

# Semantic Conformance Testing of Relational DBMS

Shuang Liu  
Renmin University of China  
Beijing, China  
shuang.liu@ruc.edu.cn

Chenglin Tian  
Tianjin University  
Tianjin, China  
chunqiubf@gmail.com

Jun Sun  
Singapore Management University  
Singapore, Singapore  
junsun@smu.edu.sg

Ruifeng Wang  
Tianjin University  
Tianjin, China  
ruifeng\_wong@tju.edu.cn

Wei Lu  
Renmin University of China  
Beijing, China  
lu-wei@ruc.edu.cn

Yongxin Zhao  
East China Normal University  
Shanghai, China  
yxzhao@sei.ecnu.edu.cn

Yinxing Xue  
University of Science and Technology  
of China  
Suzhou, China  
yxxue@ustc.edu.cn

Junjie Wang  
Tianjin University  
Tianjin, China  
junjie.wang@tju.edu.cn

Xiaoyong Du  
Renmin University of China  
Beijing, China  
duyong@ruc.edu.cn

## ABSTRACT

The standard semantics of SQL are documented in SQL specifications such as ISO/IEC 9075, and the relational DBMSs' implementations should adhere to these documented semantics. However, no tools currently exist that can automatically check the alignment of SQL semantics implemented in RDBMSs with those specified in the SQL standards. There are two primary challenges in automating this conformance checking problem. First, the SQL specification, documented in natural language, tends to be ambiguous and is not directly executable. Second, it is difficult to generate test queries that thoroughly cover all aspects defined in the SQL specification, including descriptions of keywords and parameters. In this work, we introduce the first method for semantic conformance testing of RDBMSs to assess whether their behavior aligns with the intended SQL semantics. Our contributions are threefold. Firstly, We have formally defined the denotational semantics of SQL and implemented them in Prolog, creating an executable reference RDBMS for differential testing against existing systems. Secondly, We proposed three coverage criteria based on these formal semantics, along with a coverage-guided query generation algorithm that effectively generates queries achieving high semantic coverage and detecting new bugs. Lastly, We applied our approach to five widely-used and thoroughly tested RDBMSs, i.e., MySQL, TiDB, SQLite, DuckDB and PostgreSQL, uncovering 19 bugs and 11 inconsistencies, all of which were confirmed by RDBMS developers.

## PVLDB Reference Format:

Shuang Liu, Chenglin Tian, Jun Sun, Ruifeng Wang, Wei Lu, Yongxin Zhao, Yinxing Xue, Junjie Wang, and Xiaoyong Du. Semantic Conformance Testing of Relational DBMS. PVLDB, 14(1): XXX-XXX, 2020. doi:XX.XX/XXX.XX

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](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097. doi:XX.XX/XXX.XX

```
1 SELECT 3 > (-5 | -4);  
2 -- expected: 1 ✓, actual: 0 ✗
```

(a) A query triggering bug (ID 39259) in TiDB 6.6.0

```
1 SELECT 'Hello' || NULL;  
2 -- SQLite: NULL, PostgreSQL: 'Hello'
```

<binary concatenation> is an operator, ||, that returns a binary string by joining its binary string operands in the order given.  
---ISO/IEC 9075, Foundation, 4.3.3

(b) A query causing an inconsistency between SQLite version 3.39.0 and PostgreSQL version 16.2, and the SQL specification

**Figure 1: A bug and an inconsistency detected by our method**  
**PVLDB Artifact Availability:**

The source code, data, and/or other artifacts have been made available at URL\_TO\_YOUR\_ARTIFACTS.

## 1 INTRODUCTION

Relational Database Management Systems (RDBMSs) [3, 5, 6, 10, 11] are fundamental infrastructures for a wide range of application systems, such as web applications and embedded systems [1, 2, 4, 7]. Structured Query Language (SQL) is the standard programming language for relational databases, and its specification is formally documented in ISO/IEC 9075:2016 [9]. As the parser and executor of SQL queries, a RDBMS should conform to the SQL specification to ensure correct implementations of the semantics. As of 2023, there are more than 416 different implementations of relational databases, yet many of these implementations deviate from the specification [17, 40, 45]. In practice, large online systems, such as that of Alibaba, may integrate and adopt different RDBMSs (sometimes dynamically) for efficiency or comparability reasons, and such deviations in implementations may result in unexpected system behaviors. Bugs in RDBMSs, which may potentially lead to data integrity issues or even security vulnerabilities, are also reported due to violations of the SQL specifications [13].

As the golden standard for correct SQL behavior, the specification of SQL should be clearly described, and an RDBMS should

conform to the SQL specification. Inconsistencies between RDBMS implementations and the specification can lead to unexpected results. Figure 1 presents two motivating examples, one bug and one inconsistency that our approach detected. Figure 1a is a SQL query that triggers a bug in TiDB version 6.6.0<sup>1</sup>, which occurs when performing a bitwise operation on negative numbers, which are by default signed numbers. The root cause is that TiDB incorrectly represents the result of bitwise OR (`|`) on two signed 64-bit integer (`-5` and `-4`) as an unsigned 64-bit integer (18446744073709551615). Regarding this bug, the expected result of `-5|-4` is the signed integer `-1`, the binary representation of which is 64 bits of 1. However, as TiDB treats the result of bitwise OR as an unsigned integer, it returns the unsigned 64-bit integer 18446744073709551615, which is the decimal representation of the binary number of 64 bits 1. Therefore, `3>(-5|-4)` is evaluated to 0 (false) in TiDB. For other RDBMSs that we have tested, e.g., SQLite and PostgreSQL, the given query is correctly executed (`-5|-4` returns `-1` and `3>(-5|-4)` returns 1). The underlying reason for this inconsistency is that the bitwise operation on signed numbers in the SQL specification is under specified. Therefore, different RDBMSs have different implementations. To avoid problems caused by such inconsistencies between RDBMS implementations, we need a method to systematically identify such under-specification in SQL semantics.

Figure 1b shows an inconsistency between two RDBMS implementations, SQLite and PostgreSQL, when dealing with a NULL value. In SQLite, concatenating any string with NULL returns NULL by default. In contrast, PostgreSQL treats NULL as an empty string. Thus, concatenating it with string `Hello` results in `Hello`. In the SQL specification, there is only one sentence describing the binary concatenation operator (as shown in Figure 1b), and it fails to clearly specify how to process NULL, which is a specific data type in SQL representing unknown data. This inconsistency poses substantial challenges for users of database systems. When transitioning between databases and leveraging features with inconsistent implementations, users may encounter unexpected outcomes.

From the motivational examples, we observe that failing to respect the SQL specification or unclearly documented specifications can result in bugs or inconsistent implementations across different RDBMSs, potentially confusing users. Therefore, it is critical to test the conformance of RDBMS implementations with the SQL specification. However, existing approaches on testing RDBMSs either use different RDBMSs as the test oracle [40] or propose metamorphic relations [30, 34–36]. None of those approaches consider testing the conformance of RDBMS implementations with SQL specifications. Consequently, they are only scratching the surface in evaluating the correctness of RDBMS.

To address the issue of automatic conformance testing between SQL specifications and RDBMS implementations, two main challenges arise. Firstly, the SQL specification is written in natural language, which is not directly executable. Secondly, it is challenging to generate test queries that comprehensively cover all aspects defined in the SQL specification, including descriptions of keywords and parameters. This complexity makes comprehensive conformance testing a challenging task.

In this work, we propose the first automatic conformance testing approach for relational DBMSs. To address the first challenge, we develop a formal denotational semantics of SQL and implemented the formalized semantics in Prolog. This executable SQL semantics is then used as an oracle to detect inconsistencies between RDBMS implementations and the SQL specification. To overcome the second challenge, we propose three coverage criteria based on the defined semantics, which are then utilized to guide the test query generation process, ensuring comprehensive coverage of SQL specifications.

To evaluate the effectiveness of our approach, we conducted experiments on five popular and well-tested RDBMSs, successfully detecting 19 bugs—18 of which are reported for the first time—and 11 inconsistencies. Further examination revealed that 8 bugs and 2 inconsistencies are due to deviations in RDBMS implementations from the SQL specification, 11 bugs and 9 inconsistencies are attributed to missing or unclear descriptions in the SQL specification itself. Additionally, we evaluated the effectiveness of the proposed coverage criteria. The results indicate that all three coverage criteria contribute to generating more diverse test queries, which in turn help uncover more bugs and inconsistencies, and combining all three coverage criteria yields the most effective testing results. To summarize, we make the following contributions.

- We propose the first method, SEMCONT, on semantic conformance testing of RDBMS implementations with the SQL specification, for which we formalize the semantics of SQL and implement the formalized semantics in Prolog, enabling automatic conformance testing.
- We introduce three coverage criteria based on the formal semantics, which effectively guide test query generation.
- We evaluate SEMCONT on four popular and thoroughly-tested RDBMSs, and detected 19 bugs, 18 of which are reported for the first time, and 11 inconsistencies.
- We released our Prolog implementation at <https://github.com/DBMSTesting/sql-prolog-implement> to inspire further research in this area.

## 2 PRELIMINARY ON PROLOG

Prolog (Programming in logic) [23] is a logical programming language based on first-order predicate calculus that focuses on deductive reasoning. A Prolog program consists of three components, i.e., facts, rules, and queries. Facts and rules describe the axioms of a given domain, while queries represent propositions to be proven. In the context of data and relationships, facts and rules define the logic and relations of a given domain. Computations are then conducted by applying queries to these facts and rules. Similarly, when facts and rules are used to capture the laws governing state changes, queries represent the desired target state.

We use the example in Figure 2 to illustrate the basic components of Prolog. The code snippet in line 2 is a fact in Prolog, and it represents a table list containing table `t` with initialized data. Lines 4–11 show three rules for the `SELECT` keyword, corresponding to three types of inputs, i.e., NULL (line 4), constant values (lines 5–8), and lists of columns (lines 9–11). The `column_select` functions on lines 8 and 11 correspond to specific column selection operations. The first parameter denotes the columns that the user wishes to select. The second parameter contains table metadata, including the

<sup>1</sup>We have also detected this bug in MariaDB 10.9.4 and MySQL 8.0.29.

```

1 Facts:
2 Tables=[[t,[a,b],[1,4],[2,5],[3,8]]]
3 Rules:
4 select_clause((null),Tb,[]).
5 select_clause(X,Tb,Z) :-
6     isConstant(X),
7     add_X(X,Tb,T),
8     column_select(X,T,Z).
9 select_clause(X,Tb,Z) :-
10    list(X),
11    column_select(X,Tb,Z).
12 ...
13 from_clause(T,Z) :-
14    list(T),
15    table_select(T,Tables,Z).
16 ...
17 Queries:
18 from_clause(t,TableList).
19 select_clause(t.b,TableList,Filtered).
20 --Return Result: Filtered = [[4],[5],[8]].

```

**Figure 2: An example of implementing the semantics of SQL keywords SELECT and FROM using Prolog**

- (1)  $\langle queryexp \rangle ::= \{ \langle collection clause \rangle \mid \langle select clause \rangle \langle from clause \rangle [ \langle where clause \rangle [ \langle group by clause \rangle ] [ \langle having clause \rangle ] ] [ \langle order by clause \rangle ] \}$
- (2)  $\langle collection clause \rangle ::= \langle queryexp \rangle \langle cop \rangle \langle queryexp \rangle$
- (3)  $\langle cop \rangle ::= \text{UNION [ALL]} | \text{EXCEPT [ALL]} | \text{INTERSECT [ALL]}$
- (4)  $\langle from clause \rangle ::= \text{FROM} \langle tref \rangle [ , \langle tref \rangle \dots ]$
- (5)  $\langle tref \rangle ::= \langle tname \rangle \mid \langle joined table \rangle$
- (6)  $\langle joined table \rangle ::= \langle cross join \rangle \mid \langle qualified join \rangle \mid \langle natural join \rangle$
- (7)  $\langle cross join \rangle ::= \langle tname \rangle \text{CROSS JOIN} \langle tname \rangle$
- (8)  $\langle qualified join \rangle ::= \langle tname \rangle [ \text{INNER} | \text{LEFT} | \text{RIGHT} | \text{FULL} ] \text{JOIN} \langle tname \rangle \langle on clause \rangle$
- (9)  $\langle natural join \rangle ::= \langle tname \rangle \text{NATURAL JOIN} \langle tname \rangle$
- (10)  $\langle on clause \rangle ::= \text{ON} \langle bvexp \rangle$
- (11)  $\langle where clause \rangle ::= \text{WHERE} \langle bvexp \rangle$
- (12)  $\langle select clause \rangle ::= \text{SELECT} \langle sop \rangle [ \langle af \rangle \langle slist \rangle ]$
- (13)  $\langle slist \rangle ::= * [ \langle cname \rangle [ , \langle cname \rangle \dots ] ]$
- (14)  $\langle sop \rangle ::= \text{DISTINCT} | \text{ALL}$
- (15)  $\langle af \rangle ::= \text{MAX} | \text{MIN} | \text{SUM} | \text{COUNT} | \text{AVG}$
- (16)  $\langle group by clause \rangle ::= \text{GROUP BY} \langle cname \rangle$
- (17)  $\langle having clause \rangle ::= \text{HAVING} \langle bvexp \rangle$
- (18)  $\langle order by clause \rangle ::= \text{ORDER BY} \langle cname \rangle$
- (19)  $\langle bvexp \rangle ::= \langle logical expression \rangle \mid \langle is expression \rangle \mid \langle comparison expression \rangle \mid \langle between expression \rangle \mid \langle in expression \rangle \mid \langle exists expression \rangle \mid \langle null expression \rangle \mid \langle vexp \rangle \mid \text{true} \mid \text{false} \mid \text{null}$
- (20)  $\langle tname \rangle ::= \text{identifier}$
- (21)  $\langle cname \rangle ::= \text{identifier}$

**Figure 3: SQL syntax**

table name, column names, and the data items in the table. The third parameter is utilized to store the return values. Lines 13-15 present a rule for the FROM keyword, where the first parameter T represents the name of the target table, and the second parameter signifies the output produced by the FROM clause. This rule specifically addresses the case of a single table input. It returns the required table (line 15) based on the information provided in the facts. Lines 17-18 contain Prolog queries, which corresponds to the SQL query SELECT

**Table 1: Symbol notations**

| Symbol               | Description   |
|----------------------|---|
| $T$                  | A table instance                                    |
| $\alpha$             | A data record in a table                            |
| $\hat{\alpha}$       | Bags of data records                                |
| $\beta$              | An attribute in a table                             |
| $\hat{\beta}$        | Bags of attributes                                  |
| $\Theta$             | An expression                                       |
| $T.\beta_i$          | The $i_{th}$ attribute in T                         |
| $T.\hat{\beta}$      | All attributes in T                                 |
| $\pi_{\beta}(T)$     | Data projection on attribute $\beta$                |
| $\sigma_{\Theta}(T)$ | Data selection on expression $\Theta$               |
| $\xi_{\alpha}(T)$    | The multiplicity of data record $\alpha$ in table T |
| $\xi_{\beta}(T)$     | The multiplicity of attribute $\beta$ in table T    |
| $[a, \dots, a]_n$    | A list of element a with length n                   |
| $\bowtie$            | Data join operation                                 |
| $\times$             | Cartesian product operation                         |

t.b FROM t. The from\_clause rule (lines 13-15) is first triggered to return table t and then the select\_clause rule (lines 9-11) is activated to return the required column t.b. In Prolog, the answers to a query can be automatically computed based on rules and facts through a unification algorithm. As a result, the values in column b are returned and stored in the list Filtered (line 20).

We choose to implement the semantics using Prolog for three reasons. First, Prolog is intuitive and straightforward to implement. Second, it supports a variety of operations, such as list operations and queries, which are naturally suited for specifying tables and queries in SQL. Last and most importantly, Prolog is a declarative programming language, fundamentally different from the imperative programming languages like C, which are used to implement RDBMSs. This distinction makes our semantic implementation less likely to encounter the same errors present in existing RDBMSs.

### 3 FORMAL SEMANTICS OF SQL

Following the SQL specification, we present the SQL syntax in Figure 3. Due to space limitations, we only provide the list of syntax and the corresponding semantics regarding SQL keywords, and the full list of syntax and semantic definitions, including those expressions, can be found in our technical report [14].

We manually analyze the semantics of keywords in the SQL specification and defined the denotational semantics for 138 SQL keywords in the SQL specification. Although non-trivial manual effort is required to define the semantics, this is a one-off effort. Furthermore, since the semantics of SQL is mostly stable, the defined semantics can be easily maintained as well. We adopt the bag semantics of SQL according to the SQL specification, allowing duplicate elements in the result set. We also support the null semantic, which is considered as a special unknown value in SQL.

#### 3.1 Basic symbol definition

Table 1 lists the basic symbols used for semantic definitions in this work.  $T$ ,  $\alpha$ , and  $\beta$  are used to represent a table, a data record in the table, and an attribute of a table, respectively.  $\hat{\alpha}$  and  $\hat{\beta}$  represent

**Keyword operation** ( $C : \{L, OP\} \mapsto T$ )

**Join operation**

1.  $C[[\{T_1, T_2\}, \text{natural join}]] \triangleq \{\alpha_1 \circ \alpha_2 \mid \alpha_1 \in T_1, \alpha_2 \in T_2, \bar{\beta}_I = T_1.\bar{\beta} \cap T_2.\bar{\beta}\} \quad \alpha_1 \circ \alpha_2 \triangleq \begin{cases} \alpha_1 \bowtie \alpha_2; & (\bar{\beta}_I \neq \emptyset) \wedge (\pi_{\bar{\beta}_I}(\{\alpha_1\}) = \pi_{\bar{\beta}_I}(\{\alpha_2\})) \\ \text{skip}; & \text{otherwise} \end{cases}$
2.  $C[[\{T_1, T_2\}, \text{left join}]] \triangleq \{\alpha_1 \bullet \alpha_2 \mid \alpha_1 \in T_1, \alpha_2 \in T_2, \bar{\beta}_I = T_1.\bar{\beta} \cap T_2.\bar{\beta}\}$   
 $\alpha_1 \bullet \alpha_2 \triangleq \begin{cases} \alpha_1 \bowtie \alpha_2; & (\bar{\beta}_I \neq \emptyset) \wedge (\pi_{\bar{\beta}_I}(\{\alpha_1\}) = \pi_{\bar{\beta}_I}(\{\alpha_2\})) \\ \alpha_1 \bowtie [\text{null}, \dots, \text{null}]_{|\{\alpha_2\}.\bar{\beta}| - |\bar{\beta}_I|}; & (\bar{\beta}_I \neq \emptyset) \wedge (\pi_{\bar{\beta}_I}(\{\alpha_1\}) \neq \pi_{\bar{\beta}_I}(\{\alpha_2\})) \\ \text{skip}; & \text{otherwise} \end{cases}$
3.  $C[[\{T_1, T_2\}, \text{right join}]] \triangleq \{\alpha_1 \bullet \alpha_2 \mid \alpha_1 \in T_1, \alpha_2 \in T_2, \bar{\beta}_I = T_1.\bar{\beta} \cap T_2.\bar{\beta}\}$   
 $\alpha_1 \bullet \alpha_2 \triangleq \begin{cases} \alpha_1 \bowtie \alpha_2; & (\bar{\beta}_I \neq \emptyset) \wedge (\pi_{\bar{\beta}_I}(\{\alpha_1\}) = \pi_{\bar{\beta}_I}(\{\alpha_2\})) \\ [\text{null}, \dots, \text{null}]_{|\{\alpha_1\}.\bar{\beta}| - |\bar{\beta}_I|} \bowtie \alpha_2; & (\bar{\beta}_I \neq \emptyset) \wedge (\pi_{\bar{\beta}_I}(\{\alpha_1\}) \neq \pi_{\bar{\beta}_I}(\{\alpha_2\})) \\ \text{skip}; & \text{otherwise} \end{cases}$
4.  $C[[\{T_1, T_2\}, \text{cross join}]] \triangleq \{\alpha_1 \times \alpha_2 \mid \alpha_1 \in T_1, \alpha_2 \in T_2\}$     5.  $C[[\{T_1, T_2\}, \text{inner join}]] \triangleq C[[\{T_1, T_2\}, \text{cross join}]]$

**Collection operation**

6.  $C[[\{T_1, T_2\}, \text{union}]] \triangleq \{\alpha_{u(T_1, T_2)} \mid \alpha \in T_1\} \vee \{\alpha \in T_2\}$      $\alpha_{u(T_1, T_2)} = \begin{cases} \alpha; & (\alpha \in T_1) \wedge (\alpha \in T_2) \\ \text{skip}; & \text{otherwise} \end{cases}$      $\alpha_{e(T_1, T_2)} = \begin{cases} \alpha; & (\alpha \in T_1) \wedge (\alpha \notin T_2) \\ \text{skip}; & \text{otherwise} \end{cases}$      $\alpha_{ea(T_1, T_2)} = \begin{cases} \alpha; & \xi_\alpha(T_1) > \xi_\alpha(T_2) \\ \text{skip}; & \text{otherwise} \end{cases}$
7.  $C[[\{T_1, T_2\}, \text{intersect}]] \triangleq \{\alpha_{i(T_1, T_2)} \mid \xi_{\alpha_{i(T_1, T_2)}}(T) = 1\}$      $C[[\{T_1, T_2\}, \text{intersect all}]] \triangleq \{\alpha_{i(T_1, T_2)} \mid \xi_{\alpha_{i(T_1, T_2)}}(T) = \min(\xi_{\alpha_{i(T_1, T_2)}}(T_1), \xi_{\alpha_{i(T_1, T_2)}}(T_2))\}$
8.  $C[[\{T_1, T_2\}, \text{except}]] \triangleq \{\alpha_{e(T_1, T_2)} \mid \xi_{\alpha_{e(T_1, T_2)}}(T) = 1\}$      $C[[\{T_1, T_2\}, \text{except all}]] \triangleq \{\alpha_{ea(T_1, T_2)} \mid \xi_{\alpha_{ea(T_1, T_2)}}(T) = \max(0, \xi_{\alpha_{ea(T_1, T_2)}}(T_1) - \xi_{\alpha_{ea(T_1, T_2)}}(T_2))\}$

**Filter operation**

9.  $C[[\{T\}, \text{distinct}]] \triangleq \{\alpha \mid (\forall \alpha \in T, \xi_\alpha(T_1) = 1) \wedge (\forall \alpha \in T_1, \alpha \in T)\}$     10.  $C[[\{T\}, \text{all}]] \triangleq T$

**Aggregation operation**

11.  $C[[\{T\}, \text{max}]] \triangleq \{v \mid (v \in \pi_{T.\bar{\beta}}(T)) \wedge (\forall v_1 \in \pi_{T.\bar{\beta}}(T), \sigma_{(v_1 > v)}(\pi_{T.\bar{\beta}}(T)) = \emptyset)\}$
12.  $C[[\{T\}, \text{min}]] \triangleq \{v \mid (v \in \pi_{T.\bar{\beta}}(T)) \wedge (\forall v_1 \in \pi_{T.\bar{\beta}}(T), \sigma_{(v_1 < v)}(\pi_{T.\bar{\beta}}(T)) = \emptyset)\}$
13.  $C[[\{T\}, \text{sum}]] \triangleq \{v \mid v = \sum_{v_1 \in \pi_{T.\bar{\beta}}(T)} v_1\}$     14.  $C[[\{T\}, \text{count}]] \triangleq \{v \mid v = \sum_{\alpha \in \pi_{T.\bar{\beta}}(T)} (\xi_\alpha(T))\}$
15.  $C[[\{T\}, \text{avg}]] \triangleq \{v \mid v = \sum_{v_1 \in \pi_{T.\bar{\beta}}(T)} v_1 / \sum_{\alpha \in \pi_{T.\bar{\beta}}(T)} (\xi_\alpha(T))\}$

**Keyword operation** ( $\mathcal{H} : S \mapsto (A, T)$ )

**Single keyword operation**

16.  $\mathcal{H}[\text{from}(\langle tname \rangle)] \triangleq (A, A.\langle tname \rangle)$     17.  $\mathcal{H}[\text{from}(\langle tname_1 \rangle, \langle tname_2 \rangle)] \triangleq (A, C[[A.\langle tname_1 \rangle, A.\langle tname_2 \rangle], \text{cross join}]]$
18.  $\mathcal{H}[\text{select } *]] \triangleq (\{T\}, \pi_{T.\bar{\beta}}(T))$     19.  $\mathcal{H}[\text{select } \langle cname \rangle [ \langle cname \rangle \dots ]]] \triangleq (\{T\}, \pi_{\langle cname \rangle} [ \langle cname \rangle \dots ](T))$
20.  $\mathcal{H}[\text{select } \langle cname \rangle [ \langle cname \rangle \dots ]]] \triangleq (\{T\}, \pi_{\langle cname \rangle} [ \langle cname \rangle \dots ](T))$
21.  $\mathcal{H}[\text{where } \langle bvxexp \rangle]] \triangleq (\{T\}, \sigma_{\mathcal{B}(\langle bvxexp \rangle)}(T))$     22.  $\mathcal{H}[\text{having } \langle bvxexp \rangle]] \triangleq (\{T\}, \sigma_{\mathcal{B}(\langle bvxexp \rangle)}(T))$
23.  $\mathcal{H}[\text{group by } \langle cname \rangle]] \triangleq (\{T\}, (\widehat{\alpha}_1, \dots, \widehat{\alpha}_k) : \forall \widehat{\alpha}_p \in (\widehat{\alpha}_1, \dots, \widehat{\alpha}_k), \forall v_{ij} \in \pi_{\langle cname \rangle}(\widehat{\alpha}_p), (v_{ij} = v_{i1}))$
24.  $\mathcal{H}[\text{order by } \langle cname \rangle]] \triangleq (\{T\}, T_1) \text{ where } (\forall \alpha \in T_1, \xi_\alpha(T_1) = \xi_\alpha(T)) \wedge (\forall \alpha \in T, \xi_\alpha(T) = \xi_\alpha(T_1)) \wedge (\forall v_i, v_j \in \sigma_{T_1}(\langle cname \rangle)(T_1), i < j \text{ iff } v_i \leq v_j)$
25.  $\mathcal{H}[\text{subquery}]] \triangleq \mathcal{H}[\langle query \text{ expression} \rangle]]$

**Composite keyword operation**

26.  $\mathcal{H}[\langle queryexp \rangle]] =$   
 $\mathcal{H}[\text{select } [ \langle sop \rangle ] [ \langle af \rangle ] \langle cname_1 \rangle [ \langle cname_2 \rangle \dots ] \text{from } \langle tname_1 \rangle [ \langle tname_2 \rangle \dots ] \text{from } \langle tname_1 \rangle \text{ natural/cross join } \langle tname_2 \rangle$   
 $\text{from } \langle tname_1 \rangle \text{ left/right/full/inner join } \langle tname_2 \rangle \text{ on } \langle bvxexp \rangle [ \text{where } \langle bvxexp \rangle ] [ \text{group by } \langle cname \rangle ] [ \text{having } \langle bvxexp \rangle ]$   
 $[ \text{order by } \langle cname \rangle ]]] \triangleq$   
 $\mathcal{H}[\text{from}(\langle tname_1 \rangle)] \mid (\mathcal{H}[\text{from}(\langle tname_1 \rangle)], \mathcal{H}[\text{from}(\langle tname_2 \rangle)]) \mid C[[L, \text{natural/cross join}]] \mid (\mathcal{H}[\text{from}(\langle tname_1 \rangle)], \mathcal{H}[\text{from}(\langle tname_2 \rangle)])$   
 $\mid C[[L, \text{left/right/inner/full join}]] \mid \mathcal{H}[\text{on } \langle bvxexp \rangle]] \mid \mathcal{H}[\text{where } \langle bvxexp \rangle]] \mid \mathcal{H}[\text{group by } \langle cname \rangle]] \mid \mathcal{H}[\text{having } \langle bvxexp \rangle]]$   
 $\mid \mathcal{H}[\text{select } \langle cname_1 \rangle [ \langle cname_2 \rangle \dots ]]] \mid C[[\{T_1, \langle sop/af \rangle\}]] \mid \mathcal{H}[\text{order by } \langle cname \rangle]]$
27.  $\mathcal{H}[\langle queryexp_1 \rangle \langle cop \rangle \langle queryexp_2 \rangle]] \triangleq (\mathcal{H}[\langle queryexp_1 \rangle], \mathcal{H}[\langle queryexp_2 \rangle]) \mid C[[\{T_1, T_2\}, \langle cop \rangle]]$

**Figure 4: The full list of semantic definitions for SQL keywords**

bags of data and bags of attribute, respectively.  $\Theta$  represents expressions, including logical expressions, numeric expressions, constant values and function operations. The operations of data projection, data selection, and calculating the multiplicity of data records or attributes are denoted by  $\pi$ ,  $\sigma$ , and  $\xi$ , respectively. The operations of joining two data records and performing the Cartesian product are represented by  $\bowtie$  and  $\times$ , respectively. We use  $[a, \dots, a]_n$  to denote a list of  $n$  occurrences of  $a$  (which can be a data record or a constant).

### 3.2 Semantics of SQL keywords

The semantics of SQL keywords can be classified into two categories. The first category, including join operations, set operations, filtering operations, and aggregate functions, involves functionalities that directly operate on a list of tables and return a new table. The second category, encompassing the semantics of keywords such as FROM and SELECT, as well as their composite semantics, involves functionalities that operate on query expressions, and return a tuple  $(A, T)$ , where  $A$  is a set of tables and  $T$  is the resulting table.

**Definition 1 (Keyword operation ( $C : \{L, OP\} \mapsto T$ ))** Function  $C$ , which is a mapping from a list of table instances  $L$  and an operation type  $OP$  to a table  $T$ , defines the denotational semantics of SQL keywords. Fig 4 shows the semantic definition of four SQL keywords, i.e., join operations (semantic rules 1-5), set operations (semantic rules 6-8), filter operations (semantic rules 9-10), and aggregate operations (semantic rules 11-15).

**Definition 2 (Keyword operation ( $\mathcal{H} : S \mapsto (A, T)$ ))** The function  $\mathcal{H}$  is a mapping from the domain of SQL statements  $S$  to the domain of tuples  $(A, T)$ , where  $A$  represents the set of tables that are relevant or affected during the execution process and  $T$  is the result table. Fig 4 shows the semantics of the second category of SQL keywords (semantic rules 16-25), including keywords such as FROM, WHERE, ON, SELECT, GROUP BY, HAVING, ORDER BY, and their combinations in SQL statements.

**Definition 3 (Composite operation ( $\mathcal{H} : S \mapsto (A, T)$ ))** Based on the semantics of these keywords and functions, we define the composite semantics of SQL language to represent the semantics of a SQL statement. The input of the composite semantics is a SQL statement, and the output is a tuple  $(A, T)$ . As shown in Fig 4, we first introduced the combination rules of different types of keyword semantics, and then we introduce the composite semantics based on those rules (semantic rules 26-27).

## 4 CONFORMANCE TESTING

### 4.1 Overview of our approach

Figure 5 illustrates the overview of our conformance testing approach, which consists of four components. Initially, we define the formal semantics of SQL, as detailed in Section 3. Next, we implement the SQL formal semantics in Prolog, which serves as an oracle for conformance testing. The third component is dedicated to test case generation. Here, we enhance the syntax-guided generation method, SQLancer [12], with coverage-guided test case generation. This enhancement is based on three coverage criteria, i.e., keyword coverage, rule coverage, and composite rule coverage, which we proposed based on the formal semantics we defined. The final component is dedicated to query results comparison, wherein the query

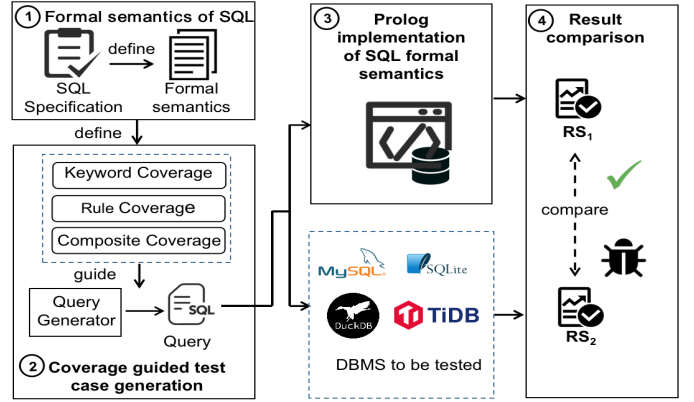


Figure 5: Overview of SEMCONT

results of the tested RDBMS and our Prolog implementation are compared to identify conformance issues.

**Conformance issues** in our work refer to bugs or inconsistencies. Both are due to violating the SQL semantics defined in the SQL specification, or unclear or missing descriptions in the SQL specification.

**Bugs** are defects that are confirmed by developers. **Inconsistencies** refer to inconsistent result produced by the tested RDBMS and SEMCONT. Inconsistencies are also confirmed by developers, yet they perceive them as deliberate design choices rather than bugs. We report these inconsistencies because various RDBMSs make differing design decisions, leading to varied query results that may potentially perplex users. This also underscores the importance of a comprehensively documented SQL specification.

### 4.2 Coverage guided test case generation

The state-of-the-art practice in test case generation involves randomly generating SQL queries guided by the syntax of SQL, among which SQLancer [12] stands out as one of the most effective tools of this kind. It is tailored to the syntactical structures of various RDBMSs. SQLancer considers database objects, such as tables, views, and indexes, as well as keywords and functionalities within query statements. Throughout the generation process, SQLancer maintains a set of keywords and functionalities, from which it randomly selects keywords to incorporate into the test cases, subject to syntactic rules of SQL (so that they remain syntactically valid). However, this generation process is entirely random and does not consider coverage of the SQL semantics. As a result, certain aspects of the semantics may never be tested.

To address this problem, we propose three coverage criteria, i.e., keyword coverage, rule coverage, and composite rule coverage, based on the formal semantics we defined. Each coverage criterion defines coverage at a different granularity level. Keyword coverage simply assesses whether each individual keyword in the SQL specification is covered. Rule coverage goes a step further by calculating whether each semantic rule, which accommodates different inputs for a SQL keyword, is covered. Composite rule coverage considers combinations of semantic rules that can form a valid SQL query.



**Table 2: Mutation Rules, with Colored ~~Deletion~~ and Addition**

| ID | Type           | Transformation                         | Example Query   |
|----|----------------|--|---|
| 01 | Keyword-level  | Replace operators                      | SELECT * FROM T WHERE T.a <del>AND</del> OR T.b   |
| 02 |                | Replace keywords                       | SELECT * FROM T <del>ORDER BY</del> GROUP BY T.a  |
| 03 |                | Add operators                          | SELECT * FROM T WHERE (T.a AND T.b) <del>IS NOT</del> TRUE                                      |
| 04 |                | Add keywords                           | SELECT * FROM T WHERE T.a = CAST(T.b as string) <del>ORDER BY</del> T.b ASC                     |
| 05 |                | Delete operators                       | SELECT * FROM T WHERE T.a <del>AND</del> T.b  |
| 06 |                | Delete keywords                        | SELECT * FROM T <del>WHERE EXP(T.a) &gt;=</del> T.b   |
| 07 | Rule-level     | Convert constants to column references | SELECT * FROM T WHERE T.a = MOD( <del>+</del> T.b,1)  |
| 08 |                | Convert parameter data types           | SELECT * FROM T WHERE T.a = POSITION( <del>+</del> 'a',1)                                       |
| 09 |                | Add parameters                         | SELECT * FROM T WHERE T.a = MOD(T.b,1) IS NOT TRUE <del>AND</del> ABS(1)                        |
| 10 |                | Delete parameters                      | SELECT * FROM T WHERE T.a = (T.b > FLOOR(T.c) <del>XOR CEILING(1.5)</del> )                     |
| 11 | Subquery-level | Replace subqueries                     | SELECT * FROM T <del>WHERE T.a = 1 AND T.b IS FALSE</del> WHERE EXISTS SELECT T.c WHERE T.b = 1 |
| 12 |                | Add subqueries                         | SELECT * FROM T <del>WHERE T.a IN (1,2) XOR T.b = exp(3)</del>                                  |
| 13 |                | Delete subqueries                      | SELECT * FROM T WHERE T.a > (T.b IS NOT UNKNOWN) <del>GROUP BY T.a HAVING LN(4)</del>           |

**Keyword coverage.** Formula (1) presents the formula for calculating keyword coverage, where  $N_{tk}$  represents the number of non-repeated keywords that are covered by the generated test queries and  $N_K$  the total number of keywords defined in our semantics.

$$Cov_k = \frac{N_{tk}}{N_K} \quad (1)$$

**Rule coverage.** The semantics of each SQL keyword implemented in Prolog is often encoded in multiple rules. Formula (2) shows the formula for measuring the percentage of rules covered by the generated test queries ( $N_{tr}$ ) in relation to the total number of rules ( $N_R$ ) defined in our semantics.  $N_{tr}$  is calculated by enumerating all keywords in the test suite and identifying the distinct semantic rules that are triggered.

$$Cov_r = \frac{N_{tr}}{N_R} \quad (2)$$

**Composite rule coverage.** We further introduce a more fine-grained coverage criterion named composite rule coverage, which takes into account the combination of semantic rules triggered by a SQL statement, offering a more comprehensive assessment of the test coverage for SQL statements. We calculate the composite rule coverage using Formula (3). The numerator ( $N_{tcr}$ ) represents the number of composite rules covered by the test queries, while the denominator ( $N_{CR}$ ) is the total number of all composite rules.

$$Cov_{cr} = \frac{N_{tcr}}{N_{CR}} \quad (3)$$

**Coverage-guided query generation.** Algorithm 1 describes the process of query generation guided by coverage. We set the initial coverage to be 0 and the declare covered set (*coveredSet*), which records the covered keywords, rules, or combined rules. We adopt SQLancer to randomly generate a large number of SQL statements (line 2), which serves as the seed pool of our query generation algorithm. The algorithm begins with randomly selecting an SQL statement from the seed pool (line 6), and mutates the query based on the mutation rules we proposed. Then we calculate coverage of the mutated query (line 8). We keep this mutation process (line 5-9) until the mutated query increases the overall coverage. Then the mutated query is added into the seed pool for future test case generation. The mutated query is executed to explore potential inconsistencies (line 11). The process terminates upon timeout.

**Algorithm 1: Coverage guided query generation**

```

1 Function SQLGeneration():
2   Initialize coverage=0, coveredSet
3   Initialize queryPool=GenerateQuery()
4   while TRUE do
5     while coverage does not increase do
6       queryInit = GetQueryFromPool()
7       query = MutateQuery(queryInit)
8       CalculateCoverage(query)
9     end
10    AddQueryIntoPool(query)
11    ExecuteQuery(query)
12    if timeout then
13      break
14    end
15  end
16 Function CalculateCoverage(query):
17   if query.pattern not in coveredSet then
18     UpdateCoverage(coverage)
19     ADD query.pattern To coveredSet
20   end

```

Function CalculateCoverage calculates the coverage of the given query. It first checks whether the given query's signature according to our definition of coverages (i.e., keyword, rule, or composite rule) is already recorded in the covered set. If it is not in the covered set, it indicates that this query increases the coverage. We then update the coverage (line 18) and add the pattern of this query to the covered set (line 19).

Table 2 lists the mutation rule examples on SQL statements we proposed. We categorize the mutation rules into three classes, i.e., keyword-level mutation rules, parameter-level mutation rules and subquery-level mutation rules. These mutation rules effectively enhance the keyword coverage, rule coverage, and combination rule coverage. In particular, keyword-level mutation rules and subquery-level mutation rules improve keyword coverage, parameter-level mutation rules improve rule coverage, all three types of mutation rules used together improve composite rule coverage. To ensure the validity of the mutated queries, we employ the SQL parser JSQLParser [8] during the mutation process to verify the syntactic validity of each mutated statement and discard the ones with syntax errors. Semantic checks, including the table references, column references, and data types, are performed to ensure that the correctness of the mutated queries.

**Algorithm 2: Executing a SQL query**


---

**Input** : *sql*: the SQL query to be executed  
**Output** : *result*: the execution result of the query

```

1 ast = ParseSQL(sql)
2 Function ExecuteQuery(ast.root):
3   keywordList = Sort(ast.root.children)
4   foreach keyword in keywordList do
5     | result = ExecuteKeyword(keyword, result)
6   end
7 Function ExecuteKeyword(keywordNode, result):
8   foreach child in keywordNode.children do
9     | if child is query then
10    | | result = ExecuteQuery(child)
11    | end
12    | if child is leaf then
13    | | result = ExecuteRule(keywordNode, result)
14    | | return result
15    | end
16    | result = ExecuteKeyword(child, result)
17  end

```

---

**4.3 Prolog implementation of SQL semantics**

We have implemented the formal semantics of SQL defined in Figure 4 in Prolog. The semantics of each keyword are implemented as a set of rules, as illustrated in Figure 2. We then implement the compositional semantics in Algorithm 2, which outlines the process of executing a SQL query in Prolog. The input to this algorithm is a SQL query, and its output is the result of executing this SQL query. Algorithm 2 initially parses the SQL statement into an Abstract Syntax Tree (AST) (line 1). The ExecuteQuery function then traverses the tree from the root node, sorting the children of the root node according to the keyword execution order (line 3). Then the semantic rules of the keywords are executed in order with the ExecuteKeyword function (lines 4-6). If a subquery is encountered (lines 9-11), the ExecuteQuery function is recursively called to initiate the sorting procedure. In other cases, ExecuteKeyword recursively calls itself (line 16) until a leaf node is reached, invoking the corresponding keyword semantic rule execution (lines 12-14).

The sorting of keywords solves the critical issue of ensuring the correct execution order of the semantics for each keyword in the query. Note that the SQL specification does not explicitly indicate the execution order of all keywords in a query, yet we can imply the execution order based on the semantic of each individual keyword. We also check the implementation of current mainstream databases, including MySQL, PostgreSQL, TiDB, SQLite and DuckDB, and confirm that they enforce the same execution order of SQL keywords, which is consistent with our understanding of keyword execution order, i.e., JOIN, FROM, WHERE, GROUP BY, Aggregate functions, HAVING, SELECT, ORDER BY, based on their semantics.

In the SQL specification, there are a total of 47 keywords whose semantics are not explicitly described, among which 4 are duplicated, e.g., AND and &&, OR and ||, LCASE and LOWER, UCASE and UPPER. The remaining 43 keywords include 4 bitwise operators, 23 string functions, and 16 numeric functions. Taking bitwise operators as an example, the SQL specification (*Part 2 Foundation, Language Opportunities*) states: "The SQL standard is missing operators on binary data types (BINARY, VARBINARY, BLOB) that allow users to bitwise manipulate values." For these keywords, we referred

**Table 3: Statistics of the target RDBMSs**

| DBMS       | Popularity rank |              | LOC  | First release | Tested version |
|------------|-----------------|--------------|------|---------------|----------------|
|            | DB-engines      | GitHub stars |      |               |                |
| MySQL      | 2               | 5.0K         | 380K | 1995          | 8.0.29         |
| TiDB       | 118             | 23.1K        | 800K | 2017          | 6.6.0          |
| SQLite     | 9               | 1.5K         | 300K | 2000          | 3.39.0         |
| DuckDB     | -               | 0.5K         | 59K  | 2018          | 0.7.0          |
| PostgreSQL | 4               | 14.9K        | 125K | 1996          | 16.1           |

to the implementation documentation of mainstream database management systems in our Prolog implementations. In cases where there were inconsistencies among different database implementations, we chose to adopt that used by the majority of databases.

Take the SQL query `SELECT * FROM T WHERE T.a=(SELECT 1 FROM T)` as an example. This query contains a subquery `SELECT 1 FROM T`. Initially, we parse this SQL statement into an AST and sort the three children nodes of the root node according to the keyword execution order of FROM, WHERE, SELECT. Taking the FROM keyword as an example, it has a single child node, which is the table name T. The rule for FROM that requires a table as input is then triggered for execution (lines 12-14). This information is subsequently relayed to the WHERE clause. During the execution of the WHERE clause, we encounter the subquery `SELECT 1 FROM T`, where we recursively call the ExecuteQuery function to process the subquery.

**4.4 Result comparison**

The final part of our method is result comparison, which entails comparing the query results from the tested RDBMS with those returned by SEMCONT. We first compare the number of records in the query results and identify an inconsistency if the numbers differ. If the numbers are identical, we proceed to compare the data records in the results. In particular, we scan both sets of query results and remove identical data pairs. An inconsistency is reported if either result set is not empty after removing all matching pairs.

For some of the SQL features, such as arithmetic operators, aggregate functions, and numerical functions, different RDBMSs may incur different implementation choices on floating point precision, which could result in false alarms in our result comparison step. To mitigate those false alarms, we impose restrictions on the return results of SQL statements that may involve floating-point outcomes during test case generation, and enforce the execution results to retain two decimal places. Meanwhile, we impose the same restrictions in our implementation of SQL semantics in Prolog to avoid potential false alarms in result comparison.

**5 EVALUATION**

We evaluate the effectiveness of SEMCONT. Specifically, our experiment is designed to answer the following four research questions.

**RQ1:** Can SEMCONT detect conformance issues, i.e., bugs and inconsistencies, in relational DBMS systems?

**RQ2:** Are the three coverage criteria effective in guiding the generation of queries for uncovering bugs and inconsistencies in RDBMSs?

**RQ3:** How does SEMCONT perform compared to state-of-the-art RDBMS testing approaches?

**RQ4:** Case studies of the bugs and inconsistencies detected by SEMCONT.

## 5.1 Experiment setup

We conducted all experiments on a server with two Intel(R) Xeon(R) Platinum 8260 CPUs at 2.30 GHz and 502 GB of memory, running Ubuntu 18.04.6 LTS. The SQL formal semantics were implemented in Prolog, while the SQL query generation program was developed in Java. We ran the experiments using Java version 11.0.15.1.

**Target RDBMS.** We selected four popular and widely used RDBMSs, each offering a range of distinct features and application scenarios, to demonstrate the effectiveness of our approach. The statistics of these RDBMSs, as obtained from their open-source repositories, are shown in Table 3. It is important to note that we used the latest release of each RDBMS, which has been extensively tested by existing approaches [34, 35]. MySQL [3] and PostgreSQL are the two most popular open-source database management systems. SQLite [6] and DuckDB [11] are both embedded DBMSs, running within the process of other applications. TiDB [10] is a distributed relational DBMS. DuckDB [11] is particularly designed for Online Analytical Processing (OLAP) workloads, and the other three DBMSs are aimed for Online Transaction Processing (OLTP) workloads.

**Compared baselines.** We compared SEMCONT with TLP [35] and NoREC [34], which are state-of-the-art metamorphic testing methods for testing RDBMS. NoREC constructs two semantically equivalent queries, one triggers the optimization and the other does not, executes the queries and compare the results. TLP, on the other hand, partitions the conditional expression of the original query into three segments, corresponding to the three possible results, i.e., TRUE, FALSE, and NULL, of the conditional expression. It then compares the union of the result sets from executing the three queries with the three segments each, with the result set of the original statement, expecting them to be identical. Both approaches have demonstrated effectiveness in RDBMS bug detection. Both NoREC and TLP are implemented in SQLancer [12] and SQLRight [30]. SQLancer adopts a generative approach for query generation and SQLRight adopts a mutation-based approach for generating queries. Therefore, in our experiment, we have four combined settings (concerning the oracle and query generation method) for the compared baselines, i.e., NoREC (SQLancer), NoREC (SQLRight), TLP (SQLancer) and TLP (SQLRight).

## 5.2 Experiment results

**RQ1: Bugs and inconsistencies.** We ran SEMCONT on four RDBMSs for a period of 3 months and reported the detected issues to the corresponding developer communities. Table 4 shows the details of the confirmed bugs and inconsistencies in four RDBMSs detected by SEMCONT. We have submitted 30 issues and 19 of them are confirmed by the developers as bugs. Among the confirmed bugs, 1 has been reported previously and 1 has been fixed. The remaining 11 issues are confirmed as inconsistencies, and the developers claim that they were their design choices. Among the 11 inconsistencies, 2 of them are due to violation of the SQL specification and 9 of them are due to unclear descriptions in the SQL specification. Developers of different RDBMSs could have different interpretations on the SQL specification, and thus design and implement their RDBMS differently. Among the nine inconsistencies arising from unclear standard descriptions, eight inconsistencies were detected in both MySQL (109147, 109148, 109836, 109845, 109962, 110711)

Table 4: Bugs and inconsistencies detected by SEMCONT

| SN | ID         | Target | Type          | Reason       | Status    |
|----|------------|--------|---------------|--------------|-----------|
| 1  | 109146     | MySQL  | Bug           | missing spec | duplicate |
| 2  | 109837     | MySQL  | Bug           | missing spec | confirmed |
| 3  | 109842     | MySQL  | Bug           | missing spec | confirmed |
| 4  | 109149     | MySQL  | Bug           | missing spec | confirmed |
| 5  | 110438     | MySQL  | Bug           | violate spec | confirmed |
| 6  | 109147     | MySQL  | Inconsistency | missing spec | confirmed |
| 7  | 109148     | MySQL  | Inconsistency | unclear spec | confirmed |
| 8  | 109836     | MySQL  | Inconsistency | missing spec | confirmed |
| 9  | 109845     | MySQL  | Inconsistency | missing spec | confirmed |
| 10 | 110439     | MySQL  | Inconsistency | violate spec | confirmed |
| 11 | 110346     | MySQL  | Inconsistency | violate spec | confirmed |
| 12 | 109962     | MySQL  | Inconsistency | missing spec | confirmed |
| 13 | 110711     | MySQL  | Inconsistency | missing spec | confirmed |
| 14 | 40996      | TiDB   | Bug           | missing spec | confirmed |
| 15 | 40995      | TiDB   | Bug           | missing spec | confirmed |
| 16 | 39260      | TiDB   | Bug           | missing spec | confirmed |
| 17 | 39259      | TiDB   | Bug           | missing spec | confirmed |
| 18 | 39258      | TiDB   | Bug           | unclear spec | confirmed |
| 19 | 42375      | TiDB   | Bug           | violate spec | confirmed |
| 20 | 42376      | TiDB   | Bug           | violate spec | confirmed |
| 21 | 42378      | TiDB   | Bug           | violate spec | confirmed |
| 22 | 42379      | TiDB   | Bug           | violate spec | confirmed |
| 23 | 42377      | TiDB   | Bug           | violate spec | confirmed |
| 24 | 42773      | TiDB   | Inconsistency | missing spec | confirmed |
| 25 | 40995      | TiDB   | Inconsistency | missing spec | confirmed |
| 26 | 7e03a4420a | SQLite | Bug           | missing spec | confirmed |
| 27 | 3f085531bf | SQLite | Bug           | missing spec | confirmed |
| 28 | 6e4d3e389e | SQLite | Bug           | violate spec | confirmed |
| 29 | 411bce39d0 | SQLite | Inconsistency | missing spec | confirmed |
| 30 | 6804       | DuckDB | Bug           | violate spec | fixed     |

and TiDB (42773, 40995). The queries triggering the inconsistencies have the same results when executed in MySQL, TiDB, and MariaDB databases, and are different from that of PostgreSQL. It is noteworthy that *all four databases have been extensively tested by existing methods [34–36], yet SEMCONT is still able to detect bugs that were not detected by those approaches.*

By a careful inspection on the detected bugs, we find that most of them indeed violate the SQL specification. One of the most representative bugs is related to the mishandling of NULL operands in keyword operations. According to the SQL specification, “*If the value of one or more, <string value expression>s, <datetime value expression>s, <interval value expression>s, and <collection value expression>s that are simply contained in a <numeric value function> is the NULL value, then the result of the <numeric value function> is the NULL value*” [27]. However, MySQL violates the specification by returning non-NULL results when operating on NULL values. One bug in MySQL (110438), three bugs in TiDB (42375, 42377, 42378) and one bug in DuckDB (6804) belong to this category. Another representative bug is due to incorrect implicit type conversion on string. When a string is converted to a signed integer type, it is mistakenly converted to a float type. There is no specific description in SQL specification on such cases. We will provide detailed analysis in the case study of section 5.2 of this type of bugs. Three bugs in MySQL (109149, 109837, 109842) and three bugs in TiDB (39260, 40995, 40996) belong to this category.

We also identified two bug in TiDB (39258, 39259) which erroneously handles bitwise operations on negative numbers and in operations on string, one bug in MySQL (109146) which is related to improper handling of newline characters by bitwise operators, and three bugs in SQLite (7e03a4420a, 3f085531bf, 6e4d3e389e) concerning the handling of data anomalies. These bugs are specifically related to the improper handling of large numbers or numbers



expressed in scientific notations. The SQL specification does not provide detailed description on those particular cases. The remaining two bugs are about column references on TiDB (42376, 42379). When RIGHT JOIN is used together with the FIELD or CONCAT\_WS keywords, if the parameters of the function contain references to a certain column, the result set will miss some data records.

The inconsistencies we identified can be categorized into two types: (1) inconsistencies violating the SQL specification, and (2) inconsistencies arise from different RDBMSs implementations due to missing or unclear descriptions in the SQL specification. One inconsistency in MySQL (109962) and one in TiDB (42773) are due to the representation of integer 0 in certain numerical functions, where the result of the integer 0 is represented as -0 because of incorrect implicit type conversion. Two inconsistencies in MySQL (110439, 110711) involve anomalies in the results returned by string functions when handling NULL parameters. This inconsistency aligns with some previously identified bugs and contradicts the semantic descriptions of NULL in the SQL specification [27, 28]. We detected three inconsistencies in MySQL (109836, 109845) and TiDB (40995), where the precision retained in floating-point results do not align with the specification. Among them, MySQL (109845) and TiDB (40995) do not obey the floating point precision specified by the parameter when using the round function to process integers. In MySQL (109836), performing arithmetic operations on string-type and numeric-type constants with the same numerical value does not yield consistent floating-point precision. Additionally, bitwise operators in both databases do not consistently return signed integer types when performing operations on negative numbers in MySQL (109147). The remaining inconsistencies relate to handling specific data types in MySQL (109148, 110346) or large numbers in SQLite (411bce39d0). The SQL specification misses descriptions on those operations, resulting in the inconsistencies.

**Answer to RQ1:** SEMCONT detects 19 bugs (18 newly reported) and 11 inconsistencies, which are all confirmed by developers, in four extensively tested RDBMSs. All detected bugs and inconsistencies are either due to RDBMS violating the SQL specification, or missing or unclear SQL specification. We did not detect any defects on PostgreSQL, which is consistent with the detection results of existing work [34].

**RQ2: Effectiveness of the coverage criteria.** We measure the effectiveness of the proposed coverage criteria in two aspects, i.e., whether they are effective in guiding generating test cases that achieve higher coverage, and whether they are effective in guiding generating test cases that uncover unknown bugs or inconsistencies. The experiment results on keyword coverage, rule coverage and composite rule coverage improvement are shown in Figure 6, Figure 7 and Figure 8, respectively.

In Figure 6, SQLancer+keyword syntax represents the setting of adding keywords and the corresponding generation rules which were not supported by SQLancer. SQLancer+keyword syntax greatly improved the keyword coverage for all four databases. Notably, within the first 1500 SQL statements, over 80% of the keywords were covered on all four databases, with SQLite achieving an impressive keyword coverage of 99%. Keyword coverage guided query generation (SQLancer+keyword coverage) further improves the

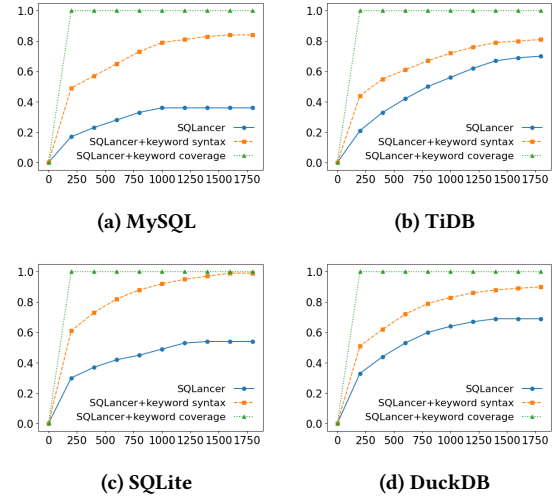


Figure 6: The keyword coverage increment (y-axis) with the number of queries (x-axis)

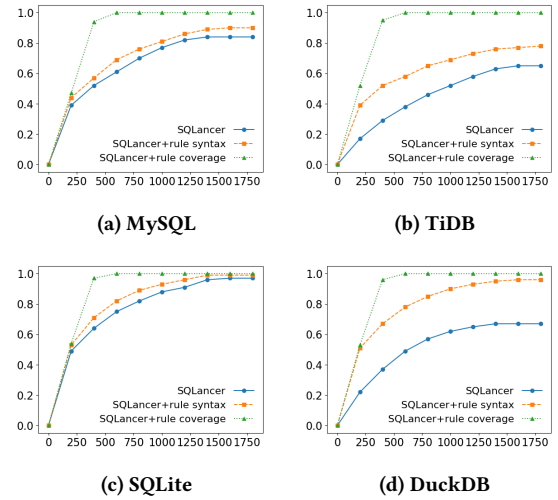


Figure 7: The rule coverage increment (y-axis) with the number of queries (x-axis).

keyword coverage, and achieved 100% keyword coverage within the first 200 generate queries for all databases, demonstrating the effectiveness of our keyword-guided query generation method.

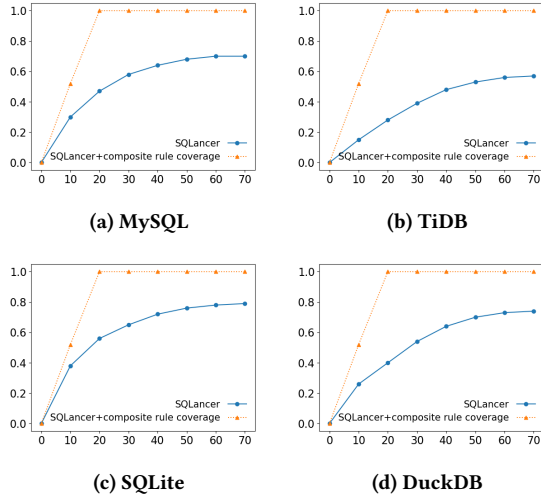
Figure 7 shows the results on rule coverage, which show similar trend with that on keyword coverage. Due to the limited support of SQL features, e.g., data types, by SQLancer, especially for DBMS such as DuckDB and TiDB, relying only on SQLancer achieves low rule coverage, as shown in Figure 7. Therefore, we add those missing features in SQLancer for the corresponding DBMS query generation and refer this as SQLancer + rule syntax. We can observe that adding those missing features improves the rule coverage, especially for DuckDB and TiDB. Rule coverage-guided query generation (SQLancer+rule coverage) achieves the highest rule coverage with the fewest number of queries. The results indicate the effectiveness of our rule coverage-guided query generation algorithm.

**Table 5: The bug and inconsistency numbers detected by SEMCONT with no coverage guided and coverage guided in 6h**

| DBMS   | SEMCONT |                 |       | SEMCONT+keyword coverage |                 |       | SEMCONT+rule coverage |                 |       | SEMCONT+composite rule coverage |                 |       |
|--------|---------|-----------------|-------|--------------------------|-----------------|-------|-----------------------|-----------------|-------|---------------------------------|-----------------|-------|
|        | Bugs    | Inconsistencies | Time  | Bugs                     | Inconsistencies | Time  | Bugs                  | Inconsistencies | Time  | Bugs                            | Inconsistencies | Time  |
| MySQL  | 3       | 3               | 12.42 | 4                        | 5               | 11.25 | 5                     | 6               | 5.75  | 5                               | 6               | 6.23  |
| TiDB   | 3       | 1               | 13.33 | 6                        | 2               | 12.17 | 5                     | 2               | 7.46  | 7                               | 2               | 8.62  |
| SQLite | 1       | 1               | 20.50 | 2                        | 1               | 17.83 | 3                     | 1               | 14.33 | 3                               | 1               | 13.05 |
| DuckDB | 1       | 0               | 29.37 | 1                        | 0               | 27.32 | 1                     | 0               | 31.68 | 1                               | 0               | 26.29 |

**Table 6: The bugs and inconsistencies detected by SQLancer, SQLRight and SEMCONT in 6h**

| DBMS   | TLP (SQLancer) |                 | NoREC (SQLancer) |                 | TLP (SQLRight) |                 | NoREC (SQLRight) |                 | SEMCONT |                 |
|--------|----------------|-----------------|------------------|-----------------|----------------|-----------------|------------------|-----------------|---------|-----------------|
|        | Bugs           | Inconsistencies | Bugs             | Inconsistencies | Bugs           | Inconsistencies | Bugs             | Inconsistencies | Bugs    | Inconsistencies |
| MySQL  | 1              | 0               | -                | -               | 1              | 0               | 0                | 0               | 5       | 6               |
| TiDB   | 2              | 0               | -                | -               | -              | -               | -                | -               | 7       | 2               |
| SQLite | 0              | 1               | 0                | 0               | 0              | 0               | 0                | 0               | 3       | 1               |
| DuckDB | 1              | 0               | 0                | 0               | -              | -               | -                | -               | 1       | 0               |



**Figure 8: The composite rule coverage increment (y-axis) with the number of queries (in million).**

Figure 8 depicts the improvements in composite rule coverage by the generated queries for the four databases. With SQLancer, which conducts random query generation, we observed that the increase in composite rule coverage tends to plateau after generating 60 million data points. At this stage, MySQL, SQLite, and DuckDB each achieved a composite rule coverage of around 70% and TiDB 50%. With the introduction of composite rule coverage guidance, all four databases were able to reach 100% composite rule coverage after generating 20 million queries (our Prolog implementation has a total of 19 million composite rules). The results indicate the effectiveness of our composite rule coverage-guided query generation algorithm.

To verify the effectiveness of the coverage-guided query generation algorithm in assisting detecting bugs and inconsistencies in relational DBMS, we conducted an ablation study of SEMCONT with and without coverage guidance. Table 5] presents the experimental results obtained from testing four databases over a period of 6 hours. We record the number of bugs and inconsistencies detected on the four settings, i.e., SEMCONT without coverage guidance, and SEMCONT with three coverage guidance. We also report the time

taken to discover the first bug or inconsistency. Note that to conduct fair comparisons, we improved SQLancer by incorporating all keywords supported by our semantics and related generation rules in SEMCONT. The experimental results indicate that within a 6-hour timeframe, all three coverage metrics successfully assist detecting more bugs and inconsistencies compared with random generation. Composite rule coverage is the most effective among all three coverage metrics. In terms of the time taken to detect the first bug or inconsistency, all three coverage guidance algorithm are faster than SEMCONT with random query generation. In particular, keyword coverage, rule coverage and composite rule coverage are 5.22%, 24.19% and 21.41% faster than random query generation.

**Answer to RQ2:** The three coverage criteria all improve the query generating process, triggering more bugs and inconsistencies with faster speed. Composite rule coverage achieves the most significant improvement.

**RQ3: Comparison with baseline approaches.** We compare SEMCONT with two state-of-the-art approaches TLP [34] and NoREC [35], which are metamorphic testing approaches for relational DBMS. For both approaches, we adopt SQLancer [12] and SQLRight [30] for query generation. Notably, SQLancer does not support the NoREC oracle for MySQL and TiDB, while SQLRight does not support TiDB and DuckDB. Therefore, we excluded these specific scenarios from our experiments. We ran the compared tools for a period of 6 hours and report the results in Table 6.

The experimental results show that both SQLancer and SQLRight using the NoREC as the oracle were unable to detect new bugs or inconsistencies. The TLP oracle with SQLancer for query generation detected 4 bugs in three databases, and with SQLRight for query generation detected 1 bug in MySQL. SEMCONT outperformed the compared approaches and detected 16 bugs and 9 inconsistencies in the four databases. The reason is that existing approaches do not consider the SQL specification and thus fail to find bugs that violated the SQL specification. For instance, One bug (109842) we detected in MySQL is related to the MOD function. When applied to negative numbers, MySQL incorrectly represents the result as -0. Both TLP and NoREC failed to detect this bug, even after successfully generated the bug triggering query.

```

1 SELECT MOD(' -12', -4);
2 --expected: 0 ✓, actual: -0 ✗

```

**Figure 9: A bug in MySQL 8.0.29**

```

1 CREATE TABLE IF NOT EXISTS t0(c0 LONGTEXT STORAGE DISK COMMENT
  'asdf' COLUMN_FORMAT FIXED);
2 CREATE TABLE IF NOT EXISTS t1 LIKE t0;
3 CREATE TABLE IF NOT EXISTS t2(c0 DECIMAL ZEROFILL COMMENT 'asdf'
  COLUMN_FORMAT FIXED PRIMARY KEY UNIQUE STORAGE DISK);
4 INSERT INTO t0(c0) VALUES(NULL);
5 INSERT INTO t1(c0) VALUES('hhhh');
6 REPLACE INTO t2(c0) VALUES(960364164);
7 SELECT t1.c0, t2.c0 FROM t1, t2 RIGHT OUTER JOIN t0 ON 0 WHERE
  (NOT ((t2.c0 IS FALSE) != ((t1.c0))));
8 -- expected:[ 'hhhh', NULL, NULL], actual:[]

```

**Figure 10: The queries triggering an inconsistency in MySQL 8.0.29 with the SQL specification**

**Answer to RQ3:** SEMCONT detected more bugs and inconsistencies than the compared baseline approaches. Baseline approaches fail to detect those bugs since they do not refer to SQL specification in their testing process.

**RQ4: Case study.** SEMCONT has identified 19 bugs and 11 inconsistencies, which arise from two reasons, i.e., (1) DBMS implementations are not consistent with SQL specification and (2) unclear or missing description in the SQL specification. These problems have resulted in variations in the specific implementations across different RDBMSs, consequently affecting the user experience.

**A bug due to missing specifications.** Figure 9 is a bug we detected in MySQL 8.0.29. The query conducts MOD function with the string type as the first parameter. The expected result of the query is 0, yet MySQL returned -0. MySQL developers confirmed this bug and explained the reason is that when the first parameter of MOD is a string type, an implicit type conversion should be triggered to convert the string type ' -12' to a signed integer -12. However, MySQL mistakenly converted ' -12' to a float type -12.0, causing this bug. SQL specification does not specify how to convert a string type to a numeric type, and thus different RDBMSs may make on their own implementation choices.

**An inconsistency violating the SQL specification.** Figure 10 shows the queries that cause an inconsistency in MySQL 8.0.29 that violates the SQL specification. The first three SQL queries create three tables t0, t1 and t2. Then t0 and t1 are inserted values NULL and string 'hhhh' (lines 4, 5), respectively. Line 6 replaces the value in t2 with value 960364164. The query in line 7 returns an empty list in MySQL 8.0.29, which violates the SQL specification [28].

The select query in line 7 involves a right outer join between t2 and t0 on condition 0, meaning false in this context, and thus no matching columns are returned from the two tables. Therefore, the resulting table will retain all the data from the right table (t0) and replace all data from the left table (t2) with NULL for the RIGHT OUTER JOIN operation, and a table with one data record [NULL, NULL] (on columns t0.c0 and t2.c0) is returned. Then natural join of table t1 with that result table is performed, resulting a table with one record ['hhhh', NULL, NULL]. The WHERE condition is the tricky part which causes the inconsistency. Since t2.c0 is NULL (after the right outer join), the result of (t2.c0 IS FALSE)

```

1 SELECT "igbnn" IN (1);
2 -- PostgreSQL: invalid input syntax for type integer
3 -- TiDB: 1

```

**(a) The query triggering different results in PostgreSQL and TiDB**

In specifies a quantified comparison. The expression RVC IN IPV is equivalent to RVC = ANY IPV  
----ISO/IEC 9075, Foundation, 8.4

**(b) The SQL specification about the IN operator**

**Figure 11: An inconsistency caused by unclear description of the IN operator**

should be FALSE. According to the SQL specification [28] (the truth table for IS BOOLEAN operator in *Part 2 Foundation, boolean value expression*), the truth value for NULL IS FALSE and NULL IS TRUE should both be false. On the left of the comparison operator != is a boolean type and on the right a string type. Therefore, MySQL will convert the boolean type false to a numeric number 0 and try to convert the string type to an integer type by default. In this case, the first character of the string 'hhhh' is a non-numeric character, it is converted to integer 0. Therefore, (t2.c0 IS FALSE) != (t1.c0) is evaluated to false and thus the WHERE condition is evaluated to true. The returned result should be ['hhhh', NULL, NULL] according to SQL specification. Yet MySQL 8.0.29 returned an empty list. We reported this inconsistency to the MySQL developers and they confirmed the reason for this inconsistency is violation of SQL specification on the truth value of the IS FALSE operator. Four inconsistencies that we detected are due to the same reason.

**An inconsistency due to unclear SQL specification.** Figure 11a shows a query which triggers different results in PostgreSQL and TiDB. PostgreSQL reports an invalid input syntax for type integer, while TiDB returns the result of 1 (representing true in this context). The documentation of TiDB states that data comparison involves implicit type conversion before the actual comparison. According to the SQL specification shown in Figure 11b, the IN operator is described as a quantitative comparison operator, but the specification does not clearly instruct on how to conduct comparison on different data types. Therefore, different RDBMS developers have different implementation choices, resulting in this inconsistency among different RDBMSs, which may confuse the users. Seven inconsistencies we detected are due to similar reasons.

We argue that both types of inconsistencies could lead to user confusion, unexpected behavior or even serious consequences in commercial or safety critical scenarios. It is thus crucial for RDBMS developers to conform to the SQL specification, and the SQL specification should also be improved to avoid potential confusion and misunderstandings.

**Answer to RQ4:** Among the 19 bugs and 11 inconsistencies we detected, 8 bugs and 2 inconsistencies are due to violating the SQL specification, 11 bugs and 9 inconsistencies are due to unclear or missing descriptions in the SQL specification. Both types of inconsistencies could confuse users and should be carefully handled.

### 5.3 Threats to Validity

**Coverage of the semantics.** We defined the denotational semantics concerning the SQL Data Query Language (DQL) commands. The other types of SQL commands, including DDL, SML, DCL and TCL, can be easily supported by extending our semantics.

**Randomness.** The number of bugs/inconsistencies detected as well as the coverage increase could be affected by the randomness of query generation. To address the issue, we run all comparison experiments (RQ2 and RQ3) 5 times and report the median number. Moreover, in the comparison experiments of different settings of SEMCONT (Table 5) and comparison with baselines (Table 6), we run each experiment for 6 hours to mitigate the effect of randomness.

**Correctness of SEMCONT’s implementation.** We conduct unit test on every semantic rule we implemented. We also conduct integrated testing by generating complex queries to test the composite semantic rules. Moreover, two of the co-authors conduct manual code inspection independently to ensure the correctness of SEMCONT’s implementation. When the SQL specification does not clearly describe the semantics of a keyword, operator or function, we need to make our own design choices. We first refer to SQL specification for similar operators, which have clear semantic descriptions. If such operators are not found, we check the implementation of existing RDBMSs and adopt the one that majority RDBMSs use. We have encounter 47 such cases.

**Fairness of experiment design.** We select two state-of-the-art metamorphic testing approaches for RDBMS as baselines. Moreover, for each baseline, we use two state-of-the-art query generation methods, which result in four settings on the compared approaches. We follow the experiment settings of the baselines presented in their papers to reproduce their performance. All experiments are conducted on the same environment, within the same time frame and configurations of the tested RDBMSs for fairness of comparison.

## 6 RELATED WORK

**Testing Relational DBMS** RAGS [40] and Apollo [29] are notable early methods that implemented differential testing to identify logical errors in database management systems. SQLSmith [39] employs a technique of continuously generating random SQL query statements for database testing. However, this method can only detect system crashes, and does not cover other types of logical errors. Ratel [43] significantly improves the robustness of SQL generation for database testing by merging SQL dictionaries with grammar-based mutations. SparkFuzz [25] introduces a fuzzing-based method that utilizes the query results from a reference database as test oracles. The effectiveness of these differential testing methods is limited by the shared functionalities and syntax supported across the databases under test. Moreover, they may yield false positives due to the varied implementation choices inherent in different database systems.

Metamorphic testing is another mainstream approaches for RDBMS testing [18, 25, 34–36]. MUTASQL [18] and EqsQL [16] construct test cases by defining mutation rules, which are used to generate or synthesize SQL query statements that are functionally equivalent to the original ones. In recent years, SQLancer [12] has emerged as the most effective black-box fuzz testing tool, distinguished by its adoption of three complementary oracles [34–36]. PQS [36] operates by first selecting a row of data, and then synthesizing a query

based on this selected data, such that it must return the initially chosen row. NoREC [34] transforms an optimized SQL query into an equivalent non-optimized version and then compares the execution results of both. TLP [35] divides a SQL query into three separate SQL statements that collectively retain the same semantics as the original query. SQLRight [30] focuses on enhancing the semantic correctness of generated SQL queries. GRIFFIN [24] executes mutation testing within the grammatical boundaries of SQL language. While metamorphic testing approaches address syntax differences arising from various database implementations, returning consistent results do not guarantee the absence of bugs. Moreover, they cannot detect bugs caused by violations of SQL specifications.

**Formal semantics for SQL.** There have been approaches [17, 21, 22, 31, 32, 41, 42] that formalize the semantics of SQL. Chinnai [20] was the first to propose a bag-based SQL operational semantics. More recent works [15, 26, 33] considered NULL values when defining SQL formal semantics. Guagliardo and Libkin [26] defined the formal semantics of SQL, considering not only the syntax of basic SQL query statements but also data structures such as subqueries, sets, and bags. SQLcoq [15] delves into grouping and aggregate functions, proving the equivalence between its proposed formal semantics and relational algebra. Additionally, Zhou et al. [44] introduces an algorithm for proving query equivalence under bag semantics. These methods [15, 19, 26, 44] have significantly enhanced the formal definition of SQL by comprehensively considering both bag and NULL semantics. However, these methods support only a subset of the functionalities defined in the SQL specification. Moreover, these work did not apply semantics for database conformance testing, since these semantics are primarily developed for correctness verification rather than efficient automatic testing.

**Semantics based testing.** Efforts have also been made to utilize executable semantics as test oracles [37, 38]. Various popular programming languages, developed using the K framework [37], offer executable semantics that can be effectively used as testing oracles. ExAIS [38] formalizes executable semantics for artificial intelligence libraries and implements them using the Prolog language. Our work presents the initiative work to use semantics for testing in the domain of database management systems. The semantics are not only employed as testing oracles but also play a pivotal role in guiding test case generation, detecting 19 bugs and 11 inconsistencies, which cannot be detected by existing approaches.

## 7 CONCLUSION

We propose the first automatic conformance testing approach, SEMCONT, for RDBMSs with the SQL specification. We define the formal semantics of SQL and implement them in Prolog, which then act as the oracle for the conformance testing. Moreover, we define three coverage criteria based on the formal semantics to guide test query generation. The evaluation with five well known and extensively tested RDBMSs show that, SEMCONT detects 19 bugs (18 of which are reported for the first time) and 11 inconsistencies, which are either due to violating SQL specification, or missing or unclear SQL specification. A comparison with state-of-the-art RDBMS testing approaches shows that SEMCONT detects more bugs and inconsistencies than baselines during the same time period, and most of the bugs and inconsistencies cannot be detected by baselines.

- [1] 1995. Amazon. <https://www.amazon.com/>. accessed on November 10, 2023.
- [2] 1995. eBay. <https://www.ebay.com/>. accessed on November 10, 2023.
- [3] 1995. MySQL. <https://www.mysql.com>. accessed on November 10, 2023.
- [4] 1996. Booking. <https://www.booking.com/>. accessed on November 10, 2023.
- [5] 1996. PostgreSQL. <https://www.postgresql.org>. accessed on November 10, 2023.
- [6] 2000. SQLite. <https://www.sqlite.org/index.html>. accessed on November 10, 2023.
- [7] 2004. Facebook. <https://www.facebook.com/>. accessed on November 10, 2023.
- [8] 2011. JSQLParser. <https://github.com/JSQLParser/JSqlParser>. accessed on November 10, 2023.
- [9] 2016. ISO/IEC 9075-2:2016. <https://www.iso.org/standard/63555.html>. accessed on November 10, 2023.
- [10] 2016. TiDB. <https://www.pingcap.com/tidb/>. accessed on November 10, 2023.
- [11] 2019. DuckDB. <https://duckdb.org>. accessed on November 10, 2023.
- [12] 2019. SQLancer. <https://github.com/sqlancer/sqlancer>. accessed on November 10, 2023.
- [13] 2022. An error occurred when the CAST function converted the numerical value in the form of scientific notation. <https://sqlite.org/forum/forumpost/3f085531bf>. accessed on November 11, 2022.
- [14] 2023. Conformance Testing of Relational DBMS with SQL Specification based on Formal Semantics (Technical Report). <https://github.com/DBMSTesting/FORMAL-SEMANTICS-FOR-SQL>. accessed on November 10, 2023.
- [15] Véronique Benzaken and Evelynne Contejean. 2019. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 249–261.
- [16] Jeroen Castelain, Mauricio Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-based test data generation for SQL queries. In *Proceedings of the 40th international conference on software engineering*. 1220–1230.
- [17] Stefano Ceri and Georg Gottlob. 1985. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on software engineering* 4 (1985), 324–345.
- [18] Xinyue Chen, Chenglong Wang, and Alvin Cheung. 2020. Testing query execution engines with mutations. In *Proceedings of the workshop on Testing Database Systems*. 1–5.
- [19] James Cheney and Wilmer Ricciotti. 2021. Comprehending nulls. In *The 18th International Symposium on Database Programming Languages*. 3–6.
- [20] Hamid R Chinaei. 2007. *An Ordered Bag Semantics for SQL*. Master’s thesis. University of Waterloo.
- [21] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.
- [22] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (2017), 510–524.
- [23] William F Clocksin and Christopher S Mellish. 2003. *Programming in PROLOG*. Springer Science & Business Media.
- [24] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free DBMS fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
- [25] Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. 2020. SparkFuzz: Searching correctness regressions in modern query engines. In *Proceedings of the workshop on Testing Database Systems*. 1–6.
- [26] Paolo Guagliardo and Leonid Libkin. 2017. A formal semantics of SQL queries, its validation, and applications. *Proceedings of the VLDB Endowment* 11, 1 (2017),

- 13