Conformance Testing of Relational DBMS Against SQL Specifications (Technical Report)

1st Shuang Liu
School of Information, Renmin University of China,
Beijing, China.
E-mail: shuang.liu@ruc.edu.cn

2nd Chenglin Tian
the College of Intelligence and Computing, Tianjin University,
Tianjin, China.
E-mail: chunqiubf@gmail.com

3rd Jun Sun
Singapore Management University,
Singapore, Singapore.
E-mail: junsun@smu.edu.sg

4th Ruifeng Wang
the College of Intelligence and Computing, Tianjin University,
Tianjin, China.
E-mail: ruifeng˙wong@tju.edu.cn

5th Wei Lu
School of Information, Renmin University of China,
Beijing, China.
E-mail: lu-wei@ruc.edu.cn

6th Yongxin Zhao
East China Normal University,
Shanghai, China.
E-mail: yxzhao@sei.ecnu.edu.cn

7th Yinxing Xue
University of Science and Technology of China,
Suzhou, China.
E-mail: yxxue@ustc.edu.cn

8th Junjie Wang
the College of Intelligence and Computing, Tianjin University,
Tianjin, China.
E-mail: junjie.wang@tju.edu.cn

9th Xiaoyong Du
School of Information, Renmin University of China,
Beijing, China.
E-mail: duyong@ruc.edu.cn

***Abstract*—A Relational Database Management System (RDBMS) is one of the fundamental software that supports a wide range of applications, making it critical to identify bugs within these systems. There has been active research on testing RDBMS, most of which employ crash or use metamorphic relations as the oracle. Although existing approaches can detect bugs in RDBMS, they are far from comprehensively evaluating the RDBMS's correctness (i.e., with respect to the semantics of SQL). In this work, we propose a method to test the semantic conformance of RDBMS i.e., whether its behavior respects the intended semantics of SQL. Specifically, we have formally defined the semantics of SQL and implemented them in Prolog. Then, the Prolog implementation serves as the reference RDBMS, enabling differential testing on existing RDBMS. We applied our approach to four widely-used and thoroughly tested RDBMSs, i.e., MySQL, TiDB, SQLite, and DuckDB. In total, our approach uncovered 19 bugs and 11 inconsistencies, which are all related to violating the SQL specification or missing/unclear specification, thereby demonstrating the effectiveness and applicability of our approach.**

I. Introduction

Relational Database Management System (RDBMS) [3], [5], [6], [12], [13] are widely adopted in various applications, including both 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 [10]. As the parser and executor of SQL queries, an RDBMS should conform to the SQL specification to ensure the correct implementation 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 [18], [47], [53]. Bugs have also been reported due to violations of the SQL specifications [15], which may potentially lead to data integrity issues or even security vulnerabilities in the database. The serious impacts of bugs in RDBMS have been discussed by various existing studies [14], [34], [40], [41], and thus it is critical to detect those bugs.

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[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. In practice, large online systems, such as that of Alibaba and Tencent, may integrate and adopt different RBDMSs (sometimes dynamically) for efficiency reasons, and such bugs may result in unexpected system behaviors. To avoid problems caused by such inconsistencies between RDBMS implementations, we need a method of systematically identifying such under-specification in SQL semantics.

[1]We have also detected this bug in MariaDB 10.9.4 and MySQL 8.0.29.

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

(a) A test case 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 test case causing an inconsistency between SQLite version 3.39.0 and PostgreSQL version 16.2, and the SQL specification

Fig. 1: A Bug and an Inconsistency detected by our approach

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 [47] or propose metamorphic relations [34], [40]–[42]. 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 four 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, in this work, 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 formalized 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.

II. PRELIMINARY

A. Prolog

Prolog (Programming in logic) [26] 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.

```
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]].
```

Fig. 2: An example of implementing the semantics of SQL keywords `SELECT` and `FROM` using Prolog

We manually analyze the semantics of keywords in the SQL specification and defined the denotational semantics for 138 SQL keywords or features related to query functionality 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 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 a rule for the `FROM` keyword, where the first parameter `T` represents the name of the desired 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 (as shown in 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`. First, 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

(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*⟩ASC|DESC
(19)⟨*vexp*⟩ ::= ⟨*nvexp*⟩ | ⟨*svexp*⟩ | ⟨*bvexp*⟩ | ⟨*caseexp*⟩ | ⟨*castexp*⟩ | ⟨*cname*⟩ | *null*
(20)⟨*svexp*⟩ ::= ⟨*concatenation*⟩ | ⟨*character substring function*⟩ | ⟨*trim function*⟩ | ⟨*fold*⟩ | ⟨*vexp*⟩
| *string literal*
(21)⟨*concatenation*⟩ ::= ⟨*svexp*⟩||⟨*svexp*⟩
(22)⟨*character substring function*⟩ ::= SUBSTRING(⟨*svexp*⟩*FROM*⟨*vexp*⟩)
(23)⟨*trim function*⟩ ::= LTRIM|RTRIM(⟨*svexp*⟩)
(24)⟨*fold*⟩ ::= UPPER|LOWER(⟨*svexp*⟩)
(25)⟨*nvexp*⟩ ::= ⟨*arithmetic expression*⟩ | ⟨*modules expression*⟩| ⟨*length expression*⟩
| ⟨*absolute value expression*⟩| ⟨*natural logarithm*⟩| ⟨*exponential function*⟩| ⟨*power function*⟩| ⟨*square root*⟩
| ⟨*floor function*⟩| ⟨*ceiling function*⟩| ⟨*vexp*⟩| *numeric literal*
(26)⟨*arithmetic expression*⟩ ::= ⟨*nvexp*⟩ + ⟨*nvexp*⟩ | ⟨*nvexp*⟩ − ⟨*nvexp*⟩ | ⟨*nvexp*⟩ ∗ ⟨*nvexp* ⟩| ⟨*nvexp*⟩/⟨*nvexp*⟩
(27)⟨*modules expression*⟩ ::= MOD(⟨*nvexp*⟩ , ⟨*nvexp*⟩)
(28)⟨*length expression*⟩ ::= {LENGTH|CHAR˙LENGTH|CHARACTER˙LENGTH}(⟨*svexp*⟩)
(29)⟨*absolute value expression*⟩ ::= ABS(⟨*nvexp*⟩)
(30)⟨*natural logarithm*⟩ ::= LN(⟨*nvexp*⟩)
(31)⟨*exponential function*⟩ ::= EXP(⟨*nvexp*⟩)
(32)⟨*power function*⟩ ::= POWER(⟨*nvexp*⟩, ⟨*nvexp*⟩)
(33)⟨*square root*⟩ ::= SQRT(⟨*nvexp*⟩)
(34)⟨*floor function*⟩ ::= FLOOR(⟨*nvexp*⟩)
(35)⟨*ceiling function*⟩ ::= {CEIL|CEILING}(⟨*nvexp*⟩)
(36)⟨*bvexp*⟩ ::= ⟨*logical expression*⟩ | ⟨*is expression*⟩ | ⟨*comparison expression*⟩ | ⟨*between expression*⟩
⟨*in expression*⟩ | ⟨*exists expression*⟩ | ⟨*null expression*⟩ | ⟨*vexp*⟩ | *true* | *false* | *null*
(37)⟨*logical expression*⟩ ::= ⟨*bvexp*⟩ OR ⟨*bvexp*⟩ | ⟨*bvexp*⟩ AND ⟨*bvexp*⟩ | ⟨*bvexp*⟩ XOR⟨*bvexp*⟩ |NOT ⟨*bvexp*⟩
(38)⟨*is expression*⟩ ::= ⟨*bvexp*⟩ IS [NOT] TRUE|FALSE|UNKNOWN
(39)⟨*comparison expression*⟩ ::= ⟨*bvexp*⟩ = |! = |< |> |>= |<= ⟨*bvexp*⟩
(40)⟨*between expression*⟩ ::= ⟨*nvexp*⟩[NOT] BETWEEN ⟨*nvexp*⟩ AND ⟨*nvexp*⟩
(41)⟨*in expression*⟩ ::= ⟨*vexp*⟩ [NOT] IN ⟨*vlist*⟩
(42)⟨*vlist*⟩ ::= (⟨*vexp*⟩[, ⟨*vexp*⟩...])
(43)⟨*exists expression*⟩ ::= EXISTS ⟨*subquery*⟩
(44)⟨*null expression*⟩ ::= ⟨*bvexp*⟩ IS [NOT] NULL
(45)⟨*subquery*⟩ ::= (⟨*query expression*⟩)
(46)⟨*caseexp*⟩ ::= CASE WHEN ⟨*vexp*⟩ THEN ⟨*vexp*⟩ ELSE ⟨*vexp*⟩
(47)⟨*castexp*⟩ ::= CAST(⟨*vexp*⟩ AS ⟨*data type*⟩)
(48)⟨*data type*⟩ ::= *string* | *numeric* | *boolean*
(50)⟨*tname*⟩ ::= *identifier*
(51)⟨*cname*⟩ ::= *identifier*

Fig. 3: The full list of syntax for SQL

Boolean value expression($\mathcal{B} : E_B \mapsto \mathbb{B}$)

1. $\mathcal{B}(\langle bvexp_1\rangle) \triangleq \mathcal{B}(\langle bvexp_1\rangle or \langle bvexp_2\rangle)|\mathcal{B}(\langle bvexp_1\rangle and \langle bvexp_2\rangle)|\mathcal{B}(\langle bvexp_1\rangle xor \langle bvexp_2\rangle)|\mathcal{B}(not \langle bvexp\rangle)|\mathcal{B}(\langle vexp\rangle\ is\ true)$
   $|\mathcal{B}(\langle bvexp\rangle\ is\ false)|\mathcal{B}(\langle bvexp\rangle\ is\ unknown)|\mathcal{B}(\langle bvexp\rangle\ is\ not\ true)|\mathcal{B}(\langle bvexp\rangle\ is\ not\ false)|\mathcal{B}(\langle bvexp\rangle\ is\ not\ unknown)$
   $|\mathcal{B}(\langle nvexp_1\rangle\ =\ \langle nvexp_2\rangle)|\mathcal{B}(\langle nvexp_1\rangle\ !=\ \langle nvexp_2\rangle)|\mathcal{B}(\langle nvexp_1\rangle\ >\ \langle nvexp_2\rangle)|\mathcal{B}(\langle nvexp_1\rangle\ <\ \langle nvexp_2\rangle)$
   $|\mathcal{B}(\langle nvexp_1\rangle\ >=\ \langle nvexp_2\rangle)|\mathcal{B}(\langle nvexp_1\rangle\ <=\ \langle nvexp_2\rangle)|\mathcal{B}(\langle nvexp\rangle\ between\ \langle nvexp_1\rangle\ and\ \langle nvexp_2\rangle)|\mathcal{B}(exists\ \langle subquery\rangle)$
   $|\mathcal{B}(\langle bvexp\rangle\ is\ null)|\mathcal{B}(\langle bvexp\rangle\ is\ not\ null)$
2. $\mathcal{B}(\langle bvexp_1\rangle or \langle bvexp_2\rangle) \triangleq \mathcal{B}(\langle bvexp_1\rangle) \vee \mathcal{B}(\langle bvexp_2\rangle)$
3. $\mathcal{B}(\langle bvexp_1\rangle and \langle bvexp_2\rangle) \triangleq \mathcal{B}(\langle bvexp_1\rangle) \wedge \mathcal{B}(\langle bvexp_2\rangle)$
4. $\mathcal{B}(\langle bvexp_1\rangle xor \langle bvexp_2\rangle) \triangleq \mathcal{B}(\langle bvexp_1\rangle) \oplus \mathcal{B}(\langle bvexp_2\rangle)$
5. $\mathcal{B}(not \langle bvexp\rangle) \triangleq \neg\mathcal{B}(\langle bvexp\rangle)$
6. $\mathcal{B}(\langle vexp\rangle\ is\ true) \triangleq \mathcal{B}(\langle vexp\rangle)\ =\ true$
7. $\mathcal{B}(\langle bvexp\rangle\ is\ false) \triangleq \mathcal{B}(\langle bvexp\rangle)\ =\ false$
8. $\mathcal{B}(\langle bvexp\rangle\ is\ unknown) \triangleq \mathcal{B}(\langle bvexp\rangle)\ =\ null$
9. $\mathcal{B}(\langle bvexp\rangle\ is\ not\ true) \triangleq \mathcal{B}(\langle bvexp\rangle)\ \neq\ true$
10. $\mathcal{B}(\langle bvexp\rangle\ is\ not\ false) \triangleq \mathcal{B}(\langle bvexp\rangle)\ \neq\ false$
11. $\mathcal{B}(\langle bvexp\rangle\ is\ not\ unknown) \triangleq \mathcal{B}(\langle bvexp\rangle)\ \neq\ null$
12. $\mathcal{B}(\langle nvexp_1\rangle\ =\ \langle nvexp_2\rangle) \triangleq \mathcal{N}(\langle nvexp_1\rangle) = \mathcal{N}(\langle nvexp_2\rangle)$
13. $\mathcal{B}(\langle nvexp_1\rangle\ !=\ \langle nvexp_2\rangle) \triangleq \mathcal{N}(\langle nvexp_1\rangle) \neq \mathcal{N}(\langle nvexp_2\rangle)$
14. $\mathcal{B}(\langle nvexp_1\rangle\ >\ \langle nvexp_2\rangle) \triangleq \mathcal{N}(\langle nvexp_1\rangle) > \mathcal{N}(\langle nvexp_2\rangle)$
15. $\mathcal{B}(\langle nvexp_1\rangle\ <\ \langle nvexp_2\rangle) \triangleq \mathcal{N}(\langle nvexp_1\rangle) < \mathcal{N}(\langle nvexp_2\rangle)$
16. $\mathcal{B}(\langle nvexp_1\rangle\ >=\ \langle nvexp_2\rangle) \triangleq \mathcal{N}(\langle nvexp_1\rangle) \geq \mathcal{N}(\langle nvexp_2\rangle)$
17. $\mathcal{B}(\langle nvexp_1\rangle\ <=\ \langle nvexp_2\rangle) \triangleq \mathcal{N}(\langle nvexp_1\rangle) \leq \mathcal{N}(\langle nvexp_2\rangle)$
18. $\mathcal{B}(\langle nvexp\rangle\ between\ \langle nvexp_1\rangle\ and\ \langle nvexp_2\rangle) \triangleq (\mathcal{N}(\langle nvexp\rangle) \leq \mathcal{N}(\langle nvexp_2\rangle)) \wedge (\mathcal{N}(\langle nvexp\rangle) \geq \mathcal{N}(\langle nvexp_1\rangle))$
19. $\mathcal{B}(\langle vexp\rangle\ in\ (\langle vexp_1\rangle[, \langle vexp_2\rangle...])) \triangleq \langle vexp\rangle \in (\langle vexp_1\rangle[, \langle vexp_2\rangle...])$
20. $\mathcal{B}(exists\ \langle subquery\rangle) \triangleq \mathcal{H}[\![\langle subquery\rangle]\!] \neq \oslash$
21. $\mathcal{B}(\langle bvexp\rangle\ is\ null) \triangleq \mathcal{B}(\langle bvexp\rangle)\ =\ null$
22. $\mathcal{B}(\langle bvexp\rangle\ is\ not\ null) \triangleq \mathcal{B}(\langle bvexp\rangle)\ \neq\ null$
23. $\mathcal{B}(\langle vexp\rangle) \triangleq \mathcal{B}(\langle nvexp\rangle)|\mathcal{B}(\langle svexp\rangle)|\mathcal{B}(\langle caseexp\rangle)|\mathcal{B}(\langle castexp\rangle)|\mathcal{B}(\langle cname\rangle)|\mathcal{B}(null)$
24. $\mathcal{B}(\langle nvexp\rangle) \triangleq \mathcal{B}(cast(\langle nvexp\rangle\ as\ boolean))$
25. $\mathcal{B}(\langle svexp\rangle) \triangleq \mathcal{B}(cast(\langle svexp\rangle\ as\ boolean))$
26. $\mathcal{B}(caseexp) \triangleq \mathcal{B}(case\ when\ \langle bvexp\rangle\ then\ \langle vexp_1\rangle\ else\ \langle vexp_2\rangle)$
27. $\mathcal{B}(case\ when\ \langle bvexp\rangle\ then\ \langle vexp_1\rangle\ else\ \langle vexp_2\rangle) \triangleq \begin{cases} \mathcal{B}(\langle vexp_1\rangle); & \mathcal{B}(\langle bvexp\rangle) = true \\ \mathcal{B}(\langle vexp_2\rangle); & \mathcal{B}(\langle bvexp\rangle) = false \end{cases}$
28. $\mathcal{B}(\langle castexp\rangle) \triangleq \mathcal{B}(cast(\langle nvexp\rangle\ as\ boolean))|\mathcal{B}(cast(\langle svexp\rangle\ as\ boolean))$
29. $\mathcal{B}(cast(\langle nvexp\rangle\ as\ boolean)) \triangleq \begin{cases} null; & \mathcal{N}(\langle nvexp\rangle) = null \\ false; & \mathcal{N}(\langle nvexp\rangle) = 0 \\ true; & otherwise \end{cases}$
30. $\mathcal{B}(cast(\langle svexp\rangle\ as\ boolean)) \triangleq \begin{cases} null; & \mathcal{S}(\langle svexp\rangle) = null \\ false; & \mathcal{S}(\langle svexp\rangle) = \text{“0”}|\text{“}false\text{”}|\text{“”} \\ true; & otherwise \end{cases}$
31. $\mathcal{B}(\langle cname\rangle) \triangleq \mathcal{B}(cast(S(\langle cname\rangle)\ as\ boolean))$
32. $\mathcal{B}(true) \triangleq true$
33. $\mathcal{B}(false) \triangleq false$
34. $\mathcal{B}(null) \triangleq null$

Fig. 4: The full list semantic definition of boolean expression

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 [32], [44].

III. Formal semantics of SQL

igure 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) [9].

We categorize the denotational semantics of the SQL language into two categories, i.e., SQL expression semantics and SQL keyword semantics. SQL expressions are basic elements used to construct constraints in queries, while SQL keywords form the main logic of SQL statements. We adopt the bag semantics of SQL according to the SQL specification, allowing duplicate elements in the result set. Moreover, we support the `null` semantic, which is considered as a special unkown value

Numeric value expression($\mathcal{N} : E_N \mapsto \mathbb{N}$)

35. $\mathcal{N}(\langle nvexp \rangle) \triangleq \mathcal{N}(\langle nvexp_1 \rangle + \langle nvexp_2 \rangle) | \mathcal{N}(\langle nvexp_1 \rangle - \langle nvexp_2 \rangle) | \mathcal{N}(\langle nvexp_1 \rangle * \langle nvexp_2 \rangle) | \mathcal{N}(\langle nvexp_1 \rangle / \langle nvexp_2 \rangle)$
$| \mathcal{N}(length | char\_length | charactor\_length(\langle svexp \rangle) | \mathcal{N}(mod(\langle nvexp_1 \rangle, \langle nvexp_2 \rangle) | \mathcal{N}(abs(\langle nvexp \rangle) | \mathcal{N}(ln(\langle nvexp \rangle))$
$| \mathcal{N}(exp(\langle nvexp \rangle) | \mathcal{N}(power(\langle nvexp_1 \rangle, \langle nvexp_2 \rangle) | \mathcal{N}(sqrt(\langle nvexp \rangle)) | \mathcal{N}(floor(\langle nvexp \rangle)) | \mathcal{N}(ceil | ceiling(\langle nvexp \rangle))$
36. $\mathcal{N}(\langle nvexp_1 \rangle + \langle nvexp_2 \rangle) \triangleq \mathcal{N}(\langle nvexp_1 \rangle) + \mathcal{N}(\langle nvexp_2 \rangle)$
37. $\mathcal{N}(\langle nvexp_1 \rangle - \langle nvexp_2 \rangle) \triangleq \mathcal{N}(\langle nvexp_1 \rangle) - \mathcal{N}(\langle nvexp_2 \rangle)$
38. $\mathcal{N}(\langle nvexp_1 \rangle * \langle nvexp_2 \rangle) \triangleq \mathcal{N}(\langle nvexp_1 \rangle) * \mathcal{N}(\langle nvexp_2 \rangle)$
39. $\mathcal{N}(\langle vexp_1 \rangle / \langle nvexp_2 \rangle) \triangleq \mathcal{N}(\langle nvexp_1 \rangle) / \mathcal{N}(\langle nvexp_2 \rangle)$
40. $\mathcal{N}(length | char\_length | charactor\_length(\langle svexp \rangle) \triangleq len(\mathcal{S}(\langle svexp \rangle))$
41. $\mathcal{N}(mod(\langle nvexp_1 \rangle, \langle nvexp_2 \rangle) \triangleq (\mathcal{N}(\langle nvexp_1 \rangle) \% \mathcal{N}(\langle nvexp_2 \rangle)$
42. $\mathcal{N}(abs(\langle nvexp \rangle) \triangleq |\mathcal{N}(\langle nvexp \rangle)|$
43. $\mathcal{N}(ln(\langle nvexp \rangle) \triangleq ln(\mathcal{N}(\langle nvexp \rangle))$
44. $\mathcal{N}(exp(\langle nvexp \rangle) \triangleq e^{\mathcal{N}(\langle nvexp \rangle)}$
45. $\mathcal{N}(power(\langle nvexp_1 \rangle, \langle nvexp_2 \rangle) \triangleq \mathcal{N}(\langle nvexp_1 \rangle)^{\mathcal{N}(\langle nvexp_2 \rangle)}$
46. $\mathcal{N}(sqrt(\langle nvexp \rangle) \triangleq \sqrt{\mathcal{N}(\langle nvexp \rangle)}$
47. $\mathcal{N}(floor(\langle nvexp \rangle) \triangleq \lfloor \mathcal{N}(\langle nvexp \rangle) \rfloor$
48. $\mathcal{N}(ceil | ceiling(\langle nvexp \rangle) \triangleq \lceil \mathcal{N}(\langle nvexp \rangle) \rceil$
49. $\mathcal{N}(\langle vexp \rangle) \triangleq \mathcal{N}(\langle bvexp \rangle) | \mathcal{N}(\langle svexp \rangle) | \mathcal{N}(\langle caseexp \rangle) | \mathcal{N}(\langle castexp \rangle) | \mathcal{N}(\langle cname \rangle) | \mathcal{N}(null)$
50. $\mathcal{N}(\langle bvexp \rangle) \triangleq \mathcal{N}(cast(\langle bvexp \rangle \ as \ numeric))$
51. $\mathcal{N}(\langle svexp \rangle) \triangleq \mathcal{N}(cast(\langle svexp \rangle \ as \ numeric))$
52. $\mathcal{N}(\langle cname \rangle) \triangleq \mathcal{N}(cast(S(\langle cname \rangle) \ as \ numeric))$
53. $\mathcal{N}(caseexp) \triangleq \mathcal{N}(case \ when \ \langle bvexp \rangle \ then \ \langle vexp_1 \rangle \ else \ \langle vexp_2 \rangle)$
54. $\mathcal{N}(case \ when \ \langle bvexp \rangle \ then \ \langle vexp_1 \rangle \ else \ \langle vexp_2 \rangle) \triangleq \begin{cases} \mathcal{N}(\langle vexp_1 \rangle); & \mathcal{B}(\langle bvexp \rangle) = true \\ \mathcal{N}(\langle vexp_2 \rangle); & \mathcal{B}(\langle bvexp \rangle) = false \end{cases}$
55. $\mathcal{N}(\langle castexp \rangle) \triangleq \mathcal{N}(cast(\langle bvexp \rangle \ as \ numeric)) | \mathcal{N}(cast(\langle svexp \rangle \ as \ numeric))$
56. $\mathcal{N}(cast(\langle bvexp \rangle \ as \ numeric)) \triangleq \begin{cases} 1; & \mathcal{B}(\langle bvexp \rangle) = true \\ 0; & \mathcal{B}(\langle bvexp \rangle) = false \\ null; & \mathcal{B}(\langle bvexp \rangle) = null \end{cases}$
57. $\mathcal{N}(cast(\langle svexp \rangle \ as \ numeric)) \triangleq \begin{cases} null; & \mathcal{S}(\langle svexp \rangle) = null \\ str2num(\mathcal{S}(\langle svexp \rangle)); & otherwise \end{cases}$
58. $\mathcal{N}(numeric \ literal) \triangleq numeric \ literal$
59. $\mathcal{N}(null) \triangleq null$

String value expression($\mathcal{S} : E_S \mapsto \mathbb{S}$)

60. $\mathcal{S}(\langle svexp \rangle) \triangleq \mathcal{S}(\langle svexp_1 \rangle || \langle svexp_2 \rangle) | \mathcal{S}(substring(\langle svexp_1 \rangle \ in \ \langle nvexp_2 \rangle)) | \mathcal{S}(ltrim(\langle svexp \rangle)) | \mathcal{S}(rtrim(\langle svexp \rangle))$
$| \mathcal{S}(lower(\langle svexp \rangle)) | \mathcal{S}(upper(\langle svexp \rangle))$
61. $\mathcal{S}(\langle svexp_1 \rangle || \langle svexp_2 \rangle) \triangleq \mathcal{S}(\langle svexp_1 \rangle) || \mathcal{S}(\langle svexp_2 \rangle)$
62. $\mathcal{S}(substring(\langle svexp_1 \rangle \ in \ \langle nvexp_2 \rangle)) \triangleq \mathcal{S}(\langle svexp_1 \rangle)[\mathcal{N}(\langle nvexp_2 \rangle), len(\mathcal{S}(\langle svexp_1 \rangle))]$
63. $\mathcal{S}(lower(\langle svexp \rangle)) \triangleq s_{lower}$
$s_{lower} : (len(s_{lower}) = len(\mathcal{S}(\langle svexp \rangle))) \wedge (\forall i \in (0, len(\mathcal{S}(\langle svexp \rangle))), s_{lower}[i] = \mathcal{S}(\langle svexp \rangle)[i] + 32$
64. $\mathcal{S}(upper(\langle svexp \rangle)) \triangleq s_{upper}$
$s_{upper} : (len(s_{upper}) = len(\mathcal{S}(\langle svexp \rangle))) \wedge (\forall i \in (0, len(\mathcal{S}(\langle svexp \rangle))), s_{upper}[i] = \mathcal{S}(\langle svexp \rangle)[i] - 32$
65. $\mathcal{S}(\langle vexp \rangle) \triangleq \mathcal{S}(\langle bvexp \rangle) | \mathcal{S}(\langle nvexp \rangle) | \mathcal{S}(\langle caseexp \rangle) | \mathcal{S}(\langle castexp \rangle) | \mathcal{S}(\langle cname \rangle) | \mathcal{S}(null)$
66. $\mathcal{S}(\langle bvexp \rangle) \triangleq \mathcal{S}(cast(\langle bvexp \rangle \ as \ string))$
67. $\mathcal{S}(\langle nvexp \rangle) \triangleq \mathcal{S}(cast(\langle nvexp \rangle) \ as \ string)$
68. $\mathcal{S}(caseexp) \triangleq \mathcal{S}(case \ when \ \langle bvexp \rangle \ then \ \langle vexp_1 \rangle \ else \ \langle vexp_2 \rangle)$
69. $\mathcal{S}(case \ when \ \langle bvexp \rangle \ then \ \langle vexp_1 \rangle \ else \ \langle vexp_2 \rangle) \triangleq \begin{cases} \mathcal{S}(\langle vexp_1 \rangle); & \mathcal{B}(\langle bvexp \rangle) = true \\ \mathcal{S}(\langle vexp_2 \rangle); & \mathcal{B}(\langle bvexp \rangle) = false \end{cases}$
70. $\mathcal{S}(\langle castexp \rangle) \triangleq \mathcal{S}(cast(\langle bvexp \rangle \ as \ string)) | \mathcal{S}(cast(\langle nvexp \rangle \ as \ string))$
71. $\mathcal{S}(cast(\langle bvexp \rangle \ as \ string)) \triangleq \begin{cases} 1; & \mathcal{B}(\langle bvexp \rangle) = true \\ 0; & \mathcal{B}(\langle bvexp \rangle) = false \\ null; & \mathcal{B}(\langle bvexp \rangle) = null \end{cases}$
72. $\mathcal{S}(cast(\langle nvexp \rangle \ as \ string)) \triangleq \begin{cases} null; & \mathcal{N}(\langle nvexp \rangle) = null \\ num2str(\mathcal{N}(\langle nvexp \rangle)); & otherwise \end{cases}$
73. $\mathcal{S}(\langle cname \rangle) \triangleq \mathcal{S}(cast(S(\langle cname \rangle) \ as \ string))$
74. $\mathcal{S}(string \ literal) \triangleq string \ literal$
75. $\mathcal{S}(null) \triangleq null$

Fig. 5: The full list of semantic definition for numeric expression and string expression

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

Join operation

76. $\mathcal{C}[\![\{[T_1,T_2],\ natural\ join\}]\!] \triangleq \{\alpha_1\circ\alpha_2|\alpha_1\in T_1,\alpha_2\in T_2,\overline{\beta}_I=T_1.\overline{\beta}\cap T_2.\overline{\beta}\}$
$$\alpha_1\circ\alpha_2\triangleq\begin{cases}\alpha_1\ \bowtie\alpha_2; & (\overline{\beta}_I\neq\emptyset)\wedge(\pi_{\overline{\beta}_I}(\{\alpha_1\})=\pi_{\overline{\beta}_I}(\{\alpha_2\}))\\ skip; & otherwise\end{cases}$$
77. $\mathcal{C}[\![\{[T_1,T_2],\ left\ join\}]\!] \triangleq \{\alpha_1\bullet\alpha_2|\alpha_1\in T_1,\alpha_2\in T_2,\overline{\beta}_I=T_1.\overline{\beta}\cap T_2.\overline{\beta}\}$
$$\alpha_1\bullet\alpha_2\triangleq\begin{cases}\alpha_1\ \bowtie\alpha_2; & (\overline{\beta}_I\neq\emptyset)\wedge(\pi_{\overline{\beta}_I}(\{\alpha_1\})=\pi_{\overline{\beta}_I}(\{\alpha_2\}))\\ \alpha_1\bowtie[null,...,null]_{|\{\alpha_2\}.\overline{\beta}|-|\overline{\beta}_I|}; & (\overline{\beta}_I\neq\emptyset)\wedge(\pi_{\overline{\beta}_I}(\{\alpha_1\})\neq\pi_{\overline{\beta}_I}(\{\alpha_2\}))\\ skip; \quad otherwise\end{cases}$$
78. $\mathcal{C}[\![\{[T_1,T_2],\ right\ join\}]\!] \triangleq \{\alpha_1\bullet\alpha_2|\alpha_1\in T_1,\alpha_2\in T_2,\overline{\beta}_I=T_1.\overline{\beta}\cap T_2.\overline{\beta}\}$
$$\alpha_1\bullet\alpha_2\triangleq\begin{cases}\alpha_1\ \bowtie\alpha_2; & (\overline{\beta}_I\neq\emptyset)\wedge(\pi_{\overline{\beta}_I}(\{\alpha_1\})=\pi_{\overline{\beta}_I}(\{\alpha_2\}))\\ [null,...,null]_{|\{\alpha_1\}.\overline{\beta}|-|\overline{\beta}_I|}\bowtie\alpha_2; & (\overline{\beta}_I\neq\emptyset)\wedge(\pi_{\overline{\beta}_I}(\{\alpha_1\})\neq\pi_{\overline{\beta}_I}(\{\alpha_2\}))\\ skip; \quad otherwise\end{cases}$$
79. $\mathcal{C}[\![\{[T_1,T_2],\ cross\ join\}]\!] \triangleq \{\alpha_1\times\alpha_2|\alpha_1\in T_1,\alpha_2\in T_2\}$
80. $\mathcal{C}[\![\{[T_1,T_2],\ inner\ join\}]\!] \triangleq \mathcal{C}[\![\{[T_1,T_2],\ cross\ join\}]\!]$

Collection operation

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

Filter operation

84. $\mathcal{C}[\![\{[T],\ distinct\}]\!] \triangleq \{\alpha|(\forall\alpha\in T,\ \xi_\alpha(T_1)=1)\wedge(\forall\alpha\in T_1,\ \alpha\in T)\}$
85. $\mathcal{C}[\![\{[T],\ all\}]\!] \triangleq T$

Aggregation operation

86. $\mathcal{C}[\![\{[T],\ max\}]\!] \triangleq \{v|(v\in\pi_{T.\overline{\beta}}(T))\wedge(\forall v_1\in\pi_{T.\overline{\beta}}(T),\sigma_{(v_1>v)}(\pi_{T.\overline{\beta}}(T))=\oslash)\}$
87. $\mathcal{C}[\![\{[T],\ min\}]\!] \triangleq \{v|(v\in\pi_{T.\overline{\beta}}(T))\wedge(\forall v_1\in\pi_{T.\overline{\beta}}(T),\sigma_{(v_1<v)}(\pi_{T.\overline{\beta}}(T))=\oslash)\}$
88. $\mathcal{C}[\![\{[T],\ sum\}]\!] \triangleq \{v|v=\sum_{v_1\in\pi_{T.\overline{\beta}}(T)}v_1\}$
89. $\mathcal{C}[\![\{[T],\ count\}]\!] \triangleq \{v|v=\sum_{\alpha\in\pi_{T.\overline{\beta}}(T)}(\xi_\alpha(T))\}$
90. $\mathcal{C}[\![\{[T],\ avg\}]\!] \triangleq \{v|v=\sum_{v_1\in\pi_{T.\overline{\beta}}(T)}v_1/\sum_{\alpha\in\pi_{T.\overline{\beta}}(T)}(\xi_\alpha(T))\}$

Fig. 6: The full list of semantic definitions for join operation, collection operation, filtering operation, and aggregation operation

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

91. $\mathcal{H}[\![from\langle tname\rangle]\!] \triangleq (A, A(\langle tname\rangle))$
92. $\mathcal{H}[\![from\langle tname_1\rangle, \langle tname_2\rangle]\!] \triangleq (A, \mathcal{C}[\![A(\langle tname_1\rangle), A(\langle tname_2\rangle)],\ cross\ join\ ]\!])$
93. $\mathcal{H}[\![select\ *]\!] \triangleq (\{T\}, \pi_{T.\overline{\beta}}(T))$
94. $\mathcal{H}[\![select\ \langle cname\rangle\ [,\ \langle cname\rangle ...]]\!] \triangleq (\{T\}, \pi_{\langle cname\rangle\ [,\ \langle cname\rangle ...]}(T))$
95. $\mathcal{H}[\![on\ \langle bvexp\rangle]\!] \triangleq (\{T\}, \sigma_{\mathcal{B}(\langle bvexp\rangle)}(T))$
96. $\mathcal{H}[\![where\ \langle bvexp\rangle]\!] \triangleq (\{T\}, \sigma_{\mathcal{B}(\langle bvexp\rangle)}(T))$
97. $\mathcal{H}[\![group\ by\ \langle cname\rangle]\!] \triangleq (\{T\}, (\widehat{\alpha_1}, ..., \widehat{\alpha_k}) : \forall \widehat{\alpha_p} \in\ (\widehat{\alpha_1}, ..., \widehat{\alpha_k}), \forall\ v_{ij} \in\ \pi_{\langle cname\rangle}(\widehat{\alpha_p}), (v_{ij} = v_{i1}))$
98. $\mathcal{H}[\![having\ \langle bvexp\rangle]\!] \triangleq (\{T\}, \sigma_{\mathcal{B}(\langle bvexp\rangle)}(T))$
99. $\mathcal{H}[\![order\ by\ \langle cname\rangle\ asc]\!] \triangleq (\{T\}, T_1)\ where\ (\forall \alpha \in T_1,\ \xi_\alpha(T_1) = \xi_\alpha(T)) \wedge (\forall \alpha \in T,$ $\xi_\alpha(T) = \xi_\alpha(T_1)) \wedge (\forall v_i, v_j \in \sigma_{T_1.\langle cname\rangle}(T_1), i > j$ iff $v_i >= v_j)$
100. $\mathcal{H}[\![order\ by\ \langle cname\rangle\ desc]\!] \triangleq (\{T\}, T_1)\ where\ (\forall \alpha \in T_1,\ \xi_\alpha(T_1) = \xi_\alpha(T)) \wedge (\forall \alpha$ $\in T, \xi_\alpha(T) = \xi_\alpha(T_1)) \wedge (\forall v_i, v_j \in \sigma_{T_1.\langle cname\rangle}(T_1), i < j$ iff $v_i <= v_j)$
101. $\mathcal{H}[\![\langle subquery\rangle]\!] \triangleq \mathcal{H}[\![\langle query\ expression\rangle]\!]$

Composite keyword operation

102. $\diamond$: (1) $\dfrac{\mathcal{H}[\![expression_1]\!] \triangleq (A, T),\ \mathcal{H}[\![expression_2]\!] \triangleq (\{T\}, T')}{\mathcal{H}[\![expression_1]\!] \diamond \mathcal{H}[\![expression_2]\!] \triangleq (A, T')}$

(2) $\dfrac{\mathcal{H}[\![expression]\!] \triangleq (A, T),\ \mathcal{C}[\![\{\{T\}, OP\}]\!] \triangleq T'}{\mathcal{H}[\![expression]\!] \diamond \mathcal{C}[\![\{\{T\}, OP\}]\!] \triangleq (A, T')}$

(3) $\dfrac{\mathcal{C}[\![\{L, OP\}]\!] \triangleq T,\ \mathcal{H}[\![expression]\!] \triangleq (\{T\}, T')}{\mathcal{C}[\![\{L, OP\}]\!] \diamond \mathcal{H}[\![expression]\!] \triangleq (L, T')}$

(4) $\dfrac{\mathcal{H}[\![expression_1]\!] \triangleq (A, T_1),\ \mathcal{H}[\![expression_2]\!] \triangleq (A, T_2),\ \mathcal{C}[\![\{\{T_1, T_2\}, OP\}]\!] \triangleq T_3}{(\mathcal{H}[\![expression_1]\!], \mathcal{H}[\![expression_2]\!]) \diamond \mathcal{C}[\![\{\{T_1, T_2\}, OP\}]\!] \triangleq (A, T_3)}$

103. $\mathcal{H}[\![\langle queryexp\rangle]\!] =$
$\mathcal{H}[\![select\ [\langle sop\rangle|\langle af\rangle]\ \langle cname_1\rangle[, \langle cname_2\rangle ...]\ from\langle tname_1\rangle[, \langle tname_2\rangle ...]|from\ \langle tname_1\rangle\ natural/cross\ join\ \langle tname_2\rangle$
$|from\ \langle tname_1\rangle\ left/right/full/inner\ join\ \langle tname_2\rangle\ on\langle bvexp\rangle\ [where\langle bvexp\rangle]\ [group\ by\ \langle cname\rangle]\ [having\ \langle bvexp\rangle]$
$[order\ by\ \langle cname\rangle\ [asc|desc]]]\!]$
$\triangleq$
$\mathcal{H}[\![from\langle tname_1\rangle]\!]$
$|(\mathcal{H}[\![from\ \langle tname_1\rangle]\!], \mathcal{H}[\![from\ \langle tname_2\rangle]\!]) \diamond \mathcal{C}[\![L, natural/cross\ join\ ]\!]$
$|(\mathcal{H}[\![from\ \langle tname_1\rangle]\!], \mathcal{H}[\![from\ \langle tname_2\rangle]\!]) \diamond \mathcal{C}[\![L, left/right/inner/full\ join\ ]\!] \diamond \mathcal{H}[\![on\ \langle bvexp\rangle]\!]$
$[\diamond\ \mathcal{H}[\![where\ \langle bvexp\rangle]\!]]$
$[\diamond \mathcal{H}[\![group\ by\ \langle cname\rangle]\!]]$
$[\diamond \mathcal{H}[\![having\ \langle bvexp\rangle]\!]]$
$\diamond\ \mathcal{H}[\![select\ \langle cname_1\rangle[, \langle cname_2\rangle ...]]\!][\diamond \mathcal{C}[\![\{T_1, \langle sop/af\rangle\}]\!]]$
$[\diamond\ \mathcal{H}[\![order\ by\ \langle cname\rangle\ [asc|desc]]\!]]$
104. $\mathcal{H}[\![\langle queryexp_1\rangle\ \langle cop\rangle\ \langle queryexp_2\rangle]\!] \triangleq (\mathcal{H}[\![\langle queryexp_1\rangle]\!], \mathcal{H}[\![\langle queryexp_2\rangle]\!]) \diamond \mathcal{C}[\![\{[T_1, T_2], \langle cop\rangle\}]\!]$

Fig. 7: The full list of semantic definitions for SQL keywords `from`, `select`, `on`, `group by`, `having`, `order by`, subquery operations, and composite semantics on SQL queries

in SQL. We manually analyze the semantics of keywords in the SQL specification and defined the denotational semantics for 138 SQL keywords or features related to query functionality in the SQL specification. In the following, we first introduce the basic symbols defined for our semantic definitions, then we introduce the semantics of SQL expressions and SQL keywords in detail.

A. Basic symbol definition

Table I 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 defined in SQL. The operations of data projection, 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.

Furthermore, this work uses $[a, ..., a]_n$ to denote a list of n occurrences of a (which can be a data record or a constant), $max(a_1, a_2)$ and $min(a_1, a_2)$ to represent the operation of finding the maximum or minimum value between a_1 and a_2. The functions $num2str(n)$ and $str2num(s)$ are used for converting between numeric and string types, $type(a)$ returns the type of data record a.

B. Semantics of Expressions

Fig 4 and 5 show our semantic definition for the expressions in SQL. In SQL, expressions are essential structural elements used to construct queries and compute return values. The

TABLE I: Symbol notations

Symbol	Description
$\mathtt{T}$	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
$\mathtt{T}.\beta_i$	The i_{th} attribute in $\mathtt{T}$
$\mathtt{T}.\overline{\beta}$	All attributes in $\mathtt{T}$
$\pi_{\beta}(\mathtt{T})$	Data projection on attribute β
$\sigma_{\Theta}(\mathtt{T})$	Data selection on expression Θ
$\xi_{\alpha}(\mathtt{T})$	The multiplicity of data record α in table $\mathtt{T}$
$\xi_{\beta}(\mathtt{T})$	The multiplicity of attribute β in table $\mathtt{T}$
$[a, ..., a]_n$	A list of element a with length n
$\bowtie$	Data join operation
$\times$	Cartesian product operation
$max(a_1, a_2)$	The maximum value in $a_1$and a_2
$min(a_1, a_2)$	The minimum value in $a_1$and a_2
$num2str(n)$	Converting the numeric type n to a string
$str2num(s)$	Converting the string type s to a number
$type(a)$	The data type of a
$[a, ..., a]_n$	A data list of a of length n

complexity of an expression can vary, ranging from simple numeric or string values to complex combinations of functions, operators, and subqueries. We categorize the semantics of SQL expressions into boolean, numeric, and string value expression according to the specific return value type.

Definition 1 (Boolean value expression ($\mathcal{B} : E_B \mapsto \mathbb{B}$)) Fig 4 (Semantic rules 1-34) lists the semantics of SQL expressions, including logical expressions, comparison expressions, IS expressions, BETWEEN expressions, IN expressions, NULL expressions, EXISTS expressions, and relevant functionalities that perform implicit type conversions on other types of expressions, that return boolean values. The function $\mathcal{B}$ maps from the domain of boolean value expressions E_B to the domain of Boolean values $\mathbb{B}$.

Definition 2 (Numeric value expression ($\mathcal{N} : E_N \mapsto \mathbb{N}$)) Fig 5 (semantics rules 35-59) show the semantics of SQL expressions, including arithmetic expressions, numeric functions, and relevant functionalities that perform implicit type conversions on the results of other types of expressions, that return numeric values. The function $\mathcal{N}$ maps from the domain of numeric value expressions E_N to the domain of numeric values $\mathbb{N}$.

Definition 3 (String value expression ($\mathcal{S} : E_S \mapsto \mathbb{S}$)) Fig 5 (Semantic rules 60-75) shows the semantic rules of SQL expressions, including include string concatenation expressions, string functions, and relevant functionalities that perform implicit type conversions on the results of other types of expressions, that return string values. The function $\mathcal{S}$ maps from the domain of string value expression E_S to the domian of string values $\mathbb{S}$.

C. 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 the set of all tables and T is the resulting table.

Definition 4 (Keyword operation ($\mathcal{C} : \{L, OP\} \mapsto T$)) Fig 6 shows the semantic definition of four SQL keywords, i.e., JOIN operations (semantic rules 76-80), set operations (semantic rules 81-83), filter operations (semantic rules 84-85), and aggregate operations (semantic rules 86-90). These operations take a list of table instances L and an operation type OP as input and produce a new table instance T as output.

Definition 5 (Keyword operation ($\mathcal{H} : S \mapsto (A, T)$)) Fig 7 shows the semantics of the second category of SQL keywords (semantic rules 91-101), including keywords such as FROM, WHERE, ON, SELECT, GROUP BY, HAVING, ORDER BY, and their combinations in SQL statements. 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. T represents the table obtained after the execution.

Definition 6 (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 7, we first introduced the combination rules of different types of keyword semantics (semantic rule 102), and then we introduce the composite semantics based on those rules (semantic rules 103-104).

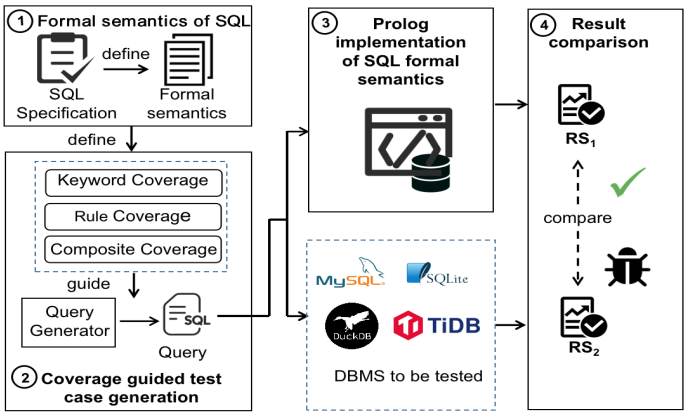


Fig. 8: Overview of SEMCONT

IV. CONFORMANCE TESTING

A. Overview of our approach

Figure 8 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 III. 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 [14], 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 two types of issues, i.e., bugs or inconsistencies. Both issues 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.

B. 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 [14] 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 keywords that are covered by the generated test queries and N_K the total number of keywords defined in our semantics.

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

To calculate keyword coverage, we count the total non-repeated occurrences of keywords in all test queries as the number of N_{tk}. In our case, N_K, which represents the total count of SQL keywords defined by our semantics, is 138.

Algorithm 1: Calculating the number of composite rules

```
Input : grammar: SQL grammar
Output: N_CR: the number of composite rules
1  Initialize N_CR=0, path=empty
2  Function CRNumberCalculation(grammar):
3      TraverseGrammar (grammar.headNode, path)
4      return N_CR
5  Function TraverseGrammar(node, path):
6      if node is Non-terminal then
7          if node is SQLKeyword then
8              path.add(node)
9          end
10         if node.children is not Null then
11             foreach child in node.children do
12                 TraverseGrammar(child, path)
13             end
14         end
15     end
16     else
17         N_pcr = 1
18         foreach Keyword in path do
19             N_pcr = N_pcr * Keyword.ruleNumber
20         end
21         N_CR = N_CR + N_pcr
22     end
23     path.remove(node)
```

Rule coverage. The semantics of each SQL keyword, as defined in Prolog, is often captured using 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). The total number of rules, N_R, is the sum of all rules for all keywords we have 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.

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

Note that we retain all duplicate keywords when collecting the semantic rules, as SQL queries with the same keyword may trigger different rules. We only remove duplicate semantic rules after collecting the complete set. For instance, as shown in Figure 2, the semantics for the keyword `SELECT` include three rules, i.e., line 4 for `SELECT NULL`, lines 5-8 for the case of a constant value input, and lines 9-11 for the case of column selection in the table. The SQL query `SELECT t.b FROM t` covers the third `SELECT` rule and the first `FROM` rule, resulting a rule coverage of 2/420, where 420 is the total number of different rules defined in our semantics.

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.

Algorithm 2: 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()
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
```

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

Algorithm 1 outlines the process of calculating the total number of composite rules. The input to the algorithm is the SQL grammar, and its output is the total number of composite rules (N_{CR}). The *path* variable declared in line 1 records each traversed paths, which are SQL keyword sequences according to grammar rules. Our algorithm conducts a depth-first traversal following the SQL grammar rules by recursively calling the `TraverseGrammar` function from the headnode of the grammar (lines 2-4). `TraverseGrammar` initially checks whether a node in the grammar is a non-terminal node. If it is both a non-terminal node and a SQL keyword, we include it in the path. A depth-first search is then performed by recursively invoking function `CRNumberCalculation` (lines 11-13). When a leaf node is incurred, we calculate the composite rules of the path by multiplying the number of rules of each keyword, i.e., node in the path (lines 18-20). The composite rules for this path are then added to the total number of composite rules (line 21). Upon completion of the traversal, the current value of N_{CR} is returned.

Take the statement `SELECT t.b FROM t` and the Prolog rules in Figure 2 as an example. To identify the composite rules covered by this query, we enumerate the rules of the `SELECT` and `FROM` keywords and match the rules based on the parameter or input of that keyword. In this case, the `SELECT` rule that takes a column list as input and the `FROM` rule that takes one table as input are matched to form the composite rule covered by the given query.

The time complexity of Algorithm 3 is O(n) with n being the number of operators in the given SQL query.

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 (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 [37], 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 II 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.

C. Prolog implementation of SQL formal semantics

We have implemented the formal semantics of SQL defined in Figure 4-7 in Prolog. The semantics of each keyword are implemented as a set of rules, as illustrated in Figure 2. We then implement the compound semantics in Algorithm 3, 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 3 initially

TABLE II: 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)~~

Algorithm 3: 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
```

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, which invokes 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 to the implementation documentation of current mainstream database management systems in our Prolog implementation of semantics. In cases where there were inconsistencies among different database implementations, we chose to adopt the approach 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 II, Prolog, being declarative, is naturally suited to specify denotational semantics we defined. Taking the 'select' keyword as an example, rule 19 of Figure 6 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 and code review following the software engineering standard procedure, covering all the semantic rules we've

implemented. We also conducted thorough experiments with 6 different RDBMS systems, including MySQL, PostgreSQL, TiDB, SQLite, DuckDB and OceanBase, validating the correctness our implementation using 18 millions of test cases.

D. Result comparison

The final part of our method involves result comparison. This process 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 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.

E. 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 [9] 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.

TABLE III: 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

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.

V. EVALUATION

In this section, 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.

A. 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 six 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 III. It is important to note that we used the latest release of each RDBMS, which has been extensively tested by existing approaches [40], [41]. MySQL [3] and PostgreSQL [5] are the two most popular open-source database management systems. SQLite [6] and DuckDB [13] are both embedded DBMSs, running within the process of other applications. TiDB [12] and OceanBase [11] are popular distributed RDBMSs.

TABLE IV: 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

Compared baselines. We compared SEMCONT with TLP [41] and NoREC [40], 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 [14] and SQLRight [34]. 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).

B. 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 IV 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. 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 [34], [40]–[42]. 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 [34], [40]–[42], making it more challenging to uncover new inconsistencies.

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) 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 [40]–[42], 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*" [30]. 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 V-B 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. The first type of inconsistency violates the SQL specification, and the second type of inconsistency arises from variations in the implementations across different RDBMSs 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 [30], [31]. 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, which results 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.

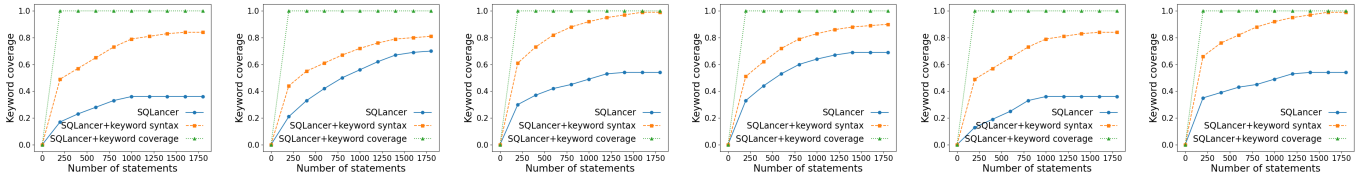
RQ2: Effectiveness of the coverage criteria? We measure the effectiveness of the proposed coverage criteria in two aspects, i.e., whether they are effective in guiding generating test cases that achieve higher coverage, and whether they are effective in guiding generating test cases that uncover unknown bugs or inconsistencies. The experiment results on keyword coverage, rule coverage and composite rule coverage improvement are shown in Figure 9, Figure 10 and Figure 11, respectively.

In Figure 9, 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 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 10 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 10. 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.

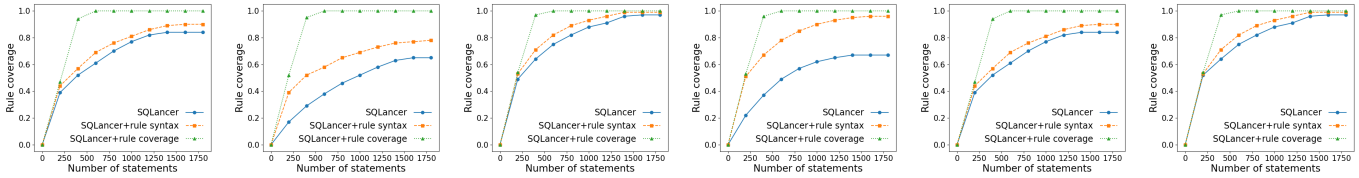
Figure 11 depicts the improvements in composite rule coverage by the generated queries for the four databases. With SQLancer, which conducts random query generation, we observed that the increase in composite rule coverage tends to plateau after generating 60 million data points. At this stage, MySQL, SQLite, 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 V] 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



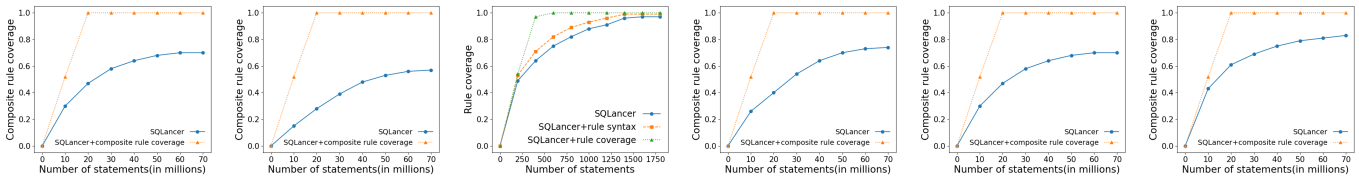
(a) MySQL (b) TiDB (c) SQLite (d) DuckDB (e) OceanBase (f) PostgreSQL

Fig. 9: 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

Fig. 10: The rule coverage increment (y-axis) with the number of queries (x-axis)



(a) mysql (b) TiDB (c) SQLite (d) DuckDB (e) OceanBase (f) PostgreSQL

Fig. 11: The composite rule coverage increment (y-axis) with the number of queries (in million).

TABLE V: 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

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: How does SEMCONT perform compared to baseline approaches?

We compare SEMCONT with two state-of-the-art approaches TLP [40] and NoREC [41], which are metamorphic testing approaches for relational DBMS. For both approaches, we adopt SQLancer [14] and SQLRight [34] 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 VI.

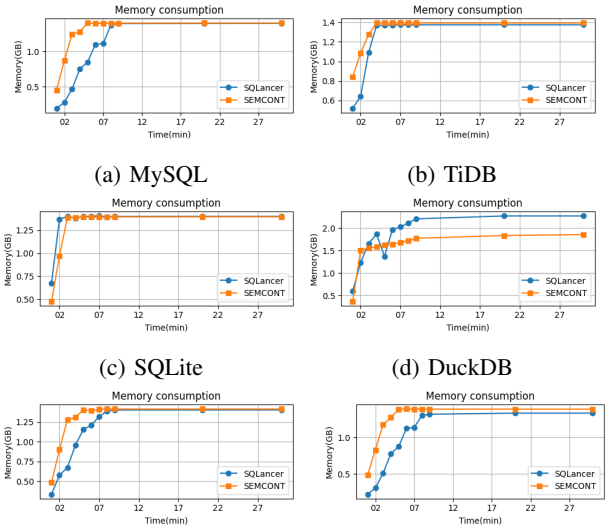
The experimental results show that both SQLancer and SQLRight using the NoREC as the oracle were unable to detect new bugs or inconsistencies. The TLP oracle with SQLancer for query generation detected 4 bugs in three databases, and with SQLRight for query generation detectd 1 bug in MySQL. SEMCONT outperformed the compared approaches and detected 16 bugs and 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.

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

TABLE VI: The bugs and inconsistencies detected by SQLancer, SQLRight and SemCon T in 6h

DBMS	TLP (SQLancer)		NoREC (SQLancer)		TLP (SQLRight)		NoREC (SQLRight)		SemCon T	
	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

with fewer test cases, demonstrating its effectiveness and efficiency in detecting bugs and inconsistencies that violating SQL specifications.



(a) MySQL (b) TiDB

(c) SQLite (d) DuckDB

(e) OceanBase (f) PostgreSQL

Fig. 12: The memory consumption of SEMCONT with SQLancer.

Memory Consumption. Figure 12 and Figure 13 shows the memory usage of SemCon T and SQLancer during a 6-hour test on six databases. Memory usage is mainly influenced by query generation, execution, and result comparison, with query generation being the most memory-intensive. We report stabilized memory consumption, with detailed time-based changes provided in our technical report [35]. Results indicate that SemCon T's memory usage is comparable to the baseline. Memory consumption is low because we use small tables (a few hundred records). We do not test concurrent query execution, resulting in stable memory usage.

Our goal is to detect compliance bugs in RDBMSs, not to evaluate performance. SemCon T is designed for offline testing during RDBMS development, and its overhead is manageable. SemCon T shows very low memory consumption and it requires far less test cases to detect a compliance issue, demonstrating its efficiency.

Answer to RQ3: SemCon T 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.

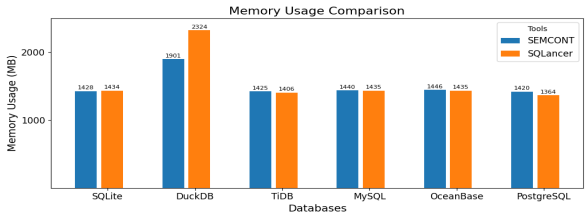


Fig. 13: Memory consumption of SemCon T and SQLancer

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

Fig. 14: A bug in MySQL 8.0.29

RQ4: Case study. SemCon T 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 14 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 15 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 [31].

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

```
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:[]
```

Fig. 15: The queries triggering an inconsistency in MySQL 8.0.29 with the SQL specification

```
SELECT "1gbnn" IN (1);
  -- PostgreSQL:invalid input syntax for type integer
  -- TiDB: 1
```

(a) The query that triggers 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

Fig. 16: An inconsistency caused by unclear description of the `IN` operator

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 [31] (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 the SQL specification. Yet MySQL 8.0.29 returned an empty list. We have reported this inconsistency to the MySQL developers and they confirmed the reason for this inconsistency is due to the 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 16a 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 16b, 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.

Table VII shows the inconsistencies we found between databases due to unclear descriptions of the standards.

> **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.

C. Threats to Validity

Coverage of the semantics. We defined the denotational semantics of all keywords concerning the SQL Data Query Language (DQL) commands. For the other types of SQL commands, including DDL, SML, DCL and TCL, they 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 test query generation. To address the issue, we run all comparison experiments, i.e., RQ2 and RQ3, 5 times and report the median number. Moreover, in the comparison experiments of different settings of SemConT (Table V) and comparison with baselines (Table VI), we run each experiment for 6 hours to mitigate the effect of randomness.

Correctness of SemConT's implementation. Firstly, 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. Another issue is when the SQL specification does not clearly describe the semantics of a keyword, operator or function. In this scenario, 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 of the compared approach. We follow the experiment settings of the baselines presented in their papers to reproduce their

TABLE VII: Inconsistencies between databases

Inconsistency Description	Example	MySQL	TiDB	DuckDB	SQLite	PostgreSQL
Implicit type conversion problem	SELECT '1' IN (1)	1	1	0	0	0
	SELECT 1 % '1E1';	1	1	1	0	1
The result representation of the bitwise operator	SELECT \n1' & 1	0	0	1	1	1
	SELECT -5 & -4;	18446744073709551608	18446744073709551608	-8	-8	-8
	SELECT -3 > ('5' '-4');	0	0	1	1	1
The result representation of zero	SELECT '0'/-4;	-0	-0	0	0	0
	SELECT mod('-12',-4);	-0	-0	0	0	0
The result representation of the round function	SELECT round(1,2);	1	1	1.00	1.00	1.00
The result representation of floating point	SELECT '1'/32;	0.03125	0.03125	0.313	0.313	0.313

best performance. All experiments are conducted on the same environment, within the same time frame and the same configurations of the tested RDBMSs for fairness of comparison.

VI. Related Work

Testing Relational DBMS Research on testing relational database systems to identify and rectify logical bugs is a pivotal and actively pursued area of study in database management systems. Existing methods for testing relational database systems primarily encompass two distinct categories, metamorphic testing, which focuses on validating the database's behavior under varying conditions without known output [19], [20], [45], [52], and differential testing, where the system's responses to identical inputs are compared across different versions or configurations.

RAGS [47] and Apollo [33] are notable early method that implemented differential testing to identify logical errors in database management systems. SQLSmith [46] employs a technique of continuously generating random SQL query statements for database testing. However, this method is limited to identifying defects that lead to system crashes, and does not cover other types of logical errors. Ratel [50] significantly improves the robustness of SQL generation for database testing by merging SQL dictionaries with grammar-based mutations. Furthermore, it boosts the accuracy of feedback through the use of binary coverage linking and bijective block mapping, thereby enhancing the overall performance of RDBMS testing. SparkFuzz [28] 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.

Numerous testing methods involve constructing a test oracle by proposing a variety of metamorphic relations [21], [28], [40]–[42]. MUTASQL [21] and Eqsql [17] are tools that construct test cases by defining mutation rules. These rules are used to generate or synthesize SQL query statements that are functionally equivalent to the original ones. In recent years, SQLancer [14] has emerged as the most effective black-box fuzz testing tool, distinguished by its adoption of three complementary oracles [40]–[42]. PQS [42] operates by first selecting a row of data, and then synthesizing a query based on this selected data. The design of the query is such that it must return the initially chosen row. This approach detects bugs by verifying whether the returned result includes the specific row of data. NoREC [40] transforms an optimized SQL query into an equivalent non-optimized version and then compares the execution results of both. TLP [41] divides a SQL query into three separate SQL statements that collectively retain the same semantics as the original query. If the results of executing the original query differ from those obtained from the three divided queries, it likely indicates the presence of logical errors. SQLRight [34] focuses on enhancing the semantic correctness of generated SQL queries and adopts the oracles proposed by PQS [42], NoREC [40], and TLP [41]. GRIFFIN [27] executes mutation testing within the grammatical boundaries of SQL language, transforming data into metadata for this purpose. While metamorphic testing approaches address syntax differences arising from various database implementations, consistently returned results do not always guarantee the absence of bugs. Furthermore, these methods fall short in detecting bugs caused by violations of SQL specifications.

Formal semantics for SQL. There have been several approaches [18], [24], [25], [36], [38], [48], [49] that made attempts to formalize the semantics of SQL. Chinaei [23] was the first to propose a bag-based SQL operational semantics. More recent works [16], [29], [39] have taken into account NULL values when defining SQL formal semantics. Guagliardo and Libkin [29] defined the formal semantics of SQL, taking into account not only the syntax of basic SQL query statements but also data structures such as subqueries, sets, and bags, which were not supported in previous research. This work also implemented and verified the correctness of its operational semantics in programming. SQLcoq [16] delves into grouping and aggregate functions, proving the equivalence between its proposed formal semantics and relational algebra. Additionally, Zhou et al. [51] introduces an algorithm for proving query equivalence under bag semantics. These methods [16], [22], [29], [51] 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 [43], [44]. Various popular programming languages, developed using the

K framework [43], offer executable semantics that can be effectively used as testing oracles. ExAIS [44] formalizes executable semantics for artificial intelligence libraries and implements them using the Prolog language. These semantics are not only employed as testing oracles but also play a pivotal role in guiding test case generation. Our work presents the first initiative to use semantics for testing in the domain of database management systems and detect 19 bugs and 11 inconsistencies, most of which cannot be detected by existing approaches.

VII. CONCLUSION

In this work, 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 four 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 those baselines during the same time period, and most of the bugs and inconsistencies cannot be detected by those baselines. We have made SEMCONT public available to inspire further research.

REFERENCES

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

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

[3] "Mysql," https://www.mysql.com, 1995, accessed on November 10, 2023.

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

[5] "Postgresql," https://www.postgresql.org, 1996, accessed on November 10, 2023.

[6] "Sqlite," https://www.sqlite.org/index.html, 2000, accessed on November 10, 2023.

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

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

[9] "International organization for standardization. (2016). iso/iec 9075-2:2016: Information technology – database languages – sql/foundation," 2016.

[10] "Iso/iec 9075-2:2016," https://www.iso.org/standard/63555.html, 2016, accessed on November 10, 2023.

[11] "Oceanbase," https://en.oceanbase.com/, 2016, accessed on November 10, 2023.

[12] "Tidb," https://www.pingcap.com/tidb/, 2016, accessed on November 10, 2023.

[13] "Duckdb," https://duckdb.org, 2019, accessed on November 10, 2023.

[14] "Sqlancer," https://github.com/sqlancer/sqlancer, 2019, accessed on November 10, 2023.

[15] "An error occurred when the cast function converted the numerical value in the form of scientific notation," https://sqlite.org/forum/forumpost/3f085531bf, 2022, accessed on November 11, 2022.

[16] V. Benzaken and E. Contejean, "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*, 2019, pp. 249–261.

[17] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen, "Search-based test data generation for sql queries," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1220–1230.

[18] S. Ceri and G. Gottlob, "Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries," *IEEE Transactions on software engineering*, no. 4, pp. 324–345, 1985.

[19] T. Y. Chen, S. C. Cheung, and S. Yiu, "Metamorphic testing: A new approach for generating next test cases," *CoRR*, vol. abs/2002.12543, 2020.

[20] T. Y. Chen, F. Kuo, H. Liu, P. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 4:1–4:27, 2018.

[21] X. Chen, C. Wang, and A. Cheung, "Testing query execution engines with mutations," in *Proceedings of the workshop on Testing Database Systems*, 2020, pp. 1–5.

[22] J. Cheney and W. Ricciotti, "Comprehending nulls," in *The 18th International Symposium on Database Programming Languages*, 2021, pp. 3–6.

[23] H. R. Chinaei, "An ordered bag semantics for sql," Master's thesis, University of Waterloo, 2007.

[24] S. Chu, C. Wang, K. Weitz, and A. Cheung, "Cosette: An automated prover for sql." in *CIDR*, 2017.

[25] S. Chu, K. Weitz, A. Cheung, and D. Suciu, "Hottsql: Proving query rewrites with univalent sql semantics," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 510–524, 2017.

[26] W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*. Springer Science & Business Media, 2003.

[27] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-free dbms fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[28] B. Ghit, N. Poggi, J. Rosen, R. Xin, and P. Boncz, "Sparkfuzz: Searching correctness regressions in modern query engines," in *Proceedings of the workshop on Testing Database Systems*, 2020, pp. 1–6.

[29] P. Guagliardo and L. Libkin, "A formal semantics of sql queries, its validation, and applications," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 27–39, 2017.

[30] International Organization for Standardization, *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*, 2016, no. 9075-2:2016.

[31] ——, *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*, 2016, no. 9075-2:2016.

[32] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1695–1712.

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

[34] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of {DBMS} with coverage-based guidance," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4309–4326.

[35] S. Liu, C. Tian, J. Sun, R. Wang, W. Lu, Y. Zhao, Y. Xue, J. Wang, and X. Du, "Conformance testing of relational dbms against sql specifications (technical report)," https://github.com/DBMSTesting/Technical-report, 2024.

[36] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky, "Toward a verified relational database management system," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2010, pp. 237–248.

[37] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*, 2nd ed. USA: Cambridge University Press, 2017.

[38] M. Negri, G. Pelagatti, and L. Sbattella, "Formal semantics of sql queries," *ACM Transactions on Database Systems (TODS)*, vol. 16, no. 3, pp. 513–534, 1991.

[39] W. Ricciotti and J. Cheney, "A formalization of sql with nulls," *Journal of Automated Reasoning*, vol. 66, no. 4, pp. 989–1030, 2022.

[40] M. Rigger and Z. Su, "Detecting optimization bugs in database engines via non-optimizing reference engine construction," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1140–1152.

[41] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[42] M. Rigger and Z. Su, "Testing database engines via pivoted query synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 667–682.

[43] G. Roșu and T. F. Șerbănuță, "An overview of the k semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.
[44] R. Schumi and J. Sun, "Exais: executable ai semantics," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 859–870.
[45] S. Segura, G. Fraser, A. B. Sánchez, and A. R. Cortés, "A survey on metamorphic testing," *IEEE Trans. Software Eng.*, vol. 42, no. 9, pp. 805–824, 2016.
[46] A. Seltenreich, B. Tang, and S. Mullender, "Sqlsmith," 2019.
[47] D. R. Slutz, "Massive stochastic testing of sql," in *VLDB*, vol. 98. Citeseer, 1998, pp. 618–622.
[48] J. Van den Bussche and S. Vansummeren, "Translating sql into the relational algebra," *Course notes, Hasselt University and Université Libre de Bruxelles*, 2009.
[49] M. Veanes, N. Tillmann, and J. De Halleux, "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, 2010, pp. 425–446.
[50] M. Wang, Z. Wu, X. Xu, J. Liang, C. Zhou, H. Zhang, and Y. Jiang, "Industry practice of coverage-guided enterprise-level dbms fuzzing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 328–337.
[51] Q. Zhou, J. Arulraj, S. Navathe, W. Harris, and J. Wu, "A symbolic approach to proving query equivalence under bag semantics," *arXiv preprint arXiv:2004.00481*, 2020.
[52] Z. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," *IEEE Trans. Software Eng.*, vol. 42, no. 3, pp. 264–284, 2016.
[53] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.