
# Detecting Data-Type-Related Logic Bugs in Relational DBMSs via Compatible Database Construction

Jiansen Song[*†]
Wensheng Dou[*†‡§]
Institute of Software at CAS, China
{songjiansen20,wsdou}@otcaix.iscas.ac.cn

Yingying Zheng[*†§]
Institute of Software at CAS, China
zhengyingying14@otcaix.iscas.ac.cn

Yu Gao[*†‡]
Institute of Software at CAS, China
gaoyu15@otcaix.iscas.ac.cn

Quanqing Xu
OceanBase, Ant Group
xuquanqing.xqq@oceanbase.com

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

Xudong Xie[*†]
Institute of Software at CAS, China
xiexudong23@otcaix.iscas.ac.cn

Hongtao Zhou[*†]
Institute of Software at CAS, China
zhouhongtao24@otcaix.iscas.ac.cn

Jiaying Zhou[*†]
Institute of Software at CAS, China
zhoujiaying25@otcaix.iscas.ac.cn

Jun Wei[*†‡]
Wei Wang[*†]
Institute of Software at CAS, China
{wangwei,wj}@otcaix.iscas.ac.cn

## ABSTRACT

Relational Database Management Systems (DBMSs) serve as foundational systems for data storage and management, supporting a rich variety of data types to specify storage formats and value ranges. These data types play a critical role in both data storage and computation. However, complex data computation operations (e.g., explicit and implicit data type conversions) can introduce data-type-related logic bugs (TypeBugs for brevity). Specifically, TypeBugs can cause `SELECT` statements to return incorrect query results, which can easily be overlooked by DBMS developers. Unfortunately, existing DBMS testing approaches do not examine data type conversions, rendering them ineffective at detecting TypeBugs.

We observe that database columns with different data types (e.g., `INT` and `BIGINT`) can store identical data values within specific value ranges, and executing identical SQL operations on these database columns should produce consistent results. Inspired by this observation, we propose `TypeCheck`, a novel testing approach for effectively detecting TypeBugs in relational DBMSs. We first identify storage-compatible data types $\mathcal{T}$ that can preserve identical data values and operation-oriented type compatibility rules *opRules* that define SQL operations producing consistent results when applied to different data types. Using $\mathcal{T}$ and *opRules*, we construct type-compatible databases and execute identical type-compatible `SELECT` statements on them. When these `SELECT` statements yield divergent execution results, we detect a TypeBug. We evaluate `TypeCheck` on six widely used relational DBMSs, and have detected 37 TypeBugs, of which 34 have been confirmed as previously unknown bugs.

**PVLDB Reference Format:**
Jiansen Song, Wensheng Dou, Yingying Zheng, Yu Gao, Quanqing Xu, Ziyu Cui, Xudong Xie, Hongtao Zhou, Jiaying Zhou, Jun Wei, and Wei Wang.

[*]Affiliated with Beijing Key Laboratory of Intelligent Software Engineering.
[†]Affiliated with Key Lab of System 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 Yingying Zheng are the corresponding authors.

Detecting Data-Type-Related Logic Bugs in Relational DBMSs via Compatible Database Construction. PVLDB, 19(11): 3440 - 3453, 2026.
doi:10.14778/3836663.3836700

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://figshare.com/s/fcc65b82637c1d5da462.

## 1 INTRODUCTION

Relational Database Management Systems (DBMSs) stand as the dominant database systems [1], underpinning a wide range of data-intensive applications [3]. These data-intensive applications rely heavily on DBMSs, utilizing SQL statements to perform complex data operations, such as data storage, retrieval, and computation. This paper focuses on relational DBMSs, and unless otherwise specified, the term 'DBMS' in this paper refers to relational DBMSs.

DBMSs offer a rich variety of data types to accommodate diverse data storage and computation needs. For example, MySQL and PostgreSQL each support over 30 data types. These different data types specify distinct storage formats and value ranges, including numeric types (e.g., `INT`, `DECIMAL`, `FLOAT`), string types (e.g., `VARCHAR`, `CHAR`, `TEXT`), temporal types (e.g., `DATE`, `TIMESTAMP`), and other specialized types (e.g., `BLOB`, `JSON`).

DBMSs also support a rich set of SQL operations for performing complex data computations, such as arithmetic operations (e.g., $+, -, *, /$), logical operations (e.g., `EXISTS`, `NOT`), and built-in functions (e.g., `SUM`, `NOW`, `CAST`). The semantics, validity, and performance of these operations are governed by the type system, which defines valid operations, derives result types, handles type conversions,


Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836700

```
1. -- Database db1
2. CREATE TABLE table1 (c1 BIGINT);
3. INSERT INTO table1 (c1) VALUES (17044);
4. SELECT TIMESTAMP(c1) FROM table1; -- {null} ✘

5. -- Database db2
6. CREATE TABLE table1 (c1 INT);
7. INSERT INTO table1 (c1) VALUES (17044);
8. SELECT TIMESTAMP(c1) FROM table1; -- {2017-04-04 00:00:00} ✔
```

**Listing 1: TiDB#59832. Inconsistent datetime conversion between `INT` and `BIGINT` data types in TiDB.**

and shapes execution strategies. Thus, data types serve as both a correctness safeguard and a performance enabler.

Incorrect data-type-related implementation during storage and computation can cause data-type-related logic bugs, hereafter denoted as TypeBugs. When a `SELECT` statement uses complex computation operations, TypeBugs may cause undesired type conversions or erroneous execution semantics, ultimately producing incorrect query results. These bugs are particularly insidious because they typically produce incorrect query results without triggering explicit errors, evading detection during testing and potentially leading to erroneous decisions in data-driven applications.

Listing 1 illustrates a real-world TypeBug TiDB#59832 detected by our approach in TiDB. The `TIMESTAMP` function is designed to accept a date or datetime expression and return a result in datetime format. When applying the `TIMESTAMP` function to a `BIGINT` column with value 17044, it must implicitly convert the integer to a datetime. However, the incorrect implicit type conversion during the execution of the `TIMESTAMP` function causes TiDB to return `NULL` (Lines 2–4). In contrast, the expected result is `'2017-04-04 00:00:00'` (Lines 6–8).

Existing DBMS testing approaches primarily focus on detecting logic bugs in `SELECT` statements [10, 16, 17, 21, 32–34, 36, 37, 44, 50]. Differential testing [15, 35, 49] compares query results across multiple DBMSs, but cannot test DBMS-specific features or detect bugs that manifest uniformly. Metamorphic testing [12, 22, 30, 32, 34] detects logic bugs within a single DBMS by constructing equivalent `SELECT` statements. However, existing metamorphic relations target query optimization bugs and lack the specificity for TypeBugs, which involve subtle data type conversions.

In DBMSs, the same value can be stored in columns with different data types, e.g., any integer in $[-2^{31}, 2^{31}-1]$ fits both `INT` and `BIGINT`. SQL operations applied to the same value with different types should yield identical results; otherwise, a TypeBug is revealed. As demonstrated in Listing 1, applying the `TIMESTAMP` function to the same integer (e.g., 17044) stored as `INT` versus `BIGINT` produces inconsistent results in TiDB, revealing a TypeBug.

Building upon this key observation, we propose `TypeCheck`, the first testing approach to effectively detect TypeBugs in DBMSs. Specifically, we first identify storage-compatible data types $\mathcal{T}$ that can store identical data values without information loss, and operation-oriented type compatibility rules *opRules* that define SQL operations producing consistent results across different data types in $\mathcal{T}$. With $\mathcal{T}$ and *opRules*, `TypeCheck` performs testing in three steps: (1) constructing two type-compatible databases using $\mathcal{T}$ and populating them with identical data values; (2) generating type-compatible `SELECT` statements (*stmts*) by applying randomly selected rules in *opRules*; and (3) executing *stmts* on both databases and comparing the results for consistency. Any inconsistency indicates a TypeBug.

**Table 1: Target relational DBMSs**

| DBMS | DB-Engines Ranking | GitHub Stars | Type |
|---|---|---|---|
| MySQL | 2 | 10.8K | Traditional |
| PostgreSQL | 4 | 16.1K | Traditional |
| MariaDB | 15 | 5.6K | Traditional |
| CockroachDB | 64 | 30.1K | NewSQL |
| TiDB | 77 | 37.2K | NewSQL |
| OceanBase | 99 | 9.8K | NewSQL |

To evaluate the effectiveness of `TypeCheck`, we implement and evaluate `TypeCheck` on six widely used DBMSs, i.e., MySQL [4], PostgreSQL [5], MariaDB [2], CockroachDB [39], TiDB [9], and OceanBase [45, 46]. In total, we have detected 37 TypeBugs, of which 34 have been confirmed as previously unknown bugs, with 10 of these confirmed bugs already fixed. Through a comprehensive root cause analysis of these 34 confirmed TypeBugs, we find that 8 stem from explicit data type conversions, 24 from implicit data type conversions, and 2 from specific computation logic, thus illustrating that TypeBugs can emerge across diverse data type handling scenarios. We further compare `TypeCheck` against three state-of-the-art approaches (NoREC [32], Radar [37], and DQP [12]) using the 34 confirmed TypeBugs. Existing approaches detect at most 6 of these bugs, thereby demonstrating that `TypeCheck` can detect TypeBugs that are missed by existing approaches.

In summary, we make the following contributions.

- We propose `TypeCheck`, the first testing approach to effectively identify data-type-related logic bugs in relational DBMSs. `TypeCheck` detects TypeBugs by identifying inconsistencies in execution results yielded by type-compatible `SELECT` statements on type-compatible databases.
- We implement and evaluate `TypeCheck` on six widely used relational DBMSs. `TypeCheck` has detected 37 TypeBugs, of which 34 have been confirmed as previously unknown bugs, with 10 of these confirmed bugs already fixed.
- We conduct a comprehensive analysis of the root causes of these confirmed TypeBugs, revealing that TypeBugs emerge across diverse data type handling scenarios, including explicit data type conversions, implicit data type conversions, and specific computation logic.

## 2 PRELIMINARIES

This section outlines the key concepts in `TypeCheck`. We first describe relational DBMSs, SQL, and our target systems (Section 2.1) and then discuss data types and SQL operations (Section 2.2).

### 2.1 Relational Database Management Systems

Based on the DB-Engines Ranking [1], Relational Database Management Systems (DBMSs) are the predominant DBMSs and form the backbone for many modern applications [3], such as e-commerce and social platforms. Relational DBMSs store data in tables [14] with typed columns and use Structured Query Language (SQL) [13] to define, manipulate, and query data.

Our target DBMSs comprise MySQL [4], PostgreSQL [5], MariaDB [2], CockroachDB [39], TiDB [9], and OceanBase [45, 46], as

shown in Table 1. Starting from the seven relational DBMSs ranked in the DB-Engines Top 10 [1], the first criterion narrows the candidates to MySQL and PostgreSQL as fully open-source systems with public bug trackers. We exclude SQLite [7] because it supports only 5 distinct data types, which restricts our bug detection capabilities. We also exclude the remaining four relational DBMSs (Oracle, Microsoft SQL Server, Snowflake, and IBM Db2) because they are commercial and closed-source. To satisfy the second and third criteria, we further include MariaDB, CockroachDB, TiDB, and OceanBase, all of which have served as evaluation targets in prior DBMS testing studies [20, 21, 27, 28, 32, 33, 36, 37, 50]. These six DBMSs cover two categories: MySQL, PostgreSQL, and MariaDB represent traditional single-node DBMSs; TiDB, CockroachDB, and OceanBase represent NewSQL distributed DBMSs.

## 2.2 Data Types and SQL Operations

In this subsection, we first present a comprehensive overview of data types in relational DBMSs, examining their storage formats and value ranges (Section 2.2.1). Next, we discuss SQL operations and how these operations relate to data types (Section 2.2.2).

*2.2.1 Data Types in Relational DBMSs.* In relational DBMSs, each column in a database table must be assigned a data type that defines the kind of data it can store. A data type specifies the allowable data formats and value ranges, the required storage space, and the set of operations that can be performed on those values.

Modern relational DBMSs support rich data type systems across the following four categories:

- **Numeric types** store numerical values with distinct precision levels and value ranges. For example, the `SMALLINT` type occupies 2 bytes per value and stores integers within the range $[-2^{15}, 2^{15}-1]$, while the `FLOAT` type stores floating-point numbers with the range $[1.175494351 \times 10^{-38}, 3.402823466 \times 10^{38}]$.
- **String types** store text data, e.g., the fixed-length string type `CHAR(N)` and the variable-length string type `VARCHAR(N)`. Here, $N$ specifies the upper limit of the string that can be stored.
- **Temporal types** store temporal information. For example, the `TIMESTAMP` data type in MySQL supports the time range from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC.
- **Specialized types** cater to domain-specific use cases, e.g., the `BOOLEAN` type for true/false values, the `JSON` type for semi-structured data, the `UUID` type for universally unique identifiers, and the `BLOB` type for binary large objects.

Note that different data types can store identical logical values within their overlapping ranges. For example, the numeric value 17044 fits in `SMALLINT`, `INT`, and `BIGINT` columns. Similarly, the string `'hello'` can be stored in `CHAR`, `VARCHAR`, or `TEXT` columns.

Our target DBMSs (Table 1) all offer comprehensive data type systems. Specifically, our target traditional DBMSs (i.e., MySQL, PostgreSQL, and MariaDB) support approximately 30 to 40 data types, with PostgreSQL offering the most extensible system. Our target NewSQL DBMSs (i.e., CockroachDB, TiDB, and OceanBase) provide from 18 to more than 30 data types that balance compatibility with distributed architectures, including recent additions like vector types emerging for AI/ML workloads.

*2.2.2 SQL Operations.* SQL statements encompass diverse SQL operations that enable data manipulation and analysis. These SQL operations include arithmetic operators (e.g., $+, -, *, /$) for numerical calculations, comparison operators (e.g., $=, \neq, <, \leq, >, \geq$) for value evaluation, logical operators (e.g., `EXISTS`, `AND`, `OR`, and `NOT`) for boolean combinations, string functions (e.g., `CONCAT`, `SUBSTRING`, `TRIM`) for text processing, temporal functions (e.g., `TIMESTAMP`) for temporal operations, and aggregate functions (e.g., `SUM`, `AVG`, `COUNT`) for statistical computations.

SQL operations are inherently type-dependent: their return types, computation logic, and conversion rules all depend on the input data types. For example, MySQL's `ROUND` returns `BIGINT` for integer inputs but `DOUBLE` for `DOUBLE` inputs; operators such as `+` implicitly convert operands (e.g., `INT` to `BIGINT`). Moreover, the comparison operator = requires identical non-null values for integer types (e.g., 1 = 1), but may match non-identical strings under case-insensitive collation (e.g., `'A' = 'a'`). These type-dependent behaviors significantly increase the complexity of data computation in DBMSs.

# 3 APPROACH

In DBMSs, the same values within a specific value range can be stored in database columns with different data types. Some SQL operations should yield identical results when applied to the same data values with different data types. Based on this observation, we propose `TypeCheck` to effectively detect data-type-related bugs (TypeBugs) in DBMSs. As illustrated in Figure 1, `TypeCheck` operates in three key phases:

(1) **Extracting storage-compatible data types**: `TypeCheck` takes the data types supported by the target DBMS (e.g., `BIGINT` and `INT` in TiDB) as input, and extracts storage-compatible data types $\mathcal{T}$ that can store identical data values without information loss by calculating data constraints (e.g., value range) between each pair of data types. For example, the data types `INT` and `BIGINT` can both store integers within the range $[-2^{31}, 2^{31}-1]$ without information loss.

(2) **Extracting operation-oriented type compatibility rules**: `TypeCheck` analyzes SQL operations (e.g., `TIMESTAMP`) supported by the target DBMS by executing each SQL operation with different parameter data types. Based on these operation instances, `TypeCheck` identifies operation-oriented type compatibility rules *opRules*, which define SQL operations that produce identical result types across storage-compatible parameter types from $\mathcal{T}$. For example, one operation-oriented type compatibility rule is that the `TIMESTAMP` function can return datetime values when taking `INT` and `BIGINT` values as input parameters.

(3) **Detecting TypeBugs**: `TypeCheck` leverages $\mathcal{T}$ and *opRules* to detect TypeBugs. First, `TypeCheck` generates an initial database (e.g., a table $table1$ that has a column $c1$ with type of `BIGINT`) and constructs its corresponding type-compatible database (e.g., a table $table1$ that has a column $c1$ with type of `INT`) using $\mathcal{T}$, and populates identical data values (e.g., a numeric value 17044) for the tables in these two type-compatible databases. Then, `TypeCheck` generates type-compatible `SELECT` statements using SQL operations defined in *opRules* to detect TypeBugs. If the target DBMS returns inconsistent execution results on these type-compatible databases for a certain `SELECT` statement, `TypeCheck` reports a TypeBug. For example,

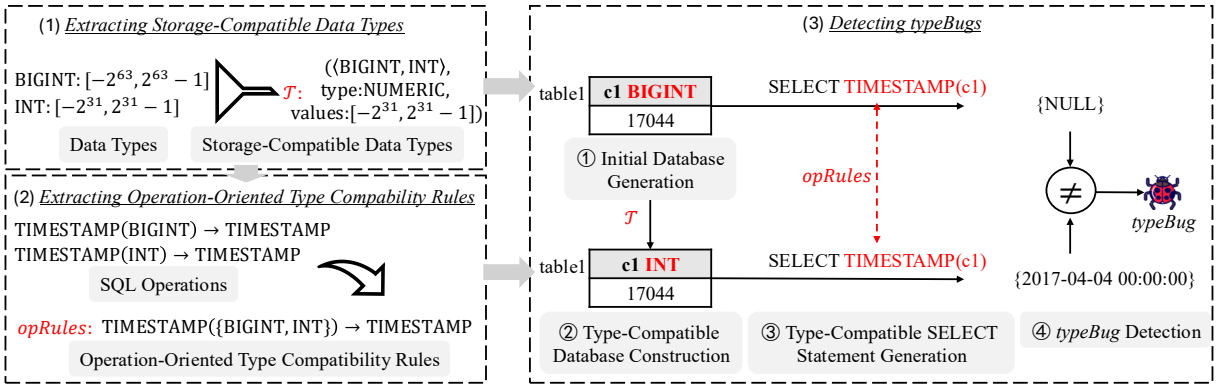


**Figure 1: The architecture of TypeCheck.**

TiDB returns NULL for the operation TIMESTAMP(c1) on the column $c1$ with type of BIGINT, while a different result '2017-04-04 00:00:00' is returned against the INT type. Thus, TypeCheck can detect this TypeBug.

## 3.1 Extracting Storage-Compatible Data Types

Some database columns with different data types can store identical data values within specific value ranges. For example, in MySQL, database columns with TINYINT and INT can both store all values in the range $[-128, 127]$. To extract storage-compatible data types in the target DBMS, we first formalize storage-compatible data types, and then present our approach to automatically identify the storage-compatible data types in the target DBMS by analyzing the supported data types and their value range constraints.

*3.1.1 Storage-Compatible Data Types.* Data types in DBMSs define how data values are stored in database columns, specifying their storage formats and value ranges. However, for some data types, the retrieved values are different from the original stored values. For example, when we store the value 1 into a column with the YEAR type, MySQL implicitly converts 1 to 2001 so that we will retrieve 2001 instead of 1. When storing the double-precision value 1.00000001 into a FLOAT column, MySQL implicitly casts it to a 4-byte single-precision float, and the value is truncated to 1.0. In addition, when data values violate the constraints (e.g., value ranges) of a data type, DBMSs typically raise errors. For example, when storing the value 2026 into a column with the TINYINT type in MySQL, an overflow error is raised since 2026 exceeds the value range of TINYINT. To determine whether a data type can successfully store and consistently retrieve data values, we define the store-retrieve consistency property as follows.

**Definition 1. Store-Retrieve Consistency** For a given range *range* (including allowed data types *range.type* and data values *range.values*), a data type $t$ can successfully store and consistently retrieve all values within *range* without information loss if and only if for every value $v \in range.values$, $Store(col_t, v) \wedge Retrieve(col_t) = v$. Here, $col_t$ denotes a column with data type $t$. $Store(col_t, v)$ represents that we store a value $v$ in an empty column $col_t$, and $Retrieve(col_t)$ represents that we retrieve data from column $col_t$.

```
-- Create tables with different data types
1. CREATE TABLE table1 (c1 TINYINT);
2. CREATE TABLE table2 (c1 INT);

-- Insert sample values
3. INSERT INTO table1 VALUES (-128), (0), (127);
4. INSERT INTO table2 VALUES (-128), (0), (127);

-- Query results
5. SELECT c1 FROM table1; -- {-128, 0, 127}
6. SELECT c1 FROM table2; -- {-128, 0, 127}
```

**Listing 2: Range verification for the storage-compatible data types TINYINT and INT.**

We use the following method to verify whether a data type $t$ satisfies store-retrieve consistency within a given range *range*: (1) We first create a table *table* with column *col* of data type $t$. (2) Then, for each value $v \in range.values$ formatted by *range.type*, we store $v$ into *col* (i.e., $Store(col_t, v)$) and retrieve the value from *col* using SELECT statements (i.e., $Retrieve(col_t)$). When the store operation is executed successfully and the retrieved value is the same as the stored value (i.e., $Store(col_t, v) \wedge Retrieve(col_t) = v$), we proceed to the next value in *range.values*. (3) Finally, if all values in *range.values* satisfy the store-retrieve consistency property, we can conclude that data type $t$ can successfully store and retrieve all values within *range* without information loss. These values within *range* are valid and do not cause rounding or overflow issues.

Based on the store-retrieve consistency property, we define storage-compatible data types as follows.

**Definition 2. Storage-Compatible Data Types** For a given range *range*, two data types $t_i$ and $t_j$ are storage-compatible over *range* when they both satisfy the store-retrieve consistency property for all values $v \in range$.

To verify the storage compatibility of two data types over a specific range *range*, one approach is to enumerate all values $v$ in *range* and verify that each value satisfies the store-retrieve consistency property for both data types. However, this approach becomes infeasible when *range* contains an extremely large number of possible values (e.g., MySQL's INT type range includes over 4 billion values).

To efficiently determine whether two data types $t_i$ and $t_j$ are storage-compatible over a specific range *range*, we sample a set of values from *range.values* and determine whether $t_i$ and $t_j$ are

storage-compatible data types by verifying if both satisfy store-retrieve consistency for these values. As demonstrated in Listing 2, given the range *range* with allowed data type NUMERIC and allowed data values $\in [-128, 127]$, along with two data types TINYINT and INT, we first create two tables *table*1 and *table*2 with the two data types, respectively (Lines 1–2). Then, we sample some values $\{-128, 0, 127\}$ from *range* including boundary values for NUMERIC values and random values within *range.values*, and insert them into both tables (Lines 3–4). Finally, we retrieve the values from both tables using SELECT statements (Lines 5–6) and check whether the retrieved values are identical to the inserted values. When all generated values maintain their integrity across both data types, we assume that the two data types are storage-compatible over the range *range*. For example, as shown in Listing 2, the TINYINT and INT data types demonstrate storage compatibility over the range $[-128, 127]$.

Sampling-based compatibility checking could theoretically introduce both false positives (incorrectly declaring two types compatible) and false negatives (missing compatible pairs). To ensure correctness, we manually verify our obtained compatible types.

*3.1.2 Extracting Storage-Compatible Data Types.* To automatically identify storage-compatible data types in the target DBMS, we first precisely describe the data constraints of each data type, including their storage formats and value ranges. TypeCheck currently supports three major data type categories: numeric, temporal and string. These types dominate real-world database workloads [41]. It is important to note that for numeric types, we only consider cases involving integer values. This is because, as discussed in Section 3.1.1, certain data types do not maintain store-retrieve consistency for floating-point values. We define the following three types of data constraints:

- **Numeric constraints** define the minimum and maximum values ($minValue$, $maxValue$) that can be stored by the data type, establishing the valid range $[minValue, maxValue]$. For example, in MySQL, the INT type has the value range of $[-2^{31}, 2^{31} - 1]$.
- **Temporal constraints** define the earliest and latest time points ($minTime$, $maxTime$) that can be stored by the data type, establishing the valid range $[minTime, maxTime]$. Additionally, temporal types may include a precision attribute *precision* that specifies the granularity of time representation (e.g., day, second). For example, in MySQL, the DATE type has the time range of ['1000-01-01', '9999-12-31'] with day precision.
- **String constraints** specify the maximum length $strLen$ that a data type can store, establishing the valid range $[0, strLen]$. For example, in MySQL, the VARCHAR(10) type can store strings with a maximum length of 10 characters.

For each data type, we describe its data constraints using the above six constraints, i.e., $minValue$, $maxValue$, $minTime$, $maxTime$, $precision$, and $strLen$. If a specific data type does not have certain constraints, we set the corresponding constraints to NULL. For example, since the INT type only exhibits numeric constraints, we set $strLen$, $minTime$, $maxTime$, and $precision$ to NULL.

We then identify storage-compatible data type pairs by calculating the data constraints of all supported data types in the target DBMS, as shown in Algorithm 1. Specifically, we first initialize an empty set $\mathcal{T}$ to store the identified storage-compatible data type pairs (Line 1). Then, we conduct pairwise analysis for all data types in the target DBMS (Lines 2–12).

For each data type pair $(t_i, t_j)$, we first determine the range type $rangeType$ according to the following rules (Line 3):

- If both $t_i$ and $t_j$ are numeric types, we set $rangeType$ to NUMERIC. For example, TINYINT and INT are both numeric types, and their range type is NUMERIC.
- If both $t_i$ and $t_j$ are temporal types, we set $rangeType$ to TEMPORAL. For example, DATE and DATETIME are both temporal types, and their range type is TEMPORAL.

**Algorithm 1:** Extract Storage-Compatible Data Types

**Input:** Data types $T$
**Output:** Storage-compatible data type pairs $\mathcal{T}$

```
1  T ← ∅;
2  for t_i, t_j ∈ T do
3      rangeType ← getRangeDataType(t_i, t_j);
4      if rangeType = NULL then
5          continue;
6      rangeCons ← getRangeConstraints(t_i, t_j);
7      if rangeCons = NULL then
8          continue;
9      range ← computeRange(rangeType, rangeCons);
10     if satisfyStoreRetrieve(t_i, range) ∧
        satisfyStoreRetrieve(t_j, range) then
11         pair ← new Pair(< t_i, t_j >, range);
12         T ← T ∪ pair;
13 return T;
14 Function getRangeConstraints(t_i, t_j) do
15     cons ← new Constraints();
16     cons.originalTypes ← [t_i, t_j];
17     cons.minValue ← max(t_i.minValue, t_j.minValue);
18     cons.maxValue ← min(t_i.maxValue, t_j.maxValue);
19     cons.minTime ← max(t_i.minTime, t_j.minTime);
20     cons.maxTime ← min(t_i.maxTime, t_j.maxTime);
21     cons.precision ← min(t_i.precision, t_j.precision);
22     cons.strLen ← min(t_i.strLen, t_j.strLen);
23     if hasConflictingConstraints (cons) then
24         return NULL;
25     return cons;
26 Function computeRange(type, cons) do
27     range.type ← type;
28     if type.isNumeric() then
29         range.values ← [cons.minValue, cons.maxValue];
30         if cons.originalTypes.contains(STRING) then
31             strValues ← getStrToValueRange(cons.strLen);
32             range.values ← range.values ∩ strValues;
33     else if type.isTemporal() then
34         range.values ← ...;
35     else if type.isString() then
36         range.values ← ...;
37     return range;
```

- If both $t_i$ and $t_j$ are string types, we set *rangeType* to STRING. For example, CHAR(10) and VARCHAR(20) are both string types, and their range type is STRING.
- If one of $t_i$ and $t_j$ is a string type and the other is a numeric or temporal type, we set *rangeType* to NUMERIC or TEMPORAL, respectively, since string types can represent numeric and temporal values.
- Otherwise, we set *rangeType* to NULL.

When *rangeType* is NULL, we skip this type pair and process the next iteration (Lines 4–5). In this case, $t_i$ and $t_j$ are not storage-compatible, e.g., INT and DATE. These types represent different data categories (e.g., numeric vs. temporal) and cannot be mapped to a common range type, making store-retrieve consistency impossible.

After determining the range type *rangeType*, we compute the range constraints that are defined by the data constraints of both types (Line 6). Specifically, we calculate a constraint *rangeCons* that satisfies all the data constraints of $t_i$ and $t_j$. Specifically, for the numeric constraints, we calculate the *minValue* and *maxValue* of *rangeCons* as $max(t_i.minValue, t_j.minValue)$ (Line 17) and $min(t_i.maxValue, t_j.maxValue)$ (Line 18), respectively. For example, consider the data type pair (INT, BIGINT). The INT type has a value range of $[-2^{31}, 2^{31}-1]$, while BIGINT has a range of $[-2^{63}, 2^{63}-1]$. The computed constraints yield $cons.minValue = -2^{31}$ and $cons.maxValue = 2^{31}-1$. Thus, the final range is $[-2^{31}, 2^{31}-1]$, constrained by the more restrictive INT type. For other constraints, we also utilize a similar calculation method (Line 19–22). For example, for string types, consider the data type pair (CHAR(10), VARCHAR(20)). The fixed-length CHAR(10) requires exactly 10 characters, while VARCHAR(20) allows up to 20 characters. The computed string constraint is $cons.strLen = \min(10, 20) = 10$, resulting in a compatible range of strings with length at most 10 characters.

Theoretically, if there are conflicts in the data constraints of both types (e.g., the derived *minValue* is greater than *maxValue*), the two types cannot be storage-compatible (Lines 23–24). Thus, we skip this data type pair and process the next iteration (Lines 7–8).

After computing the range constraints *rangeCons*, we calculate a more constrained range *range* defined by *rangeType* and *rangeCons* (Line 9). If *rangeType* is numeric, we can define the value range as $[rangeCons.minValue, rangeCons.maxValue]$ (Line 29). If the original data types contain a STRING type, we should further apply the string constraints to limit the range of values (Line 30–32). Specifically, we obtain the value range according to the length of characters that the data types can store (Line 31), and compute its intersection with the existing value range (Line 32). For example, given the data type pair (VARCHAR(5), TINYINT), the compatible range becomes $[-128, 127]$. The VARCHAR(5) can represent 5-digit decimal strings in the range $[-9999, 99999]$, and the TINYINT range is $[-128, 127]$. The intersection yields $[-128, 127]$, constrained by the TINYINT range. If *rangeType* is a temporal or string type, we utilize a similar method to calculate the constrained range *range*.

Finally, we verify whether both data types $t_i$ and $t_j$ satisfy store-retrieve consistency over *range* using the method described in Section 3.1.1 (Line 10). When both data types satisfy store-retrieve consistency for *range*, we add the data type pair $(t_i, t_j)$ along with the range *range* into the set $\mathcal{T}$ (Lines 11–12).

## 3.2 Extracting Operation-Oriented Type Compatibility Rules

Ideally, an SQL operation demonstrates operation-oriented type compatibility across two tuples of data types if it yields identical results, both in result values and result data types, when applied to the same logical data values stored in the corresponding positions of the two type tuples. For example, the TIMESTAMP function returns the exact same datetime value and type when applied to the same integer value stored in both INT and BIGINT columns. However, storage-compatible types do not necessarily exhibit operation-oriented type compatibility, as certain operations may be unsupported on some types (e.g., CockroachDB's ROUND() supports DECIMAL but not INT2, despite both being storage-compatible).

To detect TypeBugs, we need to define compatibility rules in advance. Specifically, we must identify which parameter type tuples for SQL operations in the system should consistently return the same result types and values when storing identical input values. However, these rules are often not explicitly documented. If we derive compatibility rules based on the above definition, bugs in the system could lead to the exclusion of valid rules.

We observe that if the result types of SQL operations are the same, then performing operations on identical data values is typically expected to yield the same result values. Based on this observation, we relax the compatibility constraints and propose a relaxed definition of operation-oriented type compatibility to automate the construction of compatibility rules. The definition is as follows.

**Definition 3. Operation-Oriented Type Compatibility** For an $n$-ary SQL operation $op$, it exhibits operation-oriented type compatibility between two tuples of data types $ParamTypes^i = (t_1^i, t_2^i, \ldots, t_n^i)$ and $ParamTypes^j = (t_1^j, t_2^j, \ldots, t_n^j)$, if, for all $t_x^i$ and $t_x^j$, the data types $t_x^i$ and $t_x^j$ are storage-compatible. Additionally, the operation $op$ yields an identical result type on both tuples $ParamTypes^i$ and $ParamTypes^j$. For example, in MySQL, applying the unary ROUND operation to INT and TINYINT columns produces BIGINT results in both cases. Then for any tuple of identical values $(v_1, v_2, \ldots, v_n)$, where each $v_x$ can be stored successfully in both $t_x^i$ and $t_x^j$ without information loss, the operation $op$ is expected to yield identical result values. A violation of this expectation may indicate a TypeBug in the system.

We further use the following operation-oriented type compatibility rule to represent operation-oriented type compatibility for SQL operation $op$ with respect to the result type $R$: $OpRule(op, R) = (T_1, T_2, \ldots, T_n)$, where each $T_x$ is a set of data types (e.g., $T_1 = \{INT, BIGINT, SMALLINT\}$). For any $t_x^i \in T_x$, there exists a type $t_x^j \in T_x$ such that $t_x^i$ and $t_x^j$ are storage-compatible. Then for a tuple of values $(v_1, v_2, \ldots, v_n)$ that can be stored without information loss in two tuples of data types $ParamTypes^i = (t_1^i, t_2^i, \ldots, t_n^i)$ and $ParamTypes^j = (t_1^j, t_2^j, \ldots, t_n^j)$ (with $t_x^i, t_x^j \in T_x$), if $op$ can successfully execute on both inputs and yield the same result type $R$, then $op$ is expected to produce identical result values for these inputs.

Algorithm 2 presents the process of extracting operation-oriented type compatibility rules. Given the input of SQL operations $OP$ and data types $T$ supported by the target system, along with the storage-compatible data type pairs $\mathcal{T}$ identified in Section 3.1, we first initialize an empty set *opRules* to store the obtained operation-oriented type compatibility rules (Line 1). Then, we iterate over

**Algorithm 2:** Extract Operation-Oriented Type Compatibility Rules

**Input:** SQL operations $OP$, Data types $T$, Storage-compatible data type pairs $\mathcal{T}$
**Output:** Operation-oriented type compatibility rules $opRules$

```
1  opRules ← ∅;
2  for op ∈ OP do
3      tempRules ← ∅;
4      n ← getArity(op);
5      for t_1, t_2, ..., t_n ∈ T do
6          paramTypes ← [t_1, t_2, ..., t_n];
7          resultType ← getResultType(op, paramTypes);
8          if resultType = NULL then
9              continue;
10         findCompatibleRule ← false;
11         for tempRule ∈ tempRules do
12             if resultType = tempRule.resultType
                ∧isStorageCompatible(paramTypes,
                tempRule.paramOpts, 𝒯) then
13                 updateRule (tempRule, paramTypes);
14                 findCompatibleRule ← true;
15                 break;
16         if !findCompatibleRule then
17             newRule ← new
                OpRule(op, paramTypes, resultType);
18             tempRules ← tempRules ∪ newRule;
19     opRules ← opRules ∪ tempRules;
20 return opRules;
21 Function isStorageCompatible(types, paramOpts, 𝒯) do
22     for k ← 1 to types.length do
23         if !isCompatible(types[k], paramOpts.get(k), 𝒯)
            then
24             return false;
25     return true;
26 Function updateRule(opRule, paramTypes) do
27     for k ← 1 to opRule.paramOpts.length do
28         opRule.getParamOpt(k).add(paramTypes[k]);
```

each operation $op$ in SQL operations $OP$ and extract $op$-oriented type compatibility rules (Lines 3–18).

For each operation $op$, we initialize an empty set $tempRules$ to store $op$-oriented compatibility rules (Line 3). Then we obtain the number $n$ of parameters of $op$ (Line 4). Based on $n$, we use the data types from $T$ to enumerate all parameter type tuples supported by the operation $op$ (Line 5). For example, if $T = \{\texttt{INT}, \texttt{BIGINT}\}$, for the unary operation `TIMESTAMP`, we enumerate all parameter type tuples, i.e., (`INT`) and (`BIGINT`).

For each parameter type tuple $paramTypes$, we obtain the result type $resultType$ of $op$ on $paramTypes$ using the JDBC API (Line 7). Specifically, we first create a table with columns of the specified types in $paramTypes$, and then construct a `SELECT` statement $stmt$ that applies the operation $op$ to these columns, executing them on the table to obtain the execution result. Finally, we use standard JDBC APIs to identify its SQL data type. If $stmt$ cannot be executed successfully, it indicates that the operation $op$ does not support the parameter type tuple $paramTypes$. For example, the `gcd()` function in PostgreSQL does not support the parameter type tuple ($SMALLINT$, $SMALLINT$). We then proceed to analyze the next parameter type tuple (Lines 8–9).

For each parameter type tuple $paramTypes$ and its corresponding result type $resultType$ for $op$, we check whether $tempRules$ contains a rule that shares the same result type as $resultType$ and has parameter types storage-compatible with $paramTypes$ in each parameter location (Lines 21–25). When such a rule $tempRule$ exists, we update $tempRule$ by adding each parameter type from $paramTypes$ to the corresponding position in $tempRule$, serving as an optional parameter type for that position. (Lines 26–28). For example, for the operation `add`, both parameter type tuples ($INT$, $BIGINT$) and ($BIGINT$, $INT$) result in the same result type `BIGINT`. We first create a rule for the first tuple with $OpRule(add, BIGINT) = (\{INT\}, \{BIGINT\})$. When processing the second tuple, we find that it is storage-compatible with the existing rule and produces an identical result type. We then update $paramOpts$ to $[\{INT, BIGINT\}, \{INT, BIGINT\}]$, indicating that the `add` operation accepts both `INT` and `BIGINT` types for its two parameters.

After iterating all possible parameter type tuples for operation $op$, we obtain $tempRules$ and add it to the final operation-oriented type compatibility rules $opRules$ (Line 19).

**Algorithm 3:** Detect TypeBugs

**Input:** Storage-compatible data type pairs $\mathcal{T}$, Operation-oriented type compatibility rules $opRules$
**Output:** Detected TypeBugs

```
1  repeat
2      db_1 ← generateRandomDatabase (𝒯);
3      db_2 ← mutateWithCompatibleTypes (db_1, 𝒯);
4      for k ← 1 to maxStatements do
5          stmt ← generateSelectStatement (opRules);
6          result_1, error_1 ← executeSQL (db_1, stmt);
7          result_2, error_2 ← executeSQL (db_2, stmt);
8          if isExpected (error_1) or isExpected (error_2) then
9              continue;
10         if error_1 ≠ error_2 or result_1 ≠ result_2 then
11             Report TypeBug;
12 until timeBudget;
```

## 3.3 Detecting TypeBugs

After extracting storage-compatible data types $\mathcal{T}$ and operation-oriented type compatibility rules $opRules$, `TypeCheck` iteratively generates pairs of type-compatible databases using data types from $\mathcal{T}$ and tests them with SQL statements that incorporate operations defined in $opRules$ to detect TypeBugs.

Algorithm 3 presents the overall workflow of detecting TypeBugs. First, we generate an initial database $db_1$ by creating `CREATE TABLE` statements for a random number of tables with different data types selected from $\mathcal{T}$. Second, we construct the corresponding type-compatible database $db_2$ by mutating data types in $db_1$ according to compatibility relationships in $\mathcal{T}$. Third, we populate both databases with identical data values using `INSERT` statements,

where the values are drawn from the common data range specified in $\mathcal{T}$. Finally, we execute at most *maxStatements* type-compatible SELECT statements with operations defined by *opRules* on both databases and compare their execution results to reveal inconsistencies. When an inconsistency is revealed, a TypeBug is reported. We repeat the above process until the time budget *timeBudget* is exhausted. As shown in the right part of Figure 1, TypeCheck creates type-compatible databases with INT and BIGINT data types and detects a TypeBug on the TIMESTAMP function due to the inconsistent execution results.

Note that both the parameters *timeBudget* and *maxStatements* are configurable. In our evaluation, we set *timeBudget* to 24 hours and *maxStatements* to 20,000. These values were determined empirically to balance coverage and efficiency.

*3.3.1 Initial Database Generation.* The initial database generation step creates a database $db_1$ with multiple tables using CREATE TABLE statements. Each table in $db_1$ contains multiple columns with data types randomly selected from the storage-compatible data types $\mathcal{T}$.

TypeCheck first initializes an empty database $db_1$, an empty schema *schema*, and an empty CREATE TABLE statement set $S$. TypeCheck then randomly determines the number of tables to create, ranging from 1 to *maxTables*. For each table, TypeCheck generates a CREATE TABLE statement containing multiple columns with different data types randomly selected from $\mathcal{T}$, along with random indexes (e.g., PRIMARY KEY, UNIQUE INDEX, and INDEX) and constraints (e.g., NOT NULL, FOREIGN KEY, and CHECK). When a CREATE TABLE statement includes optional features, TypeCheck randomly generates appropriate values for these options to increase test diversity. We set *maxTables* to 2 in our experiments.

After executing a generated CREATE TABLE statement to create a table in $db_1$, TypeCheck updates the schema *schema* to reflect the newly added table. The evolving schema guides the generation of subsequent CREATE TABLE statements, ensuring consistency across tables (e.g., for foreign key constraints). This phase returns the created database $db_1$, the set of CREATE TABLE statements $S$ used to construct it, and the final schema *schema*.

*3.3.2 Type-Compatible Database Construction.* The compatible database construction step creates an additional database $db_2$ that is structurally identical to $db_1$ but uses different compatible data types. Starting from the CREATE TABLE statements $S$ used for constructing the initial database $db_1$, we construct $db_2$ by systematically mutating the data types in each CREATE TABLE statement in $S$ according to the compatibility relationships defined in $\mathcal{T}$.

Algorithm 4 presents the process of constructing a compatible database. We first initialize two empty sets, *createTables* and *insertStmts*, to store the modified CREATE TABLE statements used for constructing $db_2$ and the corresponding INSERT statements used for inserting data values into the database, respectively (Lines 1–2). We then iterate each CREATE TABLE statement *createTable* in the initial database $db_1$ to perform type mutations (Lines 4–10). For each table, we initialize a *rangeList* to store the range information for columns with storage-compatible data types (Line 4).

For each column in *createTable*, we first obtain the original data type $t_i$ (e.g., BIGINT) of the column (Line 6). Based on the storage-compatible data types $\mathcal{T}$, we retrieve all types compatible with $t_i$ (Line 7). We then randomly select one compatible type $t_j$ (e.g., INT)

**Algorithm 4:** Construct Type-Compatible Databases

**Input:** Initial database $db_1$, Storage-compatible data types $\mathcal{T}$
**Output:** Type-compatible database $db_2$

```
1  createTables ← ∅;
2  insertStmts ← ∅;
3  foreach createTable ∈ db1.getCreateTables() do
4      rangeList ← ∅;
5      foreach column in createTable do
6          t_i ← getDataType(column);
7          compTypes ← getCompatibleTypes(t_i, T);
8          t_j ← randomSelect(compTypes);
9          column.replaceType(t_i, t_j);
10         rangeList.append(getRange(t_i, t_j, T));
11     createTables ← createTables ∪ {createTable};
12     testValues ← generateTestValues(rangeList);
13     insertStmt ←
         generateInsertStmt(createTable, testValues);
14     insertStmts ← insertStmts ∪ insertStmt;
15 db2 ← createDatabase(createTables);
16 insertData(db2, insertStmts);
17 insertData(db1, insertStmts);
18 return db2
```

from the result set (Line 8) and replace the original type of the column with $t_j$ (Line 9). We then compute and append the range information for the pair $< t_i, t_j >$ to *rangeList* (Line 11).

After processing all columns of the CREATE TABLE statement, we add the modified *createTable* to the *createTables* set (Line 11). We then generate test values based on the *rangeList* (Line 12) and construct the corresponding INSERT statement (Lines 13–14).

After iterating all CREATE TABLE statements, we execute the modified CREATE TABLE statements to construct the table structure in the target database $db_2$ (Line 15). We then execute the INSERT statements to populate both $db_2$ and the original database $db_1$ with the same test data (Lines 16–17). Finally, we return the constructed compatible database $db_2$ (Line 18). Note that each table has at most 10 rows.

Note that apart from data types, other structural elements from the original schema, including constraints (e.g., PRIMARY KEY, UNIQUE, NOT NULL), indexes, and table attributes will remain unchanged during the mutation process. However, certain mutations may violate DBMS-specific constraints. For example, prefix indexes in MySQL require string types. Mutating a string column to a numeric type would trigger an error such as "Incorrect prefix key". When such constraint violations occur during database generation, TypeCheck discards the current database pair and regenerates a new initial database $db_1$ to restart the construction process.

*3.3.3 Type-Compatible* SELECT *Statement Generation.* After constructing the type-compatible databases, we generate type-compatible SELECT statements using a grammar-based approach that incorporates operation-oriented type compatibility rules *opRules*.

⟨select_stmt⟩ ::= SELECT ⟨col_list⟩ FROM ⟨table_list⟩
[⟨where_clause⟩] [⟨group_by_clause⟩]
⟨table_list⟩ ::= ⟨table⟩
| ⟨table_list⟩ ⟨join_type⟩ JOIN ⟨table⟩ ON expr
⟨join_type⟩ ::= INNER | LEFT | RIGHT | FULL
⟨col_list⟩ ::= ⟨col⟩ | expr | ⟨col_list⟩, ⟨col⟩
⟨where_clause⟩ ::= WHERE expr
⟨group_by_clause⟩ ::= GROUP BY ⟨col_list ⟩[HAVING expr]

**Figure 2: BNF grammar for generating SELECT statements.**

**Algorithm 5:** Generate Type-Compatible SQL Expressions

**Input:** Operation-oriented type compatibility rules $opRules$, two type-compatible databases $db_1$ and $db_2$, previously tested expressions $testedExprs$ on $db_1$ and $db_2$
**Output:** A new SQL expression $expr$

1 $rule \leftarrow$ randomSelect($opRules$) ;
2 $params \leftarrow [\,]$;
3 **for** $k \leftarrow 1$ **to** $rule.paramOpts.length$ **do**
4 $option \leftarrow rule$.getParamOpt($k$);
5 $compColumns \leftarrow$ getCompatibleColumnsFrom($db_1, db_2, option$);
6 $compConsts \leftarrow$ generateConstantsWith($option$);
7 $compExprs \leftarrow testedExprs$.getResultTypeIn($option$);
8 $params$.append(randomSelect($compColumns \cup compExprs \cup compConsts$));
9 $expr \leftarrow$ constructExpression($rule.op, params$);
10 $testedExprs$.add($< expr, rule.resultType >$);
11 **return** $expr$;

Following the BNF grammar shown in Figure 2, TypeCheck generates type-compatible SELECT statements by randomly selecting one or more tables from the database schema *schema*, constructing a column list (which may include column names or expressions with SQL operations defined by *opRules*), and optionally adding WHERE clauses for filtering conditions. TypeCheck supports both single-table queries and multi-table queries with various JOIN types (e.g., INNER, LEFT, RIGHT, and FULL), and supports GROUP BY clauses with optional HAVING conditions to enable aggregate queries. The key challenge in this process is generating complex and valid SQL expressions that incorporate SQL operations defined by *opRules*.

Algorithm 5 presents the process for generating a single SQL expression. We first randomly select a rule *rule* from the operation-oriented type compatibility rules *opRules* (Line 1). Then, we initialize an empty parameter list *params* to store the generated parameters for the selected operation in *rule* (Line 2).

Next, we generate the parameters for the operation sequentially based on *rule* (Lines 3–8). For each parameter, we obtain all the optional data types *option* for that parameter based on the compatibility rule. (Line 4). Based on *option*, we first select columns with the same column names and storage-compatible data types from databases $db_1$ and $db_2$, ensuring that the data types of both columns are included in *option* (Line 5). For example, column $c1$ in Figure 1 is such a compatible column. We then create some constants to ensure their types also match the data types in *option* (Line 6).

To generate complex expressions, we also consider previously tested expressions on $db_1$ and $db_2$ as candidate parameters. These expressions were successfully executed on both databases and did not result in TypeBugs. Specifically, we retrieve those tested expressions from $testedExprs$ whose result type matches the data types in *option* (Line 7). By reusing already validated expressions as parameters, our approach can systematically build increasingly complex SQL expressions, such as nested function calls and multi-level arithmetic operations. Finally, we randomly choose one parameter from the generated candidates to serve as the current parameter value (Line 8).

After generating all parameters, we construct a SQL expression $expr$ using the SQL operation in the selected rule and the generated parameter list *params* (Line 9). We record this expression along with its result type in $testedExprs$ for future reference (Line 10). Finally, we return the newly generated expression $expr$ (Line 11).

*3.3.4 TypeBug Detection.* The bug detection phase executes generated type-compatible SELECT statements on type-compatible databases $db_i$ and $db_j$, comparing their execution results to identify inconsistencies. For each statement, TypeCheck captures execution results ($result_i, result_j$) and errors ($error_i, error_j$) from both databases $db_i$ and $db_j$. Note that errors can happen due to rounding or overflow issues; however, they should be the same in both databases.

TypeCheck first filters out expected errors resulting from incorrect SQL semantics in the generated statements. For example, our SQL expression generation process may produce SQL operations that operate on unsupported parameter type tuples. The gcd() function in PostgreSQL returns the BIGINT type for both parameter type tuples (BIGINT, SMALLINT) and (SMALLINT, BIGINT). From this, we derive the compatibility rule $OpRule(gcd, BIGINT) = (\{BIGINT, SMALLINT\}, \{BIGINT, SMALLINT\})$. During the expression generation process, we can generate a gcd() function that operates on the type tuple (SMALLINT, SMALLINT). However, this parameter type tuple is unsupported by the gcd() function and will produce a runtime error.

After filtering expected errors, inconsistencies are detected through two systematic comparisons: asymmetric execution states (where one database succeeds and the other fails) and divergent result sets. Any detected discrepancy triggers an immediate TypeBug report.

## 4 EVALUATION

We implement TypeCheck on our target DBMSs with about 9,200 lines of Java code. TypeCheck primarily consists of five main functionalities, including extracting storage-compatible data types, extracting operation-oriented type compatibility rules, constructing type-compatible databases, generating type-compatible SELECT statements, and detecting TypeBugs.

TypeCheck supports data types across three categories: numeric types (e.g., INT, DECIMAL, and their UNSIGNED variants), temporal types (e.g., DATE, TIMESTAMP), and string types (e.g., VARCHAR, TEXT, and their TINY/MEDIUM/LONG variants). For parameterized types like VARCHAR(N), we choose some instantiations. Overall, TypeCheck covers 145 out of 207 data types and 479 functions across the six

**Table 2: Overall bug detection results for TypeCheck**

| DBMS | Submitted | Confirmed | Fixed | Duplicate |
|---|---|---|---|---|
| MySQL | 8 | 8 | 0 | 0 |
| PostgreSQL | 1 | 1 | 1 | 0 |
| MariaDB | 13 | 11 | 3 | 1 |
| CockroachDB | 1 | 1 | 1 | 0 |
| TiDB | 6 | 6 | 2 | 0 |
| OceanBase | 8 | 7 | 3 | 0 |
| **Total** | **37** | **34** | **10** | **1** |

target DBMSs, in which their ranges are hardcoded based on the official documentation.

We evaluate the effectiveness of TypeCheck by investigating the following three research questions:

- **RQ1**. What TypeBugs can TypeCheck detect in real-world relational DBMSs?
- **RQ2**. What are the manifestations and root causes of the detected TypeBugs?
- **RQ3**. How effective is TypeCheck in detecting TypeBugs compared to existing approaches?

## 4.1 Experimental Setup

**Target DBMSs**. We evaluate TypeCheck on six widely used relational DBMSs, as detailed in Table 1. Our initial prototype is developed using the latest stable releases available at the time of development: MySQL 8.0.36, PostgreSQL 16.2, MariaDB 11.3.2, CockroachDB 23.2, TiDB 8.0.0, and OceanBase 4.3.0. To maximize bug discovery, we continuously monitor for updates and evaluate newer versions of these DBMSs throughout our study.

**Testing methodology**. We run TypeCheck on each target DBMS for 24 hours, after which we analyze the generated bug reports. For each compatible database pair $(db_1, db_2)$ constructed using storage-compatible data types, TypeCheck continuously generates type-compatible SELECT statements according to operation-oriented type compatibility rules to test the target DBMS.

When a DBMS produces different execution results for a SELECT statement *stmt* on $db_1$ and $db_2$, TypeCheck generates a bug report containing the database pair $(db_1, db_2)$, the INSERT statements, *stmt*, and the execution results. We then simplify each report by removing irrelevant database objects via execution plan analysis, reducing INSERT statements through iterative bug persistence testing, and performing syntax-based reduction on *stmt* via Abstract Syntax Tree parsing. Finally, we deduplicate reports by extracting the operation sequence from *stmt* and discarding entries with matching operation sequences.

## 4.2 Overall Bug Detection Results

To address RQ1, we evaluate the effectiveness of TypeCheck in detecting TypeBugs across real-world relational DBMSs. Following the experimental methodology outlined in Section 4.1, we continuously test our target DBMSs and systematically analyze the generated bug reports. Table 2 summarizes the overall detection results. In total, we have submitted 37 bugs to DBMS developers: 8 in MySQL, 1 in PostgreSQL, 13 in MariaDB, 1 in CockroachDB, 6 in TiDB, and 8 in OceanBase. In the following, we discuss the status, severity, and categories of these submitted bugs.

**Bug status**. Of the 37 submitted bugs, 34 have been confirmed as new bugs, with 10 of these confirmed already fixed. One bug in MariaDB is a duplicate of an existing bug report in the bug tracking system of MariaDB. At the time of writing this paper, the remaining 2 bugs are still under investigation by the corresponding DBMS developers. Note that we do not report any false positives in our experiments. Since bugs in MySQL and MariaDB are rarely fixed [10], the fix rate appears relatively low.

**Bug severity**. Out of the 34 confirmed bugs, 18 are classified as *Critical* or *Major* severity: all 8 bugs in MySQL and 10 bugs in MariaDB. The remaining 16 bugs consist of 1 *Minor* bug in MariaDB, 2 *Moderate* bugs in TiDB, 1 unlabeled bug in PostgreSQL, 1 unlabeled bug in CockroachDB, 4 unlabeled bugs in TiDB, and 7 unlabeled bugs in OceanBase. These results demonstrate that TypeCheck effectively identifies high-severity bugs that are deemed critical by DBMS developers.

**Bug categories**. The 34 confirmed bugs comprise 32 bugs (8 bugs in MySQL, 11 bugs in MariaDB, 1 bug in CockroachDB, 6 bugs in TiDB, and 6 bugs in OceanBase) that produce inconsistent query results among type-compatible databases, and two crash bugs (1 bug in PostgreSQL that crashes only one database between two type-compatible databases, and 1 bug in OceanBase that produces crashes in both type-compatible databases).

## 4.3 Bug Analysis

To answer RQ2, we analyze the 34 confirmed TypeBugs, including their bug manifestations and root causes.

*4.3.1 Bug Manifestation.* We first analyze the data type pairs that appear in type-compatible databases. Then, we analyze the number of SQL operations appearing in the bug-revealing statements.

**Type-compatible databases**. We construct type-compatible databases using storage-compatible data types and populate them with the same data values. Among 34 TypeBugs, 24 TypeBugs occur in type-compatible databases where tables contain columns with different numeric types, 6 TypeBugs occur in type-compatible databases where tables contain columns with different string types, and 4 TypeBugs occur in type-compatible databases where tables contain columns with a numeric type and a string type.

**Number of SQL operations**. We count the number of SQL operations appearing in the reduced bug-revealing statements. Among the 34 confirmed TypeBugs, the bug-revealing statements include 1 to 5 SQL operations, with an average of 2.28 operations per statement. Over 88% of the bugs are detected using statements with 1 to 3 SQL operations (9 with 1, 12 with 2, and 9 with 3), while only 4 bugs require 4 or more operations. These results indicate that TypeCheck can effectively detect bugs even in relatively simple SELECT statements with few operations.

*4.3.2 Root Cause.* We further manually analyze the root causes of our confirmed 34 TypeBugs based on the following categories: (1) *Explicit Type Conversion:* the bug is triggered by an explicit type conversion operation (e.g., CAST or `::type`) that alters value semantics. (2) *Implicit Type Conversion:* the DBMS automatically converts between different data types during an operation, and

**Table 3: Distribution of root causes across target DBMSs**

| DBMS | Explicit Type Conversion | Implicit Type Conversion | Specific Data Computation |
|---|---|---|---|
| MySQL | 3 | 5 | 0 |
| PostgreSQL | 0 | 1 | 0 |
| MariaDB | 4 | 6 | 1 |
| CockroachDB | 0 | 1 | 0 |
| TiDB | 1 | 4 | 1 |
| OceanBase | 0 | 7 | 0 |
| **Total** | **8** | **24** | **2** |

this implicit promotion introduces an error. (3) *Specific Data Computation:* no type conversion occurs; the bug lies purely in the computation function implementation itself.

Table 3 shows the distribution of root causes across different DBMSs. Overall, 8 bugs (23.53%) are caused by incorrect explicit data type conversions, 24 bugs (70.59%) are caused by incorrect implicit data type conversions, and 2 bugs (5.88%) are caused by incorrect specific data computation.

We present the analysis of three representative bugs as follows.

```
1. -- Database db1
2. CREATE TABLE table1 (c1 VARCHAR(20) UNIQUE);
3. INSERT INTO table1 VALUES (-7222412721392380250);
4. SELECT (CAST(c1 AS FLOAT) - (MIN(CAST(c1 AS FLOAT)))) FROM
      table1 GROUP BY c1; -- {2606907088896} ✘

5. -- Database db2
6. CREATE TABLE table1 (c1 MEDIUMTEXT UNIQUE);
7. INSERT INTO table1 VALUES (-7222412721392380250);
8. SELECT (CAST(c1 AS FLOAT) - (MIN(CAST(c1 AS FLOAT)))) FROM
      table1 GROUP BY c1; -- {0} ✔
```

**Listing 3: MDEV-38158. Explicit type conversion error in `CAST` to FLOAT with different text types.**

**Explicit type conversion error in `CAST` operation to FLOAT.** Listing 3 shows a bug MDEV#38158, where explicit `CAST` operations to `FLOAT` yield inconsistent results when converting large integers from different text types. In database $db_1$, a table is created with a `VARCHAR(20)` column containing a large integer value. When executing `SELECT (CAST(c1 AS FLOAT) - (MIN(CAST(c1 AS FLOAT))))`, the result is {2606907088896}, which is incorrect. In database $db_2$, the same table is created with a `MEDIUMTEXT` column. With the same data value, the same query returns the correct result {0}. This inconsistency reveals that MariaDB performs incorrect explicit type conversions when casting large integer values from `VARCHAR` to `FLOAT`, leading to precision loss and incorrect results.

```
1. -- Database db1
2. CREATE TABLE table1 (c1 INT2, c2 INT4);
3. INSERT INTO table1 VALUES (-9551, -1347373370);
4. SELECT least(c1,c2) FROM table1; -- {-18746} ✘

5. -- Database db2
6. CREATE TABLE table1 (c1 INT4, c2 INT8);
7. INSERT INTO table1 VALUES (-9551, -1347373370);
8. SELECT least(c1,c2) FROM table1; -- {-1347373370} ✔
```

**Listing 4: CRDB-159153. Implicit type conversion error in `least()` function with different integer types.**

**Implicit type conversion error in `least()` function.** Listing 4 shows a bug CRDB#159153, in which the `least()` function in CockroachDB returns incorrect results when comparing values across different integer types. In the first database $db_1$, a table is created with columns of types `INT2` and `INT4`. When executing `SELECT least(c1,c2)`, the result is {−18746}, which is incorrect. In the second database $db_2$, the same table is created with columns of types `INT4` and `INT8`. With the same data values, the `least()` function returns the correct result {−1347373370}. This inconsistency reveals that CockroachDB performs incorrect implicit type conversions when comparing `INT2` with `INT4`, leading to an overflow or truncation error during the comparison.

**Table 4: Comparison with existing approaches**

| DBMS | TypeCheck | NoREC | Radar | DQP |
|---|---|---|---|---|
| MySQL | 8 | 0 | 1 | 0 |
| PostgreSQL | 1 | 1 | 1 | 1 |
| MariaDB | 11 | 0 | 1 | 2 |
| CockroachDB | 1 | 0 | 0 | 0 |
| TiDB | 6 | 0 | 2 | 0 |
| OceanBase | 7 | 1 | 1 | 1 |
| **Total** | **34** | **2** | **6** | **4** |

```
1. -- Database db1
2. CREATE TABLE table1 (c1 INT);
3. INSERT INTO table1 VALUES (1);
4. SELECT (- (c1)) * ((c1) DIV (c1)) FROM table1; -- {-1} ✔

5. -- Database db2
6. CREATE TABLE table1 (c1 INT UNSIGNED);
7. INSERT INTO table1 VALUES (1);
8. SELECT (- (c1)) * ((c1) DIV (c1)) FROM table1; --
      {18446744073709551615} ✘
```

**Listing 5: TiDB-63109. Incorrect arithmetic computation with signed vs. unsigned integers.**

**Incorrect arithmetic computation with signed vs. unsigned integers.** Listing 5 shows a bug TiDB#63109, where arithmetic operations produce incorrect results due to improper handling of negative values in unsigned integer contexts. In database $db_1$, a table is created with a signed `INT` column, and the query `SELECT (- (c1)) * ((c1) DIV (c1))` returns the correct result {−1}. However, in database $db_2$, when the same table is created with an unsigned `INT UNSIGNED` column and the same data value, the identical query returns an incorrect result {18446744073709551615}, which is the two's complement representation of −1 interpreted as an unsigned value. This inconsistency reveals that TiDB incorrectly handles negation operations on unsigned integers, causing overflow and producing incorrect arithmetic results.

## 4.4 Comparison with Existing Approaches

We answer RQ3 by comparing TypeCheck with three state-of-the-art approaches, namely NoREC [32], Radar [37], and DQP [12], since they target logic bugs in `SELECT` statements and have thoroughly tested our target DBMSs. As crashes are explicit errors, we assume these approaches can also detect the two crash bugs. As shown in Table 4, they detect at most 6 of our 34 confirmed TypeBugs, demonstrating the effectiveness of TypeCheck.

**NoREC** transforms an optimizable `SELECT` statement into a hard-to-optimize form and compares row counts. Given a bug report containing the database *db* and the `SELECT` statement *stmt*, we apply NoREC's transformation to *stmt* and execute both the original

and transformed statements on *db*. If the counts differ on either database, NoREC can detect the bug. As a result, NoREC detects 2 out of the 34 confirmed TypeBugs, of which both are crash bugs. TypeBugs typically produce the same row count.

**Radar** removes data constraints and indexes from a database to disable metadata-related query optimization and observes whether query results change. Given a bug report containing the database *db* and the `SELECT` statement *stmt*, we check whether *db* contains data constraints or indexes. If so, we remove them to create a modified database *db′*, and then execute *stmt* on both *db* and *db′*. If the query results differ, Radar can detect the bug. As a result, Radar detects 6 out of the 34 confirmed TypeBugs. TypeBugs typically persist regardless of whether constraints or indexes are present.

**DQP** enumerates possible execution plans for a given `SELECT` statement by applying query hints or setting system variables that affect the query optimizer. Given a bug report containing the database *db* and the `SELECT` statement *stmt*, we generate different execution plans for *stmt* on *db* by applying various query hints or changing system variables. If some of these execution plans produce different query results, DQP can detect the bug. As a result, DQP detects 4 out of the 34 confirmed TypeBugs. TypeBugs typically persist across some execution plans.

Unlike optimization bugs targeted by these approaches, TypeBugs stem from flaws in data type handling logic that consistently produce incorrect values across some execution plans.

### 4.5 Effectiveness of Components

**Storage-compatible Data Types.** The efficiency and effectiveness of this component depend on the number of data types in the target DBMS. In target DBMSs, the number of data types ranges from 35 to 56, yielding 68 to 171 compatible type pairs. The extraction process is efficient, completing in 0.06 to 0.6 minutes. OceanBase and TiDB reuse the same set of compatible type pairs as MySQL.

**Operation-oriented Type Compatibility Rules.** The efficiency and effectiveness of this component depend on the number of data types and functions in the target DBMS. Across our target DBMSs, the number of functions ranges from 51 to 91, yielding 409 to 1,687 rules. The extraction process takes approximately 1.7 to 7 hours. OceanBase and TiDB reuse the same set of operation-oriented type compatibility rules as MySQL.

**TypeBug Detection.** We ran the experiment three times, each lasting 24 hours, measuring throughput, time-to-first-bug, queries-to-first-bug, and average cost per bug. On average per run, `TypeCheck` tested 9,761 databases, generated 9,761 test reports, and detected 2,588 bugs corresponding to 17 unique TypeBugs. `TypeCheck` generates and executes approximately 3,500 queries per second, with the first bug typically found within 3 seconds (median 2 seconds) after executing 5,926 queries (median 3,667). The average cost per bug is approximately 0.55 minutes.

## 5 LIMITATIONS

While `TypeCheck` demonstrates effectiveness in detecting TypeBugs, it has the following limitations. (1) `TypeCheck` cannot detect bugs when both type-compatible databases return identical incorrect results. (2) `TypeCheck` does not support non-deterministic SQL functions such as `RANDOM()`, `NOW()`, and `CURRENT_DATE()`, which produce different results across executions. (3) `TypeCheck`'s applicability depends on the availability of storage-compatible data types and type compatibility rules in the target DBMS. For example, SQLite supports only five distinct data types, which constrains `TypeCheck`'s testing capabilities. (4) `TypeCheck` does not support data types that lack storage-compatible ones, e.g., `FLOAT` and `DOUBLE`. Detecting bugs involving such data types requires accurately modeling the semantics of the types and operations.

## 6 RELATED WORK

**Grammar-model-based SQL statement generation**. Grammar-model-based SQL statement generation approaches [6, 8, 26, 42, 51, 52] define a grammar model for SQL statements and randomly generate SQL statements by instantiating grammar rules. For example, SQLsmith [8] takes the built-in grammar model as input to generate SQL statements. `TypeCheck` can integrate with these approaches to generate databases and queries.

**Mutation-based SQL statement generation**. Mutation-based SQL statement generation approaches generate SQL statements by mutating existing ones using intermediate representations [18, 19, 21, 28, 50]. For example, Squirrel [50] and SQLRight [28] both take existing SQL statements as input to generate new SQL statements through various mutation strategies. `TypeCheck` can integrate with these approaches to generate SQL statements.

**Logic bugs in SQL statements**. Logic bugs in SQL statements refer to bugs that cause a DBMS to return incorrect results. Researchers have proposed various approaches to detect logic bugs in DBMSs [11, 12, 20, 22–25, 27, 31–33, 37, 38, 40, 43, 47, 48]. For example, Mozi [27] creates databases with varying configurations and observes different execution results for the same `SELECT` statements among these database variants. SQLT [29] empirically shows that RDBMS bugs rooted in SQL data types remain a first-class testing challenge, yet existing approaches ignore data-type semantics. In contrast, `TypeCheck` specifically focuses on TypeBugs by exploiting type-compatible databases and queries.

## 7 CONCLUSION

Data storage and computation in DBMSs are closely linked to data types. Incorrect handling of data type semantics and computations can lead to data-type-related logic bugs (TypeBugs). This paper presents the first data-type-related logic bug detection approach, `TypeCheck`. `TypeCheck` first constructs type-compatible databases with different data types, and then executes identical type-compatible `SELECT` statements to reveal their execution results' inconsistencies. Our evaluation on six relational DBMSs reveals 37 TypeBugs, with at most 6 being detectable by existing approaches, highlighting the effectiveness of `TypeCheck`.

## ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (62302493), Special Project of the Key Lab of System Software at Chinese Academy of Sciences (ZD202604), Basic Research Project of the Institute of Software Chinese Academy of Sciences (ISCAS-JCZD-202403), and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044).

# REFERENCES


[1] 2025. DB-Engines Ranking. https://db-engines.com/en/ranking.
[2] 2025. MariaDB Homepage. https://mariadb.org/.
[3] 2025. MySQL Customers by Industry. https://www.mysql.com/customers/.
[4] 2025. MySQL Homepage. https://www.mysql.com.
[5] 2025. PostgreSQL Homepage. https://www.postgresql.org/.
[6] 2025. SQLancer Homepage. https://github.com/sqlancer/sqlancer.
[7] 2025. SQLite Homepage. https://www.sqlite.org/index.html.
[8] 2025. SQLsmith. https://github.com/anse1/sqlsmith.
[9] 2025. TiDB Homepage. https://www.pingcap.com/?from=en.
[10] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of International Conference on Software Engineering (ICSE)*. 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174
[11] 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. https://doi.org/10.1145/3597503.3639076
[12] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proceedings of the ACM on Management of Data (SIGMOD)* (jun 2024). https://doi.org/10.1145/3654991
[13] 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. https://doi.org/10.1145/800296.811515
[14] Edgar F Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (jun 1970), 377–387. https://doi.org/10.1145/362384.362685
[15] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially Testing Database Transactions for Fun and Profit. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Article 35, 12 pages. https://doi.org/10.1145/3551349.3556924
[16] Ziyu Cui, Wensheng Dou Dou, Yu Gao, Rui Yang, Yingying Zheng, Jiansen Song, Yuan Feng, and Jun Wei. 2025. Simple Testing Can Expose Most Critical Transaction Bugs: Understanding and Detecting Write-Specific Serializability Violations in Database Systems. *Proceedings of the VLDB Endowment (PVLDB)* 18 (April 2025), 2547–2560. https://doi.org/10.14778/3742728.3742747
[17] 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. https://doi.org/10.1109/ICSE48619.2023.00101
[18] 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. https://doi.org/10.1145/3597503.3639210
[19] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Article 49, 12 pages. https://doi.org/10.1145/3551349.3560431
[20] 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.
[21] 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.
[22] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 821–835.
[23] 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. https://doi.org/10.14778/3357377.3357382
[24] Jinhui Lai, Chi Zhang, Bingyan Li, Chenglin Liang, Jie Liang, Zhiyong Wu, Jingzhou Fu, Yu Jiang, and Zichen Xu. 2025. SRS: Detecting Logic Bugs of Join Implementation in DBMSs via Set Relation Synthesis. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Vol. 3. Article 363, 24 pages. https://doi.org/10.1145/3769828
[25] Jinhui Lai, Chi Zhang, Jie Liang, Zihao Zeng, Zhiyong Wu, Jingzhou Fu, Chijin Zhou, Shuai Ma, Yu Jiang, and Zichen Xu. 2026. Finding Missed Optimizations in DBMSs through Unbalanced Short-circuit Query Construction. *Proceedings of the ACM on Management of Data (SIGMOD)* 4, 3, Article 184 (May 2026), 24 pages.
[26] 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. https://doi.org/10.1109/ICDE55515.2023.00057
[27] 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. https://doi.org/10.1145/3597503.3639112
[28] 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.
[29] Shuang Liu, Ruifeng Wang, Yuanfeng Xie, Junjie Chen, Wei Lu, Xiao Zhang, Quanqing Xu, Chuanhui Yang, and Xiaoyong Du. 2025. A Comprehensive Study of Bugs in Relational DBMS. *IEEE Transactions on Software Engineering* 51, 12 (2025), 3654–3668. https://doi.org/10.1109/TSE.2025.3625300
[30] 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. https://doi.org/10.1145/3510003.3510093
[31] Pak-Lok Poon, Man Fai Lau, Yuen Tak Yu, and Sau-Fun Tang. 2024. Spreadsheet quality assurance: a literature review. *Frontiers of Computer Science* 18, 2 (2024), 182203. https://doi.org/10.1007/s11704-023-2384-6
[32] 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. https://doi.org/10.1145/3368089.3409710
[33] 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. https://doi.org/10.1145/3428279
[34] 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.
[35] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 618–622.
[36] 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. https://doi.org/10.1109/ICSE48619.2023.00175
[37] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proceedings of the VLDB Endowment (PVLDB)* 17, 8 (may 2024), 1884–1897. https://doi.org/10.14778/3659437.3659445
[38] Jiansen Song, Wensheng Dou, Yingying Zheng, Yu Gao, Ziyu Cui, Wei Wang, and Jun Wei. 2025. Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction. *Proceedings of the VLDB Endowment (PVLDB)* 18, 7 (March 2025), 2281–2294. https://doi.org/10.14778/3734839.3734861
[39] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1493–1509. https://doi.org/10.1145/3318464.3386134
[40] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Vol. 1. Article 55, 26 pages. https://doi.org/10.1145/3588909
[41] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. Article 1, 6 pages. https://doi.org/10.1145/3209950.3209952
[42] 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. https://doi.org/10.1109/ICSE-SEIP52600.2021.00042
[43] Zhiyong Wu, Jie Liang, Jingzhou Fu, Wenqian Deng, and Yu Jiang. 2025. DDLUMOS: understanding and detecting atomic DDL bugs in DBMSs. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*. Article 78, 15 pages.
[44] Rui Yang, Ziyu Cui, Wensheng Dou, Yu Gao, Jiansen Song, Xudong Xie, and Jun Wei. 2025. Detecting Isolation Anomalies in Relational DBMSs. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 23. https://doi.org/10.1145/3728953
[45] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. 2023. OceanBase Paetica: A Hybrid Shared-nothing/Shared-everything Database for Supporting Single Machine and Distributed Cluster. *Proc. VLDB Endow.* 16, 12 (2023), 3728–3740. https://doi.org/10.14778/3611540.3611560
[46] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: A 707 Million tpmC Distributed Relational Database System. *Proc.*

*VLDB Endow.* 15, 12 (2022), 3385–3397. https://doi.org/10.14778/3554821.3554830

[47] Chi Zhang, Jie Liang, Zhiyong Wu, Dalong Shi, Linzhang Wang, and Yu Jiang. 2026. EPSC: Testing Database Management Systems via Equivalent Prepared Statement Construction. *Proceedings of the ACM on Management of Data (SIGMOD)* 4, 3, Article 176 (May 2026), 25 pages.

[48] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Vol. 3. Article 24, 24 pages. https://doi.org/10.1145/3709674

[49] 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. https://doi.org/10.1145/3533767.3534409

[50] 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. https://doi.org/10.1145/3372297.3417260

[51] Suyang Zhong and Manuel Rigger. 2026. Scaling Automated Database System Testing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1677–1692. https://doi.org/10.1145/3779212.3790215

[52] Hongtao Zhou, Yingying Zheng, Yu Gao, Jiansen Song, Xudong Xie, Rui Yang, Ziyu Cui, Wensheng Dou, and Jun Wei. 2026. Systematically Cover SQL Syntactic Structures via k-Sequence. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3832289