0. When a target DBMS releases a new version, we update it to the latest version and test it.

Experimental infrastructure. We perform our experiments on a machine with 8 CPU cores and 32GB RAM. We deploy MySQL and MariaDB using Docker containers. We deploy CockroachDB in a three-node cluster. We deploy TiDB in a distributed manner using the official command *tiup playground*. SQLite does not require additional deployments, and we embed it in Radar.

Experimental process. We run Radar on each target DBMS for 24 hours, and then stop Radar to analyze the generated bug reports and submit the detected unique bug reports to DBMS developers. After processing all the generated bug reports, we start a new testing round. We present our bug report analysis process as follows.

For a generated bug report, we first leverage SQLancer [21] to automatically simplify it. SQLancer supports two test case reduction techniques, i.e., statement-level reduction and syntax-based reduction. Specifically, given a bug report with the query Q and two databases db and $rawDb$, we first simplify db and $rawDb$, and then simplify Q. For the database db, we first adopt the statement-level reduction and attempt to remove some statements from the database initialization statements randomly, e.g., removing some data INSERT statements. Then we adopt the syntax-based method to further simplify the database initialization statements. For the database $rawDb$, we adopt the approach explained in Section 3.3 to regenerate a raw database based on the simplified database db. For the query Q, we apply syntax-based reduction on Q, e.g., reducing the number of operators or replacing constant expressions with simple constants. Note that after each step of reduction, we check if the bug can still be triggered. If not, we revert the changes and try another round of reduction. We repeat the above steps until we cannot remove any statements and operators in db and Q.

After simplifying the generated bug reports, we automatically cluster them based on the abstract database metadata of database db and SQL features in query Q. If two simplified bug reports contain the same abstract database metadata of database db and the same SQL features in query Q, we assume that they are duplicates. We further manually analyze the clustered bug reports and identify unique bug reports.

4.2 Bug Detection Capability of Radar

To evaluate the effectiveness of Radar and answer RQ1, we apply Radar on our target DBMSs and investigate whether Radar can detect real-world *metaBug*s in these DBMSs. We test each target DBMS separately for a total of 10 rounds, running Radar for 24 hours in each round.

In total, Radar generates $1,663$ bug reports. We follow the process mentioned in Section 4.1 to simplify and remove duplicate bug reports. It took us about three weeks to filter out 42 unique bugs from the generated bug reports. We consider the remaining $1,621$ bug reports as duplicate to these 42 unique bugs. We then submit these 42 unique bugs to the corresponding DBMS community. Table 2 shows the details about these submitted bugs, including 4 bugs in MySQL, 4 bugs in SQLite, 1 bug in MariaDB, 2 bugs in CockroachDB and 31 bugs in TiDB.

Bug status. 38 out of the 42 submitted bugs have been confirmed as new bugs, and 16 bugs have been fixed by DBMS developers at the timing of writing this paper. For the remaining 4 *metaBug*s, one bug in MySQL is considered as duplicate to an existing one by developers, and the remaining 3 bugs are considered as false positives (2 bugs in SQLite and 1 bug in CockroachDB). We will further discuss the details about these three false positives in Section 4.6.2. The above result shows that Radar is effective in detecting *metaBug*s.

Bug severity. 34 out of the 38 confirmed bugs are classified as critical, e.g., *Critical*, *Major*, *Serious* or *Moderate*. Specifically, MySQL developers classified 1 bug as *Critical*, and 1 bug as *Serious*. MariaDB developer classified 1 bug as *Major*. TiDB developers classified 4

Table 3: Comparison of bug detection between Radar and Radar_{rand}

DBMS	Generated Databases		Unique Databases		Total Bugs		Unique Bugs	
	Radar	Radar_{rand}	Radar	Radar_{rand}	Radar	Radar_{rand}	Radar	Radar_{rand}
MySQL	42,997	38,508	23,533	21,151	3	0	1	0
SQLite	99,872	62,643	31,102	21,495	1	1	1	1
MariaDB	20,850	11,668	10,498	6,223	1	1	1	1
CockroachDB	2,540	2,472	1,596	1,504	0	0	0	0
TiDB	2,503	2,318	1,523	1,339	120	116	19	12
Total	**168,762**	**117,609**	**68,252**	**51,712**	**125**	**118**	**22**	**14**

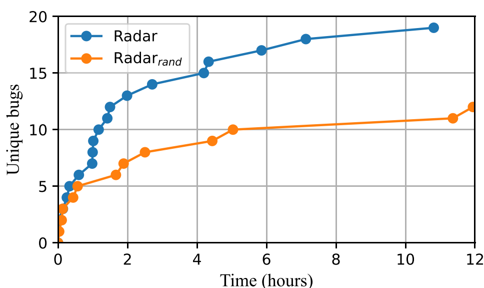

Figure 7: Unique bugs in TiDB detected by Radar and Radar_{rand}.

bugs as *Critical*, 12 bugs as *Major*, 14 bugs as *Moderate*, and 1 bug as *Minor*. For the remaining 4 bugs, 2 bugs are not classified by SQLite developers, 1 bug is not classified by CockroachDB developers, and 1 bug is classified as *Non-critical* in MySQL. The above analysis result shows that DBMS developers consider most of our submitted bugs to be important.

Bug categories. 30 out of the 38 confirmed bugs are *metaBugs*, including 3 *metaBugs* in MySQL, 2 *metaBugs* in SQLite, 1 *metaBug* in MariaDB, and 24 *metaBugs* in TiDB. The remaining 8 bugs are not *metaBugs*, since they either return the same unexpected errors or cause crashes for a database *db* and its corresponding raw database *rawDb*. Note that Radar can also detect these 8 bugs, since Radar also checks whether a query throws unexpected errors or crashes.

***metaBug* analysis**. 23 out of the confirmed 30 *metaBugs* bugs occur on the generated databases with optional database metadata (e.g, integrity constraints, and indexes), 7 bugs occur on the raw databases. Note that for these 7 bugs, the queries on their corresponding databases with optional database metadata return correct results instead. Thus, we can detect these bugs.

4.3 Effectiveness of Metadata-Oriented Testing Optimization

To answer RQ2, we implement Radar_{rand} that shares the same bug detection process as Radar except for the metadata-oriented testing optimization. That said, we disable the metadata-oriented testing optimization in Radar_{rand}, and test all the generated databases. We run Radar and Radar_{rand} on our target DBMSs for 12 hours, respectively, and compare Radar and Radar_{rand} from two aspects, i.e., bug detection (Section 4.3.1) and code coverage (Section 4.3.2).

Table 4: Code coverage comparison between Radar and Radar_{rand}

DBMS	Function Coverage		Line Coverage	
	Radar	Radar_{rand}	Radar	Radar_{rand}
MySQL	20.3%	19.7%	17.6%	17.4%
MariaDB	19.7%	19.3%	17.0%	16.9%

4.3.1 Bug Detection. We run Radar and Radar_{rand} on MySQL, SQLite, MariaDB, CockroachDB, and TiDB for 12 hours. During this experiment, we count the number of generated databases, unique databases, total bugs, and unique bugs. If a database contains unique abstract database metadata (Section 3.5.1), we assume that the generated database is unique. We follow the same method discussed in Section 4.1 to filter out unique bugs.

Table 3 shows our experimental results. Radar can effectively eliminate 45.2%, 68.8%, 49.6%, 37.1%, and 39.1% duplicate databases, in MySQL, SQLite, MariaDB, CockroachDB, and TiDB, respectively. Radar tests more unique databases than Radar_{rand} by 11.2%, 44.6%, 40.7%, 6.1%, and 13.7% in MySQL, SQLite, MariaDB, CockroachDB, and TiDB, respectively. Moreover, Radar has detected 7 more bugs and 8 more unique bugs than Radar_{rand}. Our experimental results show that our metadata-oriented testing optimization strategy can help Radar to focus on testing more databases with diverse metadata than Radar_{rand} and detecting more *metaBugs*.

We further analyze the time used to detect unique bugs in TiDB by Radar and Radar_{rand}. As shown in Figure 7, Radar takes less time to detect the same number of unique bugs as Radar_{rand}. Until exhausting the predefined experimental time (i.e., 12 hours), Radar detects 7 more unique bugs than Radar_{rand}. Note that we do not provide the details about the time used to detect unique bugs in other DBMSs, since Radar and Radar_{rand} detect few bugs on them.

4.3.2 Code Coverage. We analyze the code coverage in our target DBMSs by running Radar and Radar_{rand} in MySQL and MariaDB for 12 hours. During this experiment, we measure the function coverage and line coverage [1] in testing MySQL and MariaDB. Table 4 shows our experimental result, which indicates that Radar can explore more functions and lines in our target DBMSs than Radar_{rand}. Note that we do not find a suitable way to measure the code coverage in testing SQLite, CockroachDB, and TiDB.

In the experiment, the code coverage appears to be low but reasonable, since Radar focuses on validating the correctness of query engines, especially for metadata-related query optimizations. Our target DBMSs encompass a broad range of functionalities not

Table 5: Conceptual comparison with existing approaches

DBMS	Radar	DQE	NoREC	TLP	MutaSQL
MySQL	3	1	2	1	2
SQLite	2	0	2	2	2
MariaDB	1	0	0	0	0
CockroachDB	0	0	0	0	0
TiDB	24	7	9	0	1
Total	**30**	**8**	**13**	**3**	**5**

limited to query processing, e.g., concurrency control, backup and recovery, and user management, which we do not test in Radar.

The difference of code coverage between Radar and Radar_{rand} is small in this experiment, since Radar and Radar_{rand} share the same database and query generation approach. Even if the difference of code coverage between Radar and Radar_{rand} is small, Radar can test more databases with unique metadata and find more bugs than Radar_{rand}.

4.4 Comparing with Existing Approaches

To answer RQ3, we compare Radar with four state-of-the-art approaches (i.e., DQE [66], NoREC [61], TLP [62], and MutaSQL [34]) that aim to detect logic bugs in DBMSs. All these approaches adopt some random strategies to test DBMSs, e.g., random database generation. To avoid randomness in comparison, we first investigate whether the 30 confirmed *metaBug*s detected by Radar can also be conceptually detected by these approaches (Section 4.4.1). We then apply Radar and existing approaches on the target DBMSs for the same amount of time, and compare their actual bug detection results (Section 4.4.2).

4.4.1 Conceptual Comparison. Existing approaches can only detect at most 13 out of our confirmed 30 *metaBug*s in our conceptual comparison (Table 5), which indicates the effectiveness of Radar in detecting *metaBug*s. We show the detailed analysis as follows.

DQE evaluates a predicate on SELECT, UPDATE, and DELETE statements and detects bugs by identifying inconsistencies in their execution results. For a *metaBug*, given the database *db* and the query Q, we leverage the predicate of Q to construct UPDATE, and DELETE statements. If these statements return consistent execution results on *db*, DQE cannot detect it. Moreover, when Q queries on views or multiple tables in SQLite, DQE cannot construct corresponding UPDATE and DELETE statements. As a result, DQE can only detect 8 out of the 30 confirmed *metaBug*s.

NoREC transforms a given optimized query into another non-optimized one, and detects bugs by comparing their execution results. For a *metaBug*, given the database *db* and the query Q, we apply the same transformation on Q, and compare these two query results. NoREC cannot detect the following *metaBug*s. First, these two queries return consistent query results on *db*. Second, Q does not contain predicates, thus we cannot construct a non-optimized query for NoREC. Third, Q returns errors or causes crashes. As a result, NoREC can only detect 13 out of the 30 confirmed *metaBug*s.

TLP transforms a given query into three sub-queries, and detects bugs by comparing the original query result with the combined result of the three sub-queries. For a *metaBug*, given the database *db* and the query Q, we extract the predicate of Q to construct three sub-queries, and compare the original query result with the combined result of the three sub-queries. TLP cannot detect the following *metaBug*s. First, the combined query result of the three sub-queries is the same with the original query result. Second, Q does not contain predicates, thus we cannot construct sub-queries for TLP. Third, Q returns errors or causes crashes. As a result, TLP can only detect 3 out of the 30 confirmed *metaBug*s.

MutaSQL reports logic bugs if adding indexes changes the query result of a given query. For a *metaBug*, given the database *db* and the query Q, we first check whether *db* contains indexes. If *db* contains indexes, we remove the indexes. Otherwise, we try to add some indexes. Then we check if the result of Q is changed. As a result, MutaSQL can only detect 5 out of the 30 confirmed *metaBug*s.

Table 6: Experimental comparison with existing approaches. ★ denotes *metaBug*, and ☆ denotes other kinds of bugs. If an approach does not support testing the target DBMS, we mark the corresponding bug using -.

Bug Id	Radar	DQE	NoREC	TLP
TiDB-1	★		-	
TiDB-2	★		-	
TiDB-3	★		-	
TiDB-4	★		-	
TiDB-5	★		-	
TiDB-6	★		-	
TiDB-7	★		-	
TiDB-8	★		-	
TiDB-9	★		-	
TiDB-10	★		-	
TiDB-11	★		-	
TiDB-12	★		-	
TiDB-13	★		-	
TiDB-14	★		-	★
TiDB-15	★		-	★
TiDB-16	★	★	-	★
TiDB-17			-	☆
TiDB-18			-	☆
TiDB-19		☆	-	
TiDB-20		☆	-	
TiDB-21		☆	-	
TiDB-22		☆	-	
MySQL-1	★		-	
MySQL-2	★		-	★
MySQL-3			-	★
MySQL-4			-	☆
MariaDB-1	★		★	-
MariaDB-2			☆	-
MariaDB-3		☆		-
MariaDB-4		☆		-
MariaDB-5		☆		-
MariaDB-6		☆		-
SQLite-1	★			
Total	**20**	**9**	**2**	**8**

4.4.2 Experimental Comparison. We apply DQE, NoREC, TLP and Radar on our target DBMSs for 12 hours, respectively[1]. We utilize the method mentioned in Section 4.1 to de-duplicate bug reports

[1]We do not compare Radar with MutaSQL, since MutaSQL [34] is not publicly available.

and identify unique bugs. For a bug that was not detected by Radar, given the database *db* and query *Q* in the bug report, we attempt to construct a corresponding raw database or a database containing some metadata so that query *Q* would return different results on the initial database *db* and the database we constructed. If we can construct such a database, we consider this bug as a *metaBug*, too. Note that Radar, DQE, NoREC and TLP share the same database and query generation approach in this experiment.

Table 6 shows the results, in which all these approaches have detected 33 bugs in total, including 21 *metaBugs* that have been detected by Radar in Section 4.2. The remaining 12 bugs are other kinds of bugs, e.g., logic bugs in UPDATE or DELETE statements and error bugs. Radar detected 20 out of the 21 *metaBugs*, and 15 *metaBugs* were only detected by Radar. While the alternative approaches detected 6 out of the 21 *metaBugs*. For the only one *metaBug* that Radar did not reveal, Radar did not generate the corresponding test case due to random database and query generation.

metaBugs are specifically introduced by the incorrect optimizations related to database metadata. Although existing approaches [61, 62, 66] can effectively detect other kinds of logic bugs, they cannot effectively detect *metaBugs*.

4.5 Other Experimental Statistics

Query testing efficiency. We measure the query testing efficiency in Radar by running Radar on our target DBMSs separately for 12 hours, and counting the number of tested queries. In this experiment, Radar tested 118, 4,795, 948, 132, and 280 queries per second in MySQL, SQLite, MariaDB, CockroachDB, and TiDB, respectively.

Overhead of raw database construction. We measure the overhead of raw database construction by running Radar on our target DBMSs separately for 12 hours. In this experiment, Radar spent an average of 95ms, 33ms, 34ms, 2,645ms, and 877ms to construct a raw database in MySQL, SQLite, MariaDB, CockroachDB, and TiDB, respectively.

4.6 Representative Bugs

In this section, we discuss 6 bugs detected by Radar, of which 3 are the confirmed *metaBugs* (Section 4.6.1), and 3 bugs are false positives (Section 4.6.2).

```
1. -- Database db
2. CREATE TABLE t1(c1 DOUBLE ZEROFILL);
3. CREATE INDEX i0 ON t1(c1);
4. INSERT INTO t1 VALUES(0);

5. -- Query Q
6. SELECT c1 FROM t1 WHERE (IFNULL(-1, '')) IN (c1);
7. -- {0} in db ✘
8. -- {} in rawDb ✔
```

Listing 2: MySQL#110125. This bug relates to the INDEX on the DOUBLE ZEROFILL data type.

4.6.1 *Selected Bugs.*

Listing 2 shows a bug MySQL#110125, in which table $t1$ in database *db* contains a column $c1$ with the DOUBLE data type and an optional metadata, i.e., index $i0$ on column $c1$ (Line 3). For query *Q* (Line 6), the $IFNULL(-1, '')$ expression is expected to return the value '-1', and the predicate should be evaluated to FALSE, because column $c1$ does not contain a value equal to '-1'. However, MySQL incorrectly returns the value of column $c1$ when executing *Q* on *db* (Line 7), while returns an empty set when executing *Q* on the raw database *rawDb* (Line 8).

```
1. -- Database db
2. CREATE TABLE t1 (c1 INT UNSIGNED NOT NULL);
3. INSERT INTO t1 VALUES (0);

4. -- Query Q
5. SELECT c1 > - '7' FROM t1;
6. -- {0} in db ✘
7. -- {1} in rawDb ✔
```

Listing 3: TiDB#44219. This bug relates to the INT UNSIGNED data type.

Listing 3 shows a bug TiDB#44219, in which table $t1$ in database *db* contains a column $c1$ with the INT UNSIGNED data type and an optional metadata, i.e., the NOT NULL constraint on column $c1$ (Line 1). However, for query Q (Line 5), *db* and *rawDb* return different query results due to the incorrect data conversion in *db*.

```
1. -- Database db
2. CREATE TABLE t1(c1 FLOAT GENERATED ALWAYS AS (c2), c2 FLOAT);
3. INSERT INTO t1(c2) VALUES (0.5822439);

4. -- Query Q
5. SELECT * FROM t1 WHERE (~ (CAST(c1 AS DATETIME)));
6. -- {} in db ✘
7. -- {0.5822439} in rawDb ✔
```

Listing 4: TiDB#44135. This bug relates to the GENERATED constraint on the FLOAT data type.

Listing 4 shows a bug TiDB#44135, in which table $t1$ in database *db* contains a column $c1$ with the INT data type and an optional metadata, i.e., the GENERATED constraint on column $c1$ (Line 2). The $GENERATED(c2)$ constraint specifies the value of $c1$ should be equal to $c2$. However, for query *Q* (Line 5), the database *db* and *rawDb* return different query results.

```
1. -- Database db
2. CREATE TABLE t1 (c1 DECIMAL);
3. INSERT INTO t1 VALUES (1);
4. CREATE TABLE t2 (c2 DECIMAL PRIMARY KEY);
5. INSERT INTO t2 VALUES (1);

6. -- Query Q
7. SELECT c1 FROM t1 LEFT OUTER JOIN t2 ON (c2) IN (('a') ::
   DECIMAL);
8. -- {1} in db
9. -- error in rawDb
```

Listing 5: CockroachDB#97672. This bug relates to the PRIMARY KEY on the DECIMAL data type.

4.6.2 *False Positives.*

Listing 5 shows a bug CockroachDB#97672, in which the database *db* returns a value 1, while *rawDb* returns an error. CockroachDB developers argue that such behavior is intended, because the optimizer eliminates the join clause with its predicate in *db*. However, such optimization is disabled in *rawDb*.

```
1. -- Database db
2. CREATE TABLE t1 (c1 INTEGER PRIMARY KEY);
3. INSERT INTO t1 VALUES (0);

4. -- Query Q
5. SELECT c1 FROM t1 ORDER BY c1, json_array_length(0, 0);
6. -- {0} in db
7. -- error in rawDb
```

Listing 6: SQLite#a2bde2b8f9. This bug relates to the PRIMARY KEY on the INTEGER data type.

Listing 6 shows a bug SQLite#a2bde2b8f9, in which the database *db* returns a value 0, while *rawDb* returns an error because of the $json_array_length(0, 0)$ function. SQLite developers explain that when the ORDER BY clause can determine the data order in *db*, it does not need to invoke the JSON function.

```
1. -- Database db
2. CREATE TABLE t1 (c1 INTEGER);
3. CREATE TABLE t2 (c1 INTEGER, UNIQUE (c1));
4. INSERT INTO t1 VALUES (x''), (0.8874540680509563), (NULL), (
     '-2017888786');
5. INSERT INTO t2 VALUES (0x47d9a1ab);

6. -- Query Q
7. SELECT ALL t1.c1 FROM t1, t2 WHERE (0 OR json_patch(t1.c1,
     t2.c1)) AND (((t2.c1) BETWEEN (t1.c1) AND (t2.c1)));
8. -- {0.887454068050956, -2017888786} in db
9. -- error in rawDb
```

Listing 7: SQLite#60f85edfaf. This bug relates to the UNIQUE constraint on the INTEGER data type.

Listing 7 shows a bug SQLite#60f85edfaf, in which the database *db* returns a result, while *rawDb* returns an error due to the $json_patch(t1.c1, t2.c1)$ function. SQLite developers explain that the predicate is a short-circuit expression evaluation, and SQLite does not guarantee the evaluation order.

5 DISCUSSION

Undefined behaviors in query evaluation. The three false positives detected by Radar trigger undefined behaviors in query evaluation, since the SQL standard does not define the exact implementations of query evaluation [20]. Even though those bugs are considered as intended behaviors by DBMS developers, such undefined behaviors should be noticed, because they can lead to unpredictable execution results, and some may cause disastrous impacts.

Extend to other DBMSs. Similar to our target DBMSs, many other DBMSs also utilize database metadata to optimize query evaluation. Thus, the idea of Radar can be extended to other DBMSs that support optional database metadata to accelerate query evaluation, e.g., graph-oriented DBMSs Neo4j [18] and TigerGraph [25], document-oriented DBMSs MongoDB [15] and Google Cloud Datastore [7], and key-value stores Amazon DynamoDB [2] and Microsoft Azure Cosmos DB [4]. However, we need to modify the implementation of Radar to adapt it to the new DBMS.

Limitation. Radar may miss some bugs due to the following reasons. First, random database and query generation may cause Radar to miss some bugs. Second, Radar cannot detect bugs that occur in both the generated database and its corresponding raw database. Third, Radar cannot detect bugs that are caused by non-deterministic functions, database-related functions and index hints, since Radar does not support them during query generation.

6 RELATED WORK

Database and query generation. Database and query generation have been widely explored by existing works [10, 23, 26, 30, 31, 42, 44, 45, 50, 53, 58, 60, 69, 73]. For example, SQLsmith [23] generates queries based on its built-in abstract syntax trees. Squirrel [73] takes an input query, mutates it based on the designed intermediate representation, and utilizes a coverage-based guidance to perform their mutations. QPG [27] mutates database states with query plan guidance. An improved database and query generation approach could help our work to detect more bugs.

Differential testing on DBMSs. Some existing works apply differential testing on DBMSs [36, 52, 59, 65, 70, 72]. Differential testing on DBMSs is usually conducted by feeding the same test cases into multiple DBMSs with the same type. For example, RAGS [65], APOLLO [52] and AMOEBA [59] generate queries and execute them on multiple relational DBMSs. Our approach adopts the idea of differential testing by generating queries and evaluating them on a given database and its corresponding raw database. These existing approaches require multiple DBMSs as input to detect bugs, while Radar can detect logic bugs on a single DBMS. Thus, Radar can be applied to test DBMS-specific features.

Test oracles to detect bugs in DBMSs. Detecting bugs in DBMSs requires test oracles to validate the execution results of statements. Error bugs are usually detected by whitelists, e.g., SQLancer [21] maintains a list of expected errors for different statements. Crash bugs are usually detected by DBMS fuzzers by examining the network status of DBMS servers, e.g., checking timeouts [23, 39, 40, 50, 56, 58, 69, 73]. Logic bugs are usually detected by constructing equivalent statements, and observing discrepancies among their execution results [28, 29, 34, 43, 49, 54, 57, 61, 62, 66, 68, 71]. Particularly, PQS [63] utilizes a containment oracle to detect logic bugs by synthesizing queries to fetch pivot rows and checking their existence. PQS requires much manual effort to build its reference model. PINOLO [43] synthesizes queries with different approximations and identifies logic bugs by comparing these query results. Recent works further analyze and build test oracles for concurrent transactions, and detect transaction bugs in DBMSs [37, 38, 51, 55, 67]. The above existing approaches cannot provide an effective test oracle to detect metadata-related logic bugs, since they do not have test oracles [23, 39, 40, 50, 56, 58, 69, 73], or they require a lot of manual labor to make their test oracles effective [43, 63, 68]. But, Radar provides a general and effective test oracle to detect metadata-related logic bugs.

7 CONCLUSION

Incorrect metadata-related implementations can introduce metadata-related logic bugs, which can cause a DBMS to silently return an incorrect query result for a given query. Existing approaches cannot efficiently detect metadata-related logic bugs. In this paper, we propose a general testing approach Radar to effectively detect metadata-related logic bugs in DBMSs. We implement and evaluate Radar on five widely-used DBMSs, i.e., MySQL, SQLite, MariaDB, CockroachDB, and TiDB. In total, we have detected 42 bugs, 38 of which have been confirmed as previously unknown bugs and 16 bugs have been fixed. We expect that Radar can be widely used to improve the reliability of DBMSs.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (62072444, 62302493), Major Project of ISCAS (ISCAS-ZD-202302), Major Program (JD) of Hubei Province (2023BAA018), and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044).

REFERENCES

[1] 2023. AFL: American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.
[2] 2023. Amazon DynamoDB. https://aws.amazon.com/cn/dynamodb/.
[3] 2023. Avoiding Full Table Scans. https://dev.mysql.com/doc/refman/8.0/en/table-scan-avoidance.html.
[4] 2023. Azure Cosmos DB. https://azure.microsoft.com/.
[5] 2023. CockroachDB Homepage. https://www.cockroachlabs.com/.
[6] 2023. Database Topic in GitHub. https://github.com/topics/database.
[7] 2023. Datastore. https://cloud.google.com/datastore.
[8] 2023. DB-Engines Ranking. https://db-engines.com/en/ranking.
[9] 2023. Function Cast in MySQL. https://dev.mysql.com/doc/refman/5.7/en/cast-functions.html.
[10] 2023. go-randgen. https://github.com/pingcap/go-randgen.
[11] 2023. Incompatible query results with UNHEX function. https://github.com/pingcap/tidb/issues/45378.
[12] 2023. IS NULL Optimization. https://dev.mysql.com/doc/refman/8.0/en/is-null-optimization.html.
[13] 2023. Manifest log in RocksDB. https://github.com/facebook/rocksdb/wiki/MANIFEST.
[14] 2023. MariaDB Homepage. https://mariadb.org/.
[15] 2023. MongoDB. https://www.mongodb.com/.
[16] 2023. MySQL CREATE INDEX Statement. https://dev.mysql.com/doc/refman/8.0/en/create-index.html.
[17] 2023. MySQL Homepage. https://www.mysql.com.
[18] 2023. Neo4j Homepage. https://neo4j.com/.
[19] 2023. PostgreSQL Homepage. https://www.postgresql.org/.
[20] 2023. Quirks, Caveats, and Gotchas In SQLite. https://www.sqlite.org/quirks.html.
[21] 2023. SQLancer Homepage. https://github.com/sqlancer/sqlancer.
[22] 2023. SQLite Homepage. https://www.sqlite.org/index.html.
[23] 2023. SQLsmith. https://github.com/anse1/sqlsmith.
[24] 2023. TiDB Homepage. https://www.pingcap.com/?from=en.
[25] 2023. TigerGraph. https://www.tigergraph.com/.
[26] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–247.
[27] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of International Conference on Software Engineering (ICSE)*. 2060–2071.
[28] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of International Conference on Software Engineering (ICSE)*. Article 133, 13 pages.
[29] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proceedings of International Conference on Management of Data (SIGMOD)* (jun 2024).
[30] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 341–352.
[31] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1097–1107.
[32] Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-Based Approach to Semantic Query Optimization. *ACM Transactions on Database Systems (TODS)* 15, 2 (jun 1990), 162–207.
[33] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control (SIGFIDET)*. 249–264.
[34] Xinyue Chen, Chenglong Wang, and Alvin Cheung. 2020. Testing Query Execution Engines with Mutations. In *Proceedings of the Workshop on Testing Database Systems*. Article 6, 5 pages.
[35] Edgar F Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (jun 1970), 377–387.
[36] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2023. Differentially Testing Database Transactions for Fun and Profit. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Article 35, 12 pages.
[37] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, Jun Wei, and Tao Huang. 2024. Understanding Transaction Bugs in Database Systems. In *Proceedings of International Conference on Software Engineering (ICSE)*. Article 163, 13 pages.
[38] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *Proceedings of International Conference on Software Engineering (ICSE)*. 1123–1135.
[39] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of International Conference on Software Engineering (ICSE)*. Article 146, 12 pages.
[40] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Article 49, 12 pages.
[41] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2001. Exploiting Constraint-like Data Characterizations in Query Optimization. In *Proceedings of International Conference on Management of Data (SIGMOD)*, Vol. 30. 582–592.
[42] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. 23, 2 (may 1994), 243–252.
[43] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*. 345–358.
[44] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and Realistic Data Generation. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1243–1246.
[45] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 163–174.
[46] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (aug 2020), 3072–3084.
[47] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. 2014. Complete yet Practical Search for Minimal Query Reformulations under Constraints. In *Proceedings of International Conference on Management of Data (SIGMOD)*. 1015–1026.
[48] Anant Jhingran and Pratap Khedkar. 1992. Analysis of recovery in a database system using a write-ahead log protocol. 21, 2 (1992), 175–184.
[49] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of International Conference on Software Engineering (ICSE)*. Article 46, 12 pages.
[50] Zuming Jiang, Jiaju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *Proceedings of USENIX Security Symposium (USENIX Security)*. Article 277, 17 pages.
[51] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 397–417.
[52] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proceedings of the VLDB Endowment (PVLDB)* 13, 1 (sep 2019), 57–70.
[53] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 140–149.
[54] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 140–149.
[55] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proceedings of the VLDB Endowment (PVLDB)* 14, 3 (nov 2020), 268–280.
[56] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 668–681.
[57] 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 International Conference on Software Engineering (ICSE)*. Article 135, 12 pages.
[58] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 4309–4326.
[59] Xinyu Liu, Qi Zhou, Joy Arulrai, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *Proceedings of International Conference on Software Engineering (ICSE)*. 225–236.
[60] Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. 1993. Generating Consistent Test Data: Restricting the Search Space by a Generator Formula. *The VLDB Journal (VLDBJ)* 2, 2 (apr 1993), 173–214.
[61] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1140–1152.

[62] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vol. 4. Article 211, 30 pages.

[63] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Article 38, 16 pages.

[64] Michael Siegel, Edward Sciore, and Sharon Salveter. 1992. A Method for Automatic Rule Derivation to Support Semantic Query Optimization. *ACM Transactions on Database Systems (TODS)* 17, 4 (dec 1992), 563–600.

[65] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 618–622.

[66] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 2072–2084.

[67] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: making transactional key-value stores verifiably serializable. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Article 4, 18 pages.

[68] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. 1, 1, Article 55 (2023), 26 pages.

[69] 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 *Proceedings of IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 328–337.

[70] Rui Yang, Yingying Zheng, Lei Tang, Wensheng Dou, Wei Wang, and Jun Wei. 2023. Randomized Differential Testing of RDF Stores. In *Proceedings of International Conference on Software Engineering (ICSE Demo)*. 136–140.

[71] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Jiansen Song, Ziyue Cheng, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Differential Optimization Testing of Gremlin-Based Graph Database Systems. *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2024).

[72] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 302–313.

[73] 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 *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 58–71.