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

I. Semantics

The full list of SQL syntax is shown in Figure 7. We follow the SQL specification to obtain this syntax list. 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) [4].

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 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. $\mathtt{T}$, 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 8 and 9 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 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.

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

Definition 1 (Boolean value expression ($\mathcal{B} : E_B \mapsto \mathbb{B}$)) Fig 8 (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 9 (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 9 (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 (S, T), where S is the set of all tables and T is the resulting table.

Definition 4 (Keyword operation ($\mathcal{C} : \{L, OP\} \mapsto T$)) Fig 10 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 ($\mathcal{H} : E \mapsto (S, T)$)) Fig 11 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 expressions E to the domain of tuples (S, T), where S 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 ($\mathcal{H} : E \mapsto (S, T)$)) 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 (S, T). As shown in Fig 11, 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).

II. Reasons for Using Prolog to Implement SQL Semantics

We implement the semantics using Prolog for two main reasons. First, Prolog, like SQL, is a declarative language, which contrasts with imperative languages such as C typically used in RDBMS implementations. This distinction reduces the likelihood of replicating common errors found in traditional RDBMSs. Second, Prolog is intuitive and straightforward to implement, offering built-in support for operations like list manipulation and querying, which align well with the structure of tables and queries in SQL. For these reasons, Prolog is commonly used in existing research to formalize the semantics of various domains [9], [16].

III. Inconsistencies between databases

Correctness of SemConT. Prolog, being declarative, is naturally suited to specify denotational semantics we defined.

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

Algorithm 1: Executing a SQL query

```
Input  : $sql$: the SQL query to be executed
Output: $result$: the execution result of the query
$ast$ = ParseSQL ($sql$)
Function ExecuteQuery($ast.root$):
    $keywordList$ = Sort($ast.root.children$)
    foreach $keyword$ in $keywordList$ do
        $result$ = ExecuteKeyword($keyword, result$)
    end
Function ExecuteKeyword($keywordNode, result$):
    foreach $child$ in $keywordNode.children$ do
        if child is query then
            $result$ = ExecuteQuery($child$)
        end
        if child is leaf then
            $result$=ExecuteRule($keywordNode, result$)
            return $result$
        end
        $result$ = ExecuteKeyword($child, result$)
    end
```

Taking the 'select' keyword as an example, rule 19 of Figure 10 shows the formal semantics of 'select' and line 9-11 in Figure 1 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.

IV. Time complexity of Algorithm 1 and 2

The time complexity of Algorithm 1 is O(n) with n being the number of operators in the given SQL query.

Algorithm 2: Coverage guided query generation

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

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 [12], 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.

V. Enhancement of result explainability

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

VI. 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 [4] only provide the anomaly phenomena, which can be formalized as specific schedule templates among transactions, to be avoided in each isolation level, we need to exclude those invalid schedules in our semantics.

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

VII. Evaluation

A. Experiment setup

MySQL [1] and PostgreSQL [2] are the two most popular open-source database management systems. SQLite [3] and DuckDB [7] are both embedded DBMSs, running within the process of other applications. TiDB [6] and OceanBase [5] are popular distributed RDBMSs.

B. Experiment results

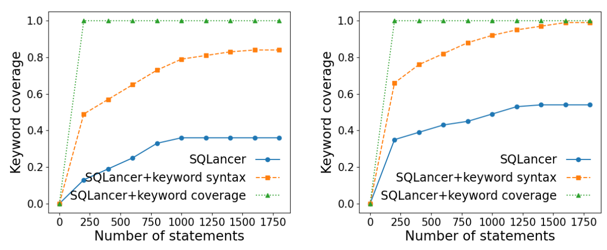
Effectiveness of SemCont. Out of the issues identified, 23 are related to scalar expressions and 9 to other keywords, including joins and various relational operators. Our primary objective is to detect inconsistencies between RDBMS implementations and the SQL specification by generating test cases that 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 [10], [13]–[15]. 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 [10], [13]–[15], making it more challenging to uncover new inconsistencies.

Bitwise operators in OceanBase(2104) do not consistently return signed integer types when performing operations on negative numbers.

TABLE II: Bugs and inconsistencies detected by SemCont in OceanBase

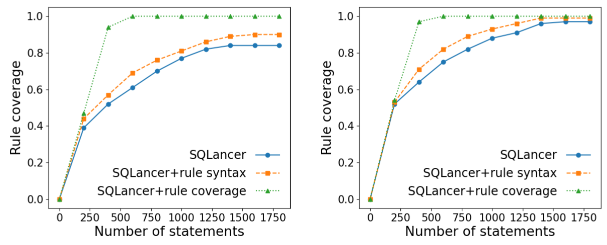
SN	ID	Target	Type	Reason	Status
1	2104	OceanBase	Inconsistency	missing spec	confirmed
2	2105	OceanBase	Inconsistency	missing spec	confirmed

Effectiveness of the coverage criteria. In Figure 2, SQLancer+keyword syntax represents the setting of adding keywords and the corresponding generation rules which



(a) OceanBase (b) PostgreSQL

Fig. 2: The keyword coverage increment (y-axis) with the number of queries (x-axis)



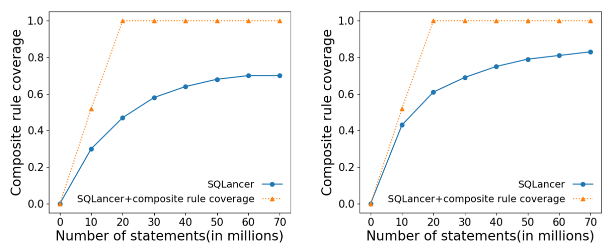
(a) OceanBase (b) PostgreSQL

Fig. 3: The rule coverage increment (y-axis) with the number of queries (x-axis)

were not supported by SQLancer. SQLancer+keyword syntax greatly improved the keyword coverage for OceanBase and PostgreSQL. Notably, within the first 1500 SQL statements, over 80% of the keywords were covered on these two databases, with PostgreSQL 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 OceanBase and PostgreSQL, demonstrating the effectiveness of our keyword-guided query generation method.

Figure 3 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 3. 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. 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 4 depicts the improvements in composite rule coverage by the generated queries for OceanBase and PostgreSQL. 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, PostgreSQL and OceanBase each achieved a composite rule coverage of around 70%. With the introduction of composite



(a) OceanBase (b) PostgreSQL

Fig. 4: The composite rule coverage increment (y-axis) with the number of queries (in million).

rule coverage guidance, both of these two 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. We detected no bugs or inconsistencies on PostgreSQL, and detected two inconsistencies on OceanBase. Table III presents the experimental results obtained from OceanBase 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. In terms of the time taken to detect the first bug or inconsistency, all three coverage guidance algorithm are faster than SEMCONT with random query generation.

Comparison with baselines. We compare SEMCONT with two state-of-the-art approaches TLP [13] and NoREC [14], which are metamorphic testing approaches for relational DBMS. For both approaches, we adopt SQLancer [8] and SQLRight [10] for query generation. Notably, SQLRight does not support 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 IV. 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 1 inconsistency in OceanBase. SEMCONT outperformed the compared approaches and detected 2 inconsistencies in OceanBase. 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 OceanBase is related to the `MOD` function. When applied to negative numbers, OceanBase incorrectly represents the result as `-0`. Both TLP and NoREC

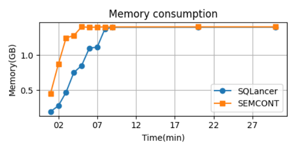
TABLE III: The bug and inconsistency numbers detected by SEMCONT with no coverage guided and coverage guided in 6h

DBMS	SEMCONT			SEMCONT+keyword coverage			SEMCONT+rule coverage			SEMCONT+composite rule coverage		
	Bugs	Inconsistencies	Time	Bugs	Inconsistencies	Time	Bugs	Inconsistencies	Time	Bugs	Inconsistencies	Time
OceanBase	0	2	13.25	0	2	13.37	0	2	7.92	0	2	9.13

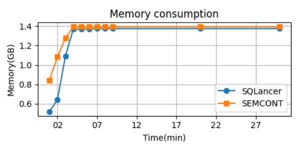
TABLE IV: The bugs and inconsistencies detected by SQLancer, SQLRight and SEMCONT in 6h

DBMS	TLP (SQLancer)		NoREC (SQLancer)		TLP (SQLRight)		NoREC (SQLRight)		SEMCONT	
	Bugs	Inconsistencies	Bugs	Inconsistencies	Bugs	Inconsistencies	Bugs	Inconsistencies	Bugs	Inconsistencies
OceanBase	0	1	0	0	-	-	-	-	0	2

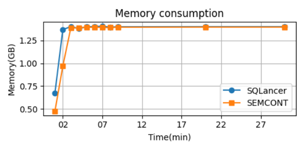
failed to detect this bug, even after successfully generated the bug triggering query. On average, our tool finds a bug in 67 minutes using 19,381 test cases, compared to 300 minutes and 1.3 million test cases for SQLancer, and 30 hours and 8.4 million test cases for SQLRight. Our approach finds more bugs/inconsistencies with fewer test cases, demonstrating its effectiveness and efficiency in detecting bugs and inconsistencies that violating SQL specifications.



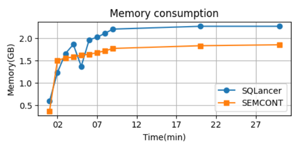
(a) MySQL



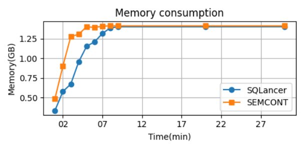
(b) TiDB



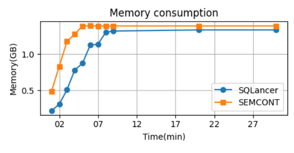
(c) SQLite



(d) DuckDB



(e) OceanBase



(f) PostgreSQL

Fig. 5: The memory consumption of SEMCONT with SQLancer.

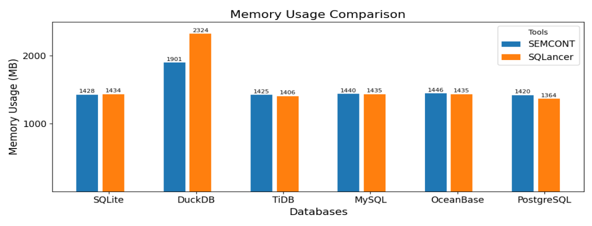


Fig. 6: Memory consumption of SEMCONT and SQLancer.

Memory Consumption. Figure 6 shows the memory usage of SEMCONT 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 [11]. Results indicate that SEMCONT'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.

VIII. INCONSISTENCIES BETWEEN DATABASES

REFERENCES

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

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

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

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

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

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

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

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

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

[10] Yu Liang, Song Liu, and Hong Hu. Detecting logical bugs of {DBMS} with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4309–4326, 2022.

[11] Shuang Liu, Chenglin Tian, Jun Sun, Ruifeng Wang, Wei Lu, Yongxin Zhao, Yinxing Xue, Junjie Wang, and Xiaoyong Du. Conformance testing of relational dbms against sql specifications (technical report). https://github.com/DBMSTesting/Technical-report, 2024.

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

[13] Manuel Rigger and Zhendong 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*, pages 1140–1152, 2020.

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

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

[16] Richard Schumi and Jun Sun. Exais: executable ai semantics. In *Proceedings of the 44th International Conference on Software Engineering*, pages 859–870, 2022.

(1)$\langle queryexp\rangle ::= \{\langle collection\ clause\rangle \mid \langle select\ clause\rangle\langle from\ clause\rangle[\langle where\ clause\rangle][\langle group\ by\ clause\rangle]$
$[\langle having\ clause\rangle]\}[\langle order\ by\ clause\rangle]$

(2)$\langle collection\ clause\rangle ::= \langle queryexp\rangle\langle cop\rangle\langle queryexp\rangle$

(3)$\langle cop\rangle ::= \text{UNION [ALL]}|\text{EXCEPT [ALL]}|\text{INTERSECT [ALL]}$

(4)$\langle from\ clause\rangle ::= \text{FROM}\langle tref\rangle[,\langle tref\rangle...]$

(5)$\langle tref\rangle ::= \langle tname\rangle \mid \langle joined\ table\rangle$

(6)$\langle joined\ table\rangle ::= \langle cross\ join\rangle \mid \langle qualified\ join\rangle \mid \langle natural\ join\rangle$

(7)$\langle cross\ join\rangle ::= \langle tname\rangle\text{CROSS JOIN}\langle tname\rangle$

(8)$\langle qualified\ join\rangle ::= \langle tname\rangle[\text{INNER}|\text{LEFT}|\text{RIGHT}|\text{FULL}]\ \text{JOIN}\langle tname\rangle\langle on\ clause\rangle$

(9)$\langle natural\ join\rangle ::= \langle tname\rangle\text{NATURAL JOIN}\ \langle tname\rangle$

(10)$\langle on\ clause\rangle ::= \text{ON}\ \langle bvexp\rangle$

(11)$\langle where\ clause\rangle ::= \text{WHERE}\langle bvexp\rangle$

(12)$\langle select\ clause\rangle ::= \text{SELECT}\ \langle sop\rangle|\langle af\rangle\ \langle slist\rangle$

(13)$\langle slist\rangle ::= *|\langle cname\rangle\ [,\ \langle cname\rangle...]$

(14)$\langle sop\rangle ::= \text{DISTINCT}\ |\text{ALL}$

(15)$\langle af\rangle ::= \text{MAX}\ |\text{MIN}\ |\text{SUM}\ |\text{COUNT}\ |\text{AVG}$

(16)$\langle group\ by\ clause\rangle ::= \text{GROUP BY}\ \langle cname\rangle$

(17)$\langle having\ clause\rangle ::= \text{HAVING}\langle bvexp\rangle$

(18)$\langle order\ by\ clause\rangle ::= \text{ORDER BY}\langle cname\rangle\text{ASC}|\text{DESC}$

(19)$\langle vexp\rangle ::= \langle nvexp\rangle \mid \langle svexp\rangle \mid \langle bvexp\rangle \mid \langle caseexp\rangle \mid \langle castexp\rangle \mid \langle cname\rangle \mid null$

(20)$\langle svexp\rangle ::= \langle concatenation\rangle \mid \langle character\ substring\ function\rangle \mid \langle trim\ function\rangle \mid \langle fold\rangle \mid \langle vexp\rangle$
$\mid string\ literal$

(21)$\langle concatenation\rangle ::= \langle svexp\rangle||\langle svexp\rangle$

(22)$\langle character\ substring\ function\rangle ::= \text{SUBSTRING}(\langle svexp\rangle FROM\langle vexp\rangle)$

(23)$\langle trim\ function\rangle ::= \text{LTRIM}|\text{RTRIM}(\langle svexp\rangle)$

(24)$\langle fold\rangle ::= \text{UPPER}|\text{LOWER}(\langle svexp\rangle)$

(25)$\langle nvexp\rangle ::= \langle arithmetic\ expression\rangle \mid \langle modules\ expression\rangle| \ \langle length\ expression\rangle$
$\mid \langle absolute\ value\ expression\rangle| \ \langle natural\ logarithm\rangle| \ \langle exponential\ function\rangle| \ \langle power\ function\rangle| \ \langle square\ root\rangle$
$\mid \langle floor\ function\rangle| \ \langle ceiling\ function\rangle| \ \langle vexp\rangle| \ numeric\ literal$

(26)$\langle arithmetic\ expression\rangle ::= \langle nvexp\rangle + \langle nvexp\rangle \mid \langle nvexp\rangle - \langle nvexp\rangle \mid \langle nvexp\rangle * \langle nvexp\ \rangle| \ \langle nvexp\rangle/\langle nvexp\rangle$

(27)$\langle modules\ expression\rangle ::= \text{MOD}(\langle nvexp\rangle\ ,\langle nvexp\rangle)$

(28)$\langle length\ expression\rangle ::= \{\text{LENGTH}|\text{CHAR˙LENGTH}|\text{CHARACTER˙LENGTH}\}(\langle svexp\rangle)$

(29)$\langle absolute\ value\ expression\rangle ::= \text{ABS}(\langle nvexp\rangle)$

(30)$\langle natural\ logarithm\rangle ::= \text{LN}(\langle nvexp\rangle)$

(31)$\langle exponential\ function\rangle ::= \text{EXP}(\langle nvexp\rangle)$

(32)$\langle power\ function\rangle ::= \text{POWER}(\langle nvexp\rangle, \langle nvexp\rangle)$

(33)$\langle square\ root\rangle ::= \text{SQRT}(\langle nvexp\rangle)$

(34)$\langle floor\ function\rangle ::= \text{FLOOR}(\langle nvexp\rangle)$

(35)$\langle ceiling\ function\rangle ::= \{\text{CEIL}|\text{CEILING}\}(\langle nvexp\rangle)$

(36)$\langle bvexp\rangle ::= \langle logical\ expression\rangle \mid \langle is\ expression\rangle \mid \langle comparison\ expression\rangle \mid \langle between\ expression\rangle$
$\langle in\ expression\rangle \mid \langle exists\ expression\rangle \mid \langle null\ expression\rangle \mid \langle vexp\rangle \mid true \mid false \mid null$

(37)$\langle logical\ expression\rangle ::= \langle bvexp\rangle\ \text{OR}\ \langle bvexp\rangle \mid \langle bvexp\rangle\ \text{AND}\ \langle bvexp\rangle \mid \langle bvexp\rangle\ \text{XOR}\langle bvexp\rangle\ |\text{NOT}\ \langle bvexp\rangle$

(38)$\langle is\ expression\rangle ::= \langle bvexp\rangle\ \text{IS [NOT] TRUE}|\text{FALSE}|\text{UNKNOWN}$

(39)$\langle comparison\ expression\rangle ::= \langle bvexp\rangle = |! = |< |> |>= |<= \langle bvexp\rangle$

(40)$\langle between\ expression\rangle ::= \langle nvexp\rangle[\text{NOT}]\ \text{BETWEEN}\ \langle nvexp\rangle\ \text{AND}\ \langle nvexp\rangle$

(41)$\langle in\ expression\rangle ::= \langle vexp\rangle\ [\text{NOT}]\ \text{IN}\ \langle vlist\rangle$

(42)$\langle vlist\rangle ::= (\langle vexp\rangle[,\langle vexp\rangle...])$

(43)$\langle exists\ expression\rangle ::= \text{EXISTS}\ \langle subquery\rangle$

(44)$\langle null\ expression\rangle ::= \langle bvexp\rangle\ \text{IS [NOT] NULL}$

(45)$\langle subquery\rangle ::= (\langle query\ expression\rangle)$

(46)$\langle caseexp\rangle ::= \text{CASE WHEN}\ \langle vexp\rangle\ \text{THEN}\ \langle vexp\rangle\ \text{ELSE}\ \langle vexp\rangle$

(47)$\langle castexp\rangle ::= \text{CAST}(\langle vexp\rangle\ \text{AS}\ \langle data\ type\rangle)$

(48)$\langle data\ type\rangle ::= string \mid numeric \mid boolean$

(50)$\langle tname\rangle ::= identifier$

(51)$\langle cname\rangle ::= identifier$

Fig. 7: The full list of syntax for SQL

TABLE V: Inconsistencies between databases

	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 of the round function represents	SELECT round(1,2);	1	1	1.00	1.00	1.00
Floating point result representation	SELECT '1'/32;	0.03125	0.03125	0.313	0.313	0.313

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. 8: The full list semantic definition of boolean expression

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. 9: 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. 10: The full list of semantic definitions for join operation, collection operation, filtering operation, and aggregation operation

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

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

Composite keyword operation

102. $\diamond$: (1) $\dfrac{\mathcal{H}\llbracket expression_1\rrbracket \triangleq (S, T),\ \mathcal{H}\llbracket expression_2\rrbracket \triangleq (\{T\}, T')}{\mathcal{H}\llbracket expression_1\rrbracket \diamond \mathcal{H}\llbracket expression_2\rrbracket \triangleq (S, T')}$

    (2) $\dfrac{\mathcal{H}\llbracket expression\rrbracket \triangleq (S, T),\ \mathcal{C}\llbracket\{\{T\}, OP\}\rrbracket \triangleq T'}{\mathcal{H}\llbracket expression\rrbracket \diamond \mathcal{C}\llbracket\{\{T\}, OP\}\rrbracket \triangleq (S, T')}$

    (3) $\dfrac{\mathcal{C}\llbracket\{L, OP\}\rrbracket \triangleq T,\ \mathcal{H}\llbracket expression\rrbracket \triangleq (\{T\}, T')}{\mathcal{C}\llbracket\{L, OP\}\rrbracket \diamond \mathcal{H}\llbracket expression\rrbracket \triangleq (L, T')}$

    (4) $\dfrac{\mathcal{H}\llbracket expression_1\rrbracket \triangleq (S, T_1),\ \mathcal{H}\llbracket expression_2\rrbracket \triangleq (S, T_2),\ \mathcal{C}\llbracket\{\{T_1, T_2\}, OP\}\rrbracket \triangleq T_3}{(\mathcal{H}\llbracket expression_1\rrbracket, \mathcal{H}\llbracket expression_2\rrbracket) \diamond \mathcal{C}\llbracket\{\{T_1, T_2\}, OP\}\rrbracket \triangleq (S, T_3)}$

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

Fig. 11: 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