SQLess: Dialect-Agnostic SQL Query Simplification

Li Lin
Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University
Xiamen, Fujian, China
linli1210@stu.xmu.edu.cn

Zongyin Hao
Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University
Xiamen, Fujian, China
haozongyin@stu.xmu.edu.cn

Chengpeng Wang
The Hong Kong University of Science and Technology
Hong Kong, China
cwangch@cse.ust.hk

Zhuangda Wang
Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University
Xiamen, Fujian, China
wurongxin@xmu.edu.cn

Rongxin Wu
Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University
Xiamen, Fujian, China
wangzhuangda@stu.xmu.edu.cn

Gang Fan
Ant Group
Shenzhen, Guangdong, China
fangang@antgroup.com

Abstract

Database Management Systems (DBMSs) are fundamental to numerous enterprise applications. Due to the significance of DBMSs, various testing techniques have been proposed to detect DBMS bugs. However, to trigger deep bugs, most of the existing techniques focus on generating lengthy and complex queries which burdens developers with the difficult of debugging. Therefore, SQL query simplification, which aims to reduce lengthy SQL queries without compromising their ability to detect bugs, is highly demanded.

To bridge this gap, we introduce SQLess, an innovative approach that employs a dialect-agnostic method for efficient and semantically correct SQL query simplification tailored for various DBMSs. Unlike previous works that have to depend on DBMS-specific grammar, SQLess utilizes an adaptive parser, which leverages error recovery and grammar extension to support DBMS dialects. Moreover, SQLess performs a semantics-sensitive SQL query trimming, which leverages alias and dependency analysis to simplify SQL queries with preserving bug-triggering capability.

We evaluate SQLess using two datasets from the state-of-the-art database bug detection studies, encompassing six widely-used DBMSs and over 32,000 complex SQL queries. The results demonstrate SQLess's superior performance: it achieves an average simplification rate of 72.45% in the *PINOLO Dataset*, which significantly outperforms the state-of-the-art approaches by 84.91%.

CCS Concepts

• **Software and its engineering** → *Software maintenance tools.*

Keywords

SQL Query Simplification, Bug Detection, Program Trimming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680317

ACM Reference Format:
Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. 2024. SQLess: Dialect-Agnostic SQL Query Simplification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680317

1 Introduction

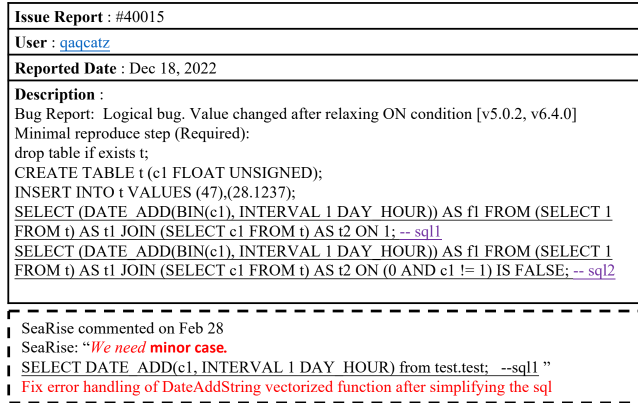
Database Management Systems (DBMSs) are widely utilized for data storage in various software applications [4]. With hundreds of DBMSs developed, each featuring distinct characteristics and functionalities [46], the complexity and large-scale nature of these systems often lead to the presence of bugs.

In database bug detection, methodologies mainly focus on identifying logical and crash errors [7]. The process [6] generates SQL queries for DBMS execution, comparing results with test oracles, to identify bugs when outcomes deviate from expectations. For queries that do not meet expectations, bug reports will be submitted to developers for validation and fixes [18, 25, 41]. However, these queries are usually lengthy because complex and valid queries are instrumental in finding deep bugs in DBMSs [6]. The lengthy query poses significant challenges for developers in pinpointing the root cause of the issue. Figure 1 gives a real-world example of bug report in TiDB. This bug was quickly confirmed by the developer within one day, but it had been lasted for more than two months without any fix due to the lengthy test case. Until a simplified SQL query was provided, this bug then was quickly fixed within two days. This example indicates that SQL query simplification is critical for developers to diagnose the bugs in DBMSs.

While many approaches concentrate on bug detection in DBMSs, most do not adequately consider the importance of the SQL query simplification. Existing studies [7, 11, 29, 31] on database bug detection merely integrate SQL query simplification into their workflows in a simplistic manner. They employ the principles of Delta debugging [50] to minimize SQL queries on structured inputs such as Abstract Syntax Tree (AST). However, a common oversight in these studies is the neglect of interdependencies among different structural elements. Such *dependency-insensitive deletion approach* without semantic checks can render the simplified SQL query non-executable. For instance, in a query like "SELECT c1, c2 from t

Table 1: Comparison of SQL query simplification tools

Tool	Simplification Strategy				Correctness Guarantee		Extensibility	Bug type	
	Expression Simplification	Clause Simplification	Subquery Simplification	Column Simplification	Syntactic Correctness	Semantic Correctness	Adaption to other DBMSs	Crash Bugs	Logical Bugs
SQLancer [29]	✓	✓	×	×	×	×	×	✓	✓
PINOLO [7]	✓	✓	×	×	×	×	×	×	✓
RAGS [31]	✓	✓	×	×	×	×	×	✓	×
APOLLO [11]	✓	✓	✓	✓	✓	×	×	✓	×



Issue Report : #40015
User : qaqcatz
Reported Date : Dec 18, 2022
Description :
Bug Report: Logical bug. Value changed after relaxing ON condition [v5.0.2, v6.4.0]
Minimal reproduce step (Required):
drop table if exists t;
CREATE TABLE t (c1 FLOAT UNSIGNED);
INSERT INTO t VALUES (47),(28.1237);
SELECT (DATE_ADD(BIN(c1), INTERVAL 1 DAY_HOUR)) AS f1 FROM (SELECT 1 FROM t) AS t1 JOIN (SELECT c1 FROM t) AS t2 ON 1; -- sql1
SELECT (DATE_ADD(BIN(c1), INTERVAL 1 DAY_HOUR)) AS f1 FROM (SELECT 1 FROM t) AS t1 JOIN (SELECT c1 FROM t) AS t2 ON (0 AND c1 != 1) IS FALSE; -- sql2

SeaRise commented on Feb 28
SeaRise: "*We need* **minor case.**
SELECT DATE_ADD(c1, INTERVAL 1 DAY_HOUR) from test.test; --sql1 "
Fix error handling of DateAddString vectorized function after simplifying the sql

Figure 1: SQL query simplification in bug report

where c2 > 30", existing tools, not analyzing the interdependencies among elements, might delete the column c2, leading to the condition "c2 > 30" in the WHERE clause becoming syntactically incorrect. Not surprisingly, such coarse-grained simplification is prone to generate invalidate SQLs. Moreover, existing studies mostly depend on the specific grammar of the targeted DBMSs, thus leading to generalizability issues. Migrating the tool to another DBMS requires much manual effort. For example, some prior study [6] showed that, to support most of MariaDB's grammar, SQUIRREL would need to add more than nine thousands lines of code [51].

To overcome the above limitations, we propose SQLess, an approach designed for SQL query simplification that ensures semantic correctness, while also possessing adaptability to various DBMSs rather than relying on a special grammar. To achieve effective simplification with semantic correctness, it is crucial to preserve the dependency relationships among syntactic elements, ensuring that subsequent clauses refer back to elements defined earlier while upholding the integrity of both syntax and semantics. In the aforementioned SELECT query, SQLess will remove the c2 column along with WHERE clause that depends on it, thereby ensuring semantic correctness. We have also observed that the reason existing tools support only specific DBMSs is due to their reliance on parsers tailored for a given DBMS. This specialization results in failure when encountering different database dialects, as the specialized parsers are unable to successfully parse these variations, thereby preventing further simplification. Our idea to resolve this problem is to leverage a standard SQL parser which covers the common core grammar of all DBMSs, together with error recovery mechanism [1] in syntax analysis, to distinguish the dialect parts. Then we expand the grammar by creating parsing rules to adapt the dialects. Once successful parsing is achieved, we can then reuse the existing simplification strategies.

At a high level, our approach is a dialect-agnostic approach, which consists of two main phases. Firstly, SQLess implements an adaptive SQL parser, building upon the foundation of ANTLR [36]. It applies error recovery mechanism [1] for grammar extension and adaptation, effectively accommodating different database dialects. Specifically, for SQL queries that fail to parse successfully due to dialect-specific variations, we establish a new rule at the node where parsing failed, taking into account the unique dialect. We then regenerate a new parser accordingly to accommodate these dialect-specific nuances. Second, we developed a range of simplification strategies based on the adaptive parser. To maintain semantic correctness, semantic analysis are performed before simplifying, forming a def-use dependency graph. This graph assists in ensuring that elements to be deleted are not referenced elsewhere. Furthermore, we employ an oracle-driven simplification verification strategy, where results are checked against different oracles. If the simplified SQL queries continue to yield erroneous results, the simplification is deemed effective.

To evaluate the effectiveness of SQLess, we constructed two datasets of bugs which have been found by the state-of-the-art DBMS bug detection tools: PINOLO [7] and SQLRight [13]. These datasets encompass a wide range of DBMSs, including MySQL [19], MariaDB [15], OceanBase [20], TiDB [43], PostgreSQL [26], and SQLite [32], totaling 32,951 complex SQL queries known to trigger bugs. Our evaluation confirms that SQLess significantly surpasses existing SQL query simplification methods, achieving a 72.45% simplification rate in the *PINOLO Dataset*, outperforming state-of-the-art by 84.91%. In summary, our contributions are as follows:

- We propose an adaptive syntax and semantics-sensitive simplification approach to address the challenges of SQL query simplification.
- We implement SQLess, a simplification tool that can simplify complex queries in the real world.
- We evaluate SQLess on six widely-used DBMSs, and the evaluation results demonstrate the effectiveness of SQLess in SQL query simplification.

We have released the source code of SQLess in the github repository [38] to help developers for SQL query simplification.

2 Motivation

In this section, we introduce the importance of SQL query simplification in the DBMS bug detection process and then illustrate the existing limitations of the techniques concerning SQL query simplification. Finally, we present the technical challenges and the key idea.

2.1 Importance of SQL Query Simplification

In the realm of database bug detection, existing techniques predominantly fall into two distinct categories: identifying logical and crash bugs [7]. Regardless of the bug types, the basic workflow for detecting bugs in a DBMS is the same. Firstly, the SQL mutators

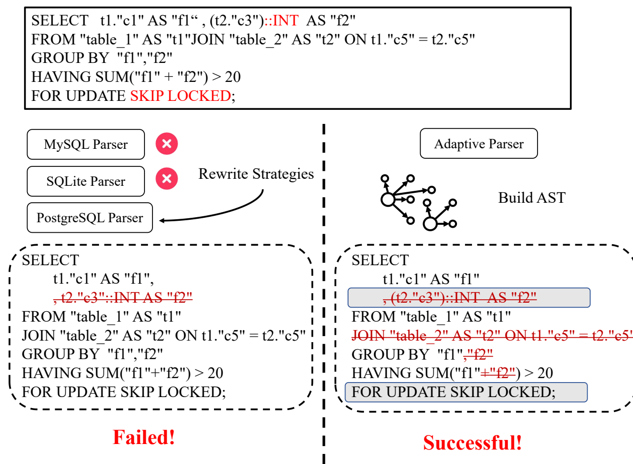


Figure 2: A motivation example

or generators consistently generate diverse SQL queries using different strategies to cover various functionalities of the DBMS as comprehensively as possible. Then, these generated SQL queries are subsequently transmitted to the DBMS, where the server executes them and returns the results. These results are then scrutinized against predefined oracles. Additionally, a crash can be seen as a special kind of test oracle. If the queries do not perform as expected, it can indicate a bug in the database [45]. For queries that do not meet expectations, bug reports will be submitted to developers for validation and fixes. However, our statistical analysis confirms that those generated SQL queries are typically lengthy; the original SQL queries generated by the state-of-the-art bug detection tool PINOLO averaged more than 160 tokens in length, as detailed in Section 5 and Table 2. It poses significant challenges for developers in pinpointing the root cause of the issue. The SQL query simplification module is used to simplify the triggering test cases to facilitate database developers to locate and resolve bugs.

Figure 1 shows a real-world bug of TiDB detected by PINOLO [7]. Although the reporter manually simplified the complex SQL query, the lengthy SQL query still made developers hard to debug. Two months passed, and the bug remained unfixed until this SQL query was simplified. Since manual simplification is a time-consuming and challenging task, especially for users who are non-experts in the domain of databases [28], there is an urgent need for an automated tool that can assist developers in simplifying SQL queries.

2.2 Challenges of SQL Query Simplification

Although SQL query simplification plays a crucial role in the detection of DBMS bugs, to date, there has yet to be a systematic study dedicated exclusively to the exploration of SQL query simplification. Existing work on database bug detection integrates SQL query simplification into their workflows, employing methods for SQL query simplification as outlined in Table 1. Table 1 provides a comparative overview of various SQL query simplification tools, detailing their strategies, correctness guarantee, DBMS extensibility, and so on. Expression simplification and clause simplification [7, 29, 31] are relatively easy to implement, but the scope for simplification is limited. The elimination of operations within subqueries and removing unused columns [11] broaden the extent of the simplification, which enhances the simplification effectiveness. However, existing tools have yet to effectively address these two main challenges: adaptability to various DBMSs and preserving semantic correctness and bug-triggering capability after simplification.

The first challenge involves achieving adaptability to various DBMSs, as existing efforts predominantly depend on the specific grammar of target DBMSs to simplify SQL queries. SQL dialects [47] represent subtle variations of the SQL standard, with each being unique to specific database systems and featuring distinct grammatical nuances. These variations manifest differences in SQL syntax and support for data types, implementations of functions and stored procedures, and proprietary features. For example, the `::INT` type cast and `SKIP LOCKED` clause are specific to PostgreSQL in Figure 2. `SKIP LOCKED` allows transactions to skip locks held by others, unavailable in DBMSs like MySQL. Similarly, `::INT` differs from standard SQL cast syntax, affecting parsing and simplification if ignored. When introducing a new DBMS or updating to a version with distinct SQL dialects, existing approaches will fail. Migrating a tool to a new DBMS is time-consuming and labor-intensive [6]. Previous approaches would first apply a specific parser such as MySQL Parser to parse the SQL query, which will lead to parsing failures when encountering the unique dialect of PostgreSQL. If parsing fails, it would require significant effort to reimplement the simplification strategies on a new parser—a process both time-consuming and labor-intensive.

Another significant challenge is ensuring semantic correctness after simplification. Semantically incorrect SQL queries will directly lead to execution errors and miss opportunities for further simplification. The key to preserve the semantic correctness is to maintain the dependency relationships between syntactic elements. For example, if they remove the alias C from a query Q but another element E uses C, the DBMS will throw a semantic error. Deletions that overlook semantic correctness can cause execution failures. For instance, previous approaches might indiscriminately delete a column like "f2" without accounting for its dependencies, as illustrated by the failure in Figure 2.

2.3 Key Idea

Our idea to resolve the first challenge is to construct an adaptive parser to accommodate various DBMS dialects. To achieve this, we first leverage a standard parser based on ANSI SQL standards [2] together with error recovery mechanism [1] to recognize the dialect parts of a given SQL query. Then, we create a new parser by performing grammatical extensions to adapt the dialect-specific elements. This enables the application of various simplification strategies across different DBMSs. Take Figure 2 as an example. We firstly employ the standard SQL grammar to parse this SQL query. Unfortunately, with the standard SQL grammar, our initial parsing attempt fails when it encounters dialect-specific elements. To mitigate this issue, we leverage error recovery mechanism which allows the parser to recognize the interruption and bypass it temporarily. Figure 2 demonstrates an example where the parser, utilizing error recovery, skips over the dialect-affected part of the query, marked in grey within the SELECTELEMENT "f2", and continues parsing the remainder. Subsequently, to handle the unrecognized parts, we incorporate a new rule into the parser's grammar. The rule, defined as "fullColumn: uid dottedId COLON COLON dataType", enables

the parser to interpret the PostgreSQL-specific type casting `::INT` which is an extension to the standard grammar that allows the parser to recognize and process type casts by defining a "dataType" non-terminal. We will further elaborate this example in Section 3.2. After that, a new parser is generated by the extended grammar and successfully parse this SQL query into AST.

To address the second challenge, i.e., ensuring semantic correctness after simplification, our approach identifies the dependencies between various elements within a query through rigorous semantic analysis. Initially, we analyze alias relationships and dependencies. A def-use graph is then constructed, serving as the foundation for applying diverse simplification strategies. This approach involves careful deletion of def elements along with their dependent uses, ensuring semantic correctness. Furthermore, we identify and remove redundant def elements, maximizing simplification efficiency. Take Figure 2 as an example. When deleting a column "f2", previous approaches might recklessly delete it without considering its dependencies, while our approach ensures that both the column and its dependent elements are simultaneously deleted. Specifically, in this case, the uses in GROUPBY clause and HAVING clause containing "f2" are deleted, and the unused definition "t2" is also deleted.

It should be noted that the design of adaptive parser is critical for maintaining the integrity and correctness of semantic analysis during the simplification process. If we were to simply disregard dialect-specific segments, such as "(t2."c3")::INT" in Figure 2, the resulting AST would be incomplete. Such incompleteness impairs semantic analysis, which in turn leads to the missing chances for simplification, particularly when it causes misunderstandings or oversights regarding key components such as aliases for columns and tables. For example, in the scenario depicted in Figure 2, if the parser overlooks the dialect-specific casting of "t2."c3"", the reference target of the alias "f2" becomes unclear since its definition relies on this dialect-specific casting. To ensure semantic correctness, one of possible methods is to preserve the column "f2" and its dependencies, leading to the missing chances of simplification. Furthermore, when removing JOIN CLAUSE, semantic analysis is performed on table "t2", which is implicated by the dialect section "t2."c3"::INT", leading to incomplete semantic analysis and thus posing a risk to semantic correctness. Different from the aforementioned methods, our adaptive parser does not simply bypass these challenging elements but integrates new rules to effectively parse and include these dialect-specific subtrees within the AST. The rule generated by our tool is not a mere placeholder but an integral part of the grammar, enabling the parser to accurately interpret the query segment that includes the dialect. These rules ensure that every part of the query, including dialect-specific syntax, is given a definite structure and meaning, thereby maintaining the integrity of the semantic analysis and the correctness of the subsequent query simplification process.

3 Approach

The goal of this paper is to design a highly adaptable and semantics-sensitive SQL query simplification approach, thereby maximizing the simplification capabilities. This section presents the design of SQLess to show how it simplifies the query statement and how it adapts to different DBMSs.

3.1 Overview

Figure 3 shows an overview of our approach SQLess, featuring two core components: adaptive parser and query trimmer. Adaptive parsing generates SQL parsers for various DBMS dialects, which then parse SQL queries to build an AST for semantic analysis. This results in a def-use graph, preserving semantic correctness. Subsequent simplification strategies prune the AST, and an oracle checker evaluates the simplified queries for bug detection, iteratively producing a set of simplified SQL queries.

The critical components of SQLess are the auto-generation of a parser capable of adapting to various DBMSs and a query trimmer effective in the simplification of SQL queries. To show more technical details, we first introduce an adaptive syntax parser generator capable of producing parsers that accommodate a variety of DBMSs' dialects in Section 3.2, and then we demonstrate how to utilize this parser for SQL query simplification in Section 3.3.

3.2 Adaptive Parsing

In contrast to simplification tools like APOLLO [11], which employs sqlparse [34], and PINOLO [7], choosing PINGCAP PARSER [24] each relies on a specific parser for SQL efficiency as the foundational premise for their simplification process. Adaptive parser develops specialized parsers for different DBMS dialects . To implement the adaptive parser, we build on the ANTLR [36] foundation. The dialect builder creates a tailored grammar file (.g4), which ANTLR [36] uses to produce a new parser. This ensures the parser can interpret SQL queries across different DBMS dialects, providing a robust solution for SQL query simplification.

3.2.1 Preliminaries. ANTLR constructs parsers from .g4 grammar files and is crucial in SQLess for parsing and simplifying SQL queries. Key ANTLR terminologies in our work include:

- **Terminal Symbols (Tokens)**: The smallest elements of the grammar, such as keywords and punctuation. As shown in Figure 4, tokens are represented by bold text in the Abstract Syntax Tree (AST) and appear as leaf nodes.
- **Nonterminal Symbols**: Nonterminal symbols are the non-leaf nodes in a parse tree, like "selectStatement" and "querySpecification" in Figure 4.
- **Production Rules**: The combinations of tokens and nonterminal symbols that define the structure of the language. For example, the "fullColumn" rule, as shown in Figure 4, illustrates a specific production rule where a "uid" is optionally followed by a "dottedId", showcasing how elements in the syntax can be structured together.
- **Error Recovery Mechanism**: To handle dialect issue, we leverage ANTLR's error recovery mechanism to achieve adaptive parsing. For example, when the parser encounters an invalid token sequence, such as an unexpected format within the "fullColumn" rule as illustrated in Figure 4, error recovery mechanism enables the parser to skip over the problematic section and continue processing the subsequent tokens. This feature ensures the parser's resilience and flexibility when dealing with various SQL dialects that may deviate from the standard grammar.

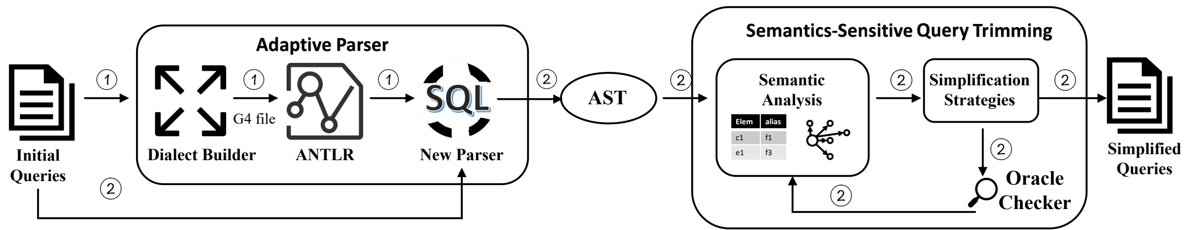


Figure 3: Workflow of SQLess

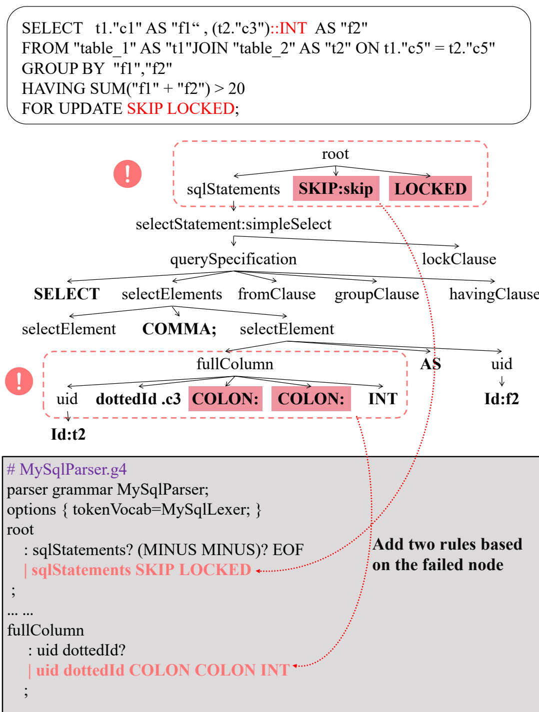


Figure 4: An example of dialect builder

3.2.2 Dialect Builder. Dialect builder is a crucial step in adaptive parser. It generates a grammar file (.g4 file) adaptable to various dialects, which is then used by the parser generator ANTLR to create a new parser. The workflow of dialect builder begins with a foundational grammar file (.g4 file) from which ANTLR generates an initial parser, denoted as Standard Parser, fundamentally based on ANSI SQL standards [2]. This parser attempts to process an initial SQL query. If the query contains DBMS-specific dialects, the parsing will fail. At this point, the dialect builder utilizes error recovery mechanism [1], which is the strategy that allow the parser to skip the point of error and continue parsing. This parser generated by ANTLR that can automatically emit rich error messages upon syntax error and successfully resynchronize much of the time [22]. This strategy facilitate the creation of new rules at the location of failure. Specifically, dialect builder identifies the top of the stack at the point of parsing failure and introduces a new production directly under the failed stack's top rule. It is important to note that our approach involves not only incorporating dialect-specific elements but also introducing new terminal and non-terminal symbols. Remarkably, even if the rule itself fails, some of its subrules may successfully parse parts of the query. Therefore, when adding new productions to the grammar, we may also introduce new non-terminal symbols to encompass these successful sub-components.

Figure 4 exemplifies this process, illustrating how new rules are integrated into the grammar after a parsing failure occurs with dialect-specific SQL constructs. The red dashed boxes emphasize the nodes where the parsing rules failed. These nodes imply the need for an enhancement of the grammar. To rectify this, the dialect builder introduces new production rules into the grammar file, signified by the "|" symbol and outlined with dashed lines, which are inserted at the point where the original parsing halted. This augmentation involves both terminal symbols, such as "COLON" and "INT" used in the type cast, and non-terminal symbols that represent structured grammar rules, such as "fullColumn" and "uid", which incorporates semantic elements like column names. By embedding these new rules directly beneath the point of failure, the adaptive parser can now successfully parse these previously unrecognized elements.

In summary, adaptive parser's methodical approach to extending the Standard Parser with dialect-specific adaptations ensures that the semantic essence of the SQL queries is preserved, thereby enhancing the efficacy of the SQL query simplification task.

3.3 Semantics-Sensitive Query Trimming

After generating a new parser capable of recognizing various dialects, query trimmer then proceeds to simplify SQL queries. SQL query simplification involves iteratively removing as many query elements as possible while ensuring that the simplified query can still trigger a bug. To ensure syntactic correctness, all our simplification operations are conducted on AST. Semantic analysis, including alias analysis and dependency analysis is performed before any node deletions to maintain semantic correctness. We also use an oracle-driven approach to ensure that the simplified queries still retain the same bug-detection capability.

3.3.1 Semantic Analysis. An effective query trimmer maintains semantic correctness by utilizing a def-use graph to accurately depict dependencies in SQL queries.

Figure 5 illustrates the dependency relationships among various elements within a SELECT statement. Firstly, we establish a tabular mapping between aliases and their actual references. A def-use graph has been constructed to represent the data dependencies. In this graph, circular nodes denote the definition of aliases, whereas square nodes indicate the use of aliases. We have identified two distinct types of relationships:

(1) Definition Dependency: Represented by solid black lines, this relationship indicates that the definition of one alias depends on the definition of another. For instance, the definition of alias "f3" relies on the definition of "f4" and "t3". Ensuring the integrity of

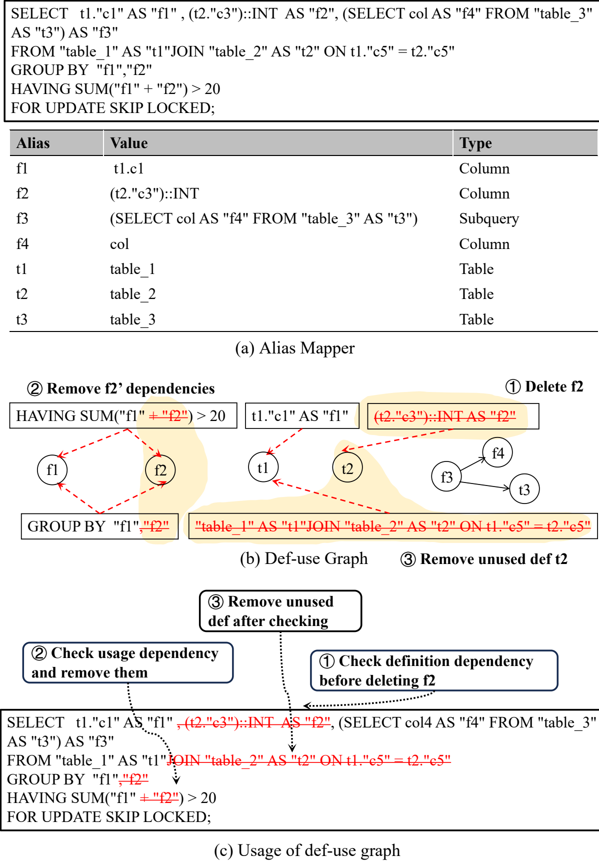


Figure 5: An example of semantic analysis

the deletion process requires removing all instances of "f4" and "t3" when deleting "f3".

(2) Usage Dependency: Represented by dashed red lines, this relationship shows the utilization of aliases within the query. For example, the expression ""f1"+"f2"" employs the alias "f1" and "f2".

Constructing a def-use graph is a cornerstone in maintaining semantic correctness within SQL query processing. **Alias analysis** is the first step. We identify and categorize the aliasing relationships defined in grammar files, including table aliases, column aliases, result-set aliases, function aliases, and expression aliases. Upon detecting an aliasing relationship, we establish a mapping record to trace the correspondence between the alias and its actual reference. Moreover, our method is recursively structured. Throughout the analysis, we unravel and identify nested alias relationships layer by layer. For instance, as depicted in Figure 5, upon recognizing the alias "f3", we delve into the subsequent aliases "f4" and "t3" associated with it. We draw two edges, one from "f3" to "f4" and the other from "f3" to "t3", to represent the dependency relationships between these aliases.

After mapping out the aliases and their actual references, **dependency analysis** necessitates pinpointing the utilization of these aliases within the SQL query elements. For example, as illustrated in Figure 5, the HAVING clause expression "HAVING SUM("f1" + "f2") > 20" employs the aliases "f1" and "f2". We record the instances of alias usage and the precise location within the query to ensure accurate deletions. Furthermore, we maintain a use counter for each alias and then track how frequently each alias is employed within the query. An alias with a use counter at zero indicates that it is no longer in use, signifying potential for safe elimination from the query without compromising its semantic correctness.

Figure 5(c) demonstrates the application of the def-use graph in the simplification of a query. When attempting to remove the column represented by the alias "f2", we first examine the definition dependencies. As the definition of "f2" does not rely on other aliases' definitions, it can be safely deleted. To remove "f2", we must also eliminate the elements that use "f2", so that it will not cause the errors of missing definitions. Then, we will update the def-use graph to reflect these changes. Finally, we inspect the related unused definitions. In this example, "t2" is not used anywhere else within the query, which means it can be safely removed as well.

Overall, semantic analysis works on each part of a query. The rules in grammar extensions can reuse non-terminals in the common core grammar, e.g., table name, column name, rename column, etc. As long as these non-terminals relevant to alias analysis and dependency analysis are successfully parsed, we can run semantic analysis. Through semantic analysis, we can trace the dependencies between elements in a query, allowing us to identify sections that can be safely simplified without compromising the query's semantic correctness.

3.3.2 *General Query Simplification Framework.*

The query trimmer framework is designed for the simplification of SQL queries with the goal of simplifying the query without reducing its bug-triggering capabilities. It takes four inputs: an initialized DBMS for executing SQL queries, an original query that is known to trigger a bug, an oracle checker that mutates SQL queries and checks if the execution results meet the expected outcomes, and a new parser generated from adaptive parser which is capable of parsing SQL queries in different DBMS dialects. The algorithm eventually returns a simplified SQL query.

Algorithm 1 outlines the simplification process. The approach begins with the original query and applies a range of simplification strategies. Each strategy is tested to reduce the query size while still triggering the bug. This is accomplished by constructing a def-use graph representation of the query's AST, deleting nodes based on the strategy, and then unparsing the modified AST back to SQL query. The original and mutated queries are executed, and if the mutated query continues to exhibit the bug, as determined by the oracle checker, the simplified query is updated. This process iterates until no further reductions are possible, resulting in the shortest query version that still can trigger the target bug.

3.3.3 *Simplification Strategies.*

Our query simplification framework is equipped with five categories of simplification strategies:

Clause simplification involves reducing the query by simplifying components such as WHERE clause and HAVING clause. This may include removing redundant conditions or simplifying complex clauses without altering the query's meaning.

Column simplification aims to minimize the SQL query by removing columns that do not affect the erroneous results. This strategy simplifies the query by ensuring only relevant columns are selected, which can improve performance and readability.

Algorithm 1 Simplifying bug-triggering queries

```
Input: DBMS: initial DBMS
    original_query: original query that can trigger bug
    Oracle_Checker: test oracle in DBMS.
    Parser_new: A parser generated by the adaptive parser
Output: simplified_query: minimal query triggering bug
simplified_query ← original_query
// Apply simplification strategies
for strategy in strategies do
    AST_temp ← Parser_new(simplified_query)
    DefUse_Graph ← ConstructGraph(AST_temp)
    AST_del ← DeleteNodeInAst(AST_temp, strategy)
    Slim_Query ← Unparser(AST_del)
    Mutated_Query ← Oracle_Checker.mutate(Slim_Query)
    Res_ori ← RunQuery(Slim_Query, DBMS)
    Res_mut ← RunQuery(Mutated_Query, DBMS)
    if Oracle_Checker.check(Res_ori, Res_mut) is bug then
        simplified_query ← Slim_Query
    end if
end for
return simplified_query
```

Subquery simplification refers to the process of minimizing or eliminating subqueries. We apply various strategies such as clause simplification or column simplification to subqueries or, where possible, to remove them entirely. For instance, with the query "`SELECT c FROM t WHERE c IN (SELECT c FROM t GROUP BY c HAVING COUNT(*) > 10);`", we may simplify by eliminating the nested GROUPBY and HAVING clauses in the subquery.

Expression simplification includes reducing the complexity of arithmetic and logical expressions within the query. It may involve applying algebraic simplifications or removing redundant expressions that do not change the outcome. For instance, in expressions like $c1 \land c2$, we may try to remove $c1$ or $c2$ to see if the result remains unchanged.

Optimizer hints and other modifiers strategy aim to remove or adjust hints and modifiers that guide the database's query optimizer. For instance, the query "SELECT /*+ index(emp ind_emp_sal) */ * FROM emp WHERE deptno = 200 AND sal > 300;" includes an optimizer hint "/*+ index(emp ind_emp_sal) */". The simplification process would remove such hints to prevent them from influencing the execution plan in a way that might mask the presence of a bug.

4 Implementation

We have implemented our approach SQLess as a tool, comprising 6,783 lines of code. In what follows, we present more details about our implementation to support the reproducibility of our approach.

Adaptive Parsing. Our adaptive parser is built upon the grammar file `MySqlParser.g4` [42], utilizing ANTLR version 4.12.0 [36] for parsing. We select MySQL grammar as the base, because its grammar encompassing the full spectrum of standard SQL syntax. The adaptive parser is powered by ANTLR version 4.12.0, a versatile tool renowned for its support of multiple target languages and its assurance of consistency across different platforms, making it invaluable for cross-platform development. The grammar framework in the adaptive parser utilizes the default ANTLR 4 strategy, ALL(*), combined with its default error recovery mechanism [37]. To clarify, while both adaptive parsing mentioned in the paper and ANTLR's ALL(*) parsing strategy share the same name, they differ in their implementation and application. Lastly, the adaptive parser in SQLess is implemented in Java, leveraging the advanced features of ANTLR.

Semantics-Sensitive Query Trimming. In the query trimmer, we perform alias analysis and dependency analysis based on the syntactic rules defined in the ANTLR grammar file `MySqlParser` [42]. The analyses facilitate the construction of a def-use graph, which is pivotal for understanding the relationships and dependencies within SQL queries. Moreover, we have applied a suite of simplification strategies directly to the AST derived from `MySqlParser`. These strategies are designed to prune the AST effectively, thereby simplifying the complexity of the SQL queries while preserving their semantic correctness.

5 Evaluation

In this section, we evaluate the efficacy of SQLess by applying it to the simplification of real-world SQL queries. Our evaluation is designed to answer the following research questions:

- **RQ1:** How does SQLess perform in simplifying real-world SQL queries in terms of effectiveness and efficiency?
- **RQ2:** Can SQLess outperform state-of-the-art query simplification tools in diverse scenarios?
- **RQ3:** How do the adaptive parsing and the semantic analysis in SQLess benefit the SQL query simplification process?
- **RQ4:** Can SQLess provide useful information for DBMS developers to facilitate the diagnosis of real-world bugs with lengthy SQL queries?

5.1 Experimental Setting

5.1.1 Dataset Preparation. We use two state-of-the-art DBMS testing techniques, PINOLO [7] and SQLRight [13], to generate complex, bug-inducing queries for our experiments.

- PINOLO constructs SQL queries with the approximation relation to address the test-oracle problem, supporting testing MySQL [19], MariaDB [15], Oceanbase [20], and TiDB [43].
- SQLRight guides validity-oriented mutations with coverage to detect logical and crash bugs in DBMS systems. It has been extensively tested on MySQL [19], PostgreSQL [26], and SQLite [32].

We test the latest version of each DBMS by running PINOLO and SQLRight for 24 hours, eventually generating problematic SQL queries that lead to unexpected results. Table 2 summarizes the outcomes: **AllN** (number of problematic queries), **AvgT** (average token length), **MaxT** (maximum token length), **AvgC** (average number of clauses), and **AvgA** (average number of aliases). These metrics gauge SQL query complexity. Both tools generate complex queries, forming two datasets for evaluating SQLess, namely *PINOLO Dataset* and *SQLRight Dataset*, respectively.

5.1.2 Metrics. We use two metrics to assess SQLess: simplification ratio and simplification time. The definitions are as follows.

- **Simplification Ratio (SimRatio)**: This metric quantifies the effectiveness of SQLess in simplifying the complexity of SQL

Table 2: Statistics of bugs found by PINOLO and SQLRight

	PINOLO Dataset					SQLRight Dataset				
DBMS	**AllN**	**AvgT**	**MaxT**	**AvgC**	**AvgA**	**AllN**	**AvgT**	**MaxT**	**AvgC**	**AvgA**
MySQL	8934	185.2	538	17.95	21.03	2	15	17	2	0
MariaDB	18507	152.2	517	15.36	18.96	–	–	–	–	–
Oceanbase	1777	190.5	464	19.10	22.50	–	–	–	–	–
TiDB	3523	165.6	490	16.43	19.34	–	–	–	–	–
PostgreSQL	–	–	–	–	–	35	42.9	128	4.08	2.00
SQLite	–	–	–	–	–	143	48.7	153	4.03	4.27

Table 3: Benchmark results of SQL query simplification

Benchmark	DBMS	ALLN	AvgT	AvgSimRatio	MaxSimRatio	AvgTime (ms)	MaxTime (ms)
PINOLO Dataset	MySQL	8,934	185.2	70.00%	93.47%	595.0	1,387
	MariaDB	18,507	152.2	76.27%	95.11%	1,041.0	4,920
	OceanBase	1,777	190.5	50.90%	90.06%	394.0	879
	TiDB	3,523	165.6	69.48%	93.16%	254.0	1,023
	All	*32,741*	*164.7*	*72.45%*	*95.11%*	*799.5*	*4,920*
SQLRight Dataset	MySQL	2	15.0	35.00%	45.00%	5.2	7
	PostgreSQL	35	42.9	40.50%	87.70%	34.2	267
	SQLite	143	48.7	37.10%	90.32%	46.1	403
	All	*180*	*47.3*	*37.74%*	*90.32%*	*43.3*	*403*

queries which is defined as the percentage reduction in the number of tokens. It is computed as follows:

$$\text{SimRatio}(\%) = \left(\frac{\text{Original Token Count} - \text{Reduced Token Count}}{\text{Original Token Count}} \right) \times 100 \quad (1)$$

- **Simplification Time**: This metric measures the temporal efficiency of the simplification process from initiation to completion. Shorter times are indicative of a more efficient simplification algorithm, which is critical for real-time database bug detection and maintenance tasks.

5.1.3 Baselines. We compare SQLess with two baselines. As illustrated in Table 1, existing tools primarily deploy two approaches:

- **ClauseDelete** : Tools such as SQLancer [29], RAGS [31], and PINOLO [7] employ a strategy of randomly deleting clauses from SQL queries. This approach is straightforward to implement and is limited to the parts of the query.
- **APOLLO**: This tool [11] takes simplification further by allowing subquery simplifications and column deletions, thereby expanding the scope of the simplification strategy.

Note that the above tools all use different parsers and a variety of error types. Since the query simplification strategies in these baselines are not implemented as an isolated component, and we have no way to run the original tools, to ensure a fair comparison, we have implemented these two baselines on the ANTLR MySQL grammar utilized by SQLess. We manually write test cases to verify the correctness of the implementation.

Environment. We conduct the experiments on one server with 104-cores Intel(R) Xeon(R) Gold 6230R CPU @2.10GHz and 500 GB memory. The server operates under the Ubuntu 18.04 OS, with the 5.4.0-135-generic version of the Linux kernel.

5.2 Effectiveness and Efficiency

We evaluate SQLess upon the two datasets, namely *PINOLO Dataset* and *SQLRight Dataset*, which cover four and three DBMSs, respectively. Table 3 shows the detailed statistics of query simplification

❖ **SQL Query Simplification Comparision**

```
WITH `MYWITH` AS ((SELECT (DATE_SUB(`f4`, INTERVAL 1 MONTH)) AS `f1`,(DATE_ADD
(REPEAT(`f5`, 2), INTERVAL 1 DAY_MINUTE)) AS `f2`,(`f4`|0.13687240936980968) AS `f3` F
ROM (SELECT `col_char(20)_undef_signed` AS `f4`,`col_double_undef_unsigned` AS `f7`,`co
l_float_key_signed` AS `f6` FROM `table_3_utf8_undef` IGNORE INDEX (`col_bigint_key_sig
ned`, `col_varchar(20)_key_signed`)) AS `t1` STRAIGHT_JOIN (SELECT (NULL) AS `f8`,(CR
C32(0)) AS `f5`,(!_UTF8MB4 'come' ) AS `f9` FROM (SELECT `col_bigint_undef_unsigned`
AS `f10`,`col_decimal(40, 20)_undef_signed` AS `f11`,`col_decimal(40, 20)_undef_unsigned`
AS `f12` FROM `table_3_utf8_undef` FORCE INDEX (`col_float_key_unsigned`)) AS `t2` WH
ERE (NOT ((LAST_DAY(_UTF8MB4 '2000-09-05 03:27:14' )) NOT BETWEEN `f10` AND _
UTF8MB4 '2016-03-14' )) OR ((DATE_SUB(`f11`, INTERVAL 1 MICROSECOND)) IN (COT
(6),SECOND(_UTF8MB4 '03:40:24' ),`f12`))) AS `t3` ON ((NOT (CAST((NULL) AS CHAR) L
IKE _UTF8MB4 '%0%' )) OR ((COLLATION(`f6`)) IN (1,`f5`,COERCIBILITY(`f4`))) OR (NOT
(ROW(COT(0.12073627913396147),LOG2(0.9688927019445049)^`f4`) IN (SELECT `col_dou
ble_key_signed`,`col_double_key_signed` FROM `table_7_utf8_undef`)))) IS TRUE) UNION A
LL (SELECT (-`f14`) AS `f1`, (BINARY 0.7339937007342865) AS `f2`,(DATE_ADD(PI(), INTE
RVAL 1 SECOND_MICROSECOND)) AS `f3` FROM (SELECT `col_double_undef_signed` AS
`f13`,`col_float_undef_unsigned` AS `f14`, `col_bigint_key_signed` AS `f15` FROM `table_3_ut
f8_undef` IGNORE INDEX (`col_float_key_signed`, `col_decimal(40, 20)_key_signed`) ) AS `t
4` WHERE ((((6557202696029309858) IN (STRCMP(`f15`, `f14`),CHARSET(`f13`),`f13`)) IS T
RUE) OR ((`f14`) BETWEEN `f13` AND `f13`)) IS FALSE HAVING (((ROW(DEGREES(0.8825
865858138652),`f3`*!TO_BASE64(`f3`)*BINARY DAYOFYEAR(_UTF8MB4'2002-02-08')) NOT
IN (SELECT `col_varchar(20)_key_signed`,`col_bigint_undef_signed` FROM `table_3_utf8_un
def`)) IS TRUE) OR ((DATE_ADD(-0, INTERVAL 1 SECOND_MICROSECOND)) IS TRUE) O
R ((EXISTS (SELECT `col_float_undef_unsigned` FROM `table_3_utf8_undef` FORCE INDE
X (`col_decimal(40, 20)_key_signed`))) IS FALSE)) IS TRUE ORDER BY `f14`)) SELECT * FR
OM `MYWITH`;
```

Note: Tokens underlined in red indicate the clauses and elements simplified by ClauseDelete, ~~the black strike-through~~ represents the parts simplified by APOLLO, and the grey boxes highlight the elements simplified by SQLess during the SQL query simplification process.

Figure 6: Queries simplified by different approaches

upon two datasets. In total, SQLess simplifies 32,741 SQL queries from *PINOLO Dataset* and 180 queries from *SQLRight Dataset*.

Simplification Ratio. The column **AvgSimRatio** in Table 3 that SQLess can achieve the average simplification ratios of 72.45% and 37.74% upon *PINOLO Dataset* and *SQLRight Dataset*, respectively. We also observe that the average simplification ratio upon *PINOLO Dataset* is much higher than the one upon *SQLRight Dataset*. The significant difference in the simplification ratios between the two datasets can be ascribed to the intrinsic differences in the SQL queries processed, where the length and complexity

Table 4: Comparison with existing tools in *PINOLO Dataset*. C, A, and S indicate ClauseDelete, APOLLO, and SQLess, respectively.

DBMS	MaxSimRatio			AvgSimRatio			AvgTime (ms)			MaxTime (ms)		
	C	A	S	C	A	S	C	A	S	C	A	S
MySQL	75.34%	89.04%	**93.47%**	23.62%	48.62%	**70.00%**	398	423	**595**	654	1275	**1,387**
MariaDB	79.71%	85.82%	**95.11%**	17.51%	35.46%	**76.27%**	486	589	**1,041**	845	3,854	**4,920**
OceanBase	62.39%	82.60%	**90.06%**	24.09%	35.64%	**50.90%**	308	398	**394**	741	756	**879**
TiDB	79.19%	86.10%	**93.16%**	27.74%	36.59%	**69.48%**	264	246	**254**	576	934	**1,023**

of the queries notably influence the potential for simplification. Longer SQL queries, which typically encompass a greater number of clauses, offer more opportunities for simplification, contributing to a potentially higher simplification ratio.

Simplification Time. As shown by the column **AvgTime** in Table 3, the average simplification time upon *PINOLO Dataset* is 799.5 ms, while it takes SQLess 43.3 ms on average to simplify the queries in *SQLRight Dataset*. Meanwhile, the column **MaxTime** in Table 3 shows that the maximal time costs upon *PINOLO Dataset* and *SQLRight Dataset* are 4,920 ms and 403 ms, respectively. The above statistics provide strong evidence that SQLess can achieve high efficiency upon our experimental subjects, showing that SQLess can be integrated into any existing DBMS testing frameworks without introducing significant extra overhead.

Answer to RQ1: SQLess can simplify real-world SQL queries effectively and efficiently. It achieves average simplification ratios of 72.45% and 37.74% upon *PINOLO Dataset* and *SQLRight Dataset* within 799.5 ms and 43.3 ms on average, respectively.

5.3 Comparison with Existing Approaches

To compare SQLess with existing SQL query simplification approaches, we implement ClauseDelete and APOLLO based on the ANTLR MySQL grammar used in SQLess, such that ClauseDelete, APOLLO, and SQLess can support parsing the same set of queries in MySQL dialect. Finally, we evaluate the three tools upon *PINOLO Dataset* and measure the simplification ratios and time accordingly.

Table 4 demonstrate the comparison results. It is evident that SQLess consistently outperforms the baseline tools in terms of maximum and average simplification ratios. Compared with ClauseDelete, SQLess , manifests a remarkable increase in the maximum simplification ratio by 24.06%(= (93.47% − 75.34%)/75.34%) for MySQL and 19.32%(= (95.11% − 79.71%)/79.71%) for MariaDB. Against APOLLO, the enhancement in the maximum simplification ratio is 4.98%(= (93.47% − 89.04%)/89.04%) for MySQL and 10.82%(= (95.11% − 85.82%)/85.82%) for MariaDB, highlighting the advanced efficacy of SQLess. The average simplification ratio, a critical indicator of the tool's overall performance, also displayed a substantial increment. Compared with ClauseDelete, SQLess improves the average simplification ratio by 196.36%(= (70.00% − 23.62%)/23.62%) for MySQL and an impressive 335.58%(= (76.27%−17.51%)/17.51%) for MariaDB. Compared with APOLLO, the two improvements are 43.97%(= (70.00% − 48.62%)/48.62%) and 115.09%(= (76.27% − 35.46%)/35.46%), respectively. We also conduct the Mann-Whitney U test [3] upon the maximum and average simplification ratios with the null hypothesis that SQLess does not outperform the other approaches significantly. The test results rejected the null hypothesis for both the maximum and average simplification ratios with a confidence level over 0.95 (i.e., p-values are less than 0.05). The statistical analysis provides a strong evidence that SQLess has a significant superiority over the two baselines in terms of simplification ratios. Figure 6 shows a concrete example of SQL query from the *PINOLO Dataset* and different versions of simplified queries made by different approaches. It is evident that the simplification results achieved by SQLess are substantially superior to those of ClauseDelete and APOLLO. We also offer more detailed cases in a public GitHub repository [40], illustrating the systematic simplification of complex SQL queries to essential components. The enhancements in simplification rates are indicative of the sophisticated strategies employed by SQLess, which leverage both alias analysis and the construction of def-use graphs to ensure the semantic correctness of the simplified queries. These strategies are pivotal to the observed gains in terms of the simplification ratio, confirming that a rich repertoire of simplification tactics backed by rigorous semantic analysis can lead to substantial improvements of the simplification.

We also compare SQLess with ClauseDelete and APOLLO in terms of simplification time. As demonstrated by the column **AvgTime** in Table 4, SQLess takes more time to simplify the queries than the two baselines, especially upon MySQL and MariaDB. The root cause is that SQLess has to conduct rigorous semantic analysis to ensure the semantic correctness, which introduces extra overhead in the simplification. However, it should be noted that the average time cost of SQLess is no more than twice average time cost of each of the baseline. Meanwhile, the absolute average time cost is quite low. Hence, it is worth achieving higher simplification ratios with acceptable extra overhead in SQLess.

Answer to RQ2: SQLess achieves higher simplification ratios upon experimental subjects than ClauseDelete and APOLLO, and meanwhile, its time cost is within a comparable and acceptable range relative to the baselines.

5.4 Ablation Study

To quantify the benefit of our technical designs, we conduct a group of ablation studies to evaluate how the adaptive parsing and the semantic analysis contribute the simplification results.

Benefit of Adaptive Parsing. To quantify the benefit of the adaptive parsing in SQLess, we implement SQLess-NoAP without adaptive parsing. Specifically, we utilize the parser generated by ANTLR for the MySQL grammar file as a foundation for simplification and implement SQLess-NoAP on top of this parser. Note that the queries in *PINOLO Dataset* are all compatible to MySQL, which implies that all of them can be parsed by SQLess-NoAP. Therefore, we only evaluate SQLess and SQLess-NoAP upon *SQLRight Dataset* and measure the success ratios of parsing.

Table 5: Comparison between the success ratios of SQLess and SQLess-NoAP upon *PINOLO Dataset*

DBMS	SQLess-NoAP	SQLess	Improvement
MySQL	100.00%	100.00%	0.00%
PostgreSQL	67.35%	100.00%	48.47% ↑
SQLite	55.29%	100.00%	80.86% ↑

Table 6: Comparison between the simplification ratios of SQLess and SQLess-NoSA upon *SQLRight Dataset*

DBMS	SQLess-NoSA	SQLess	Improvement
MySQL	54.09%	70.00%	29.41% ↑
MariaDB	41.34%	76.27%	84.49% ↑
OceanBase	33.55%	50.90%	51.71% ↑
TiDB	59.10%	69.48%	17.56% ↑

Table 5 presents the success ratios of SQLess and SQLess-NoAP in simplifying queries across different DBMS dialects in *SQLRight Dataset*. The results show substantial improvements brought by our adaptive parsing. In the SQLRight dataset, our adaptative parser required the addition of 12 new rules for PostgreSQL and 50 for SQLite to better handle these dialects. With these enhancements, SQLess successfully simplifies 100% of queries for both PostgreSQL and SQLite, whereas SQLess-NoAP, which does not incorporate these new rules, only simplifies 67.35% and 55.29% of queries, accordingly. The improvements of success ratios for simplifying the queries upon the two DBMDs are 48.47%(=(100% - 67.35%)/67.35%) and 80.86%(=(100% - 55.29%)/55.29%), respectively. The comparison results demonstrate the effectiveness of the adaptive parsing, which is introduced to augment the adaptability of SQLess across various DBMS dialects. Without the adaptive parsing, simplification attempts falter at the parsing stage, making SQLess-NoAP confined to specific DBMS dialects.

Benefit of Semantic Analysis. Similarly, we construct the other ablation SQLess-NoSA to disable the semantic analysis. We employ the *PINOLO Dataset* which contains more complex queries with a substantial number of interdependencies between query elements, to evaluate the impact of semantic analysis to SQL query simplification. Note that the queries in *SQLRight Dataset* are very simple and do not contain any dependencies between the elements. Hence, we do not evaluate the ablation SQLess-NoSA upon them.

Table 6 show the simplification ratios of SQLess and SQLess-NoSA upon *SQLRight Dataset*. The results indicate a significant improvement in the simplification ratio when the semantic analysis is enabled. Specifically, the improvements are 29.41%, 84.49%, 51.71%, and 17.56% for MySQL, MariaDB, OceanBase, and TiDB, respectively. This underscores the critical role of semantic analysis in the simplification process, as SQL queries with semantic errors are deemed invalid and cannot be executed. Benefiting from the semantic analysis, SQLess can achieve the query trimming with sufficient guidance and safely conduct an aggressive removal of SQL constructs upon the queries.

Answer to RQ3: Both the adaptive parsing and the semantic analysis are necessary and crucial to achieve an effective SQL query simplification.

5.5 RQ4: Usefulness

To evaluate the usefulness of SQLess, we applied it to simplify the real-world bug reports. To be more specific, we scrutinized the bug tracking systems of the DBMSs in our evaluation benchmarks, including MySQL, MariaDB, TiDB, OceanBase, SQLite, and PostgreSQL, and searched for bug reports with the following criterion: (1) the status of bug reports is either unresolved or unconfirmed; (2) the bug is able to be reproduced with the given SQL queries and the detailed information (e.g., DBMS version). Table 7 presents a summary of our efforts to assess the impact of SQLess in real-world scenarios. The table represents the stages in the bug reporting process for various DBMSs and highlights the timespan from bug detection to resolution for confirmed bug reports, using the **RSD** and **SRD** metrics to underline the usefulness of SQLess.

Table 7: Overview of Bug Reports Simplified by SQLess

DBMS	Submitted	Awaiting	Confirmed	RSD	SRD
MySQL	4	2	2	62.5	13
MariaDB	9	2	7	36.9	4.3
TiDB	3	0	3	219.7	1.7
OceanBase	3	0	3	246.3	0.7
SQLite	2	1	1	0	0
PostgreSQL	1	0	1	56	4
Sum	22	5	17	108.1	3.9

Note: The **Submitted** column indicates the total number of bug reports submitted. The **Awaiting** column accounts for reports pending confirmation. The **Confirmed** column indicates the bug reports that developers have recognized as valuable for confirming or fixing bugs. **RSD (Report-to-Simplification Duration)** denotes the average time, in days, from bug reporting time to the time of submitting a simplified SQL query. **SRD (Simplification-to-Resolution Duration)** represents the average time, in days, from the time of submitting a simplified SQL query to the time that the bug has been confirmed or resolved.

In total, SQLess reported the simplification queries for a total of 22 bug reports that were previously unconfirmed or unresolved. Due to our reports, 17 of them have been confirmed or resolved by developers, reflecting a substantial confirmation rate of 77%. Notably, all bug reports simplified for TiDB and OceanBase have been met with positive responses by developers, as were seven for MariaDB. This demonstrates the real-world applicability and value of SQLess in aiding developers to simplify the SQL queries effectively. With an average RSD of 108.1 days and an SRD of 3.9 days for the sum of confirmed reports, SQLess demonstrates a significant impact in expediting the bug confirmation and resolution process. For OceanBase, the usefulness of SQLess is particularly noteworthy, where our tool facilitated the confirmation or fixing of bugs that had been outstanding for an average of 246.3 days, all within an average span of just 0.7 days. Owing to the rapid resolution times by the developers of PostgreSQL and SQLite, SQLess can only simplify three bug reports for these DBMSs, focusing on issues that were still unconfirmed or unresolved. To track the status of the bugs we generated simplified queries, we published a list of these bugs in a public GitHub repository [39].

The potential users of SQLess primarily fall into two categories. First, the tool can greatly benefit designers of DBMS fuzzers. They typically suffer from a dilemma that the malformed, complex, and lengthy queries are more likely to trigger bugs but are less likely to

be accepted by DBMS developers, which is highlighted by many prior studies [7, 9, 30]. Second, SQLess can potentially benefit DBMS developers when they have difficulties in debugging bugs involving lengthy and complex queries, thereby reducing their workload. For instance, the bug report #32981 for MariaDB elicited appreciative feedback from an experienced developer: "Thank you very much! Yes, it is quite useful." Similarly, in the bug report #1678 for OceanBase, the developer acknowledged the usefulness of the simplified query. Such positive feedback underscores the practical utility of SQLess and its significance in assisting developers to effectively address and fix bugs. Potential users can easily adopt SQLess into their workflow with minimal effort by building their own oracle checker or integrating it with their fuzzing tools.

Answer to RQ4: Feedback on SQLess reveals its considerable usefulness in simplifying complex SQL queries from real-world scenarios, earning commendations from developers and underscoring its importance to both designers of DBMS fuzzers and developers.

6 Discussion

In what follows, we provide more discussion on the threats to validity and future work of SQLess.

Threats to Validity. Validity threats include the implementation of our approach. We address this by applying it to complex queries from diverse DBMSs to verify correct implementation of the adaptive parser and query trimmer. Another threat concerns the theoretical guarantee of the simplification ratio. While we lack theoretical proof, our empirical evaluation on many complex queries demonstrates its effectiveness.

Future Work. Ultimately, the usefulness of SQLess should be evaluated by real developers in actual debugging practice. Although we have conducted several case studies in this work, performing more thorough and comprehensive user studies for more debugging tasks is an important future work.

7 Related Work

Test Case Minimization. Delta debugging [12, 21, 49, 50] and Hierarchical Delta Debugging [16] target simplifying failing test cases to minimal ones that still produce failures. They can be utilized for batch debugging of independent SQL statement collections or lists of numbers. However, they cannot directly be used for SQL simplification because they require adherence to strict grammar rules that SQL's complex and nested structures often violate. Nevertheless, existing work has adapted the principles of Delta Debugging to simplify complex SQL queries that lead to certain behaviors, such as incorrect result sets or performance issues [7, 11, 27, 29, 31]. For example, Reducer [27] employs a two-step approach for minimizing queries: it sequentially removes each line and eliminates column names from SELECT statements. Besides, RAGS [31] eliminates terms within expressions and also removes WHERE and HAVING clauses. Later, APOLLO [11] improves the above approaches by adding several simplification strategies. Unfortunately, these tools can achieve effective simplification only if there are few dependencies between elements. An excessive amount of dependence yields output that is not minimized adequately. Furthermore, these tools expects structured input in the form of an AST, which makes these tools ineffective for handling SQL queries in different dialects, as they cannot generate ASTs effectively for such queries, leading to simplification failures. SQLess is the first systematic exploration of the dialect-aware SQL query simplification problem, distinguishing our work from existing studies.

SQL Parsers for Different DBMSs. SQL language is an ISO/IEC standard [8]. However, every database implements the standard differently, uses different function names for the same operation, and supports extensions accessing specific custom features. Currently, there does not exist one SQL parser for dialects of all popular databases. Sqlparse [34] is a popular Python package using regular expressions to parse SQL queries, supporting most standard queries but failing with more complex ones. There are also several parsers that only support certain dialects. For instance, the Pincap parser [23] and SQL Parser [33] in phpmyadmin cater to MySQL and MariaDB, while libpgquery [14] and its derivatives serve PostgreSQL. Different from the above parsers, JSQLParser [10] can parse multiple SQL dialects but has poor extensibility, requiring laborious manual effort for new dialects. Our work outperforms existing SQL parsers in terms of generality and extensibility. Benefiting from our adaptive parsing, SQLess does not demand any manual work to extend the parser for a new dialect, which permits automating the further simplification process.

Detecting Bugs in DBMSs. The reliability of DBMSs has garnered significant research attention. Specifically, researchers have focused on detecting two types of bugs in DBMSs: crashes and logical bugs. To induce crashes in a DBMS, previous studies have primarily concentrated on automating query generation. On the one hand, generation-based methods employ well-defined rules to create valid DBMS queries [5, 17, 35, 44], ensuring the syntactic validity of the queries. On the other hand, mutation-based approaches apply specific forms of mutators to SQL queries [6, 48, 51]. In addition to automatically generating SQL queries, detecting logical bugs requires an effective test oracle. For instance, SQLancer [29] constructs equivalent queries to ensure consistent results. Similarly, PINOLO [7] leverages the approximation relation as a specific type of metamorphic relation acting as the test oracle. Our approach, distinct from prior SQL testing studies, simplifies SQL queries using SQLess, providing concise, compelling evidence to address bugs, thanks to reduced query sizes.

8 Conclusion

SQL query simplification reduces the length of a lengthy SQL query in different contexts, facilitating debugging and patching in DBMSs. This paper presents SQLess, a dialect-agnostic approach for simplifying SQL queries. SQLess effectively reduces the sizes of queries generated by state-of-the-art DBMS fuzzers, achieving a high simplification ratio with acceptable overhead. Furthermore, SQLess has been demonstrated to support six different SQL dialects adaptively. It has shown great potential to be applied to widespread applications in the field of database bug detection.

Acknowledgement

We thank anonymous reviewers for their insightful comments. This work is supported by the Natural Science Foundation of China (62272400), Leadingedge Technology Program of Jiangsu Natural Science Foundation (BK20202001) and the fund from Ant Group. Rongxin Wu is the corresponding author.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education, Chapter 4.
[2] ANSI. 2023. The SQL Standard - ISO/IEC 9075:2023 (ANSI X3.135). https://blog.ansi.org/sql-standard-iso-iec-9075-2023-ansi-x3-135/. [Online; accessed 12-Dec-2023].
[3] Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*. 1–10.
[4] J Asplund. 2011. Data Science revealed: Data-driven glimpse into the burgeoning new field.
[5] Carsten Binnig, Donald Kossmann, Eric Lo, and M Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 341–352.
[6] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-free DBMS fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[7] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *2023 USENIX Annual Technical Conference*. 345–358.
[8] IEC ISO and N IEC. 2017. ISO/IEC. *IEEE International Standard-Systems and software engineering–Vocabulary* (2017), 1–541.
[9] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In *Proceedings of USENIX Security Symposium*. 4949–4965.
[10] JSQlParser. 2023. JSqlParser. https://jsqlparser.github.io/. [Online; accessed 23-Nov-2023].
[11] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70.
[12] Yong Lei and James H Andrews. 2005. Minimization of randomized unit test cases. In *16th IEEE International Symposium on Software Reliability Engineering*. 10–pp.
[13] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of {DBMS} with Coverage-based Guidance. In *31st USENIX Security Symposium*. 4309–4326.
[14] libpgquery. 2023. libpg_query: PostgreSQL Parser Library. https://github.com/pganalyze/libpg_query. [Online; accessed 23-Nov-2023].
[15] Mariadb. 2023. Mariadb Database. https://mariadb.org/. [Online; accessed Dec-2023].
[16] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.
[17] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 499–510.
[18] MySQL. 2023. MySQL Bug Reporting Guidelines. https://bugs.mysql.com/. [Online; accessed Dec-2023].
[19] MySQL. 2023. MySQL Database. https://www.mysql.com/. [Online; accessed Dec-2023].
[20] OceanBase. 2023. OceanBase Database. https://www.oceanbase.com/. [Online; accessed Dec-2023].
[21] Alessandro Orso, Shrinivas Joshi, Martin Burger, and Andreas Zeller. 2006. Isolating relevant component interactions with JINSI. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*. 3–10.
[22] Terence Parr. 2013. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Chapter 9.
[23] pincap. 2023. Parser - A MySQL Compatible SQL Parser. https://github.com/pingcap/tidb/tree/master/parser. [Online; accessed January-2023].
[24] pincap. 2023. TiDB Parser: The SQL Parser for TiDB. https://github.com/pingcap/tidb/tree/master/parser. [Online; accessed Nov-2023].
[25] PostgreSQL. 2023. PostgreSQL Bug Reporting Guidelines. https://www.postgresql.org/list/pgsql-bugs/. [Online; accessed Dec-2023].
[26] PostgreSQL. 2023. PostgreSQL Database. https://www.postgresql.org/. [Online; accessed Dec-2023].
[27] pquery. 2023. PQuery: Multithreaded SQL Tester / Reducer. https://github.com/Percona-QA/pquery. [Online; accessed 23-Nov-2023].
[28] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.
[29] Manuel Rigger. 2023. SQLancer. https://github.com/sqlancer/sqlancer. [Online; accessed 23-Nov-2023].
[30] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[31] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.
[32] SQLite. 2023. SQLite Database. https://sqlite.org/. [Online; accessed Dec-2023].
[33] sqlparser. 2023. phpMyAdmin SQL Parser. https://github.com/phpmyadmin/sql-parser. [Online; accessed Nov-2023].
[34] SQLParser. 2023. SQLParser. https://www.sqlparser.com/. [Online; accessed Nov-2023].
[35] SQLSmith. 2023. SQLSmith. https://github.com/anse1/sqlsmith. [Online; accessed June-2023].
[36] ANTLR Team. 2023. ANTLR (ANother Tool for Language Recognition). https://www.ANTLR.org/. [Online; accessed 23-Nov-2023].
[37] ANTLR Team. 2023. ANTLR DefaultErrorStrategy Class Documentation. https://www.ANTLR.org/api/Java/org/ANTLR/v4/runtime/DefaultErrorStrategy.html. [Online; accessed 23-Nov-2023].
[38] SQLess Team. 2023. SQLess. https://github.com/SQLess/AdaptSQLess.
[39] SQLess Team. 2023. SQLessBugReports. https://github.com/SQLess/SQLess_Bugreports.
[40] SQLess Team. 2023. SQLessExample. https://github.com/SQLess/Examples/blob/main/README.md
[41] SQLite Team. 2023. SQLite Bug Reporting Guidelines. https://www.chiark.greenend.org.uk/~sgtatham/bugs.html. [Online; accessed Dec-2023].
[42] The ANTLR Organization. 2023. ANTLR Grammars Repository. https://github.com/ANTLR/grammars-v4. [Online; accessed 23-Nov-2023].
[43] TiDB. 2023. TiDB Database. https://www.pingcap.com/tidb/. [Online; accessed Dec-2023].
[44] Jiajie Wang, Puhan Zhang, Lei Zhang, Haowen Zhu, and Xiaojun Ye. 2013. A model-based fuzzing approach for DBMS. In *2013 8th International Conference on Communications and Networking in China*. 426–431.
[45] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. 328–337.
[46] wiki. 2023. Comparison of Relational Database Management Systems. https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems. [Online; accessed Sep-2023].
[47] wikibooks. 2023. SQL Dialects Reference/Introduction. https://en.wikibooks.org/wiki/SQL_Dialects_Reference/Introduction. [Online; accessed Nov-2023].
[48] Michal Zalewski. 2021. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl. Accessed: June 2023.
[49] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.
[50] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.
[51] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.

Received 2024-04-12; accepted 2024-07-03