Efficiently Detecting DBMS Bugs through Bottom-up Syntax-based SQL Generation

Yu Liang, Peng Liu
Pennsylvania State University
{luy70, pxl20}@psu.edu

***Abstract*—Syntax-based testing is a promising technique for finding bugs in Database Management Systems (DBMSs). All existing syntax-based SQL generation tools apply a Top-down generation method. To construct a SQL query (syntax tree), the generator forward explores the SQL grammar starting from the root node, and it stops when no further grammar rule can be applied to the leaves of the syntax tree. However, the Top-down generation method tends to put more effort into exploring the shallow grammar close to the `root` and neglects the feature-rich grammar deeper in the grammar space. Therefore, it is not efficient in finding DBMS bugs.**

This paper proposes a new Bottom-up syntax-based SQL generation technique that puts more testing resources into exploring the feature-rich grammar rules. The exploration of SQL grammar begins with one interesting grammar rule that outlines the syntax of feature-rich SQL functionalities. The generator then backtracks (Bottom-up) this grammar rule to the `root` to create a syntax path that unveils this interesting grammar. Multiple Bottom-up generated syntax paths are then expanded and merged to create diverse SQL queries for fuzzing. A prototype tool, `SQLBull`, adopts the Bottom-up generation technique for fuzzing. In the evaluation, `SQLBull` found 63 zero-day bugs from 5 well-tested DBMSs: `MySQL`, `MariaDB`, `CockroachDB`, `DuckDB`, and `PostgreSQL`. It outperforms all existing tools in both bug-finding and code coverage. The evaluation results verify the effectiveness of the Bottom-up generation technique.

I. Introduction

Database Management Systems (DBMSs) power today's internet applications [1]–[4], such as enterprise resource planning (ERP) [5]–[8], E-commerce [9]–[12], and online banking [13]–[16]. However, the high complexity of DBMSs makes them prone to bugs [17]–[20]. A DBMS bug triggered in real-world applications can lead to severe consequences, such as money loss [21] or data breach [22], [23]. For example, in June 2015, the New York Stock Exchange (NYSE) froze stock trading for almost 4 hours because of a database glitch. This glitch caused the NYSE to lose at least $14 million [24]. Accordingly, proactively identifying DBMS bugs through automatic software testing (and other viable means) is a critical aspect of securing DBMS software development [25]–[29].

In the DBMS automatic testing domain, it is a well-known practice to generate SQL testing queries by following SQL grammar rules (i.e., syntax-based) [17], [30], [31]. The primary benefit is that the generated test cases have a higher chance of being syntactically correct. These test cases are more likely to pass the sanity checks conducted by the DBMS parser and subsequently trigger the execution of the internal DBMS program code. Therefore, the syntax-based testers achieve a higher code coverage and uncover more bugs than non-syntax-based ones. Because of this benefit, the DBMS testing community has developed a series of syntax-based testing tools to examine the DBMS program code. For example, `SQLancer` [30] is a representative syntax-based SQL testing platform that adopts hand-crafted SQL templates to generate testing SQL queries.

All the existing syntax-based testing tools adopt the Top-down (`TD`) generation method. To generate a SQL query (syntax tree), a `TD` generator starts its grammar exploration from the `root` symbol, where the root is always the topmost node in the syntax tree. Then, it forward traverses the grammar, constructing a syntax tree with all the grammar rules it has explored, and only stops the tree construction when no more grammar rules can be used further to expand the leaf nodes of the syntax tree. A more detailed illustration of the `TD` generation method is shown in §II-B.

However, the `TD` generator tends to put more effort into exploring the shallow SQL grammar, which is inefficient in generating feature-rich SQL queries that trigger DBMS bugs. We define feature-rich grammar as the ones that correspond to the SQL functionalities that are implemented by complex DBMS internal code and may relate to various DB optimizations. Feature-rich grammar is typically not located in the shallow layers of the grammar. Furthermore, for the `TD` generator, the operation of exploring the grammar rules further away from the `root` (i.e., deeper, require more `TD` grammar exploration steps to reach) is dependent on the exploration of rules that are closer to the `root` (i.e., shallower, require less `TD` grammar exploration steps to reach). Therefore, to explore one feature-rich deep grammar, a `TD` generator has to first comprehensively explore the shallow grammar and find out what shallow grammar this deep grammar depends on. This procedure is necessary because without this, it is challenging for the `TD` generator to expose and explore this deep grammar. This procedure, however, prevents the generator from prioritizing its exploration focus on the deeper and more interesting grammar rules. Additionally, recursive grammar rules, which introduce further complexity to the `TD` generator's exploration, are not uncommon among the

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240198
www.ndss-symposium.org

rules in the shallow layers. They further increase the difficulty for the TD generator in exploring the deeper grammar space.

Several existing works attempt to address the above issues [32]–[34]. For example, some TD generators allocate pre-defined exploration probabilities to different grammar rules, putting more weight on certain grammar rules that are known to lead to existing bugs [27], [35]. This method is more commonly used by syntax-based testing tools that rely on handwritten SQL templates, such as SQLancer [30] and SQLsmith [31]. Another method is to rewrite the grammar rules by uplifting some feature-rich grammar into the shallower layers for more frequent exploration [36]. However, both methods are manual effort-intensive to implement and are **not scalable** to the large number of grammar rules defined by the DBMSs. They demand the testers to have deep domain knowledge of the DBMS SQL grammar to identify the feature-rich SQL grammar and apply the necessary grammar annotation. Moreover, these methods are not adaptable in adjusting their weights when testing the ever-changing SQL dialects implemented by different DBMSs. Lastly, certain existing works treated the TD generation process as a Multi-arm Bandit problem [37], [38]. However, resolving the Multi-arm Bandit problem would likely lead to generating feature-simple yet correct SQL queries, which is also inefficient for detecting bugs.

Our approach. We propose a new syntax-based SQL grammar exploration technique called Bottom-up (BU) syntax path exploration to focus the testing resources on the feature-rich SQL grammar. Without wasting testing resources on overly exploring the shallow syntaxes, a BU explorer starts its execution from the interesting grammar and backtracks the parent rules repeatedly until reaching the root. The generated syntax paths thus unveil the interesting grammar to the root. Furthermore, we implement a prototype tool called SQLBull, which adopts the BU exploration technique to generate numerous SQL queries for automatic DBMS testing. SQLBull is designed with a focus on the following key points: 1) Generate diverse syntax paths that connect interesting rules to the root symbol. 2) Efficiently handle recursive grammar. 3) Generate feature-rich and bug-triggering SQL queries from the diverse syntax paths.

BU grammar exploration is more efficient than TD primarily because it prioritizes a greater focus on testing feature-rich grammar typically not located in a shallow layer. Specifically, 1) By starting BU, the operation of exploring the shallow grammar (closer to root) is dependent on the exploration of deeper feature-rich grammar, which means the SQL generator can focus more on the feature-rich grammar rules without bothering with comprehensively exploring the shallow layer grammar. 2) By starting the exploration from the feature-rich SQL grammar, the feature-rich grammar is always preserved in the generated syntax tree. 3) BU handles recursive grammar in linear exploration complexity, whereas the TD's steps could be exponential. Therefore, BU brings more exposure to feature-rich grammar than TD does while handling the issue of recursive grammar more efficiently, which enables SQLBull to find more bugs than others. Although we have acknowledged that the semantic handling of the generated SQL queries is another important factor for bug discovery [17], [27], [39], this is not the focus of this paper, and our contribution is to more efficiently explore the grammar rules that define valid syntax.

We evaluate SQLBull on 5 widely-used DBMSs, including MySQL, MariaDB, CockroachDB, DuckDB, and PostgreSQL. SQLBull discovers 63 unique zero-day bugs across five DBMSs, demonstrating the effectiveness of the BU strategy. In addition, it outperforms all existing state-of-the-art DBMS testing tools in terms of bug discovery efficiency and DBMS fuzzing coverage.

In summary, this paper makes the following contributions:

- We propose a novel Bottom-up (BU) SQL generation technique, which redirects more testing resources to explore interesting SQL grammar rules.
- We implement a prototype tool, SQLBull, that adopts the BU generation technique. SQLBull detects 63 unique zero-day bugs across 5 well-tested DBMSs, significantly outperforming all existing DBMS testing tools.

Open Source. We have released SQLBull at: https://github.com/SteveLeungYL/SQLBull.

II. Background & Motivation

In this section, we introduce the background of syntax-based DBMS testing in §II-A. Next, we then introduce a motivating example in §II-B to demonstrate a TD generator's inefficiency and a BU generator's effectiveness in detecting DBMS bugs.

A. Background of Syntax-based DBMS Testing

To improve the quality of the generated test cases, the fuzzing community adopts the syntax-based fuzzing strategy to generate syntactically correct test cases. They can be classified into two main categories: those based on handwritten SQL templates and those based on DBMSs' built-in grammar. The former uses hand-crafted SQL templates implemented in custom programming languages to output SQL query strings. For example, SQLancer uses Java to implement its SQL templates, while SQLsmith uses C++. The second category of tools borrows the built-in SQL grammar of a DBMS to guide the test case generation. The parser of a DBMS takes the grammar as input and transforms the input SQL query strings into syntax trees. Instead of matching the grammar rules to the provided seed (test case) queries, a syntax-based generator chooses one arbitrary grammar rule to execute for each parsed tree node, which eventually constructs a syntax tree from all the executed grammar rules. The generated syntax tree is then transformed into a syntactically correct SQL query for execution. A more detailed description of the syntax-based SQL generation is illustrated in §II-B. Because most DBMSs use parser generators [40]–[43] such as Bison or ANTLR to convert their grammar into parser code, the grammar code is typically located in the .y or .g4 files [44], [45] in the DBMS source repo [17], [39]. The syntax-based testing tools directly leverage the grammar to generate various SQL queries for testing.

```sql
01 CREATE TABLE v00 (c01 INT, c02 STRING);
02 FROM v00 AS ta03, LATERAL ( SELECT 'string' IN ta03.c02 )
03     AS ta04 POSITIONAL JOIN v00 AS ta05;
04 -- Internal Fatal Error: Expression with
05 --      depth > 1 detected in non-lateral join.
```

Listing 1: Proof-of-Concept (PoC) of a Fatal Error from `DuckDB`. The Fatal Error is caused by combining `POSITIONAL JOIN` and `correlated subquery` in one SQL statement, which `DuckDB` does not support. The correct behavior of `DuckDB` is to reject processing this SQL statement by using sanity checks.

B. An Example Fatal Error from DuckDB

Listing 1 shows one detected bug from `DuckDB`. The bug Proof-of-Concept (PoC) contains two SQL statements: The first SQL statement in `Line 1` creates a standard table `v00` with two columns, `c01` and `c02`. The second SQL statement in `Lines 2-3` is a `SELECT` statement that outputs the analyzed data from the created table `v00`, even though the `SELECT` keyword is omitted. The `SELECT` statement joins three tables together. The first two tables are implicitly joined within the `FROM` expression. Without the `WHERE` expression to apply the `JOIN` condition, the implicit `JOIN` between the first two tables is similar to a `CROSS JOIN` operation. Lastly, the third table is `POSITIONAL JOIN`ed with the result of the first two tables. When running with the latest version of `DuckDB`, version `v1.1.4-dev bcd6582` at the time of writing, the execution engine of `DuckDB` throws an `Internal Fatal Error`. After triggering the fatal error, `DuckDB` enters safe mode and halts executing any subsequent SQL statements. The user must explicitly restart the `DuckDB` process to resume normal `DuckDB` functionalities.

The second SQL statement from Listing 1 utilizes a unique feature, `POSITIONAL JOIN`, which is only supported by a limited number of DBMSs. Unlike the traditional `CROSS JOIN` operation which merges multiple rows from the two tables by matching values in corresponding columns, `POSITIONAL JOIN` matches rows by matching their physical positions in the two tables. If the two `POSITIONAL JOIN`ed tables have a different number of rows, `NULL` values are padded to the shorter columns. In addition, the `LATERAL` keyword presented before the `SELECT` subquery indicates the usage of the `correlated subquery`, meaning that the inner query references values that are introduced by the outer query. In this PoC, the `SELECT` subquery references `c02` without a `FROM` expression, and the `c02` column can be auto-inferred from `ta03` in the outer query. Unfortunately, the current design of the `DuckDB` execution engine does not support processing with `correlated subquery` and `POSITIONAL JOIN` in one SQL statement. The `DuckDB` developer has put sanity checks in place to reject queries that combine `correlated subquery` and `POSITIONAL JOIN` within the same joining operation. However, this PoC applies `POSITIONAL JOIN` to the result of the `correlated subquery`, where the `DuckDB` developers overlooked this scenario. Therefore, this PoC triggers an unexpected and unsupported behavior from `DuckDB`, causing it to throw an internal fatal error.

Although the semantic sanity checks of `DuckDB` should reject the execution of Listing 1 because it contains unsupported SQL features, the queries from the PoC are syntactically correct, so they can be successfully parsed and validated by the `DuckDB` parser. Therefore, the `DuckDB` grammar contains all the SQL grammar rules required to generate this PoC. Figure 1 shows a visualization of `DuckDB` grammar. The visualization focuses on illustrating the parts related to `POSITIONAL JOIN` and `correlated subquery`, and it omits the other grammar. Listing 2 shows the grammar rules from the `DuckDB` source code [46], which references the grammar rules from `root` to `Layer 3` in Figure 1. Each node in Figure 1 represents one SQL grammar rule. Each directed edge connecting a parent node to a child node corresponds to one valid grammar rule that can be applied to expand one symbol (head of the child node) from the parent node. Starting from the topmost `root` node, which typically represents the start of the grammar, the grammar keeps expanding its depth until all the contained symbols can't be further expanded by any valid grammar rules, i.e., grammar expansion terminates when it reaches

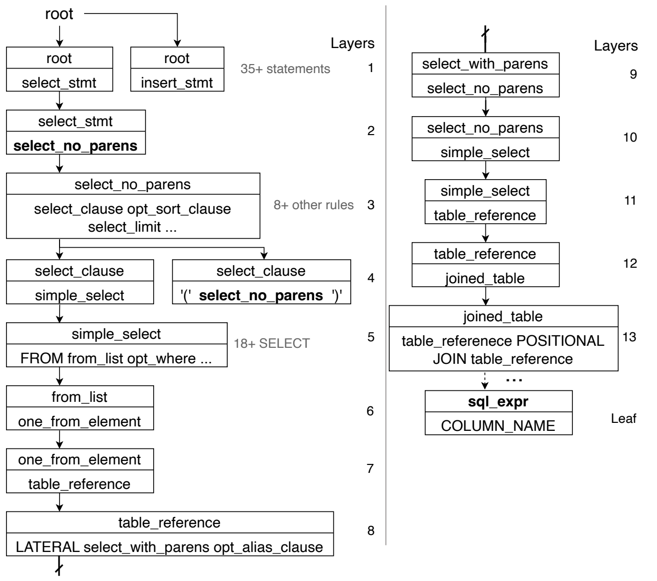


Fig. 1: Visualized `DuckDB` SQL grammar that contains the `correlated subquery` (`Layer 8`) and the `POSITIONAL JOIN` (`Layer 13`) grammar. One node represents one grammar rule. One directed edge represents one candidate child rule that can be used to expand one symbol from the parent rule. The symbol from the parent rule to be expanded by the child is also shown as the child node's header. The bold texts at `Layer 2`, `Layer 4`, and `Leaf` highlight the symbols `select_no_parens` and `sql_expr` appearing at different layers, indicating recursive grammar is present.

```
root:
    select_stmt
    | insert_stmt
    | ... 35+ other statements
    ;
select_stmt:
    select_no_parens
    | ...
    ;
select_no_parens:
    select_clause opt_sort_clause opt_limit_clause ...
    | ... 8+ other rules
    ;
```

Listing 2: Grammar rules from `DuckDB` source code. The grammar rules are visualized from `root` to `Layer 3` in Figure 1.

leaf nodes. The symbols in the leaf nodes are constructed by SQL keywords (e.g., SELECT, FROM, WHERE, etc.), identifiers (e.g., TABLE_NAME, COLUMN_NAME, etc.), and constants (e.g., number, text string, etc.). We define the layer of a node in the grammar as the distance between the node and the root. For example, the joined_table node is on Layer 12, indicating that the joined_table node is 12 expansion steps away from the root. In addition, a node at a deeper layer can reference the same symbol as a node at a shallower layer (recursive grammar), creating a cycle in the grammar and making the grammar infinite in layer and exploration space.

A syntax-based SQL generator traverses this grammar to generate a series of SQL statements. When resolving the symbols in each layer, the generator selects one and only one grammar rule to expand a particular symbol into one or more new symbols. The generation process terminates when all explored grammar branches eventually reach the leaves. The generator creates various forms of SQL statements by choosing different grammar rules at each layer.

Inefficiency of Top-down syntax-based generator. Although a TD syntax-based SQL generator can generate a series of SQL statements by traversing the grammar as mentioned above, generating the PoC from Listing 1 is still challenging. The primary reason is that the grammar rules that define the PoC's critical SQL features, POSITIONAL JOIN and correlated subquery, are hidden deep (e.g., Layer 13) in the grammar space. As shown in Figure 1, to explore the rules that define the POSITIONAL JOIN and correlated subquery, the generator needs to traverse the grammar up to Layer 8 and Layer 13, respectively, which is far from the root. Without human annotation to provide syntax paths, a TD generator has to first comprehensively explore the grammar rules at the shallower layers to find the paths that connect these rules to the root. For example, a TD generator may waste its computational resources on exploring the CREATE TABLE statements, not knowing it will never lead to POSITIONAL JOIN or correlated subquery. In addition, whether using Depth-First Search (DFS) or Breadth-First Search (BFS), a TD syntax-based SQL generator will find an exponentially increasing number of keywords to expand/explore when it reaches deeper layers of the grammar. As a result, a TD syntax-based SQL generator tends to put more effort into exploring the shallow layers of the grammar, and it is challenging for a TD syntax-based SQL generator to generate the PoC from Listing 1 within a reasonable amount of time.

Challenge in handling recursive grammar rules. The TD SQL generator faces a trade-off between abandoning the interesting grammar hidden under the recursive grammar rules or increasing the risk of exponentially increasing the grammar exploration complexity. Listing 3 further details the recursive grammar of sql_expr presented in Figure 1. Figure 2 shows the visualization of a TD generator's execution when it handles the recursive sql_expr grammar from Listing 3. If the TD generator prioritizes a thorough exploration of the grammar, it will quickly encounter the handling of recursive grammar

```
01 where_clause:
02     WHERE sql_expr;
03     | ...
04     ;
05 sql_expr: // 66 rules contain 'sql_expr', 4 does not.
06     sql_expr '+' sql_expr
07     | sql_expr '-' sql_expr
08     | sql_expr '*' sql_expr
09     | sql_expr '/' sql_expr
10     ...
11     | COLUMN_NAME
12     ;
```

Listing 3: Grammar rules from DuckDB. Line 6-9 shows certain recursive grammar rules that reference the sql_expr symbol.

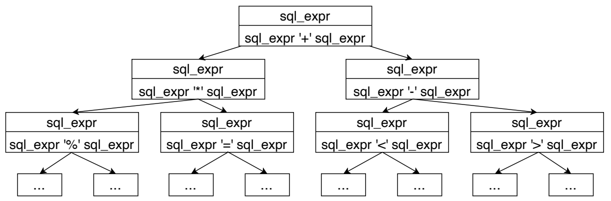


Fig. 2: Execution of Top-down query generator when handling recursive grammar of sql_expr.

rules, such as the ones shown in Listing 3. SQL expression (sql_expr) is one of the most complex SQL features provided by DuckDB, with 70 grammar rules attached to it. Furthermore, of the 70 grammar rules, 66 are recursive rules that reference the sql_expr symbol at least once. As shown in Figure 2, when handling the sql_expr, the TD generator repeatedly expands the sql_expr. It leads to an exponential number of sql_expr grammar that needs to be processed and eventually leads to an exponential growth of the syntax tree.

Unfortunately, these recursive grammar commonly occurs across multiple layers and are hard to track. Existing works rely on human efforts to annotate the grammar, e.g., annotating sql_expr with a high probability of choosing the COLUMN_NAME rule after reaching a specific exploration threshold. However, this annotation process is error-prone and time-consuming.

Our approach is capable of efficiently finding this bug. SQLBull, the first BU syntax-based SQL testing tool proposed in this paper, can find this bug by employing a Bottom-up grammar exploration technique. We configured SQLBull to start its grammar exploration from the SQL Expression grammar, because this grammar is widely used to implement various SQL features (further discussed in §V-A). The primary advantage of the BU grammar exploration is that it prioritizes more testing resources on exploring the interesting grammar rules typically located at the deeper layers of the grammar. This is achieved by starting the grammar exploration from the interesting rules, and then building the syntax paths that connect the interesting rules to the root in a BU manner (§III-A). By walking BU, the exploration of the deeper feature-rich grammar becomes the prerequisite step of exploring the shallower grammar rules, which allows SQLBull to skip the comprehensive exploration of the shallow grammar and more efficiently expose the interesting grammar. More importantly, by starting from the interesting rules, SQLBull always preserves the interesting grammar in the final generated SQL statements. Moreover, SQLBull does not suffer from the complexity issue when handling recursive grammar rules. SQLBull handles recursive grammar with linear exploration time and employs pruning strategies to prevent generating humongous syntax trees due to the recursive grammar (§III-B). These humongous syntax trees could easily lead to DBMS sanity check rejection. As a result, SQLBull can find the bug by fuzzing for less than 1 hour.

III. Design of SQLBull

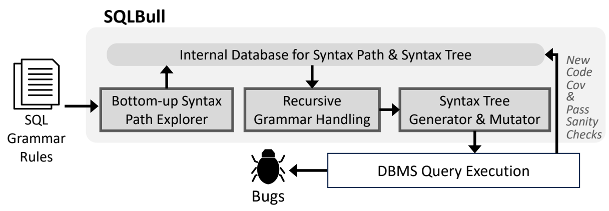

SQLBull
Internal Database for Syntax Path & Syntax Tree
SQL Grammar Rules
Bottom-up Syntax Path Explorer
Recursive Grammar Handling
Syntax Tree Generator & Mutator
New Code Cov & Pass Sanity Checks
DBMS Query Execution
Bugs

Fig. 3: System Overview of SQLBull.

Threat Model. We adopt the same threat model as Squirrel, DynSQL, and BuzzBee, in which attackers can send SQL queries to the DBMSs and conduct DoS attacks. We leave the sophisticated exploitation of bugs to future research.

SQLBull is designed based on the following insights: 1) Prioritize the exploration of the feature-rich SQL grammar. 2) Handle recursive grammar within a reasonable time and resource limit. 3) Given the interesting grammar, SQLBull should generate diverse SQL syntax trees that expose various use cases of the grammar.

System Overview. Based on these insights, we propose a new syntax-based DBMS testing tool, SQLBull. Figure 3 shows the system overview of SQLBull. SQLBull **does not** rely on SQL queries as input seeds; instead, it only accepts the grammar rules from the target DBMS source code [46]–[50] as input, such as the one shown in Listing 2. The first component of SQLBull is the Bottom-up (BU) SQL syntax path explorer, which utilizes BU grammar exploration to find the syntax paths that connect feature-rich SQL grammar to the root (§III-A). To prevent the state explosion caused by the recursive grammar rules, SQLBull then adopts Recursive Grammar Handling to prune the syntax paths and save the testing resources (§III-B). When generating new SQL queries sent to the DBMS for execution, SQLBull randomly picks one pruned syntax path from its library and expands it into a complete SQL syntax tree (§III-C). The expanded SQL syntax tree is then transformed into an SQL query string and sent to the DBMS for execution. Suppose one generated SQL query triggers new code coverage in DBMS execution and passes all sanity checks. In that case, SQLBull will promote the executed SQL query by mutating its syntax tree for more diverse SQL queries. (§III-C). Lastly, if one generated SQL query triggers a crash or an unexpected fatal error on the DBMS, SQLBull will save the sequence of SQL queries generated that lead to the bug in the file system for further analysis (§IV).

Algorithm 1: BU Syntax Path Exploration

```
1:  input $G_{int}$, $G_{SQL}$
2:  output $P_{all}$
3:  $P_{all} \leftarrow \emptyset$
4:  for all $g_{int} \in G_{int}$ do
5:      for $trial \in [0..N_{trials}]$ do
6:          $P_{cur}$.insert($g_{int}$)
7:          while $g_{int} \neq root$ do
8:              $G_{par} \leftarrow G_{SQL}$.get_parent_grammar_of($g_{int}$)
9:              $g_{int} \leftarrow G_{par}$.rand_pick()
10:             $P_{cur}$.insert($g_{int}$)
11:         end while
12:         if not $P_{all}$.find_duplicate($P_{cur}$) then
13:             $P_{all}$.insert($P_{cur}$)
14:         end if
15:     end for
16: end for
```

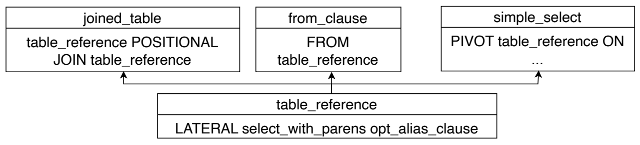

joined_table
table_reference POSITIONAL JOIN table_reference
from_clause
FROM table_reference
simple_select
PIVOT table_reference ON ...
table_reference
LATERAL select_with_parens opt_alias_clause

Fig. 4: Candidate grammar rules to explore when the BU exploration reaches table_reference. Since table_reference is used by grammar that defines joined_table, from_clause, and simple_select, SQLBull chooses one of the parent rules as the next node of the syntax path.

A. *Bottom-up SQL Syntax Path Explorer*

SQLBull employs a BU syntax path explorer to efficiently find the paths that connect the feature-rich SQL grammar to the root symbol. Algorithm 1 shows the algorithm of the BU syntax path explorer. The BU syntax path explorer accepts two inputs: 1) an array of interesting grammars that need prioritized testing, G_{int}, and 2) the entire SQL grammar extracted from the DBMS source code, G_{SQL}. For each individual interesting grammar g_{int} in G_{int}, the explorer takes N_{trials} trials to construct syntax paths connecting g_{int} to the root symbol (Lines 4-5). In each trial, the explorer starts with grammar g_{int} and inserts it to an empty syntax path P_{cur} (Line 6). Then it randomly picks one parent grammar rule of g_{int} repeatedly until it reaches the root symbol (Lines 7-11). The constructed syntax path is saved in P_{cur}, and it is added to the final syntax path set P_{all} if it is not a duplicate (Lines 12-14). To further illustrate the parent grammar rule selection process, Figure 4 shows one example execution cycle between Lines 7 and 11 in Algorithm 1. It assumes that SQLBull tries to find a path that connects the interesting grammar of table_reference (g_{int}) to the root. SQLBull identifies that the table_reference is defined by "LATERAL select_with_parens opt_alias_clause". Instead of finding another grammar to further expand the table_reference like what the TD generator would do, the BU explorer searches for all the grammar rules that contain the table_reference in their expansions (i.e., *get_parent_grammar*). As shown in Figure 4, the BU explorer finds multiple parent grammar rules, such as the "FROM table_reference" from the from_clause expansion and the "PIVOT table_reference ..." from the simple_select expansion. The BU explorer then randomly picks one and only one parent grammar as the next grammar to explore and treats it as one node of the BU-generated syntax path. The explorer continues to explore the parent grammar rules until the exploration reaches the root symbol. Finally, an example of the BU-generated syntax path (P_{cur}) can be found by including all the left-sided nodes in Figure 1. This syntax path connects grammar sql_expr to the root.

The benefits of constructing syntax paths in a Bottom-up manner are two-fold: 1) By constructing SQL queries using these syntax paths, the interesting paths will always be preserved in the final-stage SQL syntax trees. 2) Having diverse syntax paths with interesting grammar rules enables SQLBull to freely explore the different use cases of the interesting grammar rules, effectively increasing the interesting grammar's exposure in the fuzzing process. These advantages from BU can hardly be achieved by TD, which require comprehensive exploration of the shallow grammar to find the different paths that connect the interesting grammar to root. And the TD-generated queries are not guaranteed to explore the interesting grammar.

B. Recursive Grammar Rule Handling

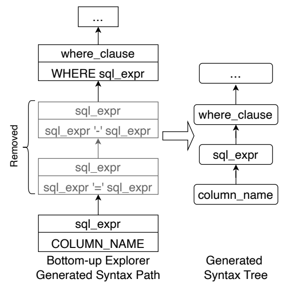


Fig. 5: A BU-generated syntax path and the process of transforming the syntax path into a syntax tree.

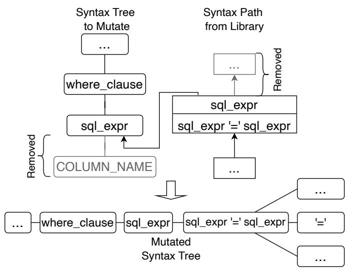


Fig. 6: Type-based syntax tree mutation process. Syntax tree node "COLUMN_NAME" is replaced with "sql_expr = sql_expr".

The BU explorer introduced in §III-A is inherently capable of handling the recursive grammar rules with linear exploration complexity. The main reason is that the exploration process will always converge at the root when traversing the grammar BU. For example, the syntax path from Figure 5 shows one BU-generated syntax path that touches on the recursive grammar of sql_expr. This syntax path follows the grammar rules from Listing 3. Even though the BU explorer is likely to encounter recursive grammar rules in its exploration process, it only needs to handle one *get_parent_grammar* at each exploration cycle. In addition, the candidate rules for the BU explorer to select include not only recursive rules but also non-recursive ones. For sql_expr, "COLUMN_NAME" is one non-recursive rule. Because the BU explorer does not choose the same rule twice in one BU exploration, the explorer is guaranteed to escape the recursive loop eventually. Therefore, the BU explorer is guaranteed to finish its exploration process by reaching the root in a reasonable time.

Although the BU explorer is capable of handling the recursive grammar rules, however, to expand the BU-generated syntax paths into concrete SQL syntax trees (further discussed in §III-C), SQLBull could still suffer from the occurrence of the recursive grammar rules and lead to the exponential growth of the syntax search space. To avoid this issue, SQLBull adopts a pruning process to remove the recursive grammar rules from the BU-generated syntax paths. The pruning process removes all the recursive grammar occurrences from the BU-generated syntax paths. For example, as shown in Figure 5, the "sql_expr -> sql_expr -> sql_expr -> where_clause" path will be pruned to "sql_expr -> where_clause". Without the pruning process, when expanding the BU-generated syntax paths into concrete SQL syntax trees (§III-C), SQLBull would frequently be trapped in path explosion due to the occurrence of the recursive grammar rules and crash because of memory exhaustion. Even if occasionally, a syntax tree is generated successfully with multiple recursive grammar rules present, the resulting SQL query could be hundreds of characters long and would be easily rejected by the DBMS's sanity checks because of a semantic error. At last, although this pruning process may lead to the absence of recursive grammar rules in the BU-generated syntax paths, the syntax tree generation and mutation processes, further discussed in §III-C, will bring back the recursive grammar in the final SQL syntax trees.

C. Syntax Tree Generation & Mutation

Algorithm 2: Syntax Tree Generation

```
1: function SyntaxTreeGeneration
2: input $P_{all}$, $G_{SQL}$
3: output $S$
4: $S \leftarrow P_{all}$.get_rand()
5: for all $node \in S$.get_unresolved_nodes() do
6:     SyntaxTreeGenerationHelper($S$, $node$, $P_{all}$, $G_{SQL}$, false)
7: end for
8: end function
9:
10: function SyntaxTreeGenerationHelper
11: input $S$, $node$, $P_{all}$, $G_{SQL}$, $use\_forward\_exp$
12: output $S$
13: if $use\_forward\_exp$ then
14:     $S \leftarrow S$.merge($G_{SQL}$.forward_gram_exp($node$))
15: else
16:     if $P_{all}$.is_contain_path_with_node($node$) then
17:         $P_{cur} \leftarrow P_{all}$.get_rand_path_with_node($node$)
18:         $S \leftarrow S$.merge($P_{cur}$)
19:         for all $node_{child} \in P_{cur}$.get_unresolved_nodes() do
20:             if check_recursive_grammar($S$, $P_{cur}$) then
21:                 $S \leftarrow$SyntaxTreeGenerationHelper ($S$, $node_{child}$, $P_{all}$, $G_{SQL}$,
                    true) // $use\_forward\_exp$ = true
22:             else
23:                 $S \leftarrow$ SyntaxTreeGenerationHelper ($S$, $node_{child}$, $P_{all}$, $G_{SQL}$,
                    false) // $use\_forward\_exp$ = false
24:             end if
25:         end for
26:     else
27:         $S \leftarrow S$.merge($G_{SQL}$.forward_gram_exp($node$))
28:     end if
29: end if
30: end function
```

After removing recursive grammar from the SQL syntax paths, SQLBull will further expand these paths into concrete SQL syntax trees. Algorithm 2 shows the algorithm of the syntax tree generation process. SQLBull randomly picks one syntax path from its internal database (Line 4). In the case shown in Figure 7, it picks Syntax Path 1, which connects where_clause, simple_select, and update_stmt. Then,

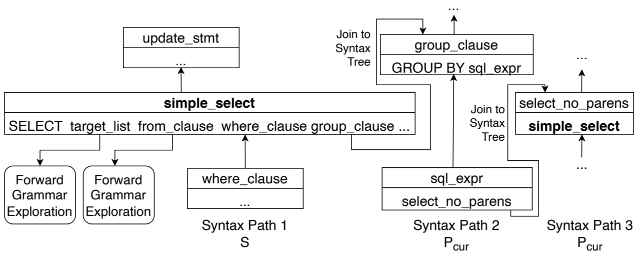


Fig. 7: Process of transforming a BU-generated syntax path into a complete SQL syntax tree. The bold text highlights a recursive grammar introduced when joining multiple syntax paths into one complete SQL syntax tree.

SQLBull scans through the syntax path and handles handle each SQL symbol that needs to be expanded by the grammar rules (Line 5). In the case shown in Figure 7, a sequence of symbols within the simple_select rule requires handling. There are two symbol handling strategies: as demonstrated by the SyntaxTreeGenerationHelper function in Algorithm 2. 1) If one saved syntax path stored in the internal database contains the grammar for the symbol, the syntax path will be merged into the current syntax tree (Line 16 - 25). 2) If no saved syntax path contains the grammar for the symbol, SQLBull will forward traverse the grammar rules starting from the symbol until it reaches the leaves of the grammar (Line 27). In the case shown in Figure 7, Syntax Path 2 merges into Syntax Path 1, which resolves the expansion of the group_clause symbol. Syntax Path 3 is further merged into the syntax tree subsequently, which resolves the expansion of the select_no_parens symbol. When no saved syntax path contains the grammar for the symbol, e.g., the target_list and from_clause symbols in Figure 7, a conservative forward traversal process is adopted to resolve the symbol.

It is worth noting that recursive grammar may be reintroduced in the syntax tree generation process. For example, the simple_select symbol in Figure 7 demonstrates one recursive grammar rule from the generated syntax tree. To reduce the occurrence of the recursive grammar and thus minimize the performance and complexity impacts caused by it, when SQLBull detects a recursive grammar rule in the current syntax tree generation or mutation process, it discards further syntax paths merging and falls back to the more conservative forward traversal approach (Line 13-14, 20-21 in Algorithm 2). The forward traversal approach prioritizes exploring grammar that leads to the leaves of the grammar tree and avoids using known recursive grammar, which makes it less likely to suffer from path explosion.

If one generated SQL query passes all the sanity checks and triggers new DBMS code coverage, SQLBull will mutate the SQL syntax tree of interest. The mutation process of SQLBull is largely the same from the previous work of Squirrel [39]. The only addition of SQLBull is that when applying type-based mutation on the syntax tree (or IR in Squirrel), SQLBull uses not only existing saved syntax trees, but also BU-generated syntax paths to mutate the syntax tree nodes. Figure 6 shows an example of the syntax tree mutation. SQLBull randomly picks one syntax tree node as the mutation target. The mutation target is the sql_expr node in Figure 6. Then SQLBull removes all the children nodes of sql_expr and replaces the children of sql_expr with another saved syntax path. As a result, the "COLUMN_NAME" is mutated to "sql_expr = sql_expr".

IV. Implementation

We implemented SQLBull as a prototype to demonstrate the effectiveness of our Bottom-up SQL generation technique. SQLBull is programmed in C++ and contains ∼60000 lines of code. SQLBull is built on the code base of Squirrel [51] and leverages the semantic-guided instantiation method introduced by Squirrel [39] to fill in the names and constants presented in the generated query syntax tree. It currently supports testing on 5 DBMSs, including MySQL, MariaDB, CockroachDB, DuckDB, and PostgreSQL, by accepting the grammar code of these DBMSs as inputs.

Once a being-tested DBMS crashes or throws an unexpected fatal error indicating memory or data corruption, SQLBull uses the PoC Simplification algorithm shown in Algorithm 3 of Appendix A to simplify the generated query sequence.

SQLBull uses the traditional AFL code coverage mechanism to instrument the C/C++ DBMSs and collect the DBMS branch coverage information. To balance the fuzzing efficiency and the coverage accuracy, we increase the code coverage metadata memory region from size 64K to 256K, which is consistent with other DBMS fuzzing works [17], [39]. For Go programming language-implemented DBMSs such as CockroachDB, we modify the line coverage instrumentation from the default Go built-in library, add the functionality to support basic block coverage, and make it compatible with SQLBull's coverage collection mechanism.

V. Evaluation

The evaluation aims to answer the following research questions.

Q1. Can BU effectively detect real-world DBMS bugs?
Q2. Can SQLBull outperform existing tools in bug detection?
Q3. What is the influence of adjusting the starting point of BU exploration?
Q4. Can BU explorer expose feature-rich SQL features?

A. Experimental Setup

To answer question **Q1**, we evaluated SQLBull on all 5 supported DBMSs, i.e., MySQL, MariaDB, CockroachDB, DuckDB, and PostgreSQL. SQLBull discovered 63 bugs across all 5 DBMSs in total, and we summarize all bug information in §V-B.

To answer question **Q2**, we compared SQLBull with the state-of-the-art DBMS testing tools in §V-C. Unfortunately, because different DBMSs share their own SQL dialects and the dialects are vastly different, no single tool can support all the DBMSs available. Therefore, we select the most advanced and open-source DBMS testing tools and compare them with SQLBull on supported DBMSs. Additionally, because most SQL templates implemented by the baselines do not support

DBMS	SQLBull's Bottom-up Grammar Exploration Starting Points
MySQL	"expr"
MariaDB	"expr"
PostgreSQL	"a_expr", "b_expr", "c_expr"
DuckDB	"a_expr", "b_expr", "c_expr", "d_expr"
CockroachDB	"a_expr", "b_expr", "c_expr", "d_expr"

TABLE I: SQLBull's Bottom-up grammar exploration starting points. SQLBull identifies which SQL grammar expands the symbols listed in the table and backtracks the identified grammar to the root to generate syntax paths.

all SQL statements, to ensure the comparisons between SQLBull and the other tools are fair, we configured SQLBull to only generate SQL statements that are commonly supported by all baselines, specifically, CREATE, INSERT, ALTER, DELETE, UPDATE, and SELECT statements. We note this configuration as SQLBull$_M$.

We include existing works from three main categories. The first category of tools generates testing SQL queries based on handwritten SQL templates. The most advanced tool in this category is SQLancer. Although SQLancer was first introduced in 2022, it has become the most popular platform for implementing the latest SQL testing techniques [52]–[56]. In this evaluation, we include two of the latest and most advanced testing techniques implemented in SQLancer which are capable of detecting DBMS memory errors, i.e., SQLancer$_{+QPG}$ [52] and SQLancer$_{+DQP}$ [53]. We compare SQLBull with SQLancer$_{+QPG}$ on CockroachDB, and SQLancer$_{+DQP}$ on MySQL and MariaDB. In addition, we include SQLsmith$_C$ as an SQL template-based testing tool maintained by the CockroachDB developer team and evaluate it on CockroachDB [57]. The second category of testing baselines is syntax-based fuzzing tools that directly learn the SQL grammar from the DBMS grammar code. We include Squirrel in this category, which has been the default choice by different DBMS vendors to fuzz their products [39]. Furthermore, we implemented a TD syntax-based generation tool that utilizes the ϵ-greedy methodology to decide which grammar to explore from each symbol handling. This tool formulates the TD syntax-based generation as a Multi-arm Bandit (MAB) problem. This methodology of handling TD generation is a well-known method that is widely used in various syntax-based generation tools, including but not necessarily limited to the DBMS testing domain [33], [34], [58]. We include SQLBull$_{TD}$ in this category, which served as a baseline to show the performance of naive TD syntax-based generation. Finally, the third category of testing baselines is the traditional bit-flip mutation-based fuzzing tools. We include the most popular fuzzing tool, AFL, to test all DBMSs that are programmed in C/C++. We use the inputs saved from the Squirrel official repository [51] as the universal input seeds for all baselines that accept SQL queries as input seeds.

To guide SQLBull in exploring the interesting SQL grammar rules, we configure the BU exploration to start from the SQL expressions's grammar by default. Table I shows the default SQL expression symbols for different DBMSs. The grammar of SQL expression is chosen as the starting point because it has been widely used in different DBMS features. Commonly executed and feature-rich SQL features such as GROUP BY, ORDER BY, and WINDOW are all defined on top of the SQL expression grammar. These features have been observed as the construction blocks of various bug-triggering SQL statements. However, to keep the evaluation complementary and fair, we answer question **Q3** by changing the BU grammar exploration to target arbitrary non-SQL expression grammar. We name this configuration of SQLBull as SQLBull$_{NE}$. We demonstrate the influence of adjusting the exploration starting point in §V-D.

To answer question **Q4**, a naive way is to present the number of depths SQLBull can navigate into the grammar. However, this metric is not a good indicator of the quality of the generated SQL queries, even though we claim that the feature-rich SQL features are not typically defined by the grammar in the shallow layers. Because the grammar in the most depth of a syntax tree is usually the ones that implement the most simple features, such as SQL keywords (e.g., NULL), identifiers (e.g., COLUMN_NAME), and constants (e.g., number, text string), being deep in the grammar does not necessarily represent a good quality of the generated SQL queries. Furthermore, the recursive rules presented in the grammar could further enlarge the depths of the generated syntax trees, and they are not helpful in bug finding. Therefore, we discard the grammar navigation depth as a metric to evaluate the quality of the generated SQL queries.

Instead, to answer question **Q4**, we conduct a statistic survey in §V-E to check the exploration frequency of a selective set of SQL features. The selected SQL features are: Subquery, logical processing expressions such as GROUP BY, ORDER BY, WINDOW, HAVING and OVER, joining multiple tables (JOIN), and conditional expressions such as CASE ... WHEN ... ELSE ... END (CASE). The features are selected based on the following criteria: 1) They are implemented on top of the SQL expression grammar rules, which matches the configuration we provided to SQLBull for the BU grammar exploration; 2) They are feature-rich and are commonly seen in the bug-triggering PoCs. We do not claim the selected SQL features represent all the interesting features that could potentially trigger bugs. However, the statistics of these selected SQL features indicate the effectiveness of SQLBull BU grammar exploration, especially when compared with the results from TD generation under similar experimental settings. For completeness, we also include the statistics of these SQL features from the queries generated by other testing tools, including mutation-based fuzzer Squirrel [39] and template-based SQL generator SQLancer [30]. These statistics infer the grammar space exploration priorities for these testing tools.

All evaluations were conducted on a machine with Intel(R) Core(TM) i5-9600KF 6 cores CPU and 48GB of RAM. Each tool was run on a single CPU core and evaluated for 24 hours. We repeated each evaluation 10 times and report only the average result. We chose the latest version of the DBMSs available at the time of the evaluation, i.e., the evaluations were run upon MySQL on version 8.0.40, MariaDB on version 11.7.2, CockroachDB on version v24.3.1, DuckDB on version v1.1.3, and PostgreSQL on version REL_17_4. We compiled all

C/C++ DBMSs with Address Sanitizer and with Assertions enabled. CockroachDB benefits from the Go runtime checks, where triggered memory corruptions are automatically detected and reported to the user.

B. Detected Bugs

SQLBull discovers 63 zero-day bugs across all 5 DBMSs, and we summarize the bug information in Table II. All bugs have been confirmed by the corresponding DBMS developers. Specifically, SQLBull discovers the most bugs on DuckDB, with 31 new unique bugs detected. In addition, SQLBull discovers 10 new unique bugs on MySQL, 5 new unique bugs on MariaDB, 14 new unique bugs on CockroachDB, and 3 new unique bugs on PostgreSQL. All Crash bugs for MySQL and MariaDB are marked with either critical or major severity (while the other DBMSs do not rate the bug severity in their bug tracking system). The bug types are categorized into Crash, Fatal Error, and Assertion Failure. Crash means that the latest release version of the DBMS will segfault when executing the bug-triggering SQL queries. Fatal Error means that when executing the bug-triggering SQL statements, the released version of the DBMS detects an unexpected memory or data corruption error, and the error stops the DBMS from continuing the query execution. For DuckDB, after triggering a Fatal Error, it enters safe execution mode. Any subsequent queries sent to DuckDB for execution will be abandoned unless the user manually restarts the DuckDB process. Assertion Failure represents that the bugs will only be triggered when running the PoC on a DBMS debug build. However, the error thrown by the Assertion indicates that an internal logic error has already occurred from the query execution. It is possible to slightly alter the PoC to trigger more severe bugs, such as Crash or Fatal Error.

All bugs are reported to the corresponding DBMS bug forums, and the developers have confirmed all the reported bugs are zero-day unique bugs, which means these bugs have not been detected by any existing tools before. Except for the bugs reported to MySQL, all other bug reports are publicly available. The MySQL bug reports will open to the public at a later date after a major version of MySQL is released with the bug fix. Among all the bugs, the bugs marked as fixed means that the DBMS developers have already released patches to address the reported bugs. It should be noted that the MySQL developers have acknowledged all the bugs detected by SQLBull. However, they typically release bug fixes altogether with the next major version release of MySQL. At the time of writing, the next major version of MySQL has not been available to the public yet, so most of the reported MySQL bugs remain confirmed.

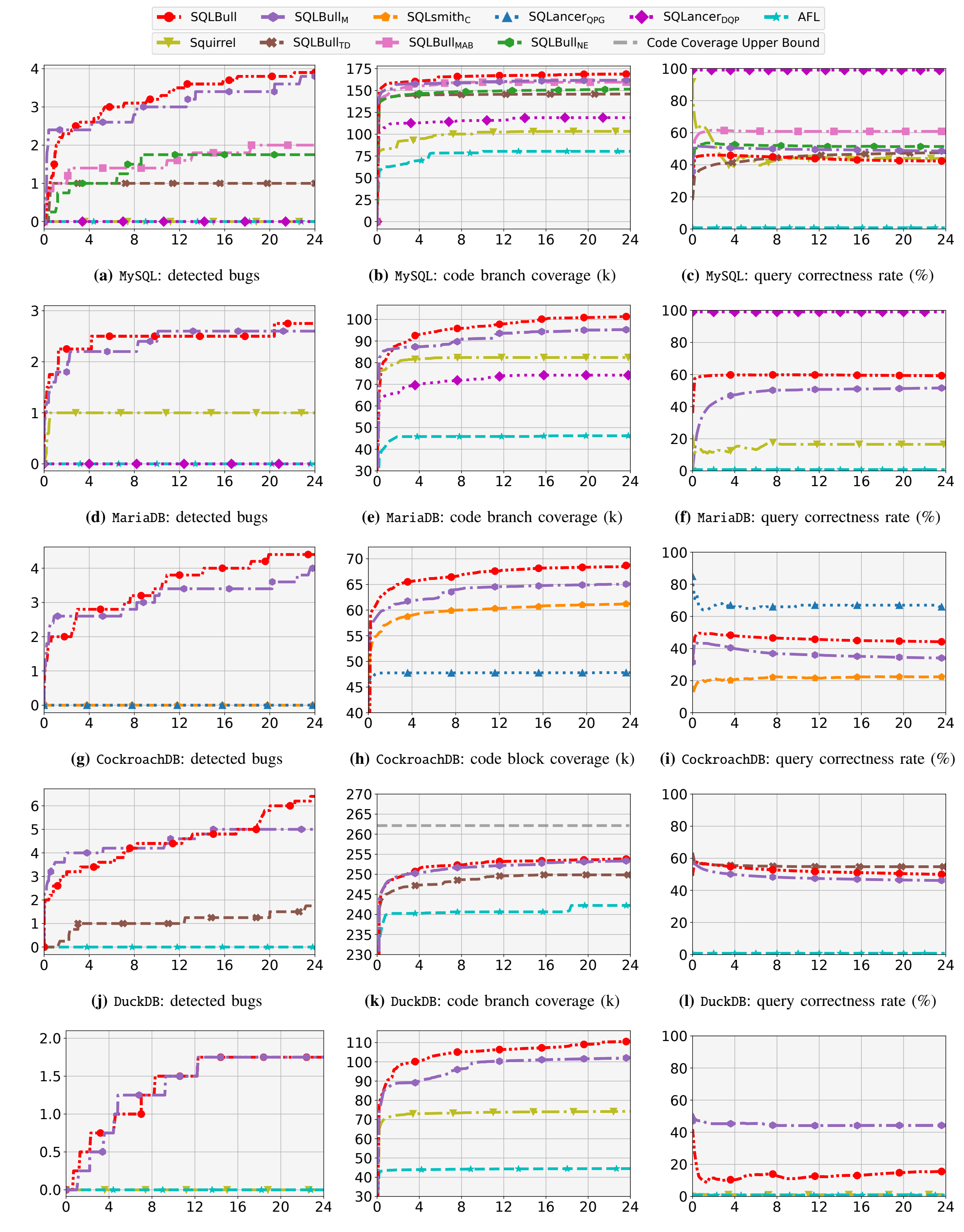
C. Comparison with Existing Tools

To answer how SQLBull performs compared to existing methods, we evaluate SQLBull and $\text{SQLBull}_{\text{M}}$ with state-of-the-art DBMS testing tools. Figure 8a, Figure 8d, Figure 8g, Figure 8j, and Figure 8m show the number of bugs detected by different tools. Figure 8b, Figure 8e, Figure 8h, Figure 8k, and Figure 8n show the branch or basic block coverage when the DBMSs run with different testing tools. Figure 8c, Figure 8f, Figure 8i, Figure 8l, and Figure 8o show the correctness rate of the queries generated or mutated by different tools. Overall, SQLBull and $\text{SQLBull}_{\text{M}}$ reproduced all the bugs detected by the baselines and they achieved the highest number of bugs detected while preserving respectable query correctness rates and the highest code coverage among all other testing tools.

We evaluate Squirrel on MySQL and PostgreSQL. Similar to other syntax-based fuzzing tools, Squirrel ensures the generated query statements are syntactically correct. However, DBMSs may reject the generated queries due to the semantic issues. It achieves a similar query correctness rate as SQLBull when testing on MySQL, about 42%. But Squirrel achieves a lower query correctness rate on PostgreSQL, which is less than 3% compared to SQLBull's $\sim$18%. Squirrel relies on input seed to bootstrap its fuzzing process. It prioritizes all the grammar presented in its input seed, regardless of whether the grammar is helpful in bug discovery or not. Using the default input seed from the Squirrel official repository [51], Squirrel explores 58.8% of the code coverage achieved by SQLBull when testing on MySQL and 68.8% on PostgreSQL. It is worth noting that Squirrel could perform better if we provide an even more comprehensive testing seed that covers more SQL grammar. However, we observed that gathering a comprehensive testing corpus is also challenging and requires a lot of manual work. In comparison, SQLBull does not require any input seed. The DBMS user only needs to configure the SQL features to be tested, and SQLBull will automatically explore the SQL features with diverse generated queries. Overall, SQLBull and $\text{SQLBull}_{\text{M}}$ can find more bugs than Squirrel by more thoroughly exploring the feature-rich grammar of the DBMSs.

$\text{SQLancer}_{\text{+QPG}}$ and $\text{SQLancer}_{\text{+DQP}}$ are the latest and most advanced DBMS testing techniques implemented on top of SQLancer. Both tools utilize the same SQL templates from SQLancer to generate SQL queries but leverage different testing strategies to explore the grammar space of the DBMSs. Interestingly, both techniques achieve a near-perfect query correctness rate close to 99.9%. Especially for $\text{SQLancer}_{\text{+DQP}}$, it is designed to execute the same valid SQL query multiple times with different optimization settings to understand the impact of the query optimization on the query execution plan, which leads to a near-perfect query correctness rate. However, the handwritten SQL templates limit the flexibility of both tools in exploring the vast grammar space of the DBMSs. Although they achieve higher query correctness rates than SQLBull in Figure 8c, Figure 8f, and Figure 8i, they fall behind by achieving a lower code coverage in Figure 8b, Figure 8e, and Figure 8h because the generated queries are less diverse. Both techniques fail to detect as many bugs as SQLBull and $\text{SQLBull}_{\text{M}}$ in our repeated 24-hour experiments.

$\text{SQLsmith}_{\text{C}}$ is yet another handwritten SQL template-based fuzzing tool, mimicking the original SQL syntax implemented in CockroachDB. It applies a custom probabilistic model to decide which grammar rules to explore, and the CockroachDB developers fine-tuned the model to generate more diverse queries. However, the Top-down generation strategy of $\text{SQLsmith}_{\text{C}}$ only focuses on selective SQL features, specifically



(a) `MySQL`: detected bugs **(b)** `MySQL`: code branch coverage (k) **(c)** `MySQL`: query correctness rate (%)

(d) `MariaDB`: detected bugs **(e)** `MariaDB`: code branch coverage (k) **(f)** `MariaDB`: query correctness rate (%)

(g) `CockroachDB`: detected bugs **(h)** `CockroachDB`: code block coverage (k) **(i)** `CockroachDB`: query correctness rate (%)

(j) `DuckDB`: detected bugs **(k)** `DuckDB`: code branch coverage (k) **(l)** `DuckDB`: query correctness rate (%)

(m) `PostgreSQL`: detected bugs **(n)** `PostgreSQL`: code branch coverage (k) **(o)** `PostgreSQL`: query correctness rate (%)

Fig. 8: Evaluation of different testing tools on `MySQL`, `MariaDB`, `CockroachDB`, `DuckDB`, and `PostgreSQL` for 24 hours. The `Code Coverage Upper Bound` line represents the `256K` memory region synchronized between the fuzzer and the DBMS to track the covered branches or basic blocks, i.e., the maximum number of branches or basic blocks coverage is `256K`.

DBMS	ID	Description	Status	Bug Type	SQL Features Under Survey
DuckDB	1	Unexpected ill-formed nested `PIVOT`	fixed (`53a85c50`)	`Crash`	`Subquery`, `JOIN`
	2	Incorrect handling of ENUM in PIVOT	fixed (`f304fe55`)	`Crash`	`Subquery`, `CASE`
	3	Ill-formed CREATE VIEW statement	fixed (`64933f52`)	`Crash`	/
	4	Uncaught exception in EXPLAIN SELECT	fixed (`292d40e2`)	`Crash`	/
	5	Subquery binding error	confirmed (`15640`)	`Fatal Error`	`Subquery`, `GROUP BY`, `HAVING`
	6	JOIN USING columns handling from BindContext	fixed (`22a67063`)	`Fatal Error`	`Subquery`, `JOIN`
	7	Incorrect handling of COALESCE with FULL JOIN	fixed (`200e8bd5`)	`Fatal Error`	`Subquery`, `JOIN`
	8	Unsupported POSITIONAL JOIN in subquery	fixed (`854885cf`)	`Fatal Error`	`Subquery`, `JOIN`
	9	Incorrect handling of subqueries	confirmed (`15525`)	`Fatal Error`	`Subquery`, `ORDER BY`
	10	Incorrect handling for subqueries in LATERAL JOIN	fixed (`ecfe7397`)	`Fatal Error`	`Subquery`, `JOIN`
	11	Issue in LATERAL JOIN handling	confirmed (`15344`)	`Fatal Error`	`Subquery`, `JOIN`
	12	Issues with unpacked columns and the NOT operator	fixed (`a1335f6f`)	`Fatal Error`	`CASE`, `JOIN`
	13	Cannot copy bound subquery node	confirmed (`15657`)	`Fatal Error`	`Subquery`, `OVER`
	14	Incorrect handling for execute_cast	fixed (`1bf1ed97`)	`Fatal Error`	`CASE`
	15	Incorrect RIGHT JOIN handling	fixed (`2035c3cc`)	`Fatal Error`	`JOIN`
	16	Incorrect type handling of subqueries	fixed (`4ddedc46`)	`Fatal Error`	`Subquery`
	17	Uncaught error within in-map casting	fixed (`5a3fde7d`)	`Fatal Error`	`CASE`
	18	INSERT BY NAME + DEFAULT VALUES	fixed (`5ebf174e`)	`Fatal Error`	`CASE`
	19	USING columns of FULL JOIN in PIVOT	fixed (`88c73e6c`)	`Fatal Error`	`JOIN`
	20	Incorrect handling of statement parameter	fixed (`9199d11a`)	`Fatal Error`	/
	21	Issue with subquery result type handling	fixed (`63f6bc2b`)	`Assertion Failure`	`Subquery`, `CASE`, `ORDER BY`, `GROUP BY`, `WINDOW`
	22	Incorrectly apply DISTINCT on empty target list	fixed (`f35f9f9a`)	`Assertion Failure`	`Subquery`, `GROUP BY`, `WINDOW`
	23	Incorrect handling of STAR expression with empty value	confirmed (`16828`)	`Assertion Failure`	`Subquery`, `GROUP BY`
	24	Issues related to USING Binding handling	fixed (`a1335f6f`)	`Assertion Failure`	`CASE`, `JOIN`
	25	Another issues related to USING context handling	fixed (`339419e9`)	`Assertion Failure`	`Subquery`, `JOIN`
	26	Incorrect handling of duplicated alias from JOINs	fixed (`3ae5cf3b`)	`Assertion Failure`	`Subquery`, `JOIN`
	27	Issue in generated column specifying	fixed (`23feff31`)	`Assertion Failure`	`CASE`
	28	Array Type handling error	confirmed (`16827`)	`Assertion Failure`	`Subquery`
	29	Incorrect handling comparison type	fixed (`ddcb3e74`)	`Assertion Failure`	/
	30	Faulty assertion in LOGICAL PROJECTION	fixed (`08b0415c`)	`Assertion Failure`	/
	31	Incorrect handling of SQL expression unfolding	confirmed (`16826`)	`Assertion Failure`	/
MySQL	32	Segfault in JOIN::refresh_base_slice	confirmed (`117082`)	`Crash`	`Subquery`, `CASE`, `JOIN`
	33	Segfault in Table_ref::fetch_number_of_rows	fixed (`8.0.42`, `8.4.5`, `9.3.0`)	`Crash`	`Subquery`
	34	Segfault in VALIDATION	confirmed (`117207`)	`Crash`	/
	35	Assertion failure within indexed tree modification	confirmed (`117066`)	`Assertion Failure`	`Subquery`, `CASE`, `ORDER BY`, `GROUP BY`, `OVER`
	36	Assertion 'sl->join == nullptr ‖ is_executed()' failed	confirmed (`117080`)	`Assertion Failure`	`Subquery`, `JOIN`, `GROUP BY`, `HAVING`, `ORDER BY`
	37	Assertion `escape_arg != nullptr` failed	confirmed (`117065`)	`Assertion Failure`	`CASE`, `OVER`
	38	MoveCompositeIteratorsFromTablePath	confirmed (`117079`)	`Assertion Failure`	`ORDER BY`, `OVER`
	39	Incorrect handling in constant propagation	confirmed (`117068`)	`Assertion Failure`	`GROUP BY`, `HAVING`
	40	Assertion `hton->flags & HTON_IS_SECONDARY` failed	confirmed (`117061`)	`Assertion Failure`	/
	41	Assertion failure in `fill_alter_inplace_info`	confirmed (`117064`)	`Assertion Failure`	/
CockroachDB	42	Panic: SHOW EXPERIMENTAL_FINGERPRINTS	fixed (`e248fccc`)	`Crash`	`Subquery`, `ORDER BY`, `WINDOW`
	43	Panic: Ill-formed CREATE TABLE statement	fixed (`ad1f8ab5`)	`Crash`	`Subquery`, `WINDOW`
	44	Panic: SELECT with JOIN and SYSTEM TIME	confirmed (`133395`)	`Crash`	`JOIN`
	45	Panic on AS OF SYSTEM TIME	fixed (`d9cd2a2f`)	`Crash`	/
	46	Panic: function in SET LOCAL SCHEMA	fixed (`5bcb5801`)	`Crash`	/
	47	Internal Error: vectorized engine error	confirmed (`130354`)	`Fatal Error`	`Subquery`
	48	Internal Error: ill-formed cursor fetch	fixed (`7dd9e95f`)	`Fatal Error`	`Subquery`
	49	Invalid memory address or nil pointer dereference	confirmed (`131121`)	`Fatal Error`	`Subquery`
	50	Internal Error: index out of range with annotation	fixed (`8847382d`)	`Fatal Error`	`Subquery`
	51	Internal Error: BUCKET_COUNT = NULL	fixed (`1bdef168`)	`Fatal Error`	/
	52	Internal Error: Error Stats from column set	confirmed (`130593`)	`Fatal Error`	/
	53	Internal Error: NULL virtual primary key column	fixed (`c7af6aaf`)	`Fatal Error`	/
	54	Internal Error when using incorrect type in REVOKE	fixed (`4b26b964`)	`Fatal Error`	/
	55	Using POSITION that returns NULL with JOIN	confirmed (`132577`)	`Fatal Error`	/
MariaDB	56	Segfault: internal_str2dec on INSERT	confirmed (`36354`)	`Crash`	`Subquery`, `JOIN`
	57	Segfault: Item_subselect::init	confirmed (`36353`)	`Crash`	`Subquery`
	58	Assertion failure `builtin_select.first...`	confirmed (`36369`)	`Assertion Failure`	`CASE`, `Subquery`, `LIMIT`
	59	Assertion failure `ASSERT(0)`	confirmed (`36370`)	`Assertion Failure`	/
	60	Assertion failure `!(thd->lex)->if_exists()`	confirmed (`36371`)	`Assertion Failure`	/
PostgreSQL	61	Segfault: Incorrect handling of duplicated SET DEFAULT list	confirmed (`18879`)	`Crash`	/
	62	Assertion Failure: `cte->ctequery...`	confirmed (`18877`)	`Assertion Failure`	`Subquery`
	63	Assertion Failure: `cte->cterecursive || ...`	confirmed (`18878`)	`Assertion Failure`	`Subquery`, `ORDER BY`, `GROUP BY`, `JOIN`

TABLE II: New Unique Zero-day Bugs Detected by `SQLBull`. `SQLBull` detects 63 bugs in total, including 15 crashes, 25 internal errors, and 23 assertion failures. Among them, 34 bugs are fixed. `SQL Features Under Survey` lists a few interesting SQL features presented in the bug-triggering PoCs. The selection of these features is discussed in §V-A. `/` means none of the SQL feature under survey is presented in the PoC. If the bug is marked as `fixed`, we provide the hash of the source control commit (in the bracket) that fixes the bug. If the bug is marked as `confirmed`, we provide the `Bug Tracking ID` for each DBMS bug reporting system in the bracket.

`GROUP BY`, `ORDER BY`, `HAVING`, and `WINDOW` expressions, thus falls short on detecting bugs related to other SQL features.

We then compare `SQLBull` with a well-known syntax-based generation technique, $\texttt{SQLBull}_{\texttt{MAB}}$, which utilizes the ϵ-greedy strategy to determine which grammar rules to explore at each step of the `TD` grammar exploration. The reward of $\texttt{SQLBull}_{\texttt{MAB}}$ is determined by the acceptance rate and the code coverage brought by the generated queries. During testing, we observed

DBMS	SQL Features Under Survey				
	Subquery	GROUP BY, ORDER BY, HAVING, WINDOW, OVER	JOIN	Conditional SQL Expressions	Total
DuckDB	17 (54.8%)	5 (16.1%)	12 (38.7%)	8 (25.8%)	25 (80.6%)
MySQL	4 (40.0%)	4 (40.0%)	2 (20.0%)	3 (30.0%)	7 (70.0%)
CockroachDB	6 (42.9%)	2 (14.3%)	1 (7.14%)	0 (0.00%)	7 (50.0%)
MariaDB	2 (40.0%)	1 (20.0%)	1 (20.0%)	1 (20.0%)	3 (60.0%)
PostgreSQL	2 (66.6%)	1 (33.3%)	1 (33.3%)	0 (0.00%)	2 (66.6%)
Total	31 (49.2%)	13 (20.6%)	17 (30.0%)	12 (19.0%)	44 (69.8%)

TABLE III: Statistics of SQL features under survey upon all the zero-day bugs detected by `SQLBull`.

DBMS	Tools	SQL Features Under Survey			
		Subquery	GROUP BY, ORDER BY, HAVING, WINDOW, OVER	JOIN	Conditional SQL Expressions
MySQL	`SQLBull`	69.5%	51.1%	21.8%	34.1%
	`SQLBull`$_{\mathtt{M}}$	69.8%	57.7%	18.0%	64.1%
	`SQLBull`$_{\mathtt{TD}}$	41.3%	13.2%	1.3%	0.38%
	`SQLBull`$_{\mathtt{MAB}}$	47.4%	42.8%	18.5%	3.9%
	`SQLBull`$_{\mathtt{NE}}$	46.0%	40.8%	4.57%	12.6%
	`Squirrel`	19.3%	19.5%	13.1%	5.6%
	`AFL`	0.00%	0.00%	0.00%	0.00%
CockroachDB	`SQLBull`	46.9%	33.0%	39.2%	39.0%
	`SQLBull`$_{\mathtt{M}}$	55.6%	57.1%	26.5%	33.4%
	`SQLancer`	3.6%	18.9%	5.3%	7.2%
	`SQLsmith`	12.8%	62.6%	9.8%	3.6%

TABLE IV: Statistics of SQL features under survey upon different tools. The statistics are calculated based on all the SQL queries generated from each testing tool running for 24 hours. We provide 0% statistics for `AFL` to demonstrate that it does not effectively explore these SQL features by mutation, but simply repeats the features presented in the seeds.

that `SQLBull`$_{\mathtt{MAB}}$ tends to generate correct but simple queries, such as "(((TABLE t0)))", where the repeated parenthesized expression is a recursive expression to hold the "`TABLE`" sub-expression. These repeated, simple yet correct expressions are promoted because they can easily pass the semantic check of the DBMS. However, `SQLBull`$_{\mathtt{MAB}}$ rarely tested the feature-rich SQL features. Because the grammar rules that define feature-rich SQL features are not in the shallow layers and the exploration of the deeper and feature-rich grammar is more likely to lead to sanity check rejections, the exploration of these feature-rich grammar would lower the reward of `SQLBull`$_{\mathtt{MAB}}$, and `SQLBull`$_{\mathtt{MAB}}$ would avoid exploring feature-rich grammar. `SQLBull`$_{\mathtt{MAB}}$ finds `2` bugs in the evaluation by only exploring the shallow grammar rules. In fact, these `2` bugs are easy to detect, as `SQLBull` and `SQLBull`$_{\mathtt{M}}$ also reproduced them. The bug IDs are `40` and `41` in Table II.

`SQLBull`$_{\mathtt{TD}}$ employs naive Top-down syntax-based SQL generation. It suffers from the performance issue introduced by the recursive grammar, resulting in only simple queries being generated using shallow grammar. This is demonstrated by the significantly lower code coverage and fewer bugs found compared to `SQLBull` in Figure 8a, Figure 8b, Figure 8j, and Figure 8k. It finds three easy-to-detect bugs also reproduced by `SQLBull` and `SQLBull`$_{\mathtt{M}}$, Bug IDs `30`, `31`, and `40` in Table II.

Lastly, we evaluate the traditional bit-flipping fuzzing tool, `AFL`, on all `C/C++` DBMSs. `AFL` does not understand the SQL grammar when mutating the inputs. Instead, it randomly flips the bits of the input SQL query or randomly concatenates part of one SQL query with another SQL query. Naturally, this mutation strategy will end up with a low query correctness rate where almost all the mutated queries are rejected by the DBMS parser. The low query correctness rate is verified in Figure 8c, Figure 8f, Figure 8l, and Figure 8o, where all the query correctness rates are lower than `1%`. The inefficient mutation strategy also stops `AFL` from discovering new feature-rich grammar (Figure 8b, Figure 8e, Figure 8k, and Figure 8n). `AFL` found no bugs. In summary, AFL's inefficiency implies the need to apply the syntax-based testing technique in DBMS bug hunting.

D. Influence of Bottom-up Explorer Starting Point

The default configuration of `SQLBull` targets SQL expressions as the `BU` starting points, as shown in Table I. However, this configuration is not the only choice. In this section, we investigate the influence of different starting points on the performance of `SQLBull`. We redirect the `BU` grammar exploration to start from arbitrary grammar rules except the default `SQL expression` ones, and evaluate the performance of `SQLBull`$_{\mathtt{NE}}$ on `MySQL`.

As shown in Figure 8b, `SQLBull`$_{\mathtt{NE}}$ achieves a lower code coverage than `SQLBull` (∼`13%` lower) because it is distracted by exploring arbitrary grammar rules that may not be feature-rich. Specific explored grammar rules may even be marked as `Unimplemented` or directly lead to sanity check rejections. Fortunately, the `BU` grammar exploration still helps `SQLBull`$_{\mathtt{NE}}$ to find `2` bugs on average in one run, proving its effectiveness by discovering Bug IDs `37`, `38`, and `40` in Table II across different runs. Consequently, we encourage the DBMS testers to freely attempt other `BU` starting points. For example, redirect `SQLBull` to target previously bug-triggering SQL grammar or target newly introduced SQL grammar changes. We believe `SQLBull` is capable of achieving surprising results given the flexibility and effectiveness of the `BU` grammar exploration.

E. Statistics of SQL Features

In this section, we investigate the effectiveness of `SQLBull` in exposing feature-rich SQL features. Because we configure `SQLBull` to explore the `SQL expression` grammar, `SQLBull` is tailored to uncover the grammar that contains `SQL expressions` in the PoCs. As a result, of all the bugs detected by `SQLBull`, 44 (69.8 %) of the bug-triggering PoCs contain the SQL features under survey (§V-A) after applying the PoC simplification steps from Algorithm 3 in Appendix A. Table III presents the statistics of SQL features under survey upon all the zero-day bugs detected by `SQLBull`, demonstrating that all SQL features under survey are heavily contributing to the bug finding. Table II further details which bug-triggering PoC contains which SQL features. Because each selected SQL feature relates to a non-trivial number of bug-triggering SQL queries, we believe `SQLBull` is capable of effectively exploring all these SQL features by traversing the SQL grammar. Table IV further shows how frequently different testing tools explore these SQL features. While most baselines prioritized generating queries with Subquery and logical processing expressions such as GROUP BY, they generated significantly fewer tests with JOIN and Conditional SQL Expressions, limiting their bug-finding effectiveness. SQLBull and SQLBull_{M}, instead, can more thoroughly explore all the SQL features, so they can detect more bugs. SQLBull_{NE} exposes fewer SQL features than SQLBull, because it is not configured to target the SQL expression grammar rules, on which all SQL features under survey are defined. AFL hardly generates any valid SQL queries, and thus does not explore any of the SQL features under survey.

VI. Related Works

Syntax-based automatic testing has been widely adopted to find bugs in various software systems [59]–[62]. We first discuss the existing syntax-based testing tools for the DBMS testing community [30], [39], [52] in §VI-A and §VI-B. We extend the scope of syntax-based testing tools to other security-oriented testing domains [63]–[66] in Appendix B.

A. SQL Template-based DBMS Testing Tools

Certain existing DBMS testing tools rely on handwritten SQL templates to generate the SQL queries. These templates are implemented with custom programming languages, such as C/C++ [31], [35], Java [27], [30], and languages designed for SAT solvers [67], [68]. The most well-known template-based query testing framework is SQLsmith [31]. The developers of CockroachDB recently ported SQLsmith to support their own DBMS [57], which is referred to as SQLsmith_{C} in this paper. SQLancer [30] is another template-based DBMS testing tool that detects DBMS memory and logic bugs [54]–[56]. Its latest improvement, SQLancer_{+QPG} [52] and SQLancer_{+DQP} [53], leverage the DBMS query plan and optimization hints to guide the query generation. All of the SQL template-based DBMS testing tools suffer from the same limitation: their hand-crafted SQL templates are limited in covering the complete set of SQL grammar implemented by the DBMSs.

B. Grammar Code-based DBMS Testing Tools

Several DBMS testing frameworks utilize the DBMS built-in grammar code to generate syntactically valid SQL queries. Squirrel is one of the state-of-the-art syntax-based DBMS fuzzing tools in this category [39]. Adopting the Top-down generation methodology, Squirrel parses the DBMS grammar into its internal representation (IR) and employs type-based IR mutation to its input corpus to generate new queries. RATEL extends Squirrel's capabilities to test enterprise-level DBMSs like GaussDB [69]. LEGO instantiates the query statements with type-affinity awareness for a higher query correctness rate [70]. WingFuzz optimizes the fuzzing feedback to reduce the coverage noise caused by concurrent DBMS threads [71]. However, all these tools mentioned above prioritize all the SQL grammar presented in the input seed, even if some are not useful for bug triggering. Thus, they waste time testing non-critical SQL features and are inefficient in exposing DBMS bugs that require feature-rich SQL features. Additionally, DynSQL [72] implements another DBMS fuzzing technique that improves the query correctness rate based on the real-time feedback from the DBMS. The idea of DynSQL is complementary to our work. We can combine the idea of DynSQL with SQLBull to improve the fuzzing efficiency further. BuzzBee is another syntax-based fuzzing tool that expands the dynamic semantic resolving technique from Squirrel to other non-relational DBMSs [73]. While BuzzBee claims to support one relational DBMS, PostgreSQL, the author of BuzzBee removes the code of PostgreSQL grammar handling in its public repo, which is the key component for BuzzBee to generate valid PostgreSQL queries. Unfortunately, except for Squirrel, none of the other tools are open-sourced. Of all the testing tools introduced, Squirrel remains one of the most effective DBMS fuzzing tool to detect DBMS memory bugs, and it is the default go-to tool for the security community to detect DBMS memory bugs. Therefore, we compare SQLBull with Squirrel in this paper.

VII. Discussion

Grammar coverage from Bottom-up grammar exploration. By building the BU grammar path from specific grammar starting points, sometimes it is infeasible to cover all the SQL statement types supported by the DBMS. For example, it is impossible to JOIN multiple tables in the ALTER TABLE DROP COLUMN statement. Therefore, giving a set of specific interesting grammar. SQLBull is not guaranteed to cover all the SQL grammar supported by the DBMS. We encourage the DBMS testers to try different sets of SQL grammar rules as the starting points of the BU grammar exploration, which potentially boosts SQLBull with better SQL feature coverage. It is also a promising research direction to dynamically introduce new BU exploration starting points based on the feedback from the fuzzing process. We will leave this as our future work.

VIII. Conclusion

In this work, we propose a new Bottom-up (BU) syntax-based SQL generation technique, which redirects more testing resources to explore SQL grammar rules that define the syntaxes of feature-rich SQL functionalities. The prototype syntax-based fuzzer, SQLBull, adopts the BU strategy with the following key designs: 1) Prioritize grammar exploration on the feature-rich grammar. 2) Handle recursive grammar efficiently. 3) Generate diverse SQL queries that explore the interesting grammar rules. In the evaluation, SQLBull discovers 63 unique zero-day bugs across 5 well-tested DBMSs, outperforming all existing tools and demonstrating the effectiveness of BU.

Acknowledgment

We want to thank our reviewers for their insightful comments. Peng Liu was supported by NSF CNS-2019340 and NSF ECCS-2140175.

References

[1] "PostgreSQL Clients," https://www.postgresql.org/about/, 2025, (visited in April 2025).

[2] "MySQL Customers," https://www.mysql.com/customers/, 2025, (visited in April 2025).

[3] "CockroachDB Customers," https://www.cockroachlabs.com/customers/, 2025, (visited in April 2025).

[4] "Awesome DuckDB," https://github.com/davidgasquez/awesome-duckdb, 2025, (visited in April 2025).

[5] "Starbucks : A Case Study on the Impact of ERP Systems," https://pangrow.com/blog/business-automation/starbucks-erp-case-study/, 2024, (visited in April 2025).

[6] Y. Yusuf, A. Gunasekaran, and M. S. Abthorpe, "Enterprise Information Systems Project Implementation: A Case Study of ERP in Rolls-Royce," *International journal of production economics*, vol. 87, no. 3, pp. 251–266, 2004.

[7] M. Ali and L. Miller, "ERP System Implementation in Large Enterprises–A Systematic Literature Review," *Journal of enterprise information management*, vol. 30, no. 4, pp. 666–692, 2017.

[8] S. V. Grabski, S. A. Leech, and P. J. Schmidt, "A Review of ERP Research: A Future Agenda for Accounting Information Systems," *Journal of information systems*, vol. 25, no. 1, pp. 37–78, 2011.

[9] I.-Y. Song and K.-Y. Whang, "Database Design for Real-world E-commerce Systems," *IEEE Data Eng. Bull.*, vol. 23, no. 1, pp. 23–28, 2000.

[10] S. DeFazio, R. Krishnan, J. Srinivasan, and S. Zeldin, "The Importance of Extensible Database Systems for E-commerce," in *Proceedings 17th International Conference on Data Engineering*. IEEE, 2001, pp. 63–70.

[11] F. Y. Ahmed, R. Sreejith, and M. I. Abdullah, "Enhancement of E-Commerce Database System During the COVID-19 Pandemic," in *2021 IEEE 11th IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. IEEE, 2021, pp. 174–179.

[12] L. Lim and M. Wang, "Managing E-commerce Catalogs in a DBMS with Native XML Support," in *IEEE International Conference on e-Business Engineering (ICEBE'05)*. IEEE, 2005, pp. 564–571.

[13] H. Wang, "Online banking: A case study for dynamic database-driven client/server system," Ph.D. dissertation, Concordia University, 2000.

[14] P. Rai, A. Singh, T. Sharma, and N. Sreenarayanan, "Banking Management System—The Web Application Way," in *Applications of Computational Methods in Manufacturing and Product Design: Select Proceedings of IPDIMS 2020*. Springer, 2022, pp. 581–590.

[15] A. Ayman, M. Sherif *et al.*, "Review of Online Banking Systems: Innovations and Challenges," *Advanced Sciences and Technology Journal*, 2024.

[16] H. Upadhyay, R. Wadhawan, and D. Gupta, "Integrating Multiple Security Features in and Online Banking Platform," 2024.

[17] 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.

[18] J. Liang, Z. Wu, J. Fu, M. Wang, C. Sun, and Y. Jiang, "Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[19] Y. Fu, Z. Wu, Y. Zhang, J. Liang, J. Fu, Y. Jiang, S. Li, and X. Liao, "THANOS: DBMS Bug Detection via Storage Engine Rotation Based Differential Testing," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 1–12.

[20] M. Rigger, "Bugs Found in Database Management Systems," https://www.manuelrigger.at/dbms-bugs/, (visited in April 2025).

[21] "The Knight Capital Disaster - Speculative Branches," https://specbranch.com/posts/knight-capital/, 2012, (visited in April 2025).

[22] Binder, Matt, "Apple Explains Why Deleted Photos were being Restored on some iPhones," https://mashable.com/article/apple-iphone-deleted-photos-restored-explained, 2024, (visited in April 2025).

[23] Federal Trade Commission, "Equifax Data Breach Settlement," https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement, 2019, (visited in April 2025).

[24] Nick Baker, "NYSE Fined $14 Million by SEC for Series of Rule Violations," https://www.bloomberg.com/news/articles/2018-03-06/nyse-fined-14-million-by-sec-for-a-series-of-rule-violations, 2018, (visited in April 2025).

[25] C. Mishra, N. Koudas, and C. Zuzarte, "Generating Targeted Queries for Database Testing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, New York, NY, USA, 2008.

[26] J. Wang, P. Zhang, L. Zhang, H. Zhu, and X. Ye, "A Model-based Fuzzing Approach for DBMS," in *2013 8th International Conference on Communications and Networking in China (CHINACOM)*. IEEE, 2013, pp. 426–431.

[27] E. Lo, C. Binnig, D. Kossmann, M. Tamer Özsu, and W.-K. Hon, "A Framework for Testing DBMS Features," *The VLDB Journal*, vol. 19, no. 2, pp. 203–230, Apr. 2010.

[28] B. Chandra, B. Chawda, B. Kar, K. V. M. Reddy, S. Shah, and S. Sudarshan, "Data Generation for Testing and Grading SQL Queries," *The VLDB Journal*, vol. 24, no. 6, Aug 2015.

[29] G. Li, X. Zhou, S. Li, and B. Gao, "QTune: A Query-aware Database Tuning System with Deep Reinforcement Learning," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2118–2130, 2019.

[30] Manuel Rigger, "SQLancer," https://github.com/sqlancer/sqlancer, (visited in April 2025).

[31] "SQLSmith," https://github.com/anse1/sqlsmith, (visited in April 2025).

[32] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller, "Inputs From Hell," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1138–1153, 2020.

[33] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[34] L. Fernandez, G. Karlsson, and D. Hübinette, "A Framework for Feedback-enabled Blackbox Fuzzing Using Context-free Grammars," 2022, (visited in April 2025).

[35] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems," in *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, 2020.

[36] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*, 2021, pp. 244–256.

[37] Y. Liang and H. Hu, "Parser knows best: Testing dbms with coverage-guided grammar-rule traversal," *arXiv preprint arXiv:2503.03893*, 2025.

[38] J. Patra and M. Pradel, "Learning to Fuzz: Application-independent Fuzz Testing with Probabilistic, Generative Models of Input Data," *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[39] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, USA, Nov. 2020.

[40] "The Lemon LALR(1) Parser Generator," https://www.sqlite.org/lemon.html, 2025, (visited in April 2025).

[41] S. C. Johnson and R. Sethi, "Yacc: a parser generator," *UNIX Vol. II: research system*, pp. 347–374, 1990.

[42] T. Parr and K. Fisher, "Ll (*) the foundation of the antlr parser generator," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[43] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[44] "GNU Bison: Languages and Context-Free Grammars," https://www.gnu.org/software/bison/manual/html_node/Language-and-Grammar.html, 2023, (visited in April 2025).

[45] "ANTLR: Parser Rules," https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md, 2023, (visited in April 2025).

[46] "DuckDB Github Repository," https://github.com/duckdb/duckdb.git, 2025, (visited in April 2025).

[47] "MySQL Parser," https://github.com/mysql/mysql-server/blob/trunk/sql/sql_yacc.yy, (visited in April 2025).

[48] "MariaDB Parser," https://github.com/MariaDB/server/blob/11.3/sql/sql_yacc.yy, (visited in April 2025).

[49] "PostgreSQL: The Parser Stage," https://www.postgresql.org/docs/current/parser-stage.html, (visited in April 2025).

[50] "CockroachDB Parser," https://github.com/cockroachdb/cockroach/blob/master/pkg/sql/parser/sql.y, (visited in April 2025).

[51] "Squirrel Github Repository," https://github.com/s3team/Squirrel, 2022, (visited in April 2025).

[52] J. Ba and M. Rigger, "Testing Database Engines via Query Plan Guidance," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023.

[53] ——, "Keep it simple: Testing databases via differential query plans," *Proc. ACM Manag. Data*, vol. 2, no. 3, May 2024. [Online]. Available: https://doi.org/10.1145/3654991

[54] M. Rigger and Z. Su, "Testing Database Engines via Pivoted Query Synthesis," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, 2020.

[55] ——, "Finding Bugs in Database Systems via Query Partitioning," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[56] ——, "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 (ESEC/FSE)*, 2020.

[57] M. Jibson, "SQLsmith: Randomized SQL Testing in CockroachDB," https://www.cockroachlabs.com/blog/sqlsmith-randomized-sql-testing/, 2019, (visited in April 2025).

[58] E. Pavese, E. Soremekun, N. Havrikov, L. Grunske, and A. Zeller, "Inputs From Hell: Generating Uncommon Inputs From Common Samples," *arXiv preprint arXiv:1812.07525*, 2018.

[59] S. Deb, K. Jain, R. Van Tonder, C. Le Goues, and A. Groce, "Syntax Is All You Need: A Universal-Language Approach to Mutant Generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 654–674, 2024.

[60] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based Whitebox Fuzzing," in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 2008, pp. 206–215.

[61] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan, "Grammar-based Fuzzing," in *2018 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 2018, pp. 32–35.

[62] Google, "Syzkaller - Kernel Fuzzer," https://github.com/google/syzkaller, 2019.

[63] S. Tang, S. Liu, J. Wang, and X. Zhang, "An Empirical Study on AST-level Mutation-based Fuzzing Techniques for JavaScript Engines," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, 2023, pp. 216–226.

[64] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware Greybox Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[65] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: A Grammar-based Open Source Fuzzer," in *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, 2018, pp. 45–48.

[66] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One Engine to Fuzz'em All: Generic Language Processor Testing with Semantic Validation," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[67] S. Abdul Khalek and S. Khurshid, "Automated SQL Query Generation for Systematic Testing of Database Engines," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010.

[68] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut, "Test Input Generation for Database Programs Using Relational Constraints," in *Proceedings of the Fifth International Workshop on Testing Database Systems (DBTest)*, Scottsdale, Arizona, 2012.

[69] 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)*, 2021.

[70] J. Liang, Y. Chen, Z. Wu, J. Fu, M. Wang, Y. Jiang, X. Huang, T. Chen, J. Wang, and J. Li, "Sequence-oriented DBMS Fuzzing," in *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2023.

[71] J. Liang, Z. Wu, J. Fu, Y. Bai, Q. Zhang, and Y. Jiang, "WingFuzz: Implementing Continuous Fuzzing for DBMSs," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 479–492.

[72] Z.-M. Jiang, J.-J. Bai, and Z. Su, "DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation," in *Proceedings of USENIX Security Symposium (USENIX Security)*, 2023.

[73] Y. Yang, Y. Chen, R. Zhong, J. Chen, and W. Lee, "Towards generic database management system fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 901–918.

[74] S. Jiang, Y. Zhang, J. Li, H. Yu, L. Luo, and G. Sun, "A Survey of Network Protocol Fuzzing: Model, Techniques and Directions," *arXiv preprint arXiv:2402.17394*, 2024.

[75] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript Engines with Aspect-preserving Mutation," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2020.

[76] M. Raselimo, J. Taljaard, and B. Fischer, "Breaking Parsers: Mutation-based Generation of Programs with Guaranteed Syntax Errors," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, 2019, pp. 83–87.

APPENDIX

A. PoC Simplification Algorithm

When the DBMS crashes or throws fatal error indicating memory or data corruption, `SQLBull` uses the `PoC Simplification` algorithm shown in Algorithm 3 to simplify the generated SQL query sequence and save the simplified queries as bug PoC.

Algorithm 3: PoC Simplification

```
1: function PoCSimplification($stmt\_seq$)
2: $simpl\_stmt\_seq \leftarrow \emptyset$
3: for all $stmt \in stmt\_seq$ do
4:     reset_database()
5:     for all $stmt' \in stmt$ do
6:         if $stmt == stmt'$ then
7:             continue
8:         end if
9:         execute($stmt'$)
10:    end for
11:    if not database_crash() then
12:        $simpl\_stmt\_seq \leftarrow simpl\_stmt\_seq + stmt$
13:    end if
14: end for
15: for all $stmt \in simpl\_stmt\_seq$ do
16:    for all $tree\_node \in stmt$.get_tree_nodes() do
17:        reset_database()
18:        for all $stmt' \in simpl\_stmt\_seq$ do
19:            if $stmt == stmt'$ then
20:                $stmt' \leftarrow stmt'$.copy().remove($tree\_node$)
21:            end if
22:            execute($stmt'$)
23:        end for
24:        if not database_crash() then
25:            $stmt \leftarrow stmt$.remove($tree\_node$)
26:        end if
27:    end for
28: end for
29: return $simpl\_stmt\_seq$
30: end function
```

B. Syntax-based Testing Tools on Other Systems

The technique of syntax-based testing has also been widely adopted in other systems, such as programming systems (e.g., JavaScript engines) [63] and network protocol implementations [74]. Specifically, [34] proposes a syntax-based fuzzing framework and applies the learned grammar to generate Link Layer Discovery Protocol (LLDP). `DIE` applies type-based AST mutation extracted from the JavaScript grammar code while preserving the context-rich parts of the input based on the intuition of the JavaScript programs [75]. `NAUTILUS` demonstrates that code coverage feedback is beneficial for guiding syntax-based input generation [33]. [76] proposes a syntax-based fuzzing tool that intentionally introduces slight syntactic errors to the generated inputs to expose faulty parser logic. Many of these previous works formulate the task of syntax-based input generation as a Multi-arm Bandit (MAB) problem [33], [34], [58]. However, they all employ Top-down generation, which is not suitable for deep bug discovery in DBMS. To prioritize feature-rich SQL grammar that is more likely to expose bugs, some existing works [32], [38], [58] prioritize the grammar rules used to construct the previous failures. However, the previous failure inputs may not contain the deep SQL grammar we want to explore in this paper. Furthermore, not all grammar used in the previous failure is useful for triggering bugs. By blindly prioritizing all the grammar distributed in these inputs without knowing which grammar is critical for bug triggering, previous failure inputs are less efficient for preserving the deep SQL grammar to uncover new deep bugs.