Semantic Conformance Testing of Relational DBMS

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

Chenglin Tian
Beijing University of Posts and Telecommunications
2024010271@bupt.edu.cn

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

Ruifeng Wang
College of Intelligence and Computing, Tianjin University
ruifeng_wong@tju.edu.cn

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

Yongxin Zhao
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
yxzhao@sei.ecnu.edu.cn

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

Junjie Wang
College of Intelligence and Computing, Tianjin University
junjie.wang@tju.edu.cn

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

ABSTRACT

Relational DBMS implementations are expected to adhere to SQL standards. However, there are currently no tools available that can automatically verify this conformance. The main reasons are twofold. First, the SQL standard 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, e.g., keywords and parameters, defined in the SQL specification. In this work, we introduce the first method for semantic conformance testing of RDBMSs. Our contributions are threefold. Firstly, we formally define the denotational semantics of SQL and implement them in Prolog, creating an executable reference RDBMS for differential testing against existing RDBMSs. Secondly, we propose three coverage criteria based on these formal semantics, along with a coverage-guided query generation algorithm that effectively generates queries achieving high semantic coverage. Lastly, we apply our approach to six widely-used and thoroughly tested RDBMSs, e.g., MySQL, PostgreSQL and OceanBase, uncovering 19 bugs and 13 inconsistencies, all of which are 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, 18(3): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBMSTesting/sql-prolog-implement.

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

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

```
SELECT 'Hello'||NULL;
    --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

1 INTRODUCTION

Relational Database Management Systems (RDBMSs) [12, 26, 29, 39, 41] are fundamental infrastructures for a wide range of application systems, such as web applications and embedded systems [1, 3, 13, 14]. Structured Query Language (SQL) is the standard programming language for relational databases, and its specification is formally documented in ISO/IEC 9075:2016 [15]. 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 [5, 37, 46]. In practice, large online systems, such as that of Alibaba, may integrate and adopt different RBDMSs (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 [40].

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

*Wei Lu is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:XX.XX/XXX.XX

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[1], 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 in 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 [37] or propose metamorphic relations [22, 31–33]. 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 13 inconsistencies. Further examination revealed that 8 bugs and 2 inconsistencies are due to deviations in RDBMS implementations from the SQL specification, 11 bugs and 11 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 six popular and thoroughly-tested RDBMSs, and detected 19 bugs, 18 of which are reported for the first time, and 13 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) [11] 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 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

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

Figure 3: SQL syntax

Table 1: Symbol notations

Symbol	Description
T	A table instance
α	A data record in a table
$\widehat{\alpha}$	Bags of data records
β	An attribute in a table
$\widehat{\beta}$	Bags of attributes
Θ	An expression
T.β_i	The i$_{th}$ attribute in T
T.$\overline{\beta}$	All attributes in T
π_β(T)	Data projection on attribute β
σ_Θ(T)	Data selection on expression Θ
ξ_α(T)	The multiplicity of data record α in table T
ξ_β(T)	The multiplicity of attribute β in table T
$[a, ..., a]_n$	A list of element a with length n
$\bowtie$	Data join operation
$\times$	Cartesian product operation

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 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 implement the semantics using Prolog for two main reasons. First, Prolog, like SQL, is a declarative language, which contrasts with imperative languages such as C typically used in RDBMS implementations. This distinction reduces the likelihood of replicating common errors found in traditional RDBMSs. Second, Prolog is intuitive and straightforward to implement, offering built-in support for operations like list manipulation and querying, which align well with the structure of tables and queries in SQL. For these reasons, Prolog is commonly used in existing research to formalize the semantics of various domains [19, 35].

3 FORMAL SEMANTICS OF SQL

Figure 3 presents the SQL syntax supported by our system. We implement all keywords and features related to the Data Query Language (DQL), including lexical elements, scalar expressions, query expressions and predicates, as defined in Part 2 of the SQL specification (ISO/IEC 9075-2:2016) [15]. Due to space constraints, we provide only a subset of the syntax and corresponding semantics for key SQL keywords here, while the complete syntax and semantic definitions are available in our full report [23].

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 unkown value in SQL.

3.1 Basic symbol definition

Table 1 lists the basic symbols used for semantic definitions in this work. T, α, and β are used to represent a table, a data record in the table, and an attribute of a table, respectively. $\widehat{\alpha}$ and $\widehat{\beta}$ represent bags of data and bags of attribute, respectively. Θ represents expressions, including logical expressions, numeric expressions, constant values and function operations. The operations of data projection,

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

1. $C\llbracket\{[T_1, T_2],\ \mathit{natural\ join}\}\rrbracket \triangleq \{\alpha_1 \circ \alpha_2 | \alpha_1 \in T_1, \alpha_2 \in T_2, \overline{\beta}_I = \mathit{joinC}(T_1.\overline{\beta}, T_2.\overline{\beta}\})$ $\quad \alpha_1 \circ \alpha_2 \triangleq \begin{cases} \alpha_1 \bowtie \alpha_2; & (\overline{\beta}_I \neq \emptyset) \wedge (\pi_{\overline{\beta}_I}(\{\alpha_1\}) = \pi_{\overline{\beta}_I}(\{\alpha_2\})) \\ \mathit{skip}; & \mathit{otherwise} \end{cases}$

2. $C\llbracket\{[T_1, T_2],\ \mathit{left\ join}\}\rrbracket \triangleq \{\alpha_1 \bullet \alpha_2 | \alpha_1 \in T_1, \alpha_2 \in T_2, \overline{\beta}_I = T_1.\overline{\beta} \cap T_2.\overline{\beta}\}$

$$\alpha_1 \bullet \alpha_2 \triangleq \begin{cases} \alpha_1 \bowtie \alpha_2; & (\overline{\beta}_I \neq \emptyset) \wedge (\pi_{\overline{\beta}_I}(\{\alpha_1\}) = \pi_{\overline{\beta}_I}(\{\alpha_2\})) \\ \alpha_1 \bowtie [\mathit{null}, ..., \mathit{null}]_{|\{\alpha_2\}.\overline{\beta}|-|\overline{\beta}_I|}; & (\overline{\beta}_I \neq \emptyset) \wedge (\pi_{\overline{\beta}_I}(\{\alpha_1\}) \neq \pi_{\overline{\beta}_I}(\{\alpha_2\})) \\ \mathit{skip}; \quad \mathit{otherwise} \end{cases}$$

3. $C\llbracket\{[T_1, T_2],\ \mathit{right\ join}\}\rrbracket \triangleq \{\alpha_1 \bullet \alpha_2 | \alpha_1 \in T_1, \alpha_2 \in T_2, \overline{\beta}_I = T_1.\overline{\beta} \cap T_2.\overline{\beta}\}$

$$\alpha_1 \bullet \alpha_2 \triangleq \begin{cases} \alpha_1 \bowtie \alpha_2; & (\overline{\beta}_I \neq \emptyset) \wedge (\pi_{\overline{\beta}_I}(\{\alpha_1\}) = \pi_{\overline{\beta}_I}(\{\alpha_2\})) \\ [\mathit{null}, ..., \mathit{null}]_{|\{\alpha_1\}.\overline{\beta}|-|\overline{\beta}_I|} \bowtie \alpha_2; & (\overline{\beta}_I \neq \emptyset) \wedge (\pi_{\overline{\beta}_I}(\{\alpha_1\}) \neq \pi_{\overline{\beta}_I}(\{\alpha_2\})) \\ \mathit{skip}; \quad \mathit{otherwise} \end{cases}$$

4. $C\llbracket\{[T_1, T_2],\ \mathit{cross\ join}\}\rrbracket \triangleq \{\alpha_1 \times \alpha_2 | \alpha_1 \in T_1, \alpha_2 \in T_2\}$ 5. $C\llbracket\{[T_1, T_2],\ \mathit{inner\ join}\}\rrbracket \triangleq C\llbracket\{[T_1, T_2],\ \mathit{cross\ join}\}\rrbracket$

Collection operation

$\alpha_{u(T_1,T_2)} \triangleq \begin{cases} \alpha; & (\alpha \in T_1) \vee (\alpha \in T_2) \\ \mathit{skip}; & \mathit{otherwise} \end{cases}$ $\quad \alpha_{i(T_1,T_2))} = \begin{cases} \alpha; & (\alpha \in T_1) \wedge (\alpha \in T_2) \\ \mathit{skip}; & \mathit{otherwise} \end{cases}$ $\quad \alpha_{e(T_1,T_2))} = \begin{cases} \alpha; & (\alpha \in T_1) \wedge (\alpha \notin T_2) \\ \mathit{skip}; & \mathit{otherwise} \end{cases}$ $\quad \alpha_{ea(T_1,T_2)} = \begin{cases} \alpha; & \xi_\alpha(T_1) > \xi_\alpha(T_2) \\ \mathit{skip}; & \mathit{otherwise} \end{cases}$

6. $C\llbracket\{[T_1, T_2],\ \mathit{union}\}\rrbracket \triangleq \{\alpha_{u(T_1,T_2)} | \xi_{\alpha_{u(T_1,T_2)}}(T) = 1\}$ $\quad C\llbracket\{[T_1, T_2],\ \mathit{union\ all}\}\rrbracket \triangleq \{\alpha_{u(T_1,T_2)}\}$

7. $C\llbracket\{[T_1, T_2],\ \mathit{intersect}\}\rrbracket = \{\alpha_{i(T_1,T_2)} | \xi_{\alpha_{i(T_1,T_2)}}(T) = 1\}$ $\quad C\llbracket\{[T_1, T_2],\ \mathit{intersect\ all}\}\rrbracket = \{\alpha_{i(T_1,T_2)} | \xi_{\alpha_{i(T_1,T_2)}}(T) = \mathit{min}(\xi_{\alpha_{i(T_1,T_2)}}(T_1), \xi_{\alpha_{i(T_1,T_2)}}(T_2))\}$

8. $C\llbracket\{[T_1, T_2],\ \mathit{except}\}\rrbracket = \{\alpha_{e(T_1,T_2} | \xi_{\alpha_{e(T_1,T_2}}(T) = 1\}$ $\quad C\llbracket\{[T_1, T_2],\ \mathit{except\ all}\}\rrbracket = \{\alpha_{ea(T_1,T_2)} | \xi_{\alpha_{ea(T_1,T_2)}}(T) = \mathit{max}(0, \xi_{\alpha_{ea(T_1,T_2)}}(T_1) - \xi_{\alpha_{ea(T_1,T_2)}}(T_2))\}$

Filter operation

9. $C\llbracket\{[T],\ \mathit{distinct}\}\rrbracket \triangleq \{\alpha | (\forall \alpha \in T,\ \xi_\alpha(T_1) = 1) \wedge (\forall \alpha \in T_1,\ \alpha \in T)\}$ $\qquad$ 10. $C\llbracket\{[T],\ \mathit{all}\}\rrbracket \triangleq T$

Aggregation operation

11. $C\llbracket\{[T],\ \mathit{max}\}\rrbracket \triangleq \{v | (v \in \pi_{T.\overline{\beta}}(T)) \wedge (\forall v_1 \in \pi_{T.\overline{\beta}}(T), \sigma_{(v_1 > v)}(\pi_{T.\overline{\beta}}(T)) = \oslash)\}$

12. $C\llbracket\{[T],\ \mathit{min}\}\rrbracket \triangleq \{v | (v \in \pi_{T.\overline{\beta}}(T)) \wedge (\forall v_1 \in \pi_{T.\overline{\beta}}(T), \sigma_{(v_1 < v)}(\pi_{T.\overline{\beta}}(T)) = \oslash)\}$

13. $C\llbracket\{[T],\ \mathit{sum}\}\rrbracket \triangleq \{v | v = \sum_{v_1 \in \pi_{T.\overline{\beta}}(T)} v_1\}$ 14. $C\llbracket\{[T],\ \mathit{count}\}\rrbracket \triangleq \{v | v = \sum_{\alpha \in \pi_{T.\overline{\beta}}(T)} (\xi_\alpha(T))\}$

15. $C\llbracket\{[T],\ \mathit{avg}\}\rrbracket \triangleq \{v | v = \sum_{v_1 \in \pi_{T.\overline{\beta}}(T)} v_1 / \sum_{\alpha \in \pi_{T.\overline{\beta}}(T)} (\xi_\alpha(T))\}$

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

16. $\mathcal{H}\llbracket \mathit{from}\langle \mathit{tname}\rangle \rrbracket \triangleq (A, A.\langle \mathit{tname}\rangle)$ $\qquad$ 17. $\mathcal{H}\llbracket \mathit{from}\langle \mathit{tname}_1\rangle, \langle \mathit{tname}_2\rangle \rrbracket \triangleq (A, C\llbracket[A.\langle \mathit{tname}_1\rangle, A.\langle \mathit{tname}_2\rangle],\ \mathit{cross\ join}\ \rrbracket)$

18. $\mathcal{H}\llbracket \mathit{select}\ * \rrbracket \triangleq (\{T\}, \pi_{T.\overline{\beta}}(T))$ 19. $\mathcal{H}\llbracket \mathit{select}\ \langle \mathit{cname}\rangle\ [,\ \langle \mathit{cname}\rangle ...] \rrbracket \triangleq (\{T\}, \pi_{\langle \mathit{cname}\rangle\ [,\ \langle \mathit{cname}\rangle ...]}(T))$

20. $\mathcal{H}\llbracket \mathit{on}\ \langle \mathit{bvexp}\rangle \rrbracket \triangleq (\{T\}, \sigma_{\mathcal{B}(\langle \mathit{bvexp}\rangle)}(T))$ 21. $\mathcal{H}\llbracket \mathit{where}\ \langle \mathit{bvexp}\rangle \rrbracket \triangleq (\{T\}, \sigma_{\mathcal{B}(\langle \mathit{bvexp}\rangle)}(T))$ 22. $\mathcal{H}\llbracket \mathit{having}\ \langle \mathit{bvexp}\rangle \rrbracket \triangleq (\{T\}, \sigma_{\mathcal{B}(\langle \mathit{bvexp}\rangle)}(T))$

23. $\mathcal{H}\llbracket \mathit{group\ by}\ \langle \mathit{cname}\rangle \rrbracket \triangleq (\{T\}, (\widehat{\alpha_1}, ..., \widehat{\alpha_k}) : \forall \widehat{\alpha_p} \in (\widehat{\alpha_1}, ..., \widehat{\alpha_k}), \forall\ v_{ij} \in \pi_{\langle \mathit{cname}\rangle}(\widehat{\alpha_p}), (v_{ij} = v_{i1}))$

24. $\mathcal{H}\llbracket \mathit{order\ by}\ \langle \mathit{cname}\rangle\ \rrbracket \triangleq (\{T\}, T_1) \mathit{where} (\forall \alpha \in T_1,\ \xi_\alpha(T_1) = \xi_\alpha(T)) \wedge (\forall \alpha \in T, \xi_\alpha(T) = \xi_\alpha(T_1)) \wedge (\forall v_i, v_j \in \sigma_{T_1.\langle \mathit{cname}\rangle}(T_1), i < j \text{ iff } v_i <= v_j)$

25. $\mathcal{H}\llbracket \langle \mathit{subquery}\rangle \rrbracket \triangleq \mathcal{H}\llbracket \langle \mathit{query\ expression}\rangle \rrbracket$

Composite keyword operation

$\diamond :$ (1) $\dfrac{\mathcal{H}\llbracket \mathit{expression}_1 \rrbracket \triangleq (A, T),\ \mathcal{H}\llbracket \mathit{expression}_2 \rrbracket \triangleq (\{T\}, T')}{\mathcal{H}\llbracket \mathit{expression}_1 \rrbracket \diamond \mathcal{H}\llbracket \mathit{expression}_2 \rrbracket \triangleq (A, T')}$ (2) $\dfrac{\mathcal{H}\llbracket \mathit{expression} \rrbracket \triangleq (A, T),\ C\llbracket\{\{T\}, OP\}\rrbracket \triangleq T'}{\mathcal{H}\llbracket \mathit{expression} \rrbracket \diamond C\llbracket\{\{T\}, OP\}\rrbracket \triangleq (A, T')}$

(3) $\dfrac{C\llbracket\{L, OP\}\rrbracket \triangleq T,\ \mathcal{H}\llbracket \mathit{expression} \rrbracket \triangleq (\{T\}, T')}{C\llbracket\{L, OP\}\rrbracket \diamond \mathcal{H}\llbracket \mathit{expression} \rrbracket \triangleq (L, T')}$ (4) $\dfrac{\mathcal{H}\llbracket \mathit{expression}_1 \rrbracket \triangleq (A, T_1),\ \mathcal{H}\llbracket \mathit{expression}_2 \rrbracket \triangleq (A, T_2),\ C\llbracket\{\{T_1, T_2\}, OP\}\rrbracket \triangleq T_3}{(\mathcal{H}\llbracket \mathit{expression}_1 \rrbracket, \mathcal{H}\llbracket \mathit{expression}_2 \rrbracket) \diamond C\llbracket\{\{T_1, T_2\}, OP\}\rrbracket \triangleq (A, T_3)}$

26. $\mathcal{H}\llbracket \langle \mathit{queryexp}\rangle \rrbracket =$
$\mathcal{H}\llbracket \mathit{select}\ [\langle \mathit{sop}\rangle | \langle \mathit{af}\rangle]\ \langle \mathit{cname}_1\rangle [, \langle \mathit{cname}_2\rangle ...]\ \mathit{from} \langle \mathit{tname}_1\rangle [, \langle \mathit{tname}_2\rangle ...] | \mathit{from}\ \langle \mathit{tname}_1\rangle\ \mathit{natural/cross\ join}\ \langle \mathit{tname}_2\rangle$
$| \mathit{from}\ \langle \mathit{tname}_1\rangle\ \mathit{left/right/full/inner\ join}\ \langle \mathit{tname}_2\rangle\ \mathit{on} \langle \mathit{bvexp}\rangle\ [\mathit{where} \langle \mathit{bvexp}\rangle]\ [\mathit{group\ by}\ \langle \mathit{cname}\rangle]\ [\mathit{having}\ \langle \mathit{bvexp}\rangle]$
$[\mathit{order\ by}\ \langle \mathit{cname}\rangle] \rrbracket \triangleq$
$\mathcal{H}\llbracket \mathit{from} \langle \mathit{tname}_1\rangle \rrbracket | (\mathcal{H}\llbracket \mathit{from}\ \langle \mathit{tname}_1\rangle \rrbracket, \mathcal{H}\llbracket \mathit{from}\ \langle \mathit{tname}_2\rangle \rrbracket) \diamond C\llbracket L, \mathit{natural/cross\ join}\ \rrbracket | (\mathcal{H}\llbracket \mathit{from}\ \langle \mathit{tname}_1\rangle \rrbracket, \mathcal{H}\llbracket \mathit{from} \langle \mathit{tname}_2\rangle \rrbracket)$
$\diamond C\llbracket L, \mathit{left/right/inner/full\ join}\ \rrbracket \diamond \mathcal{H}\llbracket \mathit{on}\ \langle \mathit{bvexp}\rangle \rrbracket [\ \diamond\ \mathcal{H}\llbracket \mathit{where}\ \langle \mathit{bvexp}\rangle \rrbracket] [\ \diamond \mathcal{H}\llbracket \mathit{group\ by}\ \langle \mathit{cname}\rangle \rrbracket] [\ \diamond \mathcal{H}\llbracket \mathit{having}\ \langle \mathit{bvexp}\rangle \rrbracket]$
$\diamond\ \mathcal{H}\llbracket \mathit{select}\ \langle \mathit{cname}_1\rangle [, \langle \mathit{cname}_2\rangle ...] \rrbracket [\ \diamond\ C\llbracket\{T_1, \langle \mathit{sop/af}\rangle\}\rrbracket] [\ \diamond\ \ \mathcal{H}\llbracket \mathit{order\ by}\ \langle \mathit{cname}\rangle \rrbracket]$

27. $\mathcal{H}\llbracket \langle \mathit{queryexp}_1\rangle\ \langle \mathit{cop}\rangle\ \langle \mathit{queryexp}_2\rangle \rrbracket \triangleq (\mathcal{H}\llbracket \langle \mathit{queryexp}_1\rangle \rrbracket, \mathcal{H}\llbracket \langle \mathit{queryexp}_2\rangle \rrbracket) \diamond C\llbracket\{[T_1, T_2], \langle \mathit{cop}\rangle\}\rrbracket$

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

data selection, and calculating the multiplicity of data records or attributes are denoted by π, σ, and ξ, respectively. The operations of joining two data records and performing the Cartesian product are represented by $\bowtie$ and $\times$, respectively. We use $[a, ..., a]_n$ 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 defines functionalities that directly operate on a list of tables and return a new table. The second category 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.

Algorithm 1: 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
```

Definition 1 (Keyword semantics ($C : \{L, OP\} \mapsto T$)) 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 (Composite semantics ($\mathcal{H} : S \mapsto (A, T)$)) 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 16-27), including keywords such as FROM, WHERE, ON, SELECT, GROUP BY, HAVING, ORDER BY, and their combinations in SQL statements.

3.3 Prolog implementation of SQL semantics

We have implemented the formal semantics of SQL defined in Figure 4 in a tool named SemConT using 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 1, 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 1 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 time complexity of Algorithm 1 is O(n) with n being the number of operators in the given SQL query.

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 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.

Correctness of SemConT. As mentioned in Section 2, Prolog, being declarative, is naturally suited to specify denotational semantics we defined. Taking the 'select' keyword as an example, rule 19 of Figure 4 shows the formal semantics of 'select' and line 9-11 in Figure 2 shows the corresponding Prolog implementation. The formal semantics defines the select keyword as a column reference using the projection operation π to select columns from a table `T`. This maps directly to the select_clause function in the Prolog implementation, which checks for column references and extracts the relevant columns from Table `Tb`. Prolog's rule-based structure ensures a one-to-one correspondence with the formal semantics, minimizing implementation errors and ensuring adherence to the SQL specification. Moreover, we have conducted comprehensive testing (on 6 different RDBMS systems, with 18 millions of test cases) and code review following the software engineering standard procedure, covering all the semantic rules we've implemented.

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

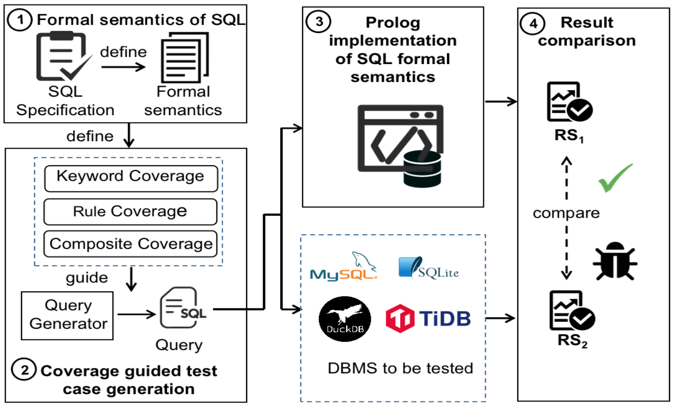


Figure 5: Overview of SemConT

Algorithm 2: Coverage guided query generation

```
Function SQLGeneration():
    Initialize $coverage$=0, $coveredSet$
    Initialize $queryPool$=GenerateQuery()
    while TRUE do
        while $coverage$ does not increase do
            $queryInit$ = GetQueryFromPool()
            $query$ = MutateQuery()
            CalculateCoverage($query$)
        end
        AddQueryIntoPool($query$)
        ExecuteQuery($query$)
        if timeout then
            break
        end
    end
Function CalculateCoverage($query$):
    if $query.pattern$ not in $coveredSet$ then
        UpdateCoverage($coverage$)
        ADD $query.pattern$ To $coveredSet$
    end
```

case generation. Here, we enhance the syntax-guided generation method, SQLancer [38], 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 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 underscore 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 [38] 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.

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.

$$Cov_k = \frac{N_{tk}}{N_K} \tag{1}$$

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.

$$Cov_r = \frac{N_{tr}}{N_R} \tag{2}$$

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.

$$Cov_{cr} = \frac{N_{tcr}}{N_{CR}} \tag{3}$$

Coverage-guided query generation. Algorithm 2 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

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

(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. 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). The time complexity of Algorithm 2 is O(nlogn), with n being the number of all possible rules for a particular coverage criterion. Our goal is to generate a set of test cases that collectively cover all semantic rules. The test case generation algorithm operates by randomly generating a test case and retaining it only if it covers a previously uncovered rule; otherwise, it is discarded. This process is analogous to the Coupon Collector's problem [25], which estimates the time required to collect n distinct coupons through random sampling. Similarly, our random generation process achieves full coverage with high probability at a time complexity of O(nlogn). In our work, the number n is 138 for keyword coverage, 556 for semantic rule coverage and 19 million for composite rule coverage.

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 [20] 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.

4.3 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.

To enhance result explainability reported by SemConT, we we log the inconsistencies and provide the semantic rule we implemented as explanations of the inconsistency. Moreover, for those under-specified keywords like `IN`, we provide multiple Prolog implementations based on popular RDBMSs, e.g., MySQL and PostgreSQL, allowing users to configure the desired semantics.

4.4 Discussion on Extensibility

In this paper, we define and implement the denotational semantics concerning the SQL Data Query Language (DQL) commands. The other types of SQL commands, including DDL, SML, and DCL can be easily supported by extending our semantics. For the semantics of transactions and concurrency, we formalize single transactions by executing SQL queries sequentially in real-time order. For concurrent transactions, the semantics should define all valid schedules. Formally, the semantics of two concurrent transactions T1 and T2 can be defined as: $T1||T2 \triangleq \{Q_1||Q_2, Q_1 \in T1, Q_2 \in T2 \wedge RTConstraint(T1) \wedge RTConstraint(T2) \wedge !\, IsolationConstraint\}$ The symbol || represents the concurrent execution of two transactions or SQL queries, *RTConstraint(T)* formalizes the realtime order constraints of SQL queries in T, and *IsolationConstraint* formalizes the schedule constraints associated with a particular isolation level.

Since the SQL specification [15] only provide the anomaly phenomena, which can be formalized as specific schedule templates among transactions, to be avoided in each isolation level, we need to exclude those invalid schedules in our semantics.

Taking dirty read as an example, any schedule that contains the sequence of `T1.w(x), T2.r(x)` should be avoided as T2 has read an uncommitted write by T1, and this potentially lead to dirty read if T1 aborts. Then this pattern can be added into the *IsolationConstraint* to filter out schedules containing this pattern. This schedule constraint is associated with all isolation levels as they all forbid dirty read. We can generate test cases that contain schedules of the phenomena to be avoided and inspect on the logs of the tested RDBMSs to check whether their implementations contain behaviors of those phenomena.

5 EVALUATION

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 RDBMSs. We selected six popular and widely used RDBMSs, each offering a range of distinct features and application scenarios, to demonstrate the effectiveness of our approach. MySQL [26] and PostgreSQL [29] are the two most popular open-source database management systems. SQLite [39] and DuckDB [12] are both embedded DBMSs, running within the process of other applications. TiDB [41] and OceanBase [28] are popular distributed RDBMSs. It is important to note that we used the latest release of each RDBMS, which has been extensively tested by existing approaches [31, 32].

Compared baselines. We compared SemConT with TLP [32] and NoREC [31], 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 [38] and SQLRight [22]. 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).

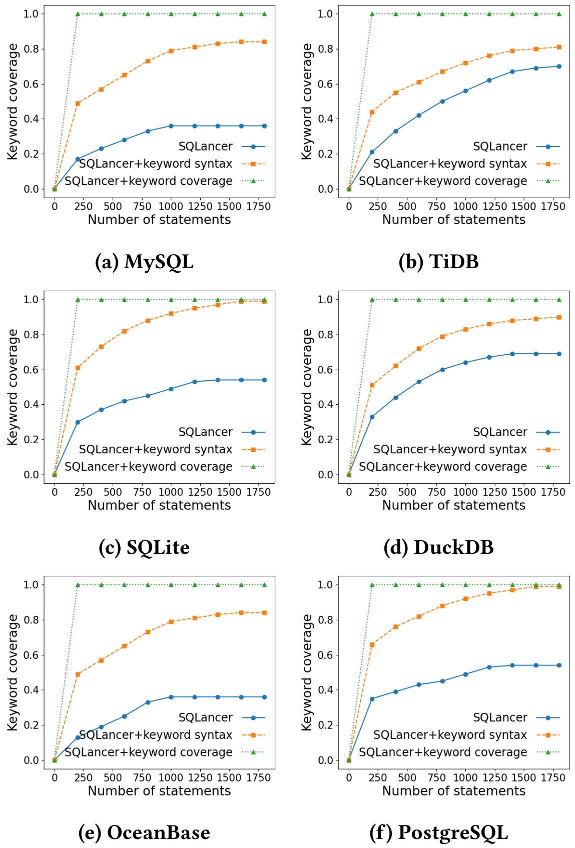
5.2 Experiment results

5.2.1 *Effectiveness of SemConT.*

We ran SemConT on six RDBMSs for a period of 3 months and reported the detected issues to the corresponding developer communities. Table 3 shows the details of the confirmed bugs and inconsistencies in six RDBMSs detected by SemConT. It detects 19 bugs (18 newly reported) and 13 inconsistencies, which are all confirmed by developers, in six extensively tested RDBMSs. All detected bugs and inconsistencies are either due to RDBMS violating the SQL specification, or missing or unclear SQL specification. Out of the issues identified, 23 are related to scalar expressions and 9 to other keywords, including joins and various relational operators. Our primary objective is to detect inconsistencies between RDBMS implementations and the SQL specification by generating test cases that achieve high coverage across different SQL keywords. Our implementation focuses on scalar expressions, query expressions, and predicates. Among these, scalar expressions are the most complex, as they often involve combinations of multiple keywords and subqueries. They are also under-specified in the SQL standard and insufficiently tested by existing approaches [22, 31–33]. In contrast, query expressions and predicates are clearly defined in the SQL specification, leading to fewer ambiguities across RDBMSs. Moreover, they have been extensively tested by prior research [22, 31–33], making it more challenging to uncover new inconsistencies. Among the confirmed bugs, 1 has been reported previously and 1 has been fixed. The remaining 13 issues are confirmed as inconsistencies, and the developers claim that they were their design choices. Among the 13 inconsistencies, 2 of them are due to violation of the SQL specification and 11 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) and TiDB (42773, 40995). The queries triggering the

Table 3: 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	comfirmed
3	109842	MySQL	Bug	missing spec	comfirmed
4	109149	MySQL	Bug	missing spec	comfirmed
5	110438	MySQL	Bug	violate spec	comfirmed
6	109147	MySQL	Inconsistency	missing spec	comfirmed
7	109148	MySQL	Inconsistency	unclear spec	comfirmed
8	109836	MySQL	Inconsistency	missing spec	comfirmed
9	109845	MySQL	Inconsistency	missing spec	comfirmed
10	110439	MySQL	Inconsistency	violate spec	comfirmed
11	110346	MySQL	Inconsistency	violate spec	comfirmed
12	109962	MySQL	Inconsistency	missing spec	comfirmed
13	110711	MySQL	Inconsistency	missing spec	comfirmed
14	40996	TiDB	Bug	missing spec	comfirmed
15	40995	TiDB	Bug	missing spec	comfirmed
16	39260	TiDB	Bug	missing spec	comfirmed
17	39259	TiDB	Bug	missing spec	comfirmed
18	39258	TiDB	Bug	unclear spec	comfirmed
19	42375	TiDB	Bug	violate spec	comfirmed
20	42376	TiDB	Bug	violate spec	comfirmed
21	42378	TiDB	Bug	violate spec	comfirmed
22	42379	TiDB	Bug	violate spec	comfirmed
23	42377	TiDB	Bug	violate spec	comfirmed
24	42773	TiDB	Inconsistency	missing spec	comfirmed
25	40995	TiDB	Inconsistency	missing spec	comfirmed
26	7e03a4420a	SQLite	Bug	missing spec	comfirmed
27	3f085531bf	SQLite	Bug	missing spec	comfirmed
28	6e4d3e389e	SQLite	Bug	violate spec	comfirmed
29	411bce39d0	SQLite	Inconsistency	missing spec	comfirmed
30	6804	DuckDB	Bug	violate spec	fixed
31	2104	OceanBase	Inconsistency	missing spec	confirmed
32	2105	OceanBase	Inconsistency	missing spec	confirmed

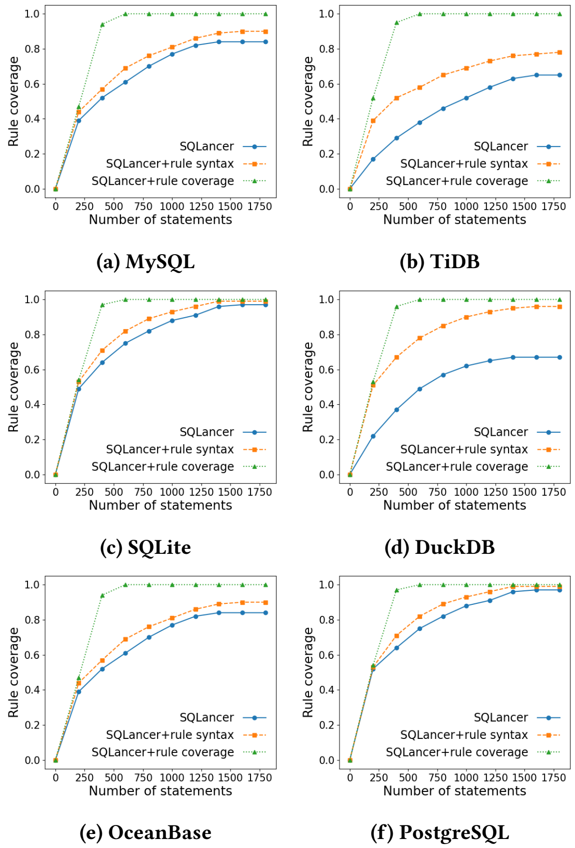


(a) MySQL **(b) TiDB**

(c) SQLite **(d) DuckDB**

(e) OceanBase **(f) PostgreSQL**

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



(a) MySQL **(b) TiDB**

(c) SQLite **(d) DuckDB**

(e) OceanBase **(f) PostgreSQL**

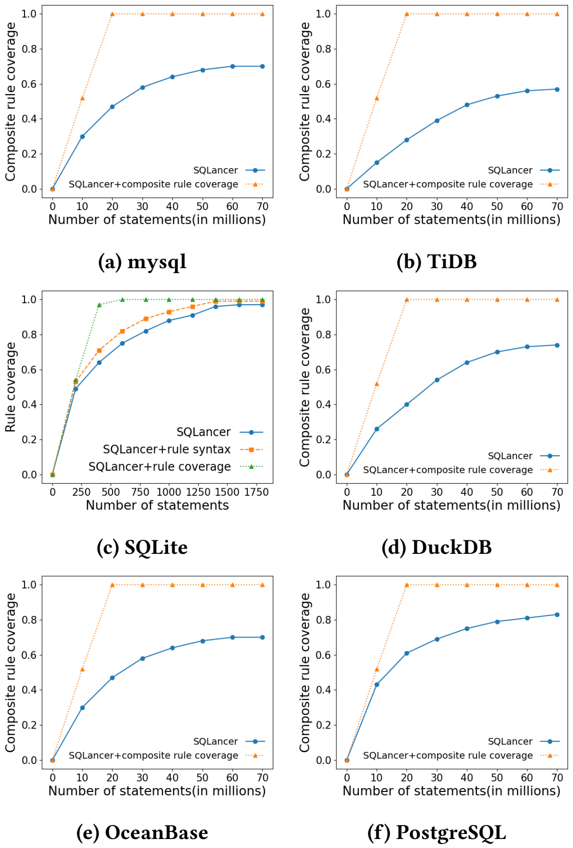
Figure 7: The rule coverage increment (y-axis) with the number of queries (x-axis)

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 six databases have been extensively tested by existing methods [31–33], 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*" [15]. 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 5.2.4 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), one in TiDB (42773) and one in OceanBase(2105) 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.



(a) mysql

(b) TiDB

(c) SQLite

(d) DuckDB

(e) OceanBase

(f) PostgreSQL

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

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 [15]. 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) and OceanBase (2104). 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.

5.2.2 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. 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.

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 six databases. Notably, within the first 1500 SQL statements, over 80% of the keywords were covered on all six databases, with SQLite achieving an impressive keyword coverage of 99%. Keyword coverage guided query generation (SQLancer+keyword coverage) further improves the 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.

Figure 8 depicts the improvements in composite rule coverage by the generated queries for the five 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, DuckDB and OceanBase 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 4 presents the experimental results obtained from testing six 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

Table 4: 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
OceanBase	0	2	13.25	0	2	13.37	0	2	7.92	0	2	9.13

Table 5: 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
OceanBase	0	1	0	0	-	-	-	-	0	2

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.

5.2.3 Comparison with baselines. We compare SemConT with two state-of-the-art approaches TLP [31] and NoREC [32], which are metamorphic testing approaches for relational DBMS. For both approaches, we adopt SQLancer [38] and SQLRight [22] for query generation. Notably, SQLancer does not support the NoREC oracle for MySQL and TiDB, while SQLRight does not support TiDB, DuckDB and OceanBase. 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 5.

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 detectd 1 bug in MySQL. SemConT outperformed the compared approaches and detected 16 bugs and 11 inconsistencies in the five 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.

On average, our tool finds a bug in 67 minutes using 19,381 test cases, compared to 300 minutes and 1.3 million test cases for SQLancer, and 30 hours and 8.4 million test cases for SQLRight. Our approach finds more bugs/inconsistencies with fewer test cases, demonstrating its effectiveness and efficiency in detecting bugs and inconsistencies that violating SQL specifications. We also compared the memory usage of SemConT and SQLancer during a 6-hour test on six databases. Results indicate that SemConT's memory usage is comparable to the baseline.

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

Figure 9: A bug in MySQL 8.0.29

5.2.4 Case study of bugs and inconsistencies. SemConT has identified 19 bugs and 13 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 [15].

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,

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

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

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) should be FALSE. According to the SQL specification [15] (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.

6 RELATED WORK

Testing Relational DBMS RAGS [37] and Apollo [21] are notable early methods that implement differential testing to identify logical errors in DBMS. SQLSmith [36] employs a technique of continuously generating random SQL queries for database testing. Ratel [44] significantly improves the robustness of SQL generation for database testing by merging SQL dictionaries with grammar-based mutations. SparkFuzz [17] introduces a fuzzing-based method that utilizes the query results from a reference database as test oracles. The effectiveness of these 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 of RDBMSs. Metamorphic testing is another mainstream approaches for RDBMS testing [6, 17, 31–33]. MUTASQL [6] and Eqsql [4] construct test cases by defining mutation rules, which are used to generate or synthesize SQL queries that are functionally equivalent to the original ones. Recently, SQLancer [38] has emerged as the most effective black-box fuzzing tool, distinguished by its adoption of three complementary oracles [31–33]. SQLRight [22] focuses on enhancing the semantic correctness of generated SQL queries. GRIFFIN [16] executes mutation testing within the grammatical boundaries of SQL language. While metamorphic testing approaches address syntax differences arising from various database implementations, they cannot detect bugs caused by violations of SQL specifications.

Formal semantics for SQL. There have been approaches [5, 9, 10, 24, 27, 42, 43] that formalize the semantics of SQL. Chinaei [8] was the first to propose a bag-based SQL operational semantics. More recent works [2, 18, 30] considered NULL values when defining SQL formal semantics. Guagliardo and Libkin [18] 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 [2] delves into grouping and aggregate functions, proving the equivalence between its proposed formal semantics and relational algebra. Additionally, Zhou et al. [45] introduces an algorithm for proving query equivalence under bag semantics. These methods [2, 7, 18, 45] 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 [34, 35]. Various popular programming languages, developed using the K framework [34], offer executable semantics that can be effectively used as testing oracles. ExAIS [35] 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 13 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 six well known and extensively tested RDBMSs show that, SemConT detects 19 bugs (18 of which are reported for the first time) and 13 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.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62472429, 61972403, 61732014, 62102283) and by Ant Group through CCF-Ant Research Fund No. CCF-AFSG RF20240103. We thank the anonymous reviewers for their constructive suggestions, which help improve the quality of this paper.

REFERENCES

[1] Amazon. 1995. https://www.amazon.com/. accessed on November 10, 2023.

[2] Véronique Benzaken and Evelyne 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.

[3] Booking. 1996. https://www.booking.com/. accessed on November 10, 2023.

[4] Jeroen Castelein, Maurício 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.

[5] 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.

[6] 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.

[7] James Cheney and Wilmer Ricciotti. 2021. Comprehending nulls. In *The 18th International Symposium on Database Programming Languages* (Copenhagen, Denmark) *(DBPL '21)*. Association for Computing Machinery, New York, NY, USA, 3–6. https://doi.org/10.1145/3475726.3475730

[8] Hamid R Chinaei. 2007. *An Ordered Bag Semantics for SQL*. Master's thesis. University of Waterloo.

[9] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. 2017. Demonstration of the Cosette Automated SQL Prover. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1591–1594. https://doi.org/10.1145/3035918.3058728

[10] 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.

[11] William F Clocksin and Christopher S Mellish. 2003. *Programming in PROLOG*. Springer Science & Business Media.

[12] DuckDB. 2019. https://duckdb.org. accessed on November 10, 2023.

[13] eBay. 1995. https://www.ebay.com/. accessed on November 10, 2023.

[14] Facebook. 2004. https://www.facebook.com/. accessed on November 10, 2023.

[15] International Organization for Standardization. 2016. ISO/IEC 9075-2:2016: Information technology – Database languages – SQL/Foundation. (2016).

[16] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 49, 12 pages. https://doi.org/10.1145/3551349.3560431

[17] 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* (Portland, Oregon) *(DBTest '20)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3395032.3395327

[18] Paolo Guagliardo and Leonid Libkin. 2017. A formal semantics of SQL queries, its validation, and applications. *Proceedings of the VLDB Endowment* 11, 1 (2017), 27–39.

[19] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2020. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1695–1712.

[20] JSQLParser. 2011. https://github.com/JSQLParser/JSqlParser. accessed on November 10, 2023.

[21] 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* 13, 1 (2019), 57–70.

[22] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of {DBMS} with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. 4309–4326.

[23] Shuang Liu, Chenglin Tian, Jun Sun, Ruifeng Wang, Wei Lu, Yongxin Zhao, Yinxing Xue, Junjie Wang, and Xiaoyong Du. 2024. Conformance Testing of Relational DBMS Against SQL Specifications (Technical Report). https://github.com/DBMSTesting/Technical-report.

[24] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a verified relational database management system. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 237–248.

[25] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis* (2nd ed.). Cambridge University Press, USA.

[26] MySQL. 1995. https://www.mysql.com. last accessed on November 10, 2023.

[27] Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. 1991. Formal semantics of SQL queries. *ACM Transactions on Database Systems (TODS)* 16, 3 (1991), 513–534.

[28] OceanBase. 2016. https://en.oceanbase.com/. accessed on November 10, 2023.

[29] PostgreSQL. 1996. https://www.postgresql.org. accessed on November 10, 2023.

[30] Wilmer Ricciotti and James Cheney. 2022. A Formalization of SQL with Nulls. *Journal of Automated Reasoning* 66, 4 (2022), 989–1030.

[31] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.

[32] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[33] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.

[34] Grigore Roșu and Traian Florin Șerbănută. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[35] Richard Schumi and Jun Sun. 2022. ExAIS: executable AI semantics. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 859–870. https://doi.org/10.1145/3510003.3510112

[36] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2019. SQLsmith: a random SQL query generator. https://github.com/anse1/sqlsmith. accessed on November 11, 2022.

[37] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.

[38] SQLancer. 2019. https://github.com/sqlancer/sqlancer. accessed on November 10, 2023.

[39] SQLite. 2000. https://www.sqlite.org/index.html. accessed on November 10, 2023.

[40] Chenglin Tian. 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.

[41] TiDB. 2016. https://www.pingcap.com/tidb/. accessed on November 10, 2023.

[42] Jan Van den Bussche and Stijn Vansummeren. 2009. Translating SQL into the relational algebra. *Course notes, Hasselt University and Université Libre de Bruxelles* (2009).

[43] Margus Veanes, Nikolai Tillmann, and Jonathan De Halleux. 2010. Qex: Symbolic SQL query explorer. In *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*. Springer, 425–446.

[44] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 328–337.

[45] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2020. A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. *arXiv preprint arXiv:2004.00481* (2020).

[46] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.