Parser Knows Best:
Testing DBMS with Coverage-Guided Grammar-Rule Traversal

Yu Liang [†] Hong Hu [†]

[†] *Pennsylvania State University*

Abstract

Database Management System (DBMS) is the key component for data-intensive applications. Recently, researchers propose many tools to comprehensively test DBMS systems for finding various bugs. However, these tools only cover a small subset of diverse syntax elements defined in DBMS-specific SQL dialects, leaving a large number of features unexplored.

In this paper, we propose `ParserFuzz`, a novel fuzzing framework that automatically extracts grammar rules from DBMSs' built-in syntax definition files for SQL query generation. Without any input corpus, `ParserFuzz` can generate diverse query statements to saturate the grammar features of the tested DBMSs, which grammar features could be missed by previous tools. Additionally, `ParserFuzz` utilizes code coverage as feedback to guide the query mutation, which combines different DBMS features extracted from the syntax rules to find more function and safety bugs. In our evaluation, `ParserFuzz` outperforms all state-of-the-art existing DBMS testing tools in terms of bug finding, grammar rule coverage and code coverage. `ParserFuzz` detects 81 previously-unknown bugs in total across 5 popular DBMSs, where all bugs are confirmed and 34 have been fixed.

1 Introduction

Database Management System (DBMS) stores, retrieves and manages data in a structured manner. They are extensively used in real-world data-intensive applications to drive trillions of Internet services and electronic devices [39, 40, 42, 44–46]. Any DBMS bugs will affect a large number of users [8, 35, 36].

Recent efforts on DBMS testing [2, 14, 47, 67] can be classified into two categories: generation-based testing and mutation-based grey-box fuzzing. `SQLsmith` [47] is the most popular generation-based testing tool to date. It generates SQL queries based on pre-defined query templates. These templates are manually crafted by `SQLsmith`'s developers, and can help generate high-quality SQL queries [57]. Another representative generation-based tool is $\texttt{SQLancer}_{\texttt{+QPG}}$, where `QPG` represents *query plan guidance* [2]. $\texttt{SQLancer}_{\texttt{+QPG}}$ adopts DBMS query plan information as the feedback to guide the query generation process, and is designed to detect logic errors from the DBMS code. Similar to `SQLsmith`, $\texttt{SQLancer}_{\texttt{+QPG}}$ also generates query sequences based on pre-defined SQL templates. However, due to the significant difference between multiple DBMS dialects, these pre-defined SQL templates cannot cover unique, complicated features from different DBMS systems [21, 38]. Further, as every DBMS keeps evolving, developers of `SQLsmith` and $\texttt{SQLancer}_{\texttt{+QPG}}$ have to track all recent updates and manually insert new templates.

Recently, coverage-guided grey-box fuzzing is widely used to detect memory errors from a wide-range of applications, including but not limited to operating systems [7, 12, 17, 27, 62], web browsers [11, 13, 37, 63] and compilers [6, 9, 28]. To conduct grey-box testing, a fuzzing tool, or *fuzzer*, first instruments the target program to record code coverage. Then, it generates a large number of random inputs and executes the target program. If one input triggers new code coverage, the fuzzer will treat the input as interesting and save it to a queue. By accumulating interesting inputs from the fuzzing queue, the fuzzer aims to trigger as many code logic from the target program as possible. Eventually, some inputs will trigger memory errors in the target program.

Researchers have adopted grey-box fuzzing to test DBMS systems [6, 14, 61, 67]. For example, `Squirrel` performs syntax-preserving mutation and semantics-guided instantiation to generate high-quality SQL queries [67]. It adopts code coverage to guide the input mutation and scheduling. However, the diversity of `Squirrel`-generated queries heavily relies on the quality of provided seed corpus [21, 27, 31]. The more features the seed corpus covers, the more diversity the mutated inputs could trigger, and therefore, the more bugs can be discovered. If the input corpus misses one grammar feature, any bug related to this feature would be missed by `Squirrel`. However, collecting a feature-rich seed corpus is a challenging task. The unit test suites provided from the target program may only cover parts of all features. Although `Squirrel` can generate queries with high validity rate, it can hardly explore the vast SQL features provided by various DBMS dialects. Therefore, how to effectively explore the vast syntax features in various SQL dialects is the key challenge to achieve efficient fuzzing.

A parser generator reads the grammar rules from a syntax definition file, and then leverage the grammar rules to a source code that can analysis any inputs according to the defined syntax rules [15, 29, 30]. If parsing successfully, the parser source code would transform the input into a parser tree, where the tree nodes represent the parsed tokens coming from the raw input. The developer is free to further transform the parser trees into their software internal structures. The main purpose of the parser is to verify the syntax correctness of the provided input, and then leverage the raw input into the program's internal structure for further operations. In our study, most of the popular DBMSs today use parser generators to define their SQL grammar rules [43, 48–52], including all the DBMSs we tested in this paper. Intuitively, these grammar rules provided for the parser generators would cover all the possible syntax regulations for the target programs, because the parsers work as verifiers to check all input sources in the software front-ends. Therefore, the grammar definition files for the parser generators can serve as an uniform interface that we can use to explore the ground truth SQL grammar rules defined for the dedicated DBMSs.

In this paper, we introduce a new query generation and mutation method, that directly learn and apply the grammar rules from the DBMSs' built-in parsers to the fuzzing process. Instead of using the grammar rules from DBMS parsers to simply verify the input queries, we use the parser rules to generate and mutate random query statements for fuzzing. By traversing all the grammar rules defined for the DBMS built-in parsers, the generated queries could saturate all the interesting syntax features supported by the original DBMS programs. Furthermore, we combine the code coverage feedback from the traditional grey-box fuzzing to guide the query grammar-based mutation. If the fuzzed query triggers a new code coverage from the DBMS program, we save the parsed syntax tree to the fuzzing queue for further query mutations.

We implemented a new fuzzing tool, `ParserFuzz`, that automatically constructs interesting query sequences based on DBMS built-in parser rules, and then uses the code coverage feedback to guide the query syntax rule-based mutation. We tested `ParserFuzz` on 5 most popular open-source DBMSs, `MySQL`, `SQLite`, `CockroachDB`, `TiDB` and `MariaDB`. `ParserFuzz` found 81 bugs from all 5 DBMSs, which contain 29 segmentation faults and 52 assertion failures. `ParserFuzz` achieves the largest detected bug numbers, the highest grammar coverage and code coverage compared to the other state-of-the-art DBMS testing tools in our evaluations.

In summary, this paper makes the following contributions.

- We propose a novel method for automatic SQL-query generation and mutation, which utilizes existing syntax rules of DBMS parsers to randomly generate feature-rich statements, without relying on high-quality seed corpus.
- We utilize code coverage as the feedback to select the promising grammar rules for generating new queries. The coverage-guided rule-based generation will help explore deep code logic of complicated DBMS programs.
- We evaluate `ParserFuzz` on 5 real-world DBMSs and find 81 new bugs. We demonstrate that our tool covers more grammar features, which helps trigger more memory errors than state-of-the-art DBMS testing tools.

Open Source. We will release the source code of `ParserFuzz` to help protect popular DBMS systems.

2 Background & Challenges

2.1 An Example Memory Error from MySQL

Listing 1 shows a query that triggers a segmentation fault on the release version of MySQL DBMS (version 8.0.33). This bug was detected by our newly proposed tool, `ParserFuzz`, without any seed corpus that explore on the similar syntax. The proof of concept (PoC) is constructed with two simple lines of SQL queries. The first query creates a new temporary table named `t0` with a column called `c1`. Temporary tables are only available within the current server-client connected session, and the data in the temporary table would be automatically dropped when the client exits the connection. In this PoC, the `CREATE TABLE` statement appends an index to the table to enhance the speed of data retrieval. Interestingly, the created index is a `composite index`, which means the created index is constituted with two different parts of data. The first part contains one single column. The second part is a functional key parts index, often referred as functional index. Different from normal index, where only columns are considered as keys. Functional index uses query expression as the indexed content, and the indexed expression receives a speed up. Functional index is useful when the DBMS user constantly employs the same SQL expression to search for table data. After caching the expression into index, the DBMS can speed up the repeated expression handling in the data retrieval. In the PoC of Listing 1, the index `i2` contains a mixed usage of normal index and functional index. In the second statement, the PoC promptly looks for the created index information.

When running this PoC on the latest version of `MySQL` (version 8.0.33), the targeted DBMS encounters `NULL pointer dereference` crash in the temporary table handling logic. The problem arises from the mixed usage of normal index and functional index, which confuses the temporary table creation handling, and the expression `c1 + c1` is not correctly saved in the index creation. When the `SHOW INDEX` query searches for the stored index information, it fails to find the columns saved into the functional index (being NULL), and therefore results in a crash. The PoC can be easily weaponized to upload to any online `MySQL` services and conduct `Denial-of-Service` attack. We have reported the bug to

```
CREATE TEMPORARY TABLE t0(c1 INT, INDEX i2(c1, (c1+c1)));
SHOW INDEXES IN t0;
```

Listing 1: A segmentation fault crashing from MySQL due to the mixed usage of `normal index` and `functional index`.

the `MySQL` developers. The developer has confirmed the bug and marked the bug severity as `Serious`.

The PoC shown in Listing 1 is interesting because, although previous works like `Squirrel` [67] have been extensively tested on `MySQL`, none of the previous tools found this bug. The key to uncovering this bug is to combine normal index creation and functional index creation in a single `CREATE TEMPORARY TABLE` statement. However, the internal representation of `Squirrel` doesn't supports using expressions as index keys, nor does `Squirrel` includes functional index in its input seeds. Therefore, not matter how hard `Squirrel` mutates on its input, it will never detect this bug. Furthermore, there are only one single instance in the `MySQL` official unit test that explores the combined use of normal index and functional index. And the unit test is not constructed with temporary table. As such, the DBMS tester is easy to overlook this one line unit test example and thus not able to include any queries that combine these two index types. Therefore, all the previous testing tools missed this bug. `ParserFuzz` automatically learns all the syntax features from the ground truth `MySQL` syntax definition file. Without relying on any pre-existing unit test corpus, it can craft queries that explore the rarely tested feature such as combined use of normal index and functional index. As a result, our tool `ParserFuzz` finds this bug in 10 hours of `MySQL` fuzzing.

2.2 Generation-based DBMS Testing Tools

There are some existing DBMS testing tools that relies on SQL templates to generate the SQL queries for testing. The most well-known generation-based query testing framework is `SQLsmith` [47]. Since its release, `SQLsmith` has been used extensively to test on different DBMSs, and found many bugs from the DBMS softwares [57]. However, the hand-written query templates are limited in covering the syntax elements from the DBMS syntax rules, and cannot fit in the complex and ever-changing SQL dialects defined in different DBMS softwares. Therefore, the queries generated from `SQLsmith` cannot cover all the SQL features from the DBMSs, and lack the capability to detect deep and unique bugs that are dedicated to the DBMSs' feature sets.

There is another generation-based DBMS testing tool called `SQLancer` [24], that focuses on detecting DBMS logic errors from DBMS systems [32–34]. DBMS logic bugs are code logic errors that cause the DBMS to return incorrect results. Unlike tools that detect memory errors, `SQLancer` doesn't generate arbitrary types of random queries for fuzzing. Instead, it focuses only on generating queries that matching its oracles' needs. In essence, `SQLancer` prefers to generate multiple syntactically different, but functionally equivalence queries, and verify their results to ensure the query execution correctness. `SQLancer` introduces a few SQL oracles for this purpose such as `NoREC`, `TLP` and `PQS`, where each shares a distinct SQL pattern to match [32–34]. With its latest configuration $\texttt{SQLancer}_{\texttt{+QPG}}$ [2], it uses the DBMS query plan to guide its query generation in order to stress test the DBMS query optimization logic. However, because `SQLancer` and $\texttt{SQLancer}_{\texttt{+QPG}}$ restricted themselves to generate queries that align with the oracles' patterns, they lack the query diversity needed to explore all the grammar features provided by the DBMSs. Therefore, neither `SQLancer` nor $\texttt{SQLancer}_{\texttt{+QPG}}$ are suitable to detect DBMS memory corruption bugs that are arise from interesting but rarely tested syntax features.

There are several other generation-based DBMS testing tools that aim to detect various kinds of bugs from the DBMSs [23, 26, 58]. Some existing works treat the SQL query generation as a boolean satisfiability problem, and use SAT solvers to produce queries that achieve high correctness rate [1, 25]. Chandra *et al.* extend the database construction technique to boost the efficiency of DBMS query testings [5]. Bruno *et al.* propose to generate queries based on Cardinality Constraints [4]. On the topic of performance issues, researchers also propose several tools to detect performance bugs in the DBMS [3, 18, 65]. `APOLLO` runs the same query on different versions of the same DBMS to detect performance regression bugs [16]. `AMOEBA` generates functional equivalence queries and checks whether they finish in a similar response times [22]. Lastly, regarding logic errors, differential testing is employed to detect logic bugs in DBMSs [38]. These testing tools check the result consistency from identical queries when running them on different DBMSs [10] or running them in one DBMS but with different versions [64]. If any result discrepancy is observed, a potential logic bug is found.

2.3 Mutation-based DBMS Testing Tools

`Squirrel` is one of the state-of-the-art mutation-based query testing tools. It supports testing 4 DBMSs including `SQLite`, `PostgreSQL`, `MySQL` and `MariaDB` [67]. The core idea of `Squirrel` is two-folded. The first contribution to transform any inputted queries into its internal representation (IR). Based on the generated IR, `Squirrel` performs type-based mutations on the IR tree nodes, in order to mutate the parsed queries but preserve the mutated queries' syntactic correctness. The second contribution is to build a dependency graph for the query arguments such as table names and column names. By resolving the dependency graph, `Squirrel` fills in the query arguments and assigns the parsed query with special semantic meanings. However, even though `Squirrel` uses an internal parser to convert the raw query input into its IR, it still heavily depends on the provided seed corpus to shape the mutated queries. If the given seeds lack a specific SQL grammar, `Squirrel` will not generate any inputs to explore this grammar feature. Given the complexity of the DBMS SQL dialects, it is challenging for Squirrel to cover all the interesting features from its seed corpus. Therefore, the DBMS features that Squirrel explores are limited. Additionally, DynSQL [14] implements another DBMS fuzzing tool. It samples the DBMS execution states after every query execution, aiming to gather more real-time feedback from the DBMS to improve the generated queries' correctness rate. The sampled query state later guides the query generation, helping to sidestep some semantic errors caused by the previous unsuccessful data creations or modifications. The idea of DynSQL is complementary to our work. While DynSQL focuses on generating queries with higher correctness rate, ParserFuzz strives to generate more diverse queries that can explore more rarely tested features. There are a few other mutation-based fuzzing tools for detecting DBMS crashing bugs [19, 59]. For example, RATEL targets enterprise-level DBMSs such as GaussDB [60]. LEGO instantiates the query statements with type-affinity awareness for higher query correctness rate [20]. Unfortunately, DynSQL, RATEL or LEGO have not released their source code, which means it is not feasible to compare our work to theirs.

2.4 Parser Generators in the DBMSs

A parser generator program takes a grammar definition file as input and generates source code that parses any input characters according to the rules defined in the grammar file [15, 29, 30]. Most parser generators use a syntax notion type similar to the Backus-Naur form (BNF) [29, 54, 55]. By defining the grammar rules in the grammar definition file, developers can implement a parser with high runtime efficiency and minimal grammar rule ambiguities. A more detailed example of grammar definition will be elaborated in §3.1. The primary purpose of the parser generator is to verify whether the input stream matches the defined syntax rules. If matched, the parser enables the developer to transform the raw input into application's internal structure.

Most of the DBMSs known today use parser generators to define their SQL grammar rules [43, 48–52]. For example, MySQL, MariaDB and PostgreSQL use bison as their parser generator [49–51]. CockroachDB and TiDB use a special GoLang implemented counterpart of the yacc parser generator [48, 52, 56]. The goyacc used by CockroachDB and TiDB shares a grammar define notation similar to the bison one, with both closely aligned with the standard BNF. SQLite uses a custom parser generator called Lemon, which was invented by the same author who originally developed SQLite [43]. It adds in a few improvements based on the grammar notations of bison or yacc, but overall maintains a similar grammar definition form. In summary, the vast usage of parser generators in DBMSs allows us to observe the ground truth grammar rules dedicated to different DBMS softwares, and offers us a uniform interface to easily leverage these pre-defined grammars for the DBMSs' fuzzing purpose.

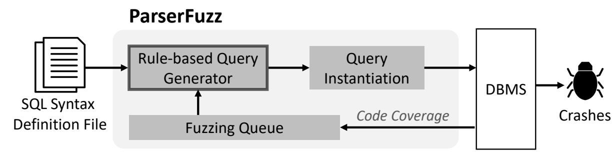


Figure 1: Overview of ParserFuzz. It automatically extracts SQL features from DBMS-specific syntax definition files, and utilizes code coverage to select promising rules for query generation.

3 Design of ParserFuzz

To generate diverse sets of queries that explore all possible grammar rules, we introduce parser grammar rule-based query generation. The query generation doesn't rely on any prior query seeds, but instead automatically constructs query statements directly based on the grammar rules defined for the parser generator. Additionally, by traversing all the parser rules defined in the parse generator, we can explore all the possible syntax features tailored for the DBMS. To steer the fuzzer towards combining different interesting syntax features, we incorporate code coverage feedback to guide the query mutation. The combination of parser rule-based query generation and code coverage feedback enables ParserFuzz to generate diverse and complex queries that can effectively find DBMS memory corruption bugs.

System overview. Figure 1 shows an overview of our tool ParserFuzz. ParserFuzz is the first tool that directly generates SQL query statements based on DBMS SQL grammar definition file, and it guarantees exploring all the grammar rules defined in the DBMSs. Unlike previous fuzzers, ParserFuzz doesn't require any queries as fuzzing seeds. Instead, all queries are produced directly from the parser rule-based query generator §3.1. These generated query are then populated with SQL arguments using the validity-oriented instantiation algorithm §4. All the produced queries are executed in the DBMS afterwards. In the current design, ParserFuzz produces SQL query sequences according to the following principle: it firstly generates 3 CREATE TABLE statements, followed by 3 INSERT statements, 2 CREATE INDEX statements, 10 randomly generated statements regardless of the statement types, and concludes with 10 SELECT statements. ParserFuzz will clean up the database contents after one query sequence finishes its execution. If one query statement activates a new code branch from the DBMS, ParserFuzz will save the statement's syntax tree to the fuzzing queue. In the next round of generating new statement, ParserFuzz offers 50% chances to mutate the queries from the fuzzing queue §3.3. If any segmentation fault or assertion failure are detected during execution, we will save the complete bug-triggering query sequence for further analysis.

```
01 simple_select:
02   SELECT_SYM SCONST
03           {/*extra developer defined logic...*/}
04 | SELECT_SYM distinct_clause target_list from_clause
05           {/*extra developer defined logic...*/}
06 | SELECT_SYM distinct_clause target_list from_clause
07         where_clause
08           {/*extra developer defined logic...*/}
09 ;
10
11 distinct_clause:
12   DISTINCT_SYM {/*extra developer defined logic...*/}
13 ;
14
15 target_list:
16   target_elem {/*extra developer defined logic...*/}
17 | target_list ',' target_elem
18         {/*extra developer defined logic...*/}
19 ;
20
21 /* other grammar rules ... */
```

Listing 2: Example grammar definition rules for `Bison`

```
static const SYMBOL symbols[] = {
  /* SQL keywords (by alphabetical order) */
  {SYM("\&\&", AND_AND_SYM)},
  {SYM("<", LT)},
  {SYM("<=", LE)},
  //...
  {SYM("SELECT", SELECT_SYM)},
  {SYM("DISTINCT", DISTINCT_SYM)},
  //...
};
```

Listing 3: Example keyword mappings from the lexer

3.1 Parser Rule-based Query Generation

Exploring the vast and rarely tested SQL features from the DBMS is key to find more interesting bugs during DBMS fuzzing. The DBMS SQL rule definition file defines the complete set of the grammar rules permitted for the SQL dialect. Listing 2 shows an example of SQL grammar rule definition. This grammar notation is adopted by parser generators such as `yacc`, `bison` and `go-yacc`, and is further used by `MySQL`, `CockroachDB` and `TiDB`. By convention, all lowercase symbols in Listing 2 represent non-terminal keywords. A non-terminal keyword means the current keyword can be further extended by sub-rules. For instance, the first line in Listing 2 defines the top non-terminal keyword `simple_select`. Line 2-9 define the sub-rules that can be expanded by the keyword `simple_select`. Additionally, all uppercase symbols in Listing 2 represent terminal keywords. A terminal keyword is one without any attached sub-rule. They can directly map to a token in the input stream. Listing 3 shows an example token mapping. Alternatively, a terminal keyword can represent arguments such as table and column names or as constant values such as string literals. The `SCONST` keyword used in Listing 3 is an example for representing any single quoted string constant in the SQL language. The parser grammar is free to define its grammar entry from any non-terminal keyword. The parser will recursively match the non-terminal keywords' grammar rules to verify the grammar correctness. A successfully parsed query traverses the defined grammar rules, constructing a tree-

```
  -- match the first rule from simple_select
  SELECT 'abc';
  -- match the second rule from simple_select
  SELECT DISTINCT c1 FROM t0;
  -- match the third rule from simple_select
  SELECT DISTINCT c1 FROM t0 WHERE TRUE;
```

Listing 4: Querie examples matching the syntax rules in Listing 2

```
a_expr:
  a_expr '+' a_expr
| a_expr AT_SYM TIME_SYM ZONE_SYM a_expr
| a_expr COLLATE_SYM any_name
| /* other grammar rules ... */
;
```

Listing 5: A keyword that recursively references itself

like structure where every tree node represents a token from the query stream. For instance, if we assume the top keyword `simple_select` is the entry point from the grammar syntax definition, Listing 4 shows some example queries that can be successfully parsed by these grammar rules.

Although the main goal of the SQL parser is to verify SQL inputs, the grammar rules can be easily tuned to serve other purposes. Most interestingly, instead of matching grammar rules from the inputs, it is feasible to generate query statements from the grammar definition by randomly choosing which grammar rule to apply. Specifically, starting from the grammar entry point, we can randomly select a grammar rule to commit every time we need to resolve a non-terminal keyword. By choosing a different set of grammar rules to apply when traversing the syntax, we can generate various forms of queries for testing. Moreover, we can easily ensure the syntax correctness of the generated query because it still adheres to the defined syntax rules. Furthermore, to provide the generated queries with semantic meanings, we label all the SQL arguments in the grammar rules, such as table names and column names, addition with constant values such as integers and string literals. These SQL arguments and constant values will later be passed to the validity-oriented instantiation algorithm to fill in the concrete values §4.

Because the grammar logics defined for the parser generator serves as the front-end of the DBMS processing pipeline, the logic represents the ground truth of the grammar rules a DBMS can support. More importantly, the grammar definition file includes all the grammar features that the DBMS currently supports. By thoroughly scanning all the grammar rules defined in the grammar file, we can generate queries that saturate the interesting syntax features dedicated to the DBMS softwares. Additionally, the grammar rule exploration does not rely on any prior input corpus, reducing the burden for DBMS testers to gather an input corpus that covers all of interesting features they care about.

```
natural_join_type:
  NATURAL_SYM opt_inner JOIN_SYM /* simple rule */
;

opt_inner:
  %empty    /* syntax termination */
| INNER_SYM /* syntax termination */
;
```

Listing 6: Example simple rule

```
subquery: '(' simple_select ')'  ; /* complex rule */
```

Listing 7: Example complex rule

3.2 Path Explosion due to Recursive Keyword

Although the BNF notation isn't as complex as programming languages such as C/C++, we also face similar issues with program static analysis when generating new queries from the grammar rules. The most notable issue is the grammar path explosion problem. Listing 5 demonstrates an example that could lead to grammar path explosion. Most of the grammar rules defined under the `a_expr` keyword reference the top keyword `a_expr`, which creates a loop in the grammar rule that continuously expands the grammar tree. Given most of the grammar rules defined in Listing 5 have recursive references, and some rules such as `a_expr '+' a_expr` even reference the top keyword twice, the query generation algorithm might get trapped in this recursive keyword and never finds an exit. Because `a_expr` is an important grammar feature in the DBMS, addressing this grammar path explosion problem becomes one important task to tackle.

To address this grammar path explosion problem, `ParserFuzz` scans through the query grammar files and classifies all the grammar rules into three categories: `simple rule`, `normal rule` and `complex rule`. Simple rules are any grammar rules that guaranteed to terminate in 2 iterations. Listing 6 shows one example. The rule defined under the `natural_join_type` keyword contains two terminal keywords: `NATURAL_SYM`, `JOIN_SYM` and one non-terminal keyword `opt_inner`. However, all grammar rules defined for `opt_inner` definitively lead to rule termination, *i.e.*, they could be either `empty` or `INNER_SYM`. In this case, by choosing the grammar rule `NATURAL_SYM opt_inner JOIN_SYM`, `ParserFuzz` ensures the grammar rule will cease expansion in 2 iterations. `ParserFuzz` therefore labels this grammar rule as simple rule. Complex rules are syntax definitions that lead to recursive keywords or more intricate grammar expressions. Apart from the aforementioned case in Listing 5, Listing 7 illustrates another complex rule example, which expands into an entire sub-select statement within the query expression, significantly complicating the syntax tree. `ParserFuzz` automatically labels all grammar rules that recursively reference keywords as complex rules. User can also manually label certain complex grammar rules, like the one shown in Listing 7. Normal rules are all the other grammar rules excluding

```
table_reference:
  table_factor   /* normal rule, preferred */
| joined_table   /* complex rule */
;

table_factor:
  table_name     /* simple rule, preferred */
| table_function /* complex rule */
;

joined_table:
  /* complex rule */
  table_reference inner_join_type table_reference
  /* complex rule */
| table_reference outer_join_type table_reference
;
```

Listing 8: Rule prioritization to resolve path explostion

```
/* interesting query saved in the fuzzing queue */
SELECT * FROM t0 INNER JOIN t1 ON t0.c1 = t1.c1;
/* the query are mutated to the following */
SELECT * FROM t0 OUTER JOIN t1 ON t0.c1 = t1.c1;
```

Listing 9: Example mutation on saved query

simple rules and complex rules. They represent an uncertain status of the current syntax rule, where the defined syntax will neither lead to immediate termination, nor lead to path explosion. User can manually define certain rules as either simple rules or complex rules to guide the query generation, and `ParserFuzz` handles the normal rule sampling automatically.

After categorizing the grammar rules, `ParserFuzz` uses the categorization information to guide the query generation. In the initial stages of the query generation, *i.e.*, within a shallow syntax rule iteration depth, `ParserFuzz` freely selects grammar rules to commit based on a Multi-Armed Bandit (MAB) solver described in §3.3. Once the query generation reaches a specific depth, `ParserFuzz` prioritizes generating *simple_rule* > *normal_rule* > *complex_rule*. Listing 8 illustrates one example that uses rule prioritization to early-exit the query generation. The `table_reference` keyword in Listing 8 contains two grammar rules. The first rule is a `normal rule`, because the grammar defined in `table_factor` could lead to both simple and complex rules. Conversely, the second rule from `table_reference` is a complex rule because all the grammar features specified in `join_table` recursively reference the top keyword `table_reference`. Therefore, once reaching specific depth, `ParserFuzz` prioritizes normal rule, `table_factor`, when resolving `table_reference`. It then commits to the rule `table_name`, which is a simple rule defined after `table_factor`, and it concludes the current syntax tree path. This categorization-based parser rule prioritization significantly enhances the success rate of generating valid SQL queries, because it effectively avoids the path explosion problem during query generation.

3.3 Query Mutation with Coverage Feedback

Lessons learned from previous grey-box fuzzing tools suggest that code coverage feedback is effective in leading fuzzers

```
join_table:
  table_reference INNER_SYM JOIN_SYM table_reference
    {/*extra user defined logic...*/}
| table_reference OUTER_SYM JOIN_SYM table_reference
| /* other grammar rules ... */
;
```

Listing 10: Grammar rules for the example in Listing 9

```
01 a_expr:
02   /* other grammar rules ... */
03 | STRING_LITERAL /* new coverage 1 time */
04 | a_expr COLLATE_SYM any_name /* new coverage 3 times */
05 | /* other grammar rules ... */
06 ;
```

Listing 11: Rule prioritization based on code coverage

into finding more bugs. After executing a query, ParserFuzz gathers the code branches triggered in the DBMS. If a query triggers a new code branch, ParserFuzz collects the syntax tree that constructs the current executed query into its fuzzing queue. Because each query acts as an independent statement in the ParserFuzz generation process, ParserFuzz does not save concrete values such as arguments or constant values into the syntax tree. On the next query generation, ParserFuzz offers 50% chances to mutate the syntax trees from the fuzzing queue instead of generating a new one. Furthermore, ParserFuzz has the autonomy to choose any tree nodes from the syntax tree to start its query mutation. To mutate, ParserFuzz refers to the SQL syntax definition rules, starts generating new query segments from the randomly chosen query mutation point, and ultimately replaces the original query segments with the newly produced one. Listing 9 shows one such mutation example. Assuming the original query from Listing 9 triggers a new code coverage from the DBMS, so it is stored in the fuzzing queue. During the mutation, ParserFuzz selects the `join_table` node to mutate. The corresponding grammar rule definition is shown in Listing 10, and it covers the query segment `t0 INNER JOIN t1`. ParserFuzz then randomly selects another parser rule that defined under `join_table`, and commits to the second rule in Listing 10. The original query from Listing 9 is ultimately mutated to the form with `OUTER JOIN`.

In addition of using the code coverage feedback to save queries for further mutations, ParserFuzz also uses the code coverage feedback to prioritize more feature-rich grammar rules that might lead to interesting DBMS behaviors. Certain grammar rules are doomed to be more feature-rich and interesting than others. For example, in the case of Listing 11, the first rule on line 3 resolves a constant string, making it less interesting than the second rule defined in line 4, where the latter rewrites the default collation for the expression's return value. Because the second rule can trigger more unique code coverage compared to the constant string resolving during the fuzzing samples, ParserFuzz should favor the `COLLATION` rule when formulating a new SQL statement. In ParserFuzz, we model the grammar rule commitment as an Multi-Arm Bandit (MAB) problem, where ParserFuzz plays a game every time it needs to opt for a grammar rule to commit. If the generated query triggers a new code coverage, all the grammar rules used to construct the test query receive a reward. The fuzzer's objective is to generate queries that maximize the code coverage(reward). However, the information of how much code coverage is achievable remains limited or unknown at the time of committing to the grammar rules. Thus, ParserFuzz needs to conduct an optimized strategy that explores all the different grammar possibilities while also favoring selecting the grammars that induce more captivating outcomes. In ParserFuzz, we use the ε-Greedy algorithm to address this MAB problem. The DBMS testers can predetermine an ε value, which represents the possibility of directly selecting the grammar rule with the highest known reward. Conversely, with a $1-\varepsilon$ probability, ParserFuzz randomly selects any rules defined to maximize the exploration. By default, the ε value for ParserFuzz is set to be 0.5. But users can adjust the ε value to suit their needs.

4 Implementation

We implemented ParserFuzz based on the logic-bug detection tool SQLRight [21]. The query-instantiation logic of SQLRight can handle a variety of semantic situations. We removed the oracle interfaces from SQLRight and expanded the fuzzer to accept any valid SQL statements. Moreover, we updated SQLRight's query mutation logic, changing the tool from a mutation-based fuzzer that relies on input corpus to a generation-based fuzzer that can generate queries by parsing grammar definition files. Here, we present more implementation details for the users that are interested in ParserFuzz.

Rule-based query generator. This generator is built based on a prototype developed by Cockroach Labs, the developer of CockroachDB. The prototype is named RSG, standing for Random Statement Generator [53]. However, this prototype supports only a limited number of features from the BNF parser notation. For example, the prototype does not recognize notation `%prec`, which is reserved for the parser generator usage. Moreover, the prototype struggles with complex syntax rules, particularly with recursive keywords. As a result, this prototype was primarily designed to generate simple and short DEMO query statements. The grammar definition files provided to the prototype rarely exceed 15 lines, making the prototype unsuitable for parsing the comprehensively grammar rules designed for CockroachDB. We enhanced the prototype's capabilities to accommodate more complex grammar definition rules, which includes adding full support for `go-yacc` grammar files used in CockroachDB and TiDB, `bison` grammar files used in MySQL and MariaDB, and even Lemon grammar file specifically designed for SQLite. The rule-based query generator in ParserFuzz can be further extended to support other grammar definition formats in the future, including the support for modern parser generation tool ANTLR.

DBMS coverage instrumentation. We used AFL LLVM mode to instrument the DBMSs that are written in C or C++ languages [66], including SQLite, MySQL and MariaDB. In addition, we enlarged the code coverage map size from 64K to 256K. However, we couldn't find any existing method to apply branch coverage instrumentation for DBMSs implemented in GoLang. Therefore, we modified the line coverage instrumentation from GoLang built-in library [41], and enhanced its capability to support branch coverage logging. We then integrated our custom branch coverage feedback logging in ParserFuzz when testing GoLang implemented DBMSs such as CockroachDB and TiDB.

5 Evaluation

We evaluate ParserFuzz on five popular open-source DBMSs, including SQLite, MySQL, CockroachDB, TiDB and MariaDB. The evaluation aims to answer the following questions.

Q1. Can ParserFuzz detect real-world DBMS bugs?
Q2. Can ParserFuzz find more bugs than existing tools?
Q3. How does code coverage guide the fuzzing process?
Q4. How do extra syntax rules contribute to bug finding?

Experimental setup. To address **Q1**, we conduct experiments of ParserFuzz on 5 popular DBMSs, SQLite, MySQL, CockroachDB, TiDB and MariaDB, and gather all the bugs detected in §5.1. To answer **Q2**, we compare ParserFuzz to existing state-of-the-arts in §5.2. Due to the diverse SQL dialects in different DBMSs, there is no universal DBMS testing tool that covers all the DBMSs we are testing. Therefore, for each evaluated DBMS, we select the latest open-source DBMS testing programs that are compatible to the DBMS as baselines. For SQLite, MySQL and MariaDB, we compare ParserFuzz against Squirrel, the most advanced grey-box mutation-based DBMS fuzzer. For CockroachDB and TiDB, we use the official query generation-based testing tools that are maintained by the DBMS developer groups, *i.e.*, we test the customized SQLsmith (SQLsmith_C for short) for CockroachDB, and test go-sqlsmith (SQLsmith_G for short) for TiDB respectively. To compare ParserFuzz against traditional bit-flips mutation-based fuzzer, we select AFL++ to test C/C++ implemented DBMSs and use LibFuzzer to test GoLang implemented ones. In addition, to understand the memory error detecting capability for DBMS logic bug detectors, we compare ParserFuzz to state-of-the-art logic bug testing tool SQLancer_{+QPG}. SQLancer_{+QPG} supports testing with SQLite, CockroachDB and TiDB, and outperforms all other logic bug detectors including SQLRight [2, 21]. We use NoREC oracle for SQLancer_{+QPG} when testing with SQLite and CockroachDB. Because NoREC oracle is claimed to be a better performer overall compared to TLP oracle [2]. But we fallback to use TLP when testing TiDB, because SQLancer_{+QPG} hasn't supported testing TiDB with NoREC oracle yet. While the most recent DBMS fuzzing tool DynSQL [14] supports testing 6 DBMSs including SQLite, MySQL and MariaDB, it is not open-source, so we cannot compare our tool to their implementation. For fuzzing tools that demand input corpus, we use the query libraries from the Squirrel repo to serve as the universal input seeds. To answer **Q3**, we disable the code coverage feedback from ParserFuzz, transforming it into a pure random query generation tool, which noted as $\text{ParserFuzz}_{\text{-cov}}$. We compare $\text{ParserFuzz}_{\text{-cov}}$ against the full-featured ParserFuzz in §5.3. Finally, we use the bugs detected to demonstrate the contribution of the diverse syntax elements from ParserFuzz §5.4, which answers the question of **Q4**.

We run all evaluations on an Ubuntu 20.04 system. The machine comes with two 28-cores Intel(R) Xeon(R) Gold 6348 CPUs and 512 GB memory. We target the latest release versions of the DBMSs at the time when we started the evaluation. Specifically, we evaluate SQLite on version 3.41.0, MySQL on version 8.0.33, CockroachDB on version v22.1.10, TiDB on version v6.1.7 and MariaDB on version 11.3.

5.1 DBMS Bugs

Due to the resource limitations, we evaluated different DBMSs over different testing durations. We fuzzed MySQL with the longest time frame, which lasted 3 months. We further tested SQLite for 2 months, CockroachDB for 2 months, TiDB for 1 month and MariaDB for 3 weeks. In total, ParserFuzz detected 81 bugs from all 5 DBMSs, containing 29 segmentation faults and 52 assertion failures.

A bug summary is presented in Table 1 and Table 2. A segmentation fault indicates a bug that brings down the DBMS server process, and forces the DBMS client to exit the ongoing session. An attacker can effectively exploit a segmentation fault PoC to conduct Denial-of-Service attack on any online DBMS services. An Assertion failure from SQLite and MySQL implies the provided PoC is reproducible only in debug build of the DBMS. Although the release build of the DBMS does not trigger the assertion crash, the failed assertion check implies that the DBMS operates in an ill-formed state. An attacker might exploit this ill-formed state to trigger higher impact exploitation. An assertion failure from CockroachDB represents an unexpected runtime error. The cause can be as fatal as index out of bounds access, however, CockroachDB automatically recovers from the error state, and it will discard the malicious changes and then resume running.

5.2 Comparison with Existing Tools

We compare ParserFuzz with state-of-the-arts on all 5 supported DBMSs, including MySQL, SQLite, MariaDB, CockroachDB and TiDB. For all experiments, we allocate 5 concurrent processes for each tool to stress-test the DBMSs. Each evaluation lasts for 24 hours, and we repeat all the experiments 3 times. Figure 2 show the results we collected.

DBMS	ID	Description	Status	Squirrel	SQLsmith	SQLsmith$_{C}$	SQLsmith$_{G}$	SQLancer$_{+QPG}$
SQLite	1	`RETURNNING` from ill-formed `VIEW`	fixed (84417bbd)	✗	✔	-	-	✗
	2	Unexpected exposed debug function	fixed (62114711)	✔	✔	-	-	✗
	3	Incorrect byte code conversion	fixed (8f637aae)	✗	✔	-	-	✗
MySQL	4	Incorrect sorting optimization	fixed (version 8.0.34)	✗	-	-	-	-
	5	Incorrect partition condition handling	confirmed	✗	-	-	-	-
	6	Incorrect partition condition handling	confirmed	✗	-	-	-	-
	7	Incorrect `REGEXP` expression handling	confirmed	✗	-	-	-	-
	8	`TEMP TABLE` created with ill-formed function index	confirmed	✗	-	-	-	-
	9	Incorrect `CHECK` condition handling in `CREATE TABLE`	confirmed	✗	-	-	-	-
	10	Incorrect charset conversion	confirmed	✗	-	-	-	-
	11	Incorrect subquery expression handling	confirmed	✗	-	-	-	-
CockroachDB	12	Incorrect temp disk storage internal value	fixed (1ee803ee)	-	-	✔	-	✗
	13	Duplicate `PRIMARY KEY`	fixed (3a3123d9)	-	-	✗	-	✗
	14	Missing name resolver to constraint validator	fixed (0c58a08d)	-	-	✔	-	✗
	15	Incorrect data type conversion	fixed (5cb5d1da)	-	-	✔	-	✔
	16	Incorrect default expression typing and backfill	fixed (51005e41)	-	-	✔	-	✗
	17	Out-of-bounds from tuple handling	fixed (58ec9687)	-	-	✔	-	✔
	18	Large number as hidden constants	fixed (318e352e)	-	-	✔	-	✔
TiDB	19	Incorrect query parsing logic	fixed (8f308ecb)	-	-	-	✗	✗
	20	Compare subquery in `SHOW`	confirmed	-	-	-	✗	✗
	21	Index out of bound access in `EXPLAIN`	fixed (762432b6)	-	-	-	✗	✗
	22	DML panic when CTE exists	fixed (25764bc8)	-	-	-	✗	✗
	23	Incorrect expression rewriter optimization	confirmed	-	-	-	✗	✔
	24	Recovery non-existing jobs	confirmed	-	-	-	✗	✗
	25	Incorrect handling for partial aggregation	confirmed	-	-	-	✗	✗
MariaDB	26	Incorrect partition condition handling	confirmed	✗	-	-	-	-
	27	Incorrect check condition handling in `CREATE TABLE`	fixed (8adb6107)	✔	-	-	-	-
	28	Incorrect remove record without match in `DELETE`	confirmed	✗	-	-	-	-
	29	Incorrect sub-select optimization	confirmed	✔	-	-	-	-

Table 1: New Segmentation Faults detected by `ParserFuzz`. `ParserFuzz` detects 81 bugs in total, including 29 crashes and 52 assertion failures. The `Squirrel`, `SQLsmith`, `SQLsmith`$_{C}$, `SQLsmith`$_{G}$ and `SQLancer`$_{+QPG}$ columns represent whether the referenced tools can theoretically detect the mentioned bug, '✔' means 'Yes', '✗' states 'No' and '-' means the tool is not applicable to the target DBMS.

Unique bug numbers. Across all evaluations conducted on the 5 DBMSs, `ParserFuzz` detects the highest number of bugs within 24 hours. As seen in Figure 2a, `ParserFuzz` detects 4 bugs in total, winning the first place of the evaluation. `Squirrel` can also find one new crashing bug from `MySQL`. For the new bug detected from `Squirrel`, we also reported it to the `MySQL` developer. In addition, `Squirrel` identifies 2 crashing bugs in Figure 2m. However, the detected bugs from `Squirrel` are old bugs that had already been known to the developer back in 2019 and 2022 respectively. Despite this, `ParserFuzz` records the highest bug count with 3 bugs detected in `MariaDB` fuzzing. Moreover, as shown in Figure 2e and Figure 2i, `ParserFuzz` detects remarkable numbers of bugs when testing on `CockroachDB` and `TiDB`, giving 6 and 4 bugs respectively. Although `SQLancer`$_{+QPG}$ can detect multiple logic bugs in `TiDB` in Figure 2i, it detects less memory errors than `ParserFuzz` in all `SQLite`, `CockroachDB` and `TiDB` testings. All baselines tools except `SQLancer`$_{+QPG}$ do not detect any issues in `SQLite` evaluation as shown in Figure 2q, where `ParserFuzz` detects 2 bugs within the set time frame.

Grammar edge number. The extensive amount of grammar edge triggered by the `ParserFuzz` fuzzing is the primary reason why it can find more memory errors compared to other baseline tools. A grammar edge represents the possible combinations between two non-terminal keywords. For example, in Listing 8, a keyword mapping from `table_reference` to `table_factor` represents one edge case, and `table_reference` to `joined_table` represents another. The upper bound lines display the total possible grammar edges for each DBMSs' grammar rules. The grammar edge coverage plots are presented in Figure 2b, Figure 2f, Figure 2j, Figure 2n and Figure 2r. While we claim that `ParserFuzz` can account for all the grammar edges from the defined grammar rules, the gaps between `ParserFuzz`'s grammar edges and the upper bounds indicate the grammar syntaxes we intensionally exclude. These syntax features are omitted primarily because they could corrupt the database source, forcing the DBMS to reboot or disrupting the DBMS server-client connection during the fuzzing loop. The excluded grammars include user modification statements, privilege modification statements, and data read-write lock modifications, among others. DBMS testers can decide whether to include these syntax elements in their testing. But including them would likely reduce the DBMS fuzzing speed. While `ParserFuzz` captures all the interesting grammar edges in our evaluation, other tools barely match its performance. The extra syntax elements learned by `ParserFuzz` enables more diverse query generation, resulting in more memory errors

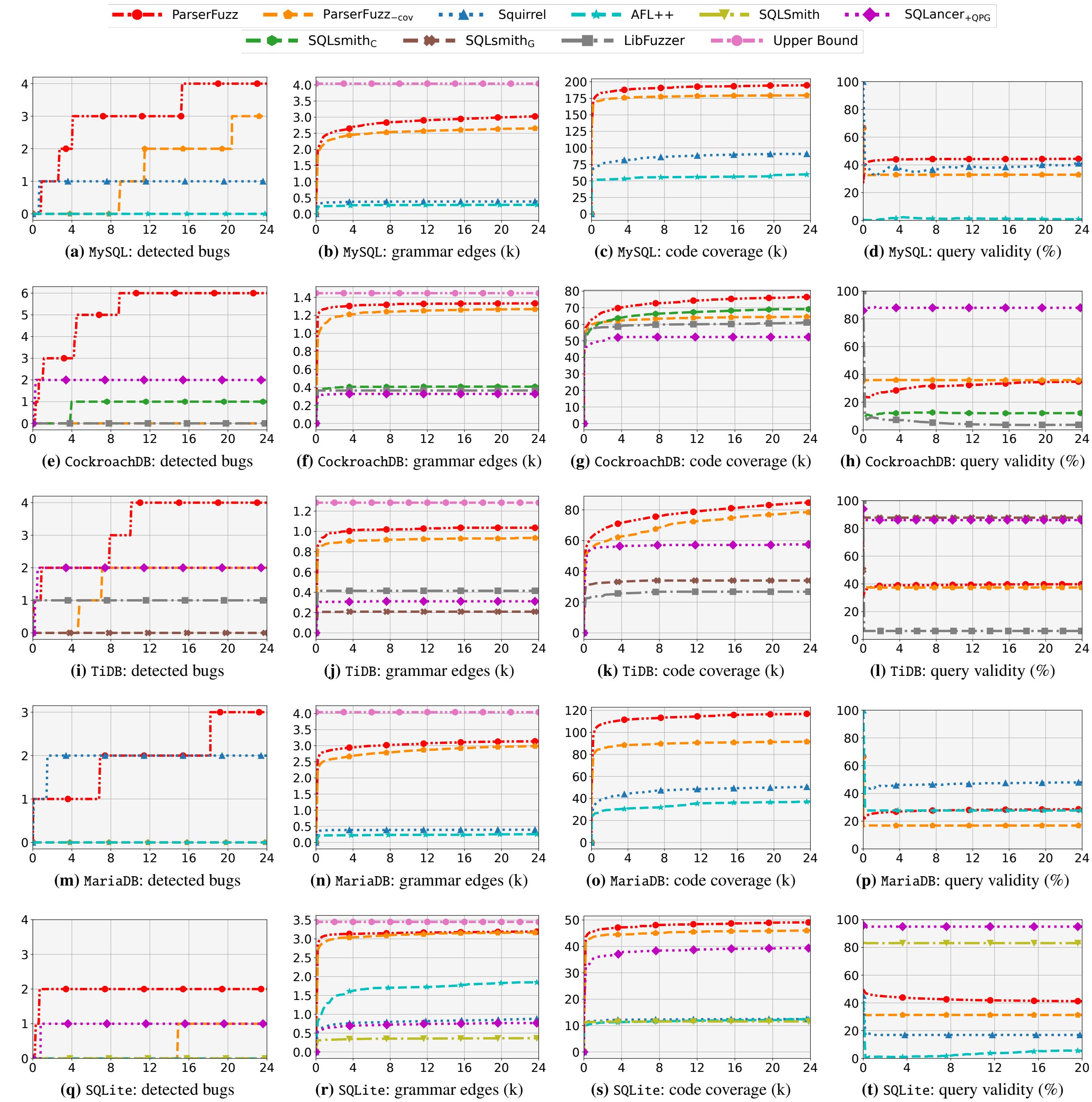
DBMS	ID	Description	Status	`Squirrel`	`SQLsmith`	`SQLsmith`$_C$	`SQLsmith`$_G$	`SQLancer`$_{+QPG}$
`SQLite`	1	'pExpr->affExpr==OE_Rollback ...'	fixed (e9543911)	✗	✗	-	-	✗
	2	sqlite3_result_blob, 'n>=0'	fixed (ab3331f4)	✗	✔	-	-	✗
	3	'sqlite3VdbeMemValidStrRep(pVal)'	fixed (3e2da8a7)	✗	✗	-	-	✗
`MySQL`	4	'escape_arg != nullptr'	fixed (version 8.2.0)	✗	-	-	-	-
	5	'm_alter_info->requested_lock'	confirmed	✗	-	-	-	-
	6	'has_error == thd->get_stmt_da()->is_error()'	confirmed	✗	-	-	-	-
	7	check_set_user_id_priv, '0'	fixed (version 8.0.35)	✔	-	-	-	-
	8	'is_prepared() && !is_optimized()'	confirmed	✗	-	-	-	-
	9	MoveCompositeIteratorsFromTablePath, 'false'	confirmed	✗	-	-	-	-
	10	'!thd->lex->is_exec_started()'	confirmed	✗	-	-	-	-
	11	'!sl->order_list.first'	confirmed	✔	-	-	-	-
	12	'm_return_field_def.auto_flags == Field::NONE'	confirmed	✗	-	-	-	-
	13	'm_relaylog_file_reader.position() == m_rli->...'	confirmed	✗	-	-	-	-
	14	'ha_alter_info->handler_flags ...'	confirmed	✔	-	-	-	-
	15	'!thd->lex->is_exec_started() ‖ thd->lex ...'	confirmed	✗	-	-	-	-
	16	'!thd->in_sub_stmt'	confirmed	✗	-	-	-	-
	17	'is_prepared()'	confirmed	✗	-	-	-	-
	18	'thd->is_error()'	fixed (79eae6a2)	✗	-	-	-	-
	19	'inited == NONE ‖ table->open_by_handler'	confirmed	✔	-	-	-	-
	20	'is_nullable()'	confirmed	✗	-	-	-	-
	21	'!is_set()'	confirmed	✗	-	-	-	-
	22	'm_deque == other.m_deque'	confirmed	✔	-	-	-	-
`CockroachDB`	23	unsupported comparison operator	fixed (7b473a8f)	-	-	✔	-	✔
	24	input to ArrayFlatten should be uncorrelated	confirmed	-	-	✔	-	✔
	25	an empty end boundary must be inclusive	confirmed	-	-	✗	-	✔
	26	runtime error: index out of range	confirmed	-	-	✗	-	✔
	27	unexpected error from the vectorized engine	fixed (8d1865fd)	-	-	✗	-	✔
	28	tuple length mismatch	confirmed	-	-	✗	-	✗
	29	use of crdb_internal_vtable_pk column not allowed	fixed (5cc456bb)	-	-	✗	-	✗
	30	top-level relational expression cannot have outer columns	confirmed	-	-	✗	-	✗
	31	cannot map variable 7 to an indexed var	confirmed	-	-	✗	-	✗
	32	expected *DString, found tree.dNull	fixed (6eabc2f3)	-	-	✗	-	✗
	33	invalid memory address or nil pointer dereference	fixed (b4d5b0b8)	-	-	✔	-	✗
	34	aggregate function is not allowed in this context	fixed (1c8dd156)	-	-	✗	-	✗
	35	invalid memory address or nil pointer dereference	fixed (de8a3c77)	-	-	✔	-	✗
	36	expected subquery to be lazily planned as routines	fixed (9f319ddb)	-	-	✔	-	✗
	37	tuple contents and labels must be of same length: [], [alias_0]	confirmed	-	-	✗	-	✗
	38	unhandled type *tree.RangeCond	fixed (0d647800)	-	-	✗	-	✗
	39	referenced descriptor ID 1: descriptor not found	confirmed	-	-	✗	-	✗
	40	invalid datum type given: inet, expected int	fixed (ff87db04)	-	-	✗	-	✗
	41	unexpected statement: *tree.SetTracing	confirmed	-	-	✗	-	✗
	42	cannot overwrite distribution ...	confirmed	-	-	✗	-	✗
	43	no output column equivalent to 6	fixed (b9b8da67)	-	-	✔	-	✔
	44	index out of range [0] with length 0 (in function handling)	confirmed	-	-	✗	-	✗
	45	unrecognized relational expression type: alter-table-unsplit-all	confirmed	-	-	✗	-	✗
	46	schema change PostCommitPhase, index out of range [1]	fixed (f0dede19)	-	-	✗	-	✗
	47	generator functions cannot be evaluated as scalars	confirmed	-	-	✗	-	✗
	48	could not parse "1 sec" as type bool: invalid bool value	confirmed	-	-	✗	-	✔
	49	SetAnnotation(), index out of range [4] with length 1	fixed (c6cf5189)	-	-	✗	-	✗
	50	locking cannot be used with virtual table	confirmed	-	-	✗	-	✗
	51	no known encoding type for array	confirmed	-	-	✔	-	✗
	52	zero transaction timestamp in EvalContext	confirmed	-	-	✗	-	✗

Table 2: New Assertion Failures detected by `ParserFuzz`. `ParserFuzz` detects 81 bugs in total, including 29 crashes and 52 assertion failures. The `Squirrel`, `SQLsmith`, `SQLsmith`$_C$, `SQLsmith`$_G$ and `SQLancer`$_{+QPG}$ columns represent whether the referenced tools can theoretically detect the mentioned bug, '✔' means 'Yes', '✗' states 'No' and '-' means the tool is not applicable to the target DBMS. `Assertion failure` for `CockroachDB` represents the bug that `CockroachDB` returns unexpected error. But the bug would not crash the whole `CockroachDB` process.

reported than the baseline tools.

Code coverage. `ParserFuzz` reaches the highest DBMS code coverage across all 5 DBMSs' experiments. The code coverage plots are shown in Figure 2c, Figure 2g, Figure 2k, Figure 2o and Figure 2s. Notably, `ParserFuzz` doesn't rely on any input corpus to reach this level of code coverage, sparing the efforts from the DBMS testers to gather interesting queries as input seeds.

Query correctness rate. The query correctness rate is illustrated in Figure 2d, Figure 2h, Figure 2l, Figure 2p and Fig-



(a) `MySQL`: detected bugs **(b)** `MySQL`: grammar edges (k) **(c)** `MySQL`: code coverage (k) **(d)** `MySQL`: query validity (%)

(e) `CockroachDB`: detected bugs **(f)** `CockroachDB`: grammar edges (k) **(g)** `CockroachDB`: code coverage (k) **(h)** `CockroachDB`: query validity (%)

(i) `TiDB`: detected bugs **(j)** `TiDB`: grammar edges (k) **(k)** `TiDB`: code coverage (k) **(l)** `TiDB`: query validity (%)

(m) `MariaDB`: detected bugs **(n)** `MariaDB`: grammar edges (k) **(o)** `MariaDB`: code coverage (k) **(p)** `MariaDB`: query validity (%)

(q) `SQLite`: detected bugs **(r)** `SQLite`: grammar edges (k) **(s)** `SQLite`: code coverage (k) **(t)** `SQLite`: query validity (%)

Figure 2: Evaluation of different testing tools on `MySQL`, `CockroachDB`, `TiDB`, `MariaDB` and `SQLite`.

ure 2t. `ParserFuzz`, along with other mutation-based fuzzing tools, generally has a lower query correctness rate compared to generation-based tools that rely on hand-written templates. For example, `SQLancer`$_{+QPG}$, `SQLsmith`$_G$ and `SQLsmith` all achieve high query validity in their own tests, with `SQLsmith`$_C$ being an exception. However, these generation-based tools lack the flexibility to produce diverse query statements, because all the generated queries patterns must be hand-written by the developers, making the process labor-intensive. Therefore, `ParserFuzz` can find the highest number of bugs by generating more diverse queries and saturating all the grammar rules defined for the parsers.

Overall, ParserFuzz can find more memory errors than other testing tools, because it thoroughly examines all the grammar rules defined in the parser, and can reach the highest number of grammar coverage upon testing. Although most generation-based DBMS testers can guarantee a high query correctness rate, the requirement for hand-written SQL templates limits their scalability, resulting in less flexible solutions in the end.

5.3 Contribution of Coverage Feedback

To understand the contribution of code coverage, we introduce an alternative configuration of ParserFuzz, labeled as $\texttt{ParserFuzz}_{\texttt{-cov}}$, to evaluate the fuzzer performance without code coverage feedback. $\texttt{ParserFuzz}_{\texttt{-cov}}$ discards all code coverage information obtained from the DBMS execution, transforming ParserFuzz into a pure random query generation tool. We compare ParserFuzz with $\texttt{ParserFuzz}_{\texttt{-cov}}$ in all the 5 supported DBMSs. The results are embedded in the same plots we used to evaluate on different tools in Figure 2.

Unique bug numbers. Without code coverage guidance, the pure generation-based testing tool $\texttt{ParserFuzz}_{\texttt{-cov}}$ degrades significantly in bug finding capability. It detects 6 bugs, as opposed to 19 bugs detected by ParserFuzz across all DBMSs. The largest difference occurs in the CockroachDB evaluation in Figure 2e. ParserFuzz detects 6 bugs in total, but without code coverage guidance, $\texttt{ParserFuzz}_{\texttt{-cov}}$ finds none.

Grammar edge coverage. Interestingly, even if ParserFuzz and $\texttt{ParserFuzz}_{\texttt{-cov}}$ both share the same grammar definition file, ParserFuzz achieves a slightly higher grammar edge coverage compared to $\texttt{ParserFuzz}_{\texttt{-cov}}$. The code coverage feedback aids ParserFuzz in exploring hard-to-triggered query syntaxes, thereby uncovering more interesting syntax elements which are absent in the pure query generation.

Code coverage. ParserFuzz achieves higher code coverage than $\texttt{ParserFuzz}_{\texttt{-cov}}$ as anticipated. Simply covering all syntax elements from the grammar definition file doesn't provide a complete picture. But smartly combines different elements together to form different interesting contexts is also crucial for generating more interesting test cases. Code coverage feedback guides the fuzzer to gradually accumulate interesting syntax elements together in the fuzzing queue. The various combinations between different DBMS features lead to more unexpected SQL contexts for the DBMS handling logic, and eventually lead to more bugs detected.

The code coverage feedback accumulates the interesting syntax features discovered from the grammar rule definition file, and it creates more interesting feature combination contexts. It helps ParserFuzz to achieve higher code coverage, and eventually leads to more bugs detected.

5.4 Contribution of Diverse Syntax Features

To demonstrate how vast query syntaxes enhance bug-finding, we refer to Table 1 and Table 2 to show the benefits. Assuming infinite resources can be allocated, the columns of Squirrel, SQLsmith, $\texttt{SQLsmith}_{\texttt{C}}$, $\texttt{SQLsmith}_{\texttt{G}}$ and $\texttt{SQLancer}_{\texttt{+QPG}}$ in Table 1 and Table 2 indicate whether each bug could theoretically be detected by referenced tools. Given the input corpus and Internal Representation (IR), Squirrel can only detect 8 out of 37 bugs that ParserFuzz found. SQLsmith can detect half of the bugs reported by ParserFuzz (3 out of 6). $\texttt{SQLsmith}_{\texttt{C}}$ can detects 13 out of 37 bugs from CockroachDB. $\texttt{SQLsmith}_{\texttt{G}}$ can detect none of the bugs from ParserFuzz. $\texttt{SQLancer}_{\texttt{+QPG}}$ can detect 11 out of 50. The diverse syntaxes enable ParserFuzz to explore more interesting features from the DBMSs, and trigger more interesting bugs that are overlooked by these baseline tools. Next, we present two case studies to demonstrate the uniqueness of bugs detected by ParserFuzz.

```sql
RECOVER TABLE BY JOB 0;
```

Listing 12: A one-line query that crashes TiDB, which aims to recover a table from a non-existing DDL JOB ID.

One-line query that crashes TiDB. Listing 12 presents a unique bug from TiDB. The PoC is surprisingly simple, consisting of just one line of SQL query. But the simple PoC crashes the TiDB query executor, and results in an immediate loss of connection between the TiDB server and client. The PoC attempts to recover a table that had been previously dropped from the database. A more commonly use case is to directly recover the table by its table name, *i.e.*, using `RECOVER TABLE table_0;` to bring back the deleted table `table_0`. However, in conner cases where the DBMS user has created another table that share the same name as the deleted one, TiDB offers an alternative form of the RECOVER statement, as shown by Listing 12, that uses `DDL JOB ID` to recover the table that were previously removed. The `DDL JOB ID` information can be fetched by using the `ADMIN SHOW DDL JOBS;` statement, where the `DDL JOB ID` saves the unique ID for all Data Definition Language (DDL) after their executions. Interestingly, the `DDL JOB ID` value can never be 0. But the parser from TiDB never checks the value from the PoC, and the value `DDL JOB ID` equals 0 is successfully set in the TiDB backend. In this case, TiDB interprets the statement as `RECOVER TABLE table_name` and then call the `getRecoverTableByTableName` function. Unfortunately, the table name variable is remain uninitialized, resulting in a nil pointer dereference bug from TiDB and crashes the TiDB worker process. This bug is interesting and was never detected before because the `RECOVER TABLE BY JOB` statement has been rarely tested. Since the feature introduced after TiDB version 3.0, there are only 13 instances in the TiDB unit tests reference this feature. Furthermore, all the unit tests are constructed with pre-defined or fixed `DDL JOB ID`, which are not helpful to trigger this bug. Additionally, the official query generation-based testing tool from the TiDB developer, $\texttt{SQLsmith}_{\texttt{G}}$, does not support this grammar feature in its query generation templates. Our tool ParserFuzz directly parses the

```
CREATE TABLE IF NOT EXISTS t0 (c1 INT) PARTITION BY
  HASH(c1);
ALTER TABLE t0 CHECK PARTITION ALL FOR UPGRADE;
ALTER TABLE t0 ORDER BY c1;
```

Listing 13: A `MariaDB` crash that corrupts the database source.

grammar rule definition file designed for `TiDB`, and automatically recognizes the `RECOVER TABLE BY JOB` syntax grammar. `ParserFuzz` then generates the `RECOVER TABLE BY JOB` statement, and fills in the `DDL JOB ID` with arbitrary integer, such as value 0, and eventually triggers this bug.

A database corruption bug from MariaDB. Listing 13 shows a segmentation fault PoC from `MariaDB` DBMS. The `CREATE TABLE` statement creates a table `t0` with one column `c1`. The table is partitioned by `HASH(c1)`. Table partitioning is used to split one table data into multiple subsets, and store them individually to ease their managements or speed up their access speed. The `PARTITION BY HASH` directive tells the DBMS to handle the data partitioning, and make sure the data are distributed evenly in the split partitions. The second statement runs `ALTER TABLE CHECK PARITION` on the just created table. It verifies whether the created partitions contain any errors. The additional syntax `FOR UPGRADE` checks if the current partitions are compatible with the currently running `MariaDB` version. Right after the partition checking, the third statement modifies the table `t0` and reorders the table contents based on the data in column `c1`. By running all three statements together, `MariaDB` crashes with corrupted memory access. What's worse, the PoC also corrupts the `DDL_LOG` section from the database source, causing future `MariaDB` crashes whenever `MariaDB` accesses this database. We have reported the PoC to the `MariaDB` developers and they are working on the patch.

This bug can only be detected by `ParserFuzz` in our evaluation. The crucial step in triggering this bug is to combine `CHECK PARTITION ALL` with `FOR UPGRADE`, and then call `ALTER TABLE` immediate after. However, the syntax features of `CHECK PARTITION ALL` and `FOR UPGRADE` are rarely touched by the existing testing tools. `Squirrel` doesn't support either syntaxes in its internal parser. These two syntax features are absent in the `Squirrel`'s input corpus either. Furthermore, the combined usage of `CHECK PARTITION ALL` and `FOR UPGRADE` is also not presented in the `MariaDB` official unit test library, where `FOR UPGRADE` is more commonly used for `CHECK TABLE` in the test instead of `CHECK PARTITION`. Our tool `ParserFuzz` does not rely on any input corpus to realize the different syntax features, where the input corpus are often gathered from the DBMS's unit test library. Instead, `ParserFuzz` learns the syntax rules from `MariaDB`'s built-in parser, and automatically constructs queries that contain the `CHECK PARTITION ALL` and `FOR UPGRADE` symbols. Therefore, `ParserFuzz` is the only tool we tested that can detect this bug from `MariaDB`.

```
joinop: JOIN_KW JOIN_SYM
      | JOIN_KW nm JOIN_SYM
      | JOIN_KW nm nm JOIN_SYM ;
nm:     IDENTIFIER /* terminal keyword for query arg */
      | JOIN_KW;   /* terminal keyword from tokens */
/* Mapped query tokens for JOIN_KW
 * "CROSS"   -> "JOIN_KW"
 * "FULL"    -> "JOIN_KW"
 * "INNER"   -> "JOIN_KW"
 * "LEFT"    -> "JOIN_KW"
 * "NATURAL" -> "JOIN_KW"
 * "OUTER"   -> "JOIN_KW"
 * "RIGHT"   -> "JOIN_KW"
 */
```

Listing 14: BNF grammar does not tell the whole story. We simplify these rules from `SQLite`'s parser and covert them to BNF.

> The diverse syntax features learned from grammar definition files enable `ParserFuzz` to generate more diverse testing queries, which cover more interesting syntax features from the DBMSs, and therefore brings more interesting bugs that are not possible from the previous tools.

6 Discussion

Syntax rules outside grammar definition file. `ParserFuzz` delivers a promising result in exploring syntax elements defined in SQL grammar definition files. However, some grammar rules can be pushed down to the DBMS backend, not in the grammar definition file. For example, `SQLite` contains a parser generator tool, `Lemon` [43], which supports grammar definition syntax similar to BNF that `Yacc` and `Bison` support. However, the grammar definition file provided to `Lemon` does not strictly represent the ground truth grammar complied by the `SQLite` frontend, since many grammar checks are pushed down to the `SQLite` back-end to handle. Listing 14 shows one case that the parser rule in 'parser.y' cannot faithfully defines all the syntax constraints from the `SQLite`'s parser. The example focuses on the `joinop` keyword, which is used to combine data from two or more tables. The valid query segments matching `joinop` could be 'LEFT JOIN', 'RIGHT JOIN', and 'LEFT INNER JOIN', 'RIGHT OUTER JOIN' etc. As we can see the terminal keyword `JOIN_KW` can map to all the tokens we mention here, so all these valid cases should pass the grammar check. However, the three rules defined in `joinop` do not enforce the order of the `JOIN_KW` tokens, which means a query such as 'LEFT RIGHT JOIN' might also pass the grammar check, and later turns out to be invalid. What's worse, the `nm` keyword brings in `IDENTIFIER` as an alternative choice that we can fill into the `joinop` rules, which would also be rejected by `SQLite` parsing in the end. It turns out `SQLite` implements a function named `sqlite3JoinType`, that are designed to verify the keyword contents passed into the `joinop` grammar rules. It effectively rejects any `IDENTIFIER` keywords in the parsed syntax tree. It also rejects corner cases such as 'LEFT RIGHT JOIN'. However, these additional logics are written in `C` language instead of using BNF grammar define notation. So there is no common pattern we can make used of to acknowledge these extra syntax restrictions. Therefore, we currently rely on human efforts to mark these hidden grammar constraints. For instance, we replace all the `nm` keywords to `JOIN_KW` in `joinop`, and filter out any 'LEFT RIGHT JOIN' in the query generation.

7 Conclusion

We design `ParserFuzz`, a novel fuzzing tool that automatically extracts syntax features from DBMSs built-in grammar definition files. By traversing these grammar files, `ParserFuzz` explores all grammar rules defined in each DBMS, and thus can generate more diverse testing queries than previous DBMS testing tools. `ParserFuzz` detects 81 bugs across five popular DBMSs, `SQLite`, `MySQL`, `CockroachDB`, `TiDB` and `MariaDB`. The evaluation shows `ParserFuzz` achieves the highest grammar coverage, the highest code coverage and reports more bugs within 24-hour experiments. We have reported all detected bugs to the corresponding DBMS developers. They have confirmed all the bugs and fixed 34 of them.

References

[1] Shadi Abdul Khalek and Sarfraz 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.

[2] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2023.

[3] Antonio Bovenzi, Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo. On the aging effects due to concurrency bugs: A case study on mysql. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 211–220. IEEE, 2012.

[4] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006.

[5] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. Data Generation for Testing and Grading SQL Queries. *The VLDB Journal*, 24(6), Aug 2015.

[6] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke 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.

[7] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NTFUZZ: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[8] Catalin Cimpanu. Google Chrome Impacted by New Magellan 2.0 Vulnerabilities. https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities/, December 2019.

[9] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, 2020.

[10] Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. SparkFuzz: Searching Correctness Regressions in Modern Query Engines. In *Proceedings of the Workshop on Testing Database Systems (DBTest)*, Portland, Oregon, 2020.

[11] Google. ClusterFuzz. https://google.github.io/clusterfuzz. (visited in October 2023).

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

[13] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *2013 Second International Conference on Informatics & Applications (ICIA)*, pages 212–215. IEEE, 2013.

[14] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. Dynsql: Stateful fuzzing for database management systems with complex and valid sql query generation. In *Proceedings of USENIX Security Symposium (USENIX Security)*, 2023.

[15] Stephen C Johnson and Ravi Sethi. Yacc: a parser generator. *UNIX Vol. II: research system*, pages 347–374, 1990.

[16] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak 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.

[17] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2020.

[18] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[19] Yabin Li, Yuanping Nie, and Xiaohui Kuang. Fuzzing dbms via nnlm. In *2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*, pages 367–374. IEEE, 2022.

[20] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. Sequence-oriented dbms fuzzing. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2023.

[21] 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.

[22] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering*, pages 225–236, 2022.

[23] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. A Framework for Testing DBMS Features. *The VLDB Journal*, 19(2):203–230, April 2010.

[24] Manuel Rigger. SQLancer. https://github.com/sqlancer/sqlancer. (visited in October 2023).

[25] Michaël Marcozzi, Wim Vanhoof, and Jean-Luc 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.

[26] Chaitanya Mishra, Nick Koudas, and Calisto 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.

[27] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, Baltimore, MD, 2018.

[28] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2020.

[29] Terence Parr and Kathleen Fisher. Ll (*) the foundation of the antlr parser generator. *ACM Sigplan Notices*, 46(6):425–436, 2011.

[30] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[31] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, 2014.

[32] 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 (ESEC/FSE)*, 2020.

[33] 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.

[34] Manuel Rigger and Zhendong Su. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, 2020.

[35] Contessa Brewer Rohan Goswami. Caesars paid millions in ransom to cybercrime group prior to MGM hack. https://www.cnbc.com/2023/09/14/caesars-paid-millions-in-ransom-to-cybercrime-group-prior-to-mgm-hack.html, September 2023.

[36] Zeeshan Shah. Over 2 million Pakistanis' data hacked from restaurants database up for sale. https://www.geo.tv/latest/510830-over-2-million-pakistanis-data-hacked-from-restaurants-database-up-for-sale, September 2023.

[37] Chaofan Shou, Ismet Burak Kadron, Qi Su, and Tevfik Bultan. Corbfuzz: Checking browser security policies with fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 215–226. IEEE, 2021.

[38] Donald R Slutz. Massive Stochastic Testing of SQL. In *VLDB*, volume 98, pages 618–622. Citeseer, 1998.

[39] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *ACM SIGMOD Record*, 42(1):44–49, 2013.

[40] Well-known Users of SQLite. https://www.sqlite.org/famous.html. (visited in October 2023).

[41] Go Fuzzing. https://www.sqlite.org/lemon.html. (visited in October 2023).

[42] MySQL Customers. https://www.mysql.com/customers/. (visited in October 2023).

[43] The Lemon LALR(1) Parser Generator. https://www.sqlite.org/lemon.html. (visited in October 2023).

[44] CockroachDB Customers. https://www.cockroachlabs.com/customers/. (visited in October 2023).

[45] PingCAP Partner. https://www.pingcap.com/partners/. (visited in October 2023).

[46] PostgreSQL Clients. https://www.postgresql.org/about/. (visited in October 2023).

[47] SQLSmith. https://github.com/anse1/sqlsmith. (visited in October 2023).

[48] CockroachDB Parser. https://github.com/cockroachdb/cockroach/blob/master/pkg/sql/parser/sql.y. (visited in October 2023).

[49] MariaDB Parser. https://github.com/MariaDB/server/blob/11.3/sql/sql_yacc.yy. (visited in October 2023).

[50] MySQL Parser. https://github.com/mysql/mysql-server/blob/trunk/sql/sql_yacc.yy. (visited in October 2023).

[51] PostgreSQL: The Parser Stage. https://www.postgresql.org/docs/current/parser-stage.html. (visited in October 2023).

[52] TiDB Parser. https://github.com/pingcap/tidb/blob/master/parser/parser.y. (visited in October 2023).

[53] Testing random, valid SQL in CockroachDB. https://www.cockroachlabs.com/blog/testing-random-valid-sql-in-cockroachdb/, 2016.

[54] Backus–Naur form. https://en.wikipedia.org/wiki/BackusâĂŞNaur_form, 2023.

[55] GNU Bison: Languages and Context-Free Grammars. https://www.gnu.org/software/bison/manual/html_node/Language-and-Grammar.html, 2023.

[56] goyacc. https://pkg.go.dev/golang.org/x/tools/cmd/goyacc, 2023.

[57] SQLsmith Bugs. https://github.com/anse1/sqlsmith/wiki#score-list, 2023.

[58] J. Wang, P. Zhang, L. Zhang, H. Zhu, and X. Ye. A Model-based Fuzzing Approach for DBMS. In *Proceedings of the 8th International Conference on Communications and Networking in China (CHINACOM)*, Aug 2013.

[59] Jiajie Wang, Puhan Zhang, Lei Zhang, Haowen Zhu, and Xiaojun Ye. A model-based fuzzing approach for dbms. In *2013 8th International Conference on Communications and Networking in China (CHINACOM)*, pages 426–431. IEEE, 2013.

[60] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu 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.

[61] Shihao Wen, Peng Jia, Pin Yang, and Chi Hu. Squill: Testing dbms with correctness feedback and accurate instantiation. *Applied Sciences*, 13(4):2519, 2023.

[62] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[63] Wen Xu, Soyeon Park, and Taesoo Kim. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, FL, November 2020.

[64] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D. Viglas, and Allison Lee. Snowtrail: Testing with Production Queries on a Cloud Database. In *Proceedings of the Workshop on Testing Database Systems (DBTest)*, New York, NY, USA, 2018.

[65] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering*, pages 800–810, 2018.

[66] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl. (visited in October 2023).

[67] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao 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, November 2020.