APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems

Jinho Jung Hong Hu Joy Arulraj Taesoo Kim Woonhak Kang
{jinho.jung, hhu86, arulraj, taesoo}@gatech.edu wokang@ebay.com
Georgia Institute of Technology eBay Inc.

ABSTRACT

The practical art of constructing database management systems (DBMSs) involves a morass of trade-offs among query execution speed, query optimization speed, standards compliance, feature parity, modularity, portability, and other goals. It is no surprise that DBMSs, like all complex software systems, contain bugs that can adversely affect their performance. The performance of DBMSs is an important metric as it determines how quickly an application can take in new information and use it to make new decisions.

Both developers and users face challenges while dealing with performance regression bugs. First, developers usually find it challenging to manually design test cases to uncover performance regressions since DBMS components tend to have complex interactions. Second, users encountering performance regressions are often unable to report them, as the regression-triggering queries could be complex and database-dependent. Third, developers have to expend a lot of effort on localizing the root cause of the reported bugs, due to the system complexity and software development complexity.

Given these challenges, this paper presents the design of APOLLO, a toolchain for automatically detecting, reporting, and diagnosing performance regressions in DBMSs. We demonstrate that APOLLO automates the generation of regression-triggering queries, simplifies the bug reporting process for users, and enables developers to quickly pinpoint the root cause of performance regressions. By automating the detection and diagnosis of performance regressions, APOLLO reduces the labor cost of developing efficient DBMSs.

PVLDB Reference Format:
Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, Woonhak Kang. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *PVLDB*, 13(1): 57-70, 2019.
DOI: https://doi.org/10.14778/3357377.3357382

1. INTRODUCTION

Database management systems (DBMSs) are the critical component of modern data-intensive applications [50, 19, 65]. The performance of these systems is measured in terms of the time for the system to respond to an application's request. Improving this metric is important, as it determines how quickly an application can take in new information and use it to make new decisions.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 1
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3357377.3357382

The theories of optimizing and processing SQL queries in relational DBMSs are well developed [42, 58]. However, the practical art of constructing DBMSs involves a morass of trade-offs among query execution speed, query optimization speed, standards compliance, feature parity achievement, modularity, portability, and other goals [4, 9]. It should be no surprise that these complex software systems contain bugs that can adversely affect their performance.

Developing DBMSs that deliver predictable performance is non-trivial because of complex interactions between different components of the system. When a user upgrades a DBMS installation, such interactions can unexpectedly slow down certain queries [8, 3]. We refer to these bugs that slow down the newer version of the DBMS as *performance regression* bugs, or *regressions* for short. To resolve regressions in the upgraded system, users should file regression reports to inform developers about the problem [2, 7]. However, users from other domains, like data scientists, may be unfamiliar with the requirements and process for reporting a regression. In that case, their productivity may be limited. A critical regression can reduce performance by orders of magnitude, in many cases converting an interactive query to an overnight execution [56].

Regression Detection. To detect performance regression bugs, developers have employed a variety of techniques in their software development process, including unit tests and final system validation tests [10, 5]. However, these tests are human-intensive and require a substantial investment of resources, and their coverage of the SQL *input domain* is minimal. For example, existing test libraries compose thousands of test scripts of SQL statements that cover both individual features and common combinations of multiple features. Unfortunately, studies show that composing each statement requires about half an hour of a developer's time [63]. Further, the coverage of these libraries is minimal for two reasons: the number of possible combinations of statements and database states is exponential; components of a DBMS tend to have complex interactions. These constraints make it challenging to uncover regressions with testing.

Regression Reporting. Performance regressions in production DBMSs are typically discovered while running *complex* SQL queries on *enormous* databases, which make the bug analysis time-consuming and challenging. Therefore, developers typically require users to simplify large bug-causing queries before reporting the problem, in a process known as *test-case reduction* [2, 7]. However, simplifying a query to its essence is often an exercise in trial and error [12, 59, 63]. A user must repeatedly experiment by removing or simplifying pieces of the query, running the reduced query, and backtracking when a change no longer triggers the performance degradation [63]. It is common that regressions go unreported because of the high difficulty of simplifying them. When confronted with a regression, a reasonable user might easily decide to find a workaround (*e.g.*, change the query), instead of being sidetracked by reporting it.

Regression Diagnosis. Even if a user successfully files a minimal bug report, it is still challenging for developers to identify the root cause of the problem [45, 64]. Currently, a developer may manually examine the control-flow and data-flow of the system, or use a performance profiler to determine *where* the system is spending its computational resources. However, such tools will not highlight *why* these resources are being spent [53, 54]. A recent study shows that a developer usually invests more than 100 days on average, a significant amount of effort, on diagnosing a bug [64] [1].

Prior research on the automatic bug detection in DBMS has focused on *correctness* bugs. The RAGS automated testing system, designed by the Microsoft SQL Server testing group, stochastically generates valid SQL queries and compares the results for the generated queries on diverse DBMSs [63]. While this technique uncovered several correctness bugs, it suffers from three limitations. First, it is not tailored for detecting performance regression bugs. It often misclassifies queries that do not suffer from performance regression as candidates for reporting. Second, it does not assist developers with the regression-diagnosis process, leading to longer bug-fixing periods. Finally, the test-case reduction algorithm employed in RAGS is not effective on complex SQL queries (§7.2).

This paper addresses these challenges by developing techniques for automatically detecting Performance regressions, minimizing the queries for bug-reporting, and assisting developers with bug diagnosis. We implemented these techniques in a prototype, called APOLLO. We demonstrate that APOLLO simplifies the regression reporting process for users and enables developers to quickly pinpoint the root cause of performance regressions. We evaluate APOLLO on two DBMSs: SQLite and PostgreSQL [6, 1]. APOLLO discovered 10 previously unknown performance regression bugs, where seven of them have been acknowledged by developers. Among these discovered regressions, it accurately pinpoints the root causes for eight bugs. By automating the detection and diagnosis of regressions, APOLLO reduces the labor cost of developing DBMSs.

In summary, we make the following contributions:

- We introduce a technique to automatically detect performance regression bug in DBMSs using domain-specific fuzzing (§4).
- We propose an algorithm to automatically reduce queries for reporting regression bugs (§5).
- We formulate a technique to automatically locate the root cause of regressions through bisecting and statistical debugging (§6).
- We demonstrate the utility of our automated techniques for detecting, reporting, and diagnosing Performance regressions on two popular DBMSs: SQLite and PostgreSQL (§7).

We will release the source code of our system at: `https://github.com/sslab-gatech/apollo`.

2. MOTIVATION & BACKGROUND

In this section, we first demonstrate the necessity of the detection and automatic diagnosis of performance regressions in DBMSs. Then, we present the challenges to achieve these goals and briefly discuss our corresponding solutions. At the end, we provide an overview of greybox fuzzing and statistical debugging techniques.

2.1 Motivating Examples

DBMSs are enigmatic for many users, as their performance is highly dependent on the complex interactions among many components. These interactions can trigger performance regressions that limit the users' productivity. For example, when a user upgrades the DBMS to a new version, a critical regression bug can slow down certain queries by orders of magnitude, in many cases converting an interactive query to an overnight one [56, 21, 8, 3].

```
SELECT R0.S_DIST_06 FROM PUBLIC.STOCK AS R0
WHERE  (R0.S_W_ID < CAST(LEAST(0, 1) AS INT8));
```

Example 1. Impact on Runtime Performance. Consider the SQL query above derived from the TPC-C benchmark [15]. When we run this query on the latest version of PostgreSQL v11.1 (maintained since NOV 2018), it takes 15800× more time to execute compared to that taken on an earlier version v9.5.0 (maintained since JAN 2016). We attribute this performance regression to the interplay between the query optimizer that overestimates the number of rows in a table and a recently introduced policy for choosing the scan algorithm. Due to the misestimation, the optimizer in the latest version selects an expensive sequential scan algorithm, while in the earlier version, it picks the cheaper bitmap scan instead. This example illustrates the impact of performance regressions on user productivity. We defer a detailed discussion of this bug to §7.3.1.

```
SELECT R0.NO_O_ID FROM MAIN.NEW_ORDER AS R0
WHERE  EXISTS (
  SELECT R1.O_OL_CNT FROM MAIN.OORDER AS R1
  WHERE  EXISTS (
    SELECT R2.H_C_ID FROM MAIN.HISTORY AS R2
    WHERE (R0.NO_W_ID IS NOT NULL) AND (R2.H_C_ID IS 'A')));
```

Example 2. Debugging Complexity. The SQL query above takes two hours to complete on the latest version of SQLite v3.27.2 (FEB 2019), while an earlier version of SQLite v3.23.0 (APR 2018) completes this query in an hour. The commit that introduced this regression contains 242 lines of additions and 116 lines of deletions, spanning 20 files. The developer has to spend a significant amount of time to pinpoint the root cause of this regression even if she knows the bug is introduced in this commit. This example illustrates the complexity of debugging performance regressions. We defer a detailed discussion of this regression to §6.2.2.

2.2 Challenges & Our Approaches

The oft-repeated struggles of DBMS users and developers with discovering, reporting, and diagnosing performance regressions motivate the need for an automated toolchain to assist with these key tasks. Unfortunately, prior research on automated bug detection in DBMSs focused on functionality-related bugs [63]. For example, the RAGS system cannot uncover any performance regression or help diagnose the root cause. We develop APOLLO to tackle the following challenges to provide automatic performance regression detection, minimization, and diagnosis in DBMS systems:

Finding Regressions. We stochastically generates SQL statements for uncovering performance regressions using *fuzzing* [34, 40, 44]. This technique consists of bombarding a system with many randomly generated inputs. Researchers have successfully used it to find security vulnerabilities and correctness bugs [70, 67]. Unlike those bugs, validating performance regressions is challenging because the ground truth of the regression is unclear and may be heavily affected by the execution environment. We tackle these problems by applying a set of validation checks, incorporating the feedback from DBMS developers, to reduce false positives.

Reducing Queries. When a regression is discovered, the next challenge is for users to report it [7, 2]. As the queries are usually large and spans multiple files, users have to perform *query reduction* to minimize the report. However, manual query reduction is time-consuming and challenging, especially for users who are non-experts in the domain of databases [12, 59]. We solve this problem by iteratively distilling a regression-causing statement to its essence. This takes out as many elements of the statement as possible while

[1] The authors attribute the delayed diagnosis process to: (1) communication delays between bug reporters and developers, (2) inability to reproduce the symptoms, and (3) lack of good diagnosis tools.

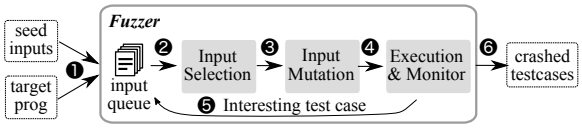


Figure 1: Overview of greybox fuzzing. A fuzzer keeps mutating known inputs to generate new ones. It feeds each input to the tested program and monitors the execution behavior. Based on the result, it updates the input generation policy to trigger desired program features.

ensuring that the reduced query still triggers the problem.

Diagnosing Causes. Once a regression report is filed, the final challenge is for developers to diagnose its root cause [45, 64]. To accomplish this, a developer either manually examines the program, or utilizes a performance profiler to determine *how* the CPU time is distributed on different functions. However, this process cannot highlight *why* the time is distributed this way [53]. To simplify the diagnosis process, we use two techniques to automatically identify the root cause. First, we bisect historical commits to locate the first one that introduces the performance decrease. Second, we leverage statistical debugging to co-relate the execution decrease to suspicious source lines within the commit [33, 53].

2.3 Background

Fuzzing. Fuzzing is an automated technique for testing software [55]. It mutates inputs in a pseudo-random fashion and monitors whether the target program shows unexpected behaviors (*e.g.*, crashes) on each mutated input. Feedback-driven fuzzing utilizes the *feedback* (*e.g.*, code coverage) from previous runs to dynamically update the policy of input selection and mutation. Figure 1 illustrates the fuzzing process in AFL, a widely used feedback-driven fuzzer [22]. The detailed steps are as follows: ❶ AFL initializes a priority queue with the given input. ❷ It selects the most interesting input from the queue. ❸ AFL mutates the input using predefined policies (*e.g.*, modifying several bytes, inserting interesting values, or deleting some blocks). ❹ It launches the target program with the newly generated input and monitors the execution status for anomaly behaviors. ❺ AFL collects feedback metrics of the execution (*e.g.*, code coverage) and ❻ compares the metric against prior runs. If the execution crashes with a new code coverage, it reports the bug-triggering input; if the execution terminates normally with a new code coverage, AFL appends the new input to the queue and go to ❶; otherwise, it returns to ❷ for another iteration.

Statistical Debugging. Statistical debugging is an effective technique for diagnosing failures in systems [33, 53, 54, 64]. It formalizes and automates the process of finding program (mis)behaviors that correlate with the failure. Statistical debugging consists of two steps. First, it uses binary instrumentation to keep track of various program behaviors. Second, it uses a statistical model to automatically identify *predicates* on the program state that highly correlated with the program failure. The overall steps of a statistical debugging pipeline are as follows: ❶ It injects code to the target program to evaluate boolean predicates (*e.g.*, true or false) at various program points. ❷ Upon termination, the instrumented program will generate a trace that records how often each predicate was observed and found to be true. ❸ It constructs a statistical model by consolidating a large number of traces (*e.g.*, across many inputs) to find predicates that are predictive of failure. ❹ It ranks these predicates based on their *sensitivity* (*i.e.*, account for many failed runs) and *specificity* (*i.e.*, do not mispredict failure in successful runs). ❺ The developer may use the list of predicates to identify buggy program components (*e.g.*, functions containing branches that are highly correlated with failures). We will further describe more details in §6.2.

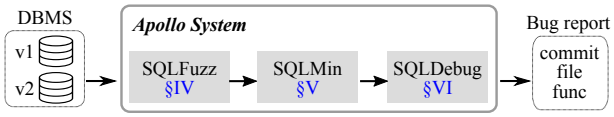


Figure 2: Architecture of APOLLO. It takes in two versions of one DBMS, and produces a set of performance regression reports. Internally, APOLLO mutates SQL queries to trigger significant speed difference. Then it minimizes the bug-triggering queries and performs diagnosis on the regression.

3. SYSTEM OVERVIEW

APOLLO will help developers build DBMSs that deliver robust performance: it detects performance regression bugs, simplifies bug reports and automates bug diagnosis. Therefore, we design APOLLO to contain three components: the fuzzing engine SQLFUZZ (§4), the query minimization framework SQLMIN (§5), and the diagnosis engine SQLDEBUG (§6), as shown in Figure 2. SQLFUZZ relies on a feedback-driven fuzzing engine to generate SQL statements to uncover performance regressions. It provides a wider coverage of the SQL input domain compared to prior work [63]. Also, it generates expressive SQL statements to support many combinations of DBMS features. This allows us to empirically measure the utility of fuzzing DBMSs. SQLMIN leverages domain-specific query-reduction algorithms to provide compelling evidence that a regression exists. Formalizing the statement-reduction problem allows us to investigate the effectiveness of a collection of complex query-reduction transformations. SQLDEBUG assists with identifying the root cause of performance regressions in DBMSs. It relies on a statistical model of program successes and failures to track down causes across versions of a DBMS. This model enables developers to isolate the relationships between specific program behaviors and the eventual success or failure of a program run.

Workflow. We anticipate that users and developers will adopt APOLLO in the following steps. A user first deploys APOLLO on her machine and connects it to the target DBMS. ❶ SQLFUZZ will perform feedback-driven mutational fuzzing to generate SQL statements and provide wider coverage of the SQL input domain (§4). The key idea is to guide the fuzzing engine based on domain-specific feedback (*i.e.*, probability for each clause in a SQL query), including runtime performance. ❷ Next, SQLMIN will automatically distill the regression-activating SQL statements discovered by SQLFUZZ to their essence for filing regression reports. The user will send the regression report to the developers containing the query reduced by SQLMIN. ❸ The developer will use SQLDEBUG to diagnose the root cause of the regression from the simplified test case produced by SQLMIN.

4. SQLFuzz: DETECTING REGRESSIONS

SQLFUZZ automatically constructs SQL queries and runs them for uncovering performance regressions in a target DBMS. Figure 3 illustrates the architecture of SQLFUZZ. It begins by stochastically generating a set of general SQL queries. These queries are based on the widely supported SQL-92 standard [11] and do not rely on any features that are only available in a particular DBMS version. Next, SQLFUZZ sends each generated query to two versions of the DBMS and records the execution time for each version. If the ratio of the latest version's execution time to the earlier version's exceeds the threshold (*e.g.*, 3 times), SQLFUZZ treats the query as a potential regression-triggering input. Finally, it applies a set of validation rules on the short-listed SQL queries to confirm that each input consistently activates a performance regression. This validation step is crucial for reducing the number of false positives and is tailored for reporting performance regressions.

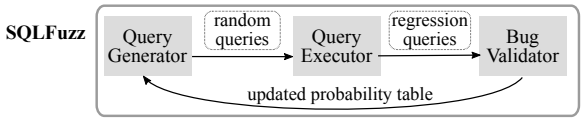


Figure 3: Overview of SQLFuzz. Our query generator keeps constructing queries based on the user specification and the probability table. The executor sends each query to both DBMS versions, and records execution plans and used times. The validator conducts a series of checks to remove false positives on each query that exhibits a significant performance drop on the newer version. Based on validated regressions, we update the probability table for guiding subsequent query generation.

4.1 SQL Query Generation

SQLFuzz employs a top-down approach to generate SQL statements. As shown in the `GenerateQuery` procedure of Algorithm 1, it first collects the meta-data regarding the target DBMS and the database (line 13). This meta-data includes information of the supported operators by the DBMS and the schema of the tables in the database. Since the latest version of the DBMS may implement new operators, we use the old version to collect the meta-data. Next, SQLFuzz derives a query specification from the meta-data (line 15). The specification consists of a set of rules for symbol substitution, which can be recursively applied for generating queries [49, 71]. SQLFuzz constructs the specification based on a *probability table*, which defines a frequency for each clause in a SQL query (*e.g.*, `WHERE` clause is used in 70% of generated queries). If the constructed query requires a *target*, like a column or a table, we randomly select an appropriate one based on the schema of the database. SQLFuzz then populates the abstract syntax tree (AST) of the query based on the specification (line 16). We use AST in query generation as it provides a simple way to mutate the query. For example, SQLFuzz easily inserts subqueries by adding subquery branches into existing AST. It then transforms the AST to the query (line 17).

SQLFuzz provides supports for controlling the query complexity (line 18). It first validates whether the generated query conforms to the SQL grammar. After that, it checks if the generated query satisfies user-defined query complexity, like the maximum depth of subqueries, number of used subqueries, number of `JOIN`s, number of clauses, and length of total query (line 18). If the query passes those checks, the procedure returns the query to the caller. Otherwise, it repeats these steps to generate another query.

Feedback-driven Query Generation Process. SQLFuzz utilizes information from prior fuzzing rounds to improve the probability of discovering queries that trigger performance regressions [27, 22, 51]. Specifically, it uses the probability table to manage this information across rounds. When SQLFuzz confirms that a query uncovers a performance regression, it extracts all the entities (*e.g.*, clauses) from the query, and increases the probabilities of these entities in the probability table. The intuition is that these entities may contain certain characteristics that lead to sought-after DBMS behaviors. By constructing queries that contain these entities, it is more likely to generate queries that trigger performance regressions. For example, if most of the regression-triggering queries contain the `JOIN` clause, the feedback-driven mechanism will gradually increase the frequency for `JOIN`, like from 50% to 60%. Therefore, the newly generated queries are more likely to contain a `JOIN` clause.

Algorithm 1 presents the procedure for discovering performance regressions. SQLFuzz starts with a probability table that assigns the same priority to all entities (line 1). It relies on the aforementioned `GenerateQuery` procedure to generate well-formed SQL queries (line 3). It sends each generated query to two versions of the DBMS, *i.e.*, DBMS_{old} and DBMS_{new} and computes the ratio of the corresponding execution times (line 6). If DBMS_{new} exhibits a significant performance drop for the current query, SQLFuzz applies a set of validation rules to confirm the performance regression (line 8). Finally, it stores the regression-triggering query after and updates the probability table to increase the frequency for each contained entity (line 10). To avoid overfitting problem (*e.g.*, generated query always contains `JOIN` clause), SQLFuzz assigns maximum probability for each clause. SQLFuzz continuously executes this loop for discovering more performance regressions.

Algorithm 1: Procedure for generating SQL queries and discovering performance regressions

Input : $DBMS_{old}$: old DBMS version, $DBMS_{new}$: new DBMS version, *DB*: given database, *threshold*: least time difference
Output : *query*: bug-triggering queries

```
1  prob_table ← InitProbTable();
2  while True do
       // Generate random queries for each round
3      query ← GenerateQuery(DBMS_old, prob_table) ;
       // Run queries on specified DBMSs
4      time_old, plan_old ← RunQuery(query, DBMS_old) ;
5      time_new, plan_old ← RunQuery(query, DBMS_new) ;
6      diff_ratio ← time_new / time_old ;
       // Store regression query after FP test
7      if diff_ratio > threshold then
8          if Validate(query) then
9              StoreQuery(query) ;
10             UpdateProbTable(prob_table, query, diff_ratio) ;
11
12 Procedure GenerateQuery(DBMS_old, prob_table)
       // Retrieve meta-data about DBMS and database
13     meta-data ← RetrieveMetaData(DBMS_old) ;
       // Construct query specification for query generation
14     while True do
15         specification ← BuildSpecification(meta-data, prob_table) ;
16         ast ← SpecificationtoAST(spec) ;
17         query ← ASTtoQuery(ast) ;
18         if SyntaxCheck(query) and ComplexityCheck(query) then
19             return query ;
```

4.2 Regression Validation

Validating queries that trigger performance regressions is challenging [63]. The reasons for this are twofold. First, developers may attempt to improve the performance of frequently executed queries, even if the changes result in slowing down other queries. We should report only regressions that affect a wide range of queries of real-world applications. This is different from uncovering correctness bugs or security vulnerabilities, where the ground truth (*i.e.*, the sought-after program behavior) is well-defined. Second, the query execution time may be affected by the environment (*e.g.*, number of CPU cores, memory capacity). This makes it challenging to reproduce the performance regression in a different environment.

SQLFuzz tackles these problems by applying a set of validation checks to reduce false positives. These checks incorporate the feedback we received from DBMS developers. We demonstrate that these validation checks effectively reduce false positives in §7.1.

- **Non-executed Plan.** When a query is submitted to the DBMS, the query optimizer attempts to find the optimal execution plan from a large number of alternatives [39]. However, the old version and the new version of the DBMS may select different plans for the same query, resulting in different execution times. One special case resulting in significant time difference is when the plan selected by the old version produces an empty result in the middle of its execution, which immediately returns the empty result without running the left subplans. Meanwhile, the plan selected by the new version does not see any empty result, and thus it will complete the whole plan without any early termination. A major source of false positives is such *non-executed plans* [30]. DBMS developers clarified that this is an inherit problem of

DBMS, and they do not consider such plans to be bugs.

- **Non-deterministic Behavior.** Non-deterministic clauses and routines may return different results for the same query, resulting in many false alarms. For example, the `LIMIT K` clause should return the top `K` records. However, without an `ORDER BY` clause, the top `K` records are randomly selected [28]. The result of statement `WHERE (timestamp>now())` depends on the non-deterministic routine `now()`. To avoid such false positives, we only use deterministic clauses and routines for query generation.
- **Catalog Statistics.** If the statistics maintained by the DBMS are not up-to-date, the optimizer may not select the optimal plan. We eliminate false positives due to out-of-date statistics by always updating the statistics (*e.g.*, using `ANALYZE` for PostgreSQL).
- **Environment Settings.** To mitigate the impact of environment settings on performance, we configure SQLFuzz to use the same settings across different DBMS versions. For example, PostgreSQL uses a 4 MB memory buffer for sorting and `JOIN`. If the executor requires more memory, it has to spill over the results to disk, leading to a high execution time. We resolve this problem by increasing the size of the memory buffer to 256 MB.

4.3 Design Details

The design goals for SQLFuzz are threefold: (1) efficiency, (2) reproducibility, and (3) extensibility. We now discuss the technical details we leverage in SQLFuzz to accomplish these goals.

Efficiency. Canonical fuzzers usually restrict the size of test cases so that the fuzzer can process more test cases, like limiting the file size to accelerate the fuzzing of a file parser [48]. However, in DBMSs, even a short query with a few `JOIN` clauses can take several hours to complete. Therefore, we leverage the following DBMS-aware techniques for maximizing the efficiency of SQLFuzz.

- **Limiting Query Complexity.** The following features of a SQL query have a heavy impact on its execution time: the number of `JOIN` clauses, the number of subqueries, and the number of statements. To accelerate fuzzing, we constrain the complexity of generated queries (line 18 in Algorithm 1) regarding these aspects. For example, a query contains at most four `JOIN` clauses).
- **Syntax Check.** Executing queries with syntax errors reduces computational efficiency. SQLFuzz circumvents this problem by applying syntax checks before running the query and discarding invalid queries (line 18 in Algorithm 1).
- **`LIMIT` Clause.** SQLFuzz uses the `LIMIT` clause to restrict the number of returned rows to accelerate query execution. However, as `LIMIT` introduces non-deterministic behavior, during validation we remove all `LIMIT` clauses from the queries and confirm whether they consistently activate the regressions.
- **Query Timeout** SQLFuzz applies a user-defined timeout for each generated query to amortize the time budget across several queries. This allows the tool to gracefully handle time-consuming queries that satisfy the above checks. We adopt two different timeout values, one for the query generation and another for each query execution. To decide the timeout value, we empirically assign them to utilize machine as much as possible. Developers can adjust the timeout number as they want.

Reproducibility. DBMS developers may want to configure the tool to reproduce queries in a deterministic way and focus on specific classes of regressions. Table 1 lists a subset of the settings currently supported by SQLFuzz: regression threshold, depth of subqueries, number of `JOIN`s, the depth of expressions, the seed of the random number generator, and other features of the validator.

- **Query Generator:** When the `Non_Deterministic` flag is enabled, the query generator refrains from constructing queries containing non-deterministic clauses and routines. The `Max_Join` and `Max_Query_Size` settings determine the maximum number of `JOIN` clauses and the query size, respectively.
- **Query Executor:** `Regression_Threshold` indicates the minimal performance gap for the tool to consider a query as regression-triggering. `DBMS` and `DBMS_Configuration` are used for connecting to the target DBMS. `Execute_Analyze` parameter contains the DBMS-specific command for updating the catalog's statistics.
- **Regression Validator:** When the `Non_Executed` flag is enabled, the validator discards any query if its plan contains any non-executed part. When `LIMIT_Clause` is enabled, we remove `LIMIT` clauses before regression validation to eliminate this randomness.

Table 1: Configurable Settings of SQLFuzz. While settings for the query generator and validator help reduce false positives, those for the query executor allow SQLFuzz to support different DBMSs.

	Configuration	Description
Query Generator	`Non_Deterministic`	Discard non-deterministic funcs
	`Max_Query_Size`	Maximum query size
	`Max_Join`	Maximum number of `JOIN`s
	`Allow_DB_Update`	Allow DB modifications
	`Timeout`	Maximum generation time
Query Executor	`Regression_Threshold`	Magnitude of regression
	`DBMS`	Targeted DBMS
	`DBMS_configuration`	DBMS-specific configuration
	`Execute_Analyze`	Update catalog statistics
	`Timeout`	Maximum execution time
Bug Validator	`Non_Executed`	Discard non-executed plans
	`LIMIT_Clause`	Discard `LIMIT` clauses

Extensibility. We leverage three techniques to improve the extensibility of SQLFuzz to support multiple DBMSs. First, the core components of SQLFuzz (*i.e.*, query generator, executor, and validator) are DBMS-agnostic. We introduce a layer of indirection between general-purpose parameters and specific commands used by a target DBMS (*e.g.*, `Execute_Analyze`). Second, the fuzzer supports both client-server DBMSs (*e.g.*, PostgreSQL, MySQL, or MariaDB) and embedded DBMSs (*e.g.*, SQLite). It communicates with client-server DBMSs using a networking component. In the case of embedded DBMSs, SQLFuzz spawns the system as a new process and directly executes queries. Third, SQLFuzz supports two types of query generators to address query dialect problem: a SQLSmith-based generator [26] for SQLite and PostgreSQL systems, and a RQG-based random query generator [20] for MySQL and MariaDB systems.

5. SQLMin: QUERY REDUCTION

After discovering a regression-triggering query of the target DBMS, the user may try to minimize the query before sending the bug report to developers. This minimization technique consists of repeatedly removing as many elements of the query as possible, while ensuring that the performance regression is still preserved. Therefore, manual effort for query-reduction is time-consuming and error-prone. SQLMin addresses these problems by automating this process in a general way. Besides queries produced by SQLFuzz, SQLMin can also minimize regression-triggering queries from other sources, like normal executions or other fuzzing tools.

Comparison with Prior Research. We note that the Rags system [63] and Reducer [23] also try to tackle the automated query-minimization problem. The minimization algorithm of Rags consists of two steps: discard terms in expressions and remove `WHERE` and `HAVING` clauses. Reducer also applies two steps for query-minimization: delete line by line and remove column name from `SELECT` or `INSERT`. Because of the simplicity, their algorithms suffer from three limitations. First, they do not consider dependencies between the removed expressions and the rest of the query, resulting in invalid queries. For example, if they remove expression E from

Algorithm 2: Procedure for reducing bug-triggering queries

Input : $DBMS_{old}$: old DBMS version, $DBMS_{new}$: new DBMS version, *query*: original query, *min_time*: minimum execution time, *threshold*: minimum regression threshold
Output : *min_query*: minimal query triggering regression

1 **Procedure** `Minimize`(*DBMS*$_{old}$, *DBMS*$_{new}$, *query, min_time, threshold*)
2 old_size ← min_size ← ∞ ;
`// Bottom-up reduction`
3 **for** *subquery* ∈ *query* **do**
4 time$_{old}$, plan$_{old}$ ← `RunQuery`(*subquery, DBMS*$_{old}$) ;
5 time$_{new}$, plan$_{new}$ ← `RunQuery`(*subquery, DBMS*$_{new}$) ;
6 **if** *time* > *min_time* **and** $\frac{time_{new}}{time_{old}}$ > *threshold* **then**
7 **if** `Length`(*subquery*) < *min_size* **then**
8 min_query ← subquery ;
9 min_size ← `Length`(*subquery*) ;
`// Top-down reduction`
10 **for** *element* ∈ `GetComponents`(*min_query*) **do**
`// Component = {subquery, clauses, lists}`
11 min_query′ ← min_query - element ;
12 time$_{old}$, plan$_{old}$ ← `RunQuery`(*min_query*′, *DBMS*$_{old}$) ;
13 time$_{new}$, plan$_{new}$ ← `RunQuery`(*min_query*′, *DBMS*$_{new}$) ;
14 **if** *time* > *min_time* **and** $\frac{time_{new}}{time_{old}}$ > *threshold* **then**
15 min_query ← min_query′ ;
`// Iterate until no reduction`
16 **if** `Length`(*min_query*) < *old_size* **then**
17 old_size ← min_size ← `Length`(*min_query*) ;
18 **go to** *line 3* ;
19 **else**
20 **return** *min_query*

a query Q but the reduced query Q' still uses E, the DBMS will throw a syntax error. Because of the lack of dependency-tracking, they also cannot handle correlated subqueries. Second, they employ a *top-down* approach that starts from the entire query and iteratively removes as many expressions as possible. If the performance regression is associated with a nested subquery, a *top-down* approach will end up with a syntax error due to the dependencies between inner and outer queries. Finally, they are tailored for correctness or functionality bugs and will not preserve the performance regression during query reduction. Specifically, the reduced query should produce the same result, or crash the program in the same way. However, for regression bugs, the reduced query just has to exhibit a performance drop over the developer-specified threshold.

5.1 General Query Reduction Framework

SQLMIN is a general framework for reducing regression-triggering queries. It takes three sets of input from the user: the two versions of the DBMS, the original regression-triggering query, and the `min_execution_time` and `regression_threshold` parameters. We require the execution time of a reduced query to be larger than `min_execution_time` (*e.g.*, 10 ms) to ensure that the reduction algorithm is not misled by the inaccuracy of time measurement or the environmental noises (*e.g.*, interference from other concurrently running processes). We require the reduced query to trigger a performance drop higher than the `regression_threshold` parameter.

Algorithm 2 illustrates the query-reduction algorithm. SQLMIN iteratively reduces the query until convergence. It initially adopts a bottom-up approach to reduce each subquery (line 3 to line 9). For each valid subquery, it checks whether the subquery exhibits the desired performance drop and takes non-trivial time to complete. If so, we set this as the best reduced query (`min_query`) (line 8). SQLMIN then adopts a top-down approach to further reduce the subquery by removing different components of the query: subqueries, clauses, and lists (line 10 to line 11). If the reduced subquery still exhibits the desired performance drop and takes non-trivial time to complete, we will update `min_query` to the reduced one (line 15). Finally, SQLMIN checks whether this loop iteration successfully reduced the query (line 17). If so, it continues on to the next iteration to check

```
1  SELECT S1.C2
2  FROM (
3  SELECT
4    CASE WHEN EXISTS (
5      SELECT S0.C0
6      FROM OORDER AS R1
7      WHERE ((S0.C0 = 10) and (S0.C1 IS NULL))
8    ) THEN S0.C0 END AS C2,
9  FROM (
10   SELECT R0.I_PRICE AS C0, R0.I_DATA AS C1,
11     (SELECT I_ID FROM ITEM) AS C2
12   FROM ITEM AS R0
13   WHERE R0.I_PRICE IS NOT NULL
14       OR (R0.I_PRICE IS NOT S1.C2)
15   LIMIT 1000) AS S0
16 ) AS S1;
```

❶ Extract subquery
❷ Remove condition
❸ Remove column list
❹ Remove subquery
Remove Dependency
❺ Remove clause

Figure 4: Query Minimization Example. SQLMIN reduces the size of the discovered query while preserving the performance regression property. It adopts both top-down and bottom-up approaches for the reduction.

whether it is possible to further reduce the `min_query` (line 18). Otherwise, it returns `min_query` after detecting convergence (line 20).

Example 3. Query Minimization. We use the example in Figure 4 to illustrate the query-reduction process. The original SQL query Q contains 3,912 bytes and the performance drop associated with it is 2.4×. We relax the regression threshold to 1.8×. With the bottom-up approach, SQLMIN ❶ extracts the largest subquery and transforms it to a valid query (shaded region) by eliminating dependencies between the outer and the inner queries. For example, it removes referred column name from the `WHERE` clause (line 14). With the top-down approach, it sequentially removes several entities from the query (❷-❺): condition in the `WHERE` clause (line 7), column list and subquery in the `SELECT` (line 10 and 11), and the `LIMIT` clause (line 15). SQLMIN iterates this reduction until convergence. This enables it to resolve dependencies between entities. For example, there is a dependency between the condition `SubQ_0.C1` (❷) and the column `Ref_0.O_I_DATA` (❸). If SQLMIN removes the referencing condition without eliminating the referenced column, the reduced query will be syntactically invalid. During the first iteration, SQLMIN only removes the referencing condition since eliminating it does not trigger a syntax error. During the second iteration, it removes the referenced column, thereby converging to a syntactically valid reduced query.

6. SQLDebug: DIAGNOSING ROOT CAUSE

Even with the reduced regression-triggering query, DBMS developers still need to invest a significant amount of effort on diagnosing the root cause of the problem [64]. In this section, we present the design of SQLDEBUG, a tool for assisting developers with root cause diagnosis. SQLDEBUG first identifies the commit (*i.e.*, a set of changes) to the DBMS source code that gave rise to the problem (§6.1). It then localizes the root cause of the regression among these changes using statistical debugging (§6.2).

6.1 Identification of Problematic Commit

Developers co-ordinate changes to the source code of a DBMS using *version control systems* (*e.g.*, `git` [13] and `fossil` [14]). Each change is identified using a unique *commit identifier* (*e.g.*, `307a94f`). Two different versions of a DBMS may be separated by tens to thousands of commits. SQLDEBUG uses a binary search algorithm across these commits to identify the one that introduced the performance regression. This technique, referred to as *commit bisecting*, has been applied to other complex software systems [16, 18, 17].

SQLDEBUG first extracts all commits between the old version and the new version of the DBMS from the version control system. It then checks whether the query triggers the performance regression between the two commits corresponding to two versions. This check

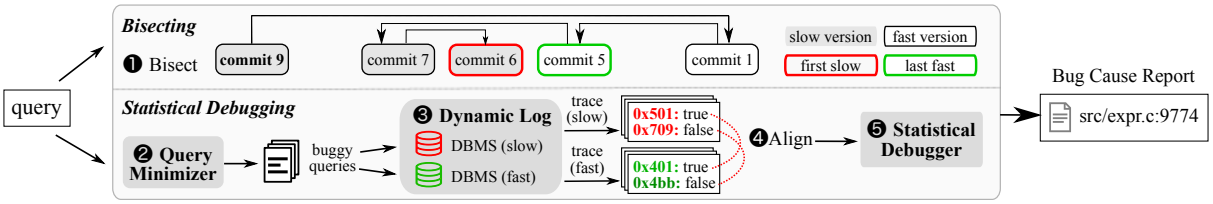


Figure 5: Architecture of SQLDEBUG. The diagnosis process consists of two techniques: (1) commit bisecting and (2) statistical debugging. First, SQLDEBUG identifies the earliest commit that introduced the performance regression. Then, it collects execution traces on the identified commit and utilizes statistical debugging to localize the root cause of the regression (*i.e.*, file name, function name, and line number).

is to confirm the source code is consistent with the released DBMS binaries regarding the regression. If the query does not trigger the performance regression, SQLDEBUG just includes more older and newer commits until it finds two commits that show the regression. After that, SQLDEBUG uses a binary search algorithm to find the first commit that introduces the regression. We call the commit corresponding to the older and faster version as *fast commit*, and call the on corresponding to the newer and slower version as *slow commit*. In each iteration of the binary search, we pick up the commit in the middle between the current fast commit and the slow commit, called *middle commit*. If the DBMS compiled from the middle commit is much slower than the fast commit, we set the middle commit as the new slow commit. Otherwise, we set the middle commit as the new fast commit. We keep this search until no commit exists between the fast commit and slow commit. The slow commit is the first one that triggers the regression.

Compilation Cache. During the commit bisecting, SQLDEBUG retrieves and compiles many versions of the DBMS, which leads to a slow diagnosis process. To address this problem, SQLDEBUG caches the compiled versions of the DBMS and reuses them when possible across searches to accelerate the diagnosis process.

Example 4. Commit Bisecting. Figure 5 illustrates a commit bisecting process. There is a performance regression between the first commit `c1` and the latest commit `c9`. Our goal is to find the first commit that introduces the regression. SQLDEBUG starts by validating the performance regression between `c1` and `c9`. It then retrieves and compiles the middle commit `c5` and runs the regression-triggering input on that version. If the query runs fast on `c5`, it updates the search to begin at `c5`. By iterating this binary search, SQLDEBUG concludes that the regression is activated by `c5`.

6.2 Localization of Root Cause

After identifying the commit that introduced the regression, a developer will further localize the root cause of the problem to a particular source line of the source code. This step is crucial for a *major commit* that contains changes spans a large number of source code files. We automate this localization process in SQLMIN by extending the traditional statistical debugging technique (Figure 2.3).

Challenges. The canonical statistical debugging technique suffers from two limitations in locating regression bugs. First, it only supports analysis in one version of the program. However, SQLMIN has to perform comparative analysis across two versions of the DBMS. Second, it requires a significant number of regression-triggering inputs to construct a statistical model. However, it is challenging to collect a large set of loosely correlated queries that trigger the same regressions in DBMSs. SQLDEBUG addresses the first challenge by aligning execution traces from two versions of the DBMS. It tackles the second challenge by using SQLMIN to derive loosely correlated queries that trigger the same regression.

Figure 5 illustrates the way we extend the statistical debugging technique for diagnosing performance regressions. ❶ SQLDEBUG begins with the first slow commit and the last fast commit identified through commit bisecting. It compiles these versions to binaries with debugging information [38]. ❷ It then uses SQLMIN to produce more regression-triggering queries during the query reduction. Specifically, it collects all the intermediate sub-optimally reduced queries no matter they trigger the performance regression or not. ❸ Next, SQLDEBUG instruments the binaries to collect the list of evaluated branches while executing these *bad* and *good* queries in both versions. ❹ For each query, it aligns the pair of traces obtained from the two versions, based on the differences from the source code. We discuss the trace collection and alignment steps in detail later in this section. ❺ Finally, it leverages statistical debugging to process the aligned traces from all queries and generates a list of branches in the source code that strongly correlate with the regression.

6.2.1 Trace Collection and Alignment

Execution Trace Collection. SQLDEBUG utilizes *dynamic binary instrumentation* to collect the execution traces from both versions of the DBMS. Specifically, it instruments the binary to record each conditional branch instruction to obtain the list of branches that are either taken or not taken during execution. Since the dynamic instrumentation tool, DynamoRIO [35], does not support multi-processed software systems, we configure the DBMS in the single-process mode for bug diagnosis (*e.g.*, single-user mode in PostgreSQL [29] launches DBMS within one process). By applying the single-process mode, DBMS runs necessary modules in multiple threads within the same process; thus, APOLLO can diagnose the root cause correctly. We introduce implementation detail in §7.

Trace Alignment. Since two binaries of a DBMS have different address space layouts, the execution traces collected by SQLDEBUG do not share the same set of addresses. SQLDEBUG aligns these different layouts using three steps for statistical debugging. First, it only considers instructions in functions that were modified across these two binaries. Second, it maps each instruction to a (`function`,`offset`) pair, where `offset` is the distance from the given instruction to the first instruction of the `function`. For example, (`TupleHashTableMatch`,`0x445`) identifies that the instruction starts at offset `0x445` from the start of the `TupleHashTableMatch` function. Finally, it sequentially aligns instructions from two versions if their function names and offsets match each other. SQLDEBUG examines both instructions with and without matches in the other version, since the latter changes may also correlate with the regression.

6.2.2 Applying Statistical Debugging

After aligning the traces, we build statistical debugging model. by following a conventional method to infer buggy locations. We calculate the probability that an execution fails when the corresponding predicate (branch) is taken. Using the calculated probability, we infer possible buggy locations with its rank. We introduce a great reference for readers who would like to know the details of statistical debugging method [64].

2-Version Statistical Debugging. Our statistical debugging model differs from the traditional one in two ways. First, since we are

Table 2: SQLDEBUG validation. Validation result on existing regressions.

DBMS	Version	Bisecting	Statistical Debugging	Validate
SQLite	v3.6.23	defaf0d99	where.c:sqlite3WhereBegin()	✓
	v3.7.14	ddd5d789e	where.c:bestBtreeIndex()	✓
PostgreSQL	v8.1.2	7ccaf13a0	nbtutils.c:_bt_checkkeys()	✓

targeting performance regressions, our notions of program success and failure correspond to fast and slow query execution, respectively. Second, we restrict the number of ranked predicates (*i.e.*, conditional branches) by only examining those related to the commit identified using bisection. The statistical debugging model returns a list of predicates and their *Importance metric* [64] (*e.g.*, predicate A: 0.887). Using the debugging information in the compiled binaries, SQLDEBUG maps the addresses of the predicates to the source code of the DBMS: (`function_name`, `line_number`).

Validation with Existing Regressions. To confirm the validity of the diagnosis result, we collect reproducible performance regressions that are submitted to the DBMS community. Then we manually compare the actual bug fix with our diagnosis result. As shown in Table 2, SQLDEBUG correctly pinpoints the root cause of the regressions. After bisecting roughly identifies multiple candidate locations, statistical debugging diagnosed the actual location, *i.e.*, the file, the function, and the related predicates.

Example 5. Diagnosing Root Cause. We use SQLDEBUG for identifying the root cause of the performance regression triggered by Example 2 in §2.1. ❶ After 11 binary search iterations, bisection reveals that the last fast-commit and the first slow-commit are `f856676` (DEC-31-2018) and `e130319` (DEC-31-2018), respectively. ❷ SQLDEBUG then constructs a set of loosely related intermediate queries using SQLMIN. Among the queries, it randomly chooses 10 that trigger the regression and 10 that do not trigger the bug. ❸ We feed these 20 queries to the corresponding binaries and collect the execution traces. ❹ SQLDEBUG filters out all instructions that are not relevant to the changes between these two commits and aligns the left instructions for each query. In this example, the filtering step reduces the number of instructions from 4,106 to 136 on average. ❺ SQLDEBUG then applies statistical debugging to localize the root cause of the regression. It returns the top three predicates with the highest *Importance* metric. ❻ With the debugging information of the binaries, it maps these predicates back to the source code [38]. We manually inspect these three predicates and confirm that two of them in functions `sqlite3VdbeJumpHere` and `sqlite3VdbeAddOp0` are responsible for the performance regression.

7. EXPERIMENTAL EVALUATION

We implement APOLLO with 3,054 lines of Python code and 156 lines of C++ code. We develop SQLFUZZ based on SQLSmith and Random Query Generator (RQG) [20]. We leverage DynamoRIO [35] to collect execution traces in SQLDEBUG. During the fuzzing, we use the TPC-C benchmark in our fuzzing and corresponding evaluations [15]. In particular, we use the queries as a corpus for bootstrapping SQLFUZZ and execute the queries on the tables contained in the benchmark. We configure the benchmark's scale factor to be one and 50 for fuzzing and validation, respectively.

Our evaluation aims to answer the following questions:

- **Regression Detection:** Is APOLLO effective at finding performance regressions in real-world DBMSs? How effective is SQLFUZZ at removing false positives? (§7.1)
- **Query Reduction:** Can SQLMIN outperform RAGS on reducing discovered queries? How effective are different strategies? (§7.2)
- **Regression Diagnosis:** Can SQLDEBUG localize the root cause of detected performance regressions? (§7.3)

Table 3: APOLLO configuration. Settings used in our evaluation.

	Configuration	Default
Query Generator	`Non_Deterministic`	exclude
	`Max_Query_Size`	< 4000 bytes, < 4 joins
	`Allow_DB_Update`	do not modify DB
	`Timeout`	0.2 second per query
Query Executor	`Threshold`	150%
	`DBMS`	PostgreSQL and SQLite
	`DB_Configuration`	WORK_MEM(128MB), SHARED_MEM(128MB)
	`Execute_Analyze`	run every 1,000 Execs.
	`Timeout`	5 seconds per query
Bug Validator	`Non_Executed`	remove
	`LIMIT_Clause`	include when query

- **Query Patterns:** Can the feedback improve the performance of fuzzing? Are there certain query patterns that are strongly correlated with performance regressions? (§7.4)

Experimental Setup. We evaluate APOLLO on two DBMSs: SQLite (v3.23 and v3.27.2) in the client-server mode, and PostgreSQL (v9.5.0 and v11.1) in the embedded mode. We evaluate APOLLO on a server with Intel(R) Xeon(R) Gold 6140 CPU (32 processors) and 384 GB of RAM. We ran APOLLO on these systems for two months using the configuration shown in Table 3.

Fuzzing Performance. On average, SQLFUZZ effectively produces and executes 5.8 queries per second. Although the query mutation is efficient, about 68% of the generated queries are discarded due to syntax or semantic errors. To improve the performance, we utilize multi-threaded fuzzing. Also, SQLMIN and SQLDEBUG can process one regression-triggering within 30 minutes.

7.1 Performance Regression Detection

APOLLO discovered 10 unique performance regressions from two tested DBMSs. Table 4 summarizes the key characteristics of these regressions. The performance regressions in SQLite lead to a 1.6× to more than 1,000× performance drop, while the regressions in PostgreSQL reduce the performance from 1.9× to more than 1,000×. The first regressions in the table for each DBMS leave the system keeps running for more than one day. We reported these 10 regressions to the corresponding developers with the minimized regression-triggering queries and the diagnosis results. Developers have already confirmed seven regressions and fixed two of them in the latest versions. Next we discuss the details of two performance regressions, one from SQLite and another from PostgreSQL.

```
/* [Fast Version] Scan STOCK -> Search CUSTOMER  */
/* [Slow Version] Scan CUSTOMER -> Search STOCK  */
SELECT COUNT(*)
FROM (SELECT R0.C_ID
  FROM MAIN.CUSTOMER AS R0 LEFT JOIN MAIN.STOCK AS R1
  ON (R0.C_STREET_2 = R1.S_DIST_01)
  WHERE R1.S_DIST_07 IS NOT NULL) AS S0
WHERE EXISTS (SELECT C_ID FROM MAIN.CUSTOMER);
```

Example 6. Performance Drop due to Bug Fix. The latest version of SQLite spends >1,000× more time to execute the query above, compared to the time taken by the older version v3.23.0. Our investigation reveals that this performance regression was inadvertently introduced when the developer attempted to fix a correctness bug. Originally, if the `WHERE` clause satisfies a particular predicate, the DBMS will degenerate the `LEFT JOIN` clause to a faster `JOIN`. However, this optimization has an unforeseen interaction with the `IS NOT NULL` operator, resulting in a correctness bug. In commit `d840e9b` (FEB-05-2019), the developer fixed this correctness bug by skipping the optimization if the `WHERE` clause contains a `IS NOT NULL` operator. After this commit, the query above exhibits a performance drop due to the lack of this optimization. The SQLite

Table 4: Discovered performance regressions. List of regressions uncovered by APOLLO in SQLite and PostgreSQL DBMSs. The `Perf. drop` column refers to the drop in performance across the two versions of the DBMS. The `Query minimization` columns indicate the size of the query before and after the minimization step. The `Commit bisecting` columns indicate the commit associated with a regression and the number of discovered queries that were found to be associated with that commit. The `Statistical Debugging` column presents the predicate reported by the diagnosis tool (file name: function name). We reported all the listed regressions to the developers who have confirmed seven of them and already fixed two of them. † indicates the cases where the commit identified via bisection is different from the actual commit that caused the regressions.

DBMS	Versions	Perf. Drop	Query Minimization			Commit Bisecting		Statistical Debugging	Bug Status
			original	reduced	reduction%	identifier	# of queries		
SQLite	3.23.0	> 1000×	3,875	270	93.0%	`d840e9b`	1,823	`expr.c:impliesNotNullRow`	Confirmed
		24.5×	1,447	706	51.2%	`172f5bd`	1	`where.c:whereLoopAddBtree`	Reported
		51.9×	1,717	626	63.5%	`57eb2ab`	1	`select.c:sqlite3Select`	Confirmed
	3.27.2	2.4×	3,912	548	86.0%	`7d9072b`	17	`expr.c:codeApplyAffinity`	Confirmed
		1.6×	923	406	56.0%	`e130319`	16	`expr.c:sqlite3VdbeJumpHere`, `expr.c:sqlite3VdbeAddOp0`	Confirmed
PostgreSQL	9.5.0	> 1000×	572	130	77.3%	`5edc63b`†	1	`costsize.c:compute_bitmap_pages`	Fixed
		3.2×	767	295	61.5%	`bf6c614`	23	`execGrouping.c:BuildTupleHashTable`	Fixed
	11.1	2.7×	1,619	205	87.3%	`77cd477`†	277	`costsize.c:max_parallel_degree`	Confirmed
		2.0×	531	409	23.0%	`0c2070c`	11	`costsize.c:cost_seqscan`	Reported
		1.9×	659	206	68.7%	`7ca25b7`	98	`selfuncs.c:neqjoinsel`	Reported

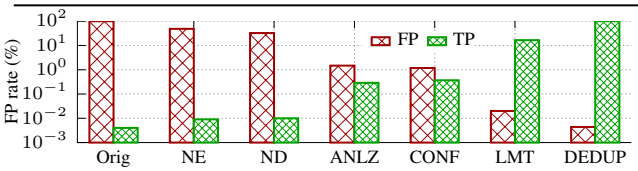


Figure 6: Factor analysis of regression validation checks. SQLFUZZ reduces false positives (FP) and increases true positives (TP). **NE**: discard queries with a non-executed plan; **ND**: discard non-deterministics; **ANLZ**: periodically update statistics; **CONF**: change configuration; **LMT**: disable `LIMIT`; **DEDUP**: deduplicate queries associated with the same problem.

community has acknowledged this problem and is seeking to concurrently solve both correctness and performance regressions [31].

```
/* Same plan but different execution time */
SELECT R0.O_D_ID  FROM PUBLIC.OORDER AS R0
WHERE EXISTS (SELECT COUNT(*)
  FROM (SELECT DISTINCT R0.O_ENTRY_D
    FROM PUBLIC.CUSTOMER AS R1
    WHERE (FALSE)) AS S1);
```

Example 7. Expensive Hash-table Construction. The query above triggers a 3 .2× performance drop in PostgreSQL compared to the time taken on v9.5.0. This regression was introduced by the commit `bf6c614` (FEB-16-2018) for improving the performance with a hashed aggregation executor. The hashed aggregation executor uses a hash table to store a representative tuple and an array of `AggStatePerGroup` structures for each distinct set of column values. To improve the query performance, this commit replaces the `execTuplesMatch` function with a faster function (`ExecQual`) for tuple comparison. To do so, it executes the `ExecBuildGroupingEqual` function to build the tuple hash table every time. Since the construction of the hashtable is computationally expensive, the query execution takes more time compared to the original hashed aggregation executor. This regression was recently fixed in commit `356687b` (FEB-09-2019) by resetting the hashtable if it already exists instead of building a new empty hashtable every time.

False Positive Reduction. We next examine the efficacy of our validation techniques for reducing false positives (§4.2). In particular, we measure the number of false positives and true positives during a 24-hour fuzzing experiment using a factor analysis to understand the impact of each validation technique. We measure the effectiveness of the following techniques: **NE** discards the query if it contains a *Non-Executed Plan*; **ND** avoids utilizing *Non-Deterministic* behaviors to generate queries; **ANLZ** periodically updates the catalog statistics; **CONF** increases memory limits compared to the default DBMS configuration; **LMT** validates the query without using the `LIMIT` clauses; **DEDUP** clusters queries associated with the same problem together using commit bisecting. During the factor analysis, we added these validation techniques one at a time. We use the same random seed in SQLFUZZ to avoid non-determinism during query generation. We pass the regression-triggering queries discovered by SQLFUZZ through SQLDEBUG to identify the unique regressions. Figure 6 summarizes the results of this experiment. NE, ND, and ANLZ reduce the percentage of false positives to 48.49%, 32.57%, and 1.49%, respectively. By combining these six checks, we are able to discard all false positives. We note that the fuzzing speed improves by 1.34× when we add the **ANLZ** check due to better plans, thereby improving the efficacy of SQLFUZZ.

7.2 Query Minimization

For the detected 10 performance regressions, SQLMIN successfully reduced 66.7% of statements from the original queries: the regression-triggering queries of SQLite are reduced by 69.9%, while the queries of PostgreSQL regressions are reduced by 63.5%. This result shows that SQLMIN is effective in reducing the query size of real-world performance regressions.

To further understand the contribution of each minimization policy, we design unit tests and perform evaluation with more examples. First, we collect all regression-triggering queries from SQLFUZZ, which has removed false positives. We randomly choose 30 queries and separate them into three groups according to their size. Then, for each group, we ran SQLMIN with multiple configurations. Starting with one policy (*e.g.*, subquery removal), we gradually add up remaining policies on top of the existing ones. We iterate the process until no further size reduction is possible.

Figure 7 shows our evaluation results. Our algorithms reduce the size of regression-triggering queries by 55% on average, specifically, 50% for small-sized, 45% for medium-sized, and 71% for large-sized queries. In the large-sized query group, subquery removal (SR) shows the best reduction. From the original query, the technique itself has a reduction of 61%. This is because the large-sized queries usually contain a multiple number of subqueries. On the other hand, list and clause removal (LCR) achieves the best reduction for small-sized queries. Since small-sized queries do not contain many subqueries, such a fine-grained approach delivers the best result.

In addition, we compare our result with RAGS, a well-known query-minimization platform [63]. As the source code of RAGS is not available, we implement our version based on the original paper. RAGS tries to remove expressions and two clauses – `WHERE` and `HAVING`) – and does not delete any element in other clauses (*e.g.*, `SELECT`, `GROUP BY`). Further, RAGS does not remove or extract subqueries, and its operations are performed once in a sequential order. The comparison result in Figure 7 shows that SQLMIN outperforms RAGS by 4.6×, where our approach achieves a 55%

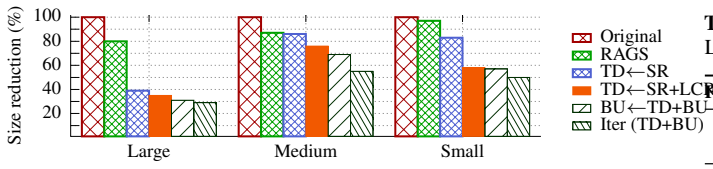


Figure 7: Effectiveness of SQLMIN. The graph shows the query reduction, unified on the original size. We use three query groups based on their size: large (1500,∞), medium (600,1500), small (1,600). RAGS is the previously developed system [63]. **TD** and **BU** indicate the top-down and bottom-up subquery extraction policy. **SR** and **LCR** indicate subquery removal and list and clause removal, respectively. **Iter** runs minimization iteratively.

reduction, while RAGS merely reduces queries by 12%. Especially, RAGS is not able to reduce any query from the small-sized group, where most of the reduction can be achieved only through column list removal in the SELECT clause.

7.3 Regression Diagnosis

We next examine the efficacy of SQLDEBUG on localizing the root cause of the discovered performance regressions.

Efficacy of Commit Bisecting. Among 1,621 commits between two evaluated versions of SQLite, our bisecting method is able to identify all five regression-inducing commits. From 6,880 commits between two evaluated versions of PostgreSQL, the bisecting successfully identifies three commits, and fails to locate another two commits. We defer a detailed analysis of the failed cases to §7.3.1. In the first time we report our bisecting results to the DBMS developers, we find that the commits we identify are off by a few commits (*e.g.*, ±5 commits) from the ones detected by the developers. After we integrated their built-in command (*e.g.*, SQLite `fossil bisect`) in SQLDEBUG, we obtained the same commits as the developers.

In addition to identifying the regression-inducing commits, bisecting can also cluster queries associated with the same regression in one group. With the bisecting, we are able to remove 225 duplicated queries per regression on average (370 queries in SQLite and 81 queries in PostgreSQL). For example, in SQLite, we found 1,823 queries that triggered the same performance regression, introduced in commit `d840e9b` (FEB-15-2019). Our bisecting technique can be used as an efficient clustering tool to avoid duplicate regression reports from the same regression, thereby saving valuable developer time. We note that the bisecting-based clustering may accidentally merge two or more different regressions as one, if they are introduced in the same commit. We discuss this limitation in §8.

Efficacy of Statistical Debugging. Using dynamic binary instrumentation, SQLDEBUG keeps track of 3,534 and 3,442 predicates in the latest and the older versions of the DBMS on average. It then selects a subset of these predicates based on the bisected commit. These reduced traces contain 71 and 47 predicates from the latest and the older versions of the DBMS on average. Using these reduced traces, statistical debugger ranks the predicates based on their importance metric. It then returns a ranked list of 12 predicates on average. The final report contains the predicate address (*i.e.*, the address of the conditional branch), the function name, the line number associated with that address, and its rank (*i.e.*, how closely it is related to the regression). To validate the efficacy of statistical debugging, we examine the reported locations and patch them back to the original code. If the patch has other dependencies, we patch all the necessary code snippets to avoid any compilation errors. Although the final report has several predicates (*e.g.*, 20), we only need to patch up to three predicates on average. After patching, we run the relevant queries and verify if patching restores their performance. We found that statistical debugging correctly identifies the problematic locations for eight out of ten regressions.

Table 5: Profiler-based diagnosis. The table shows diagnosis result from Linux Perf and Intel VTune on Example 8 and 9.

Profiler	Method	Diagnosis Result
Perf	Branch record	Recorded branches do not contain root cause location
	Tracepoint probe	Difficult to insert tracepoint for unknown root cause
VTune	Hot-spot analysis	Result does not contain actual root cause location
	Call-stack diffing	① Ex8: Does not capture root cause location
		② Ex9: Identifies parallel exec. and cost estimation

7.3.1 *Analysis of Diagnosis Failures*

We find two reasons that make SQLDEBUG fail to diagnose a performance regression: a query triggers multiple regressions in the DBMS, rendering our diagnosis incomplete; the performance regression is enabled by another recently introduced benign feature.

```
/* [Fast Version] Bitmap Heap Scan on STOCK R0  */
/* [Slow Version] Seq Scan on STOCK R0          */
SELECT R0.S_DIST_06 FROM PUBLIC.STOCK AS R0
WHERE  (R0.S_W_ID < CAST(LEAST(0, 1) AS INT8));
```

Example 8. Existence of Two Problems. We found that the query above triggers two problems in the latest version of PostgreSQL. During the commit bisecting, SQLDEBUG identifies that the problematic commit `5edc63b` (NOV-10-2017) changes the policy for launching bitmap heap scan (BHS), and leads to the performance regression. Specifically, the newer version uses BHS only if the DBMS has enough space in the working memory. Although this query does not return any rows, both versions over-estimate the number of returned rows. However, the older version uses BHS, while the newer version uses sequential scan (SS) due to the policy change. Given this property, BHS immediately skips the predicate evaluation, while SS evaluates the predicate on all the tuples in the table. Thus, the newer version is more than 1,000× slower than the older version. We attribute the misestimation problem in the optimizer to another regression. Although `LEAST(0,1)` reduces to 0, the optimizer does contain relevant constant-folding logic, thereby misestimating the number of returned tuples. When we reported this regression, PostgreSQL developers confirmed that this regression has been patched in the latest release.

```
/* [Fast Version] SEQ SCAN ON ORDER_LINE R0          */
/* [Slow Version] Gather -> Workers Plan/Launch: 2    */
SELECT R0.OL_DELIVERY_D FROM PUBLIC.ORDER_LINE AS R0
WHERE  EXISTS (
  SELECT R1.I_IM_ID FROM PUBLIC.ITEM AS R1
  WHERE  R0.OL_D_ID <=  R1.I_IM_ID);
```

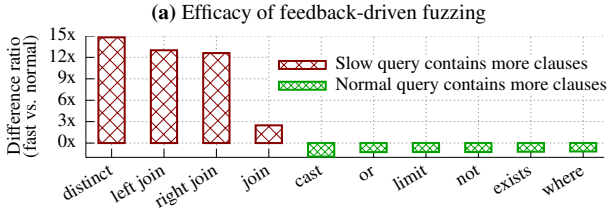
Example 9. Cascaded Performance Drop. The query above illustrates the cascading flow of control from the root cause. SQLDEBUG identifies that the problematic commit `77cd477` (APR-26-2016) enables the parallel query execution by default. However, for this query, the parallel scan is slower than the sequential scan, and the newer version is slower than the old version. Although the parallel execution seems to be the root case, the crux of this problem lies in misestimation. Specifically, the newer version over-estimates the query execution time and resorts to an expensive parallel scan operator. SQLDEBUG currently cannot handle this type of regressions.

7.3.2 *Complement with Profiler-based Diagnosis*

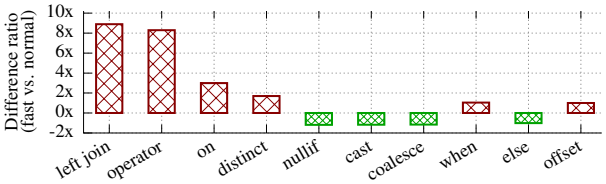
Software profiling techniques help developers understand the distribution of computing resources during the program execution. Such information can be used by a profile-based diagnosis system or performance tuning system to diagnose performance regression bugs. To understand whether the profiler-based analysis can help regression-diagnosis or not, we tried profilers Linux Perf [36] and Intel VTune [61] on Example 8 and Example 9, where SQLDEBUG failed to diagnose the regression correctly. Table 5 shows the result. First, regression diagnosis using Linux Perf failed to identify any

	Feedback-driven	Random Testing
Regression queries	441 / 347,616 (0.126%)	243 / 340,416 (0.071%)
Regression rate (Avg.)	10.8×	10.2×

(a) Efficacy of feedback-driven fuzzing



(b) Impact of query clauses in PostgreSQL



(c) Impact of query clauses in SQLite

Figure 8: Importance of feedback and query clauses. Table (a) shows impact of feedback-driven fuzzing. Figure (b) and (c) show importance of query clauses. Red and green-colored bars indicate that regression-triggering queries contain the associated clauses more and less frequently compared to normal queries, respectively.

root cause. The reason is that the recorded branch predicates missed several branches where the actual root cause stays. We also find it is challenging to insert a tracepoint to record the call-stack as we do not know the possible root cause; thus a tracepoint is not suitable for regression diagnosis. Second, diagnosis with Intel VTune successfully identified the root cause of Example 9, but failed on Example 8. When we extracted different call-stacks from the good and bad profiles, we noticed that the bad query is executed when the parallel execution was enabled by default (in `execscan.c`) and the cost estimator (in `costsize.c`) showed the difference. This result shows that profiler-based approaches may help diagnose the root cause of some regressions. However, they do not guarantee the completeness in debugging because the sampling process may miss important predicates or functions during the recording.

7.4 Analysis of Regression-inducing Queries

We next study the efficacy of the feedback and the characteristics of regression-inducing queries along three dimensions: the effect of the feedback on fuzzing, the importance of different query clauses and the importance of the query size.

Efficacy of Fuzzing Feedback. To demonstrate the potential usefulness of feedback (*i.e.*, updating the probability table for each clause) on the regression detection, we run fuzzing with and without the feedback for 24 hours on PostgreSQL. Since the fuzzing stage does not consider false positive filtering for `LIMIT` clause and `DEDUP`, we collect all queries if any query show regression than the threshold (*i.e.*, 150%) without the two validations. Overall, feedback-driven fuzzing shows better regression-detecting performance than random testing. Feedback improves the detection rate by 76%, specifically from 0.071% (*i.e.*, without feedback) to 0.125% (*i.e.*, with feedback). Also, we further investigate the regression ratio between two fuzzing setups. We found that the feedback is not relevant to enlarge the regression ratio of discovered queries as shown in Figure 8a.

Importance of Query Clauses. We construct a dataset with 1,000 normal queries (*i.e.*, no performance regression) and 2,268 regression-triggering queries for PostgreSQL and SQLite. We count the frequency of each clause in these queries and normalize the count by their size. Figure 8 illustrates the distribution of clauses across normal and regression-triggering queries. The most notable observation is that the `JOIN` clause is particularly effective at uncovering regressions in both DBMSs. In particular, the `LEFT` clause appears 13.0× more often (PostgreSQL) and 8.9× (SQLite) more often in regression-triggering queries than that in normal queries. The `DISTINCT` clause appears 14.8× more often in PostgreSQL queries and operators (*e.g.*, <>, +, and %) appear 8.3× more often in SQLite queries. In contrast, certain clauses are less frequently present in regression-triggering queries. For example, `CAST`, `OR`, or `LIMIT` are 1.3× to 1.9× less frequently present in PostgreSQL queries. These results show that certain types of clauses are more capable of uncovering performance regressions. Based on these observations, SQLFuzz dynamically increases the probabilities associated with these clauses in the probability table.

Unimportance of Query Size. We found that the query size is not relevant to uncover performance regressions. In our experiment, normal queries are 3% larger and 9% smaller than regression-triggering queries in PostgreSQL and SQLite, respectively.

8. LIMITATIONS AND FUTURE WORK

We now discuss the limitations of Apollo and present our ideas that may address the problems in the future work.

Coping with Multiple Problems in a Commit. SQLDebug currently cannot distinguish multiple performance regressions introduced in the same commit. Specifically, if two queries are bisected to the same commit, SQLDebug assumes they trigger the same regression and drop one to avoid duplicated reports. For example, we cluster together 1,823 duplicated queries related to the same commit `d840e9b` in SQLite. However, this clustering technique may lead to false negatives as we discuss in §7.3.1. We can solve this problem by passing all discovered queries to statistical debugging, without bisecting. However, this requires developers to do more subsequent analyses of the reports generated by the statistical debugger.

Alternate Statistical Debugging Models. We plan to investigate the efficacy of alternate statistical debugging models in the future. Besides the current boolean-value predicate (*i.e.*, a predicate is true or false), we will investigate integer-valued predicates (*e.g.*, the times a predicate is true in each run). Prior research has shown that Latent Dirichlet Allocation (LDA) can support integer-valued predicates [32]. Consider a predicate P that is true 10 times and false 20 times in a successful run, and true 20 times and false 10 times in a failed run. The traditional statistical debugging model will not prioritize P, as P is both true and false in both runs. In contrast, LDA will rank P higher since it takes into consideration that P is true more often during failures than during successful runs.

Augmenting the Impact of Performance Regression. Some regression-triggering queries demonstrate a significant performance drop, but suffer from a short execution time. For example, consider a query that completes in 0.1 ms and 10 ms in the older version and the latest versions of a DBMS, respectively. We currently ignore these queries since their execution time is short and DBMS developers do not take them seriously. However, we contend that such queries should not be ignored. First, if such queries are used as subqueries in larger queries, the execution time of the larger query may exhibit a similar performance drop (*e.g.*, $\langle$0.1 ms, 10 ms$\rangle \rightarrow \langle$10 s, 1,000 s $\rangle$). Second, the size of the tested database often determines the execution time. The same query can exhibit a longer execution time on a larger database. Third, DBMS may consist of massive number of short-time executions. For example, On-line Transaction Processing (OLTP) involves many short online transactions such as

UPDATE, INSERT, and DELETE. In this scenario, short-time difference may cause significant performance regressions.

Supporting Other DBMSs. We can easily extend APOLLO to support other DBMSs. Taking MySQL as an example: First, SQLFUZZ uses RQG [20] for query generation since it is geared towards MySQL. We add a DBMS-specific configuration file (*e.g.*, port number, database name, command for updating statistics). Second, our general-purpose SQLMIN can reduces the discovered query without any additional modification. Lastly, SQLDEBUG can perform commit bisecting on MySQL code repository and then executes statistical debugging on collected execution traces.

Code Coverage as a Feedback Mechanism. Code coverage is a frequently used feedback mechanism in fuzzing engines [22, 27]. We found that this metric is not particularly useful for fuzzing DBMSs, since the core components of DBMS (*e.g.*, query optimizer) already have high coverage (*e.g.*, > 95%) after running tens of queries. Thus, we instead use the clause-occurrence frequency as a feedback mechanism in SQLFUZZ, as discussed in §7.4.

Fuzzing on Fixed Dataset. The current design of APOLLO uses a fixed database to fuzz the target DBMS, which may constrain the fuzzing coverage due to the repetitious schema. Specifically, a fixed database lacks in variety of relationships between tables, defined indexes and triggers, data types for each column, and number of rows in table. If many performance regressions attribute to one of the varieties, APOLLO will unlike unearth these regressions regardless of the advance in query generation and fuzzing strategy. We plan to include random dataset generation in the future work, which will define arbitrary DB schema and insert random data automatically.

User Privacy in Regression Report. Current version of APOLLO does not handle the problem of sharing the user dataset and corresponding regression query. However, we consider this is an important problem because users may not be able to export internal dataset due to their confidentiality constraints. We believe *differential privacy* [46, 37] is a promising solution for addressing this issue, as it allows general data analytics of data (*i.e.*, same performance regression) while providing a strong guarantee of privacy. We plan to investigate the efficacy of differential privacy in the future.

Performance Regression from Statistics Collection Logic. We consider the identical statistics important for reducing the false positives; thus we periodically update the statistics during the fuzzing. Therefore, if the performance regression is caused by the logic in statistics collection, APOLLO is not able to detect the problem in the current setting. If we disable the periodical update and record sequence of queries that can affect to the internal statistics, we will be able to figure out problem in statistics collection logic.

9. RELATED WORK

The need for tools that can accelerate the testing and debugging of large-scale software systems has been well-known for decades [43, 57, 62]. As such, there is an extensive corpus on the problems of detecting and diagnosing bugs in software systems. In this section, we discuss methods for testing and debugging DBMSs with a special focus on performance regressions.

Detecting Functional Bugs. Although grammar-based testing has a long history in compiler validation, it has not been extensively studied by the DBMS community. RAGS was a system built by the Microsoft SQL Server group to explore automated testing for functional bugs in DBMSs [63]. It generates SQL statements by stochastically constructing a parse tree based on the database schema and printing it out. When a statement generated by RAGS causes an error, the debugging process is often difficult if the statement is long and complex. The paper describes a technique for simplifying the offending statement by taking out as many elements of the statement as possible while preserving the original error message. This top-down greedy approach for simplifying statements can get stuck at a local minimum. The reduced statement may require further manual reduction before it can be submitted in a bug report.

Detecting and Diagnosing Performance Bugs. BmPad [60], Snowtrail [69], and Oracle SQL Performance Analyzer [68] defined multiple test suites (*i.e.*, workloads) for testing DBMS performance. To monitor any performance anomaly, testers monitors if the executed result exceeds performance barrier (*i.e.*, baseline). FlexMin builds minimization techniques on SQL query to isolate bug by applying delta debugging and clause simplification. APOLLO differs from these approaches by adopting multiple versions of DBMS without pre-determined performance baseline and by applying the query minimization for performance regression domain.

Fuzzing DBMS. There were several fuzzing projects to discover inputs that can cause a system crash on DBMSs [25, 24, 66, 23, 41]. They showed effectiveness by discovering the vulnerabilities of DBMS and releasing them as a form of CVE (Common Vulnerabilities and Exposures). However, APOLLO shows contrast with these approaches. Unlike the traditional fuzzing projects, which attempted to identify vulnerability and designed to find problem mainly on parser, APOLLO is specially designed to discover performance regressions in planner, optimizer, and executor.

Genetic Algorithms & Query Morphing. Another line of research for testing DBMSs focuses on genetic algorithms. SQLSmith generates arbitrary SQL statements based on the database schema and has been used to find functional bugs [26]. Also, an automated and feedback-driven query generator have developed to support targeted test requirement in DB2 database [52]. More recently, researchers have developed a bug-detection tool, called SQLScalpel, that eschews randomized testing and genetic algorithms in favor of a targeted search based on stepwise query morphing [47]. However, this guided search technique limits the tool's expressiveness and thereby reduces coverage.

APOLLO differs from prior work in that it is the first to explore the problems of automatically diagnosing performance regressions in DBMSs using domain-specific input mutation, feedback-driven fuzzing, and statistical debugging techniques.

10. CONCLUSION

We presented APOLLO, a toolchain for automatically detecting and diagnosing performance regressions in DBMSs. APOLLO leverages domain-specific fuzzing to detect performance regressions. It then uses a hybrid minimization algorithm to reduce queries. Finally, it identifies the root cause of regressions using commit bisecting and statistical debugging techniques. APOLLO discovered 10 previously unknown performance regressions from SQLite and PostgreSQL. It reduced query size by 66.7% and can effectively eliminate false positives. By automating the detection and diagnosis of performance regressions, APOLLO reduces the labor cost of developing DBMSs.

11. ACKNOWLEDGMENT

We thank the anonymous reviewers and development teams of PostgreSQL and SQLite for their helpful feedback. This research was supported, in part, by the NSF awards CNS-1563848, CNS-1704701, CRI-1629851, CNS-1749711, IIS-1850342 and IIS-1908984, ONR under grants N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA TC (No. DARPA FA8650-15-C-7556), and ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla, Intel, VMware, and Google.

12. REFERENCES

[1] PostgreSQL. https://www.postgresql.org/.

[2] PostgreSQL Bug Reporting Guidelines. https://www.postgresql.org/list/pgsql-bugs/.

[3] PostgreSQL Performance Regression Reports. https://www.postgresql.org/search/?m=1&q=performance+regression&l=8&d=-1&s=r.

[4] PostgreSQL Roadmap. https://wiki.postgresql.org/wiki/Todo.

[5] PostgreSQL Testing. https://www.postgresql.org/developer/testing/.

[6] SQLite. https://www.sqlite.org/index.html.

[7] SQLite Bug Reporting Guidelines. https://www.chiark.greenend.org.uk/~sgtatham/bugs.html.

[8] SQLite Performance Regression Reports. https://www.sqlite.org/src/rptview?rn=1.

[9] SQLite Roadmap. https://sqlite.org/src4/doc/trunk/www/design.wiki.

[10] SQLite Testing. https://sqlite.org/testing.html.

[11] Database Language SQL. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, 1992.

[12] How to Report Bugs Effectively. https://www.chiark.greenend.org.uk/~sgtatham/bugs.html, 1999.

[13] Git-scm. https://git-scm.com, 2005.

[14] Fossil-scm. https://fossil-scm.org, 2006.

[15] TPC-C Benchmark. http://www.tpc.org/tpcc/spec/tpcc_current.pdf, 2007.

[16] Fighting Regressions with Git Bisect. http://www.linux-kongress.org/2009/abstracts.html#3_7_1, 2008.

[17] Fossil-bisect. https://www.fossil-scm.org/index.html/help/bisect, 2008.

[18] Git-bisect. https://git-scm.com/docs/git-bisect, 2008.

[19] A Data-Driven Glimpse into the Burgeoning New Field. http://emc.com/collateral/about/news/emc-data-science-study-wp.pdf, 2011.

[20] RQG: Random Query Generator. https://launchpad.net/randgen, 2012.

[21] 100x Faster Postgres Performance by Changing 1 Line. https://www.datadoghq.com/blog/100x-faster-postgres-performance-by-changing-1-line/, 2013.

[22] AFL: American Fuzzy Lop, 2015. http://lcamtuf.coredump.cx/afl/.

[23] PQuery: Multithreaded SQL Tester / Reducer. https://github.com/Percona-QA/pquery, 2015.

[24] Finding Bugs in SQLite, the Easy Way, 2016. https://lcamtuf.blogspot.com/2015/04/finding-bugs-in-sqlite-easy-way.html.

[25] OSS-Fuzz: Continuous Fuzzing for Open Source Software, 2016. https://github.com/google/oss-fuzz.

[26] SQLSmith. https://github.com/anse1/sqlsmith, 2016.

[27] libFuzzer. http://llvm.org/docs/LibFuzzer.html, 2018.

[28] PostgreSQL LIMIT and OFFSET. https://www.postgresql.org/docs/11/queries-limit.html, 2018.

[29] PostgreSQL Single-User Mode. https://www.postgresql.org/docs/11/app-postgres.html, 2018.

[30] PostgreSQL Table Partitioning. https://www.postgresql.org/docs/current/ddl-partitioning.html, 2019.

[31] SQLite performance bug response. http://mailinglists.sqlite.org/cgi-bin/mailman/private/sqlite-users/2019-April/083864.html, 2019.

[32] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical Debugging using Latent Topic Models. In *ECML*, pages 6–17. Springer, 2007.

[33] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical Debugging using Compound Boolean Predicates. In *ISSTA*, pages 5–15. ACM, 2007.

[34] S. Bratus, A. Hansen, and A. Shubina. LZfuzz: A Fast Compression-based Fuzzer for Poorly Documented Protocols. *Darmouth College, Hanover, NH, Tech. Rep. TR-2008*, 634, 2008.

[35] D. Bruening and S. Amarasinghe. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[36] A. C. De Melo. The new linux 'perf' tools. In *Linux Kongress*, 2010.

[37] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[38] M. J. Eager. Introduction to the DWARF Debugging Format, 2012.

[39] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[40] G. Fraser and A. Arcuri. Evosuite: Automatic Test Suite Generation for Object-oriented Software. In *ESEC/FSE*, pages 416–419. ACM, 2011.

[41] R. Garcia. Case study: Experiences on sql language fuzz testing. In *DBTEST*. ACM, 2009.

[42] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[43] K. V. Hanford. Automatic Generation of Test Cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[44] C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. In *SECURITY*, pages 445–458, 2012.

[45] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. *SIGPLAN Notices*, 47(6):77–88, 2012.

[46] N. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for sql queries. *PVLDB*, 11(5):526–539, 2018.

[47] M. L. Kersten, P. Koutsourakis, and Y. Zhang. Finding the Pitfalls in Query Performance. In *DBTEST*, page 3. ACM, 2018.

[48] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *SIGSAC CCS*, pages 2123–2138. ACM, 2018.

[49] R. Lämmel and W. Schulte. Controllable Combinatorial Coverage in Grammar-based Testing. In *IFIP ICTCS*, pages 19–38. Springer, 2006.

[50] D. Laney. 3-D Data Management: Controlling Data Volume, Velocity and Variety, Feb. 2001.

[51] C. Lemieux, R. Padhye, K. Sen, and D. Song. PerfFuzz: Automatically Generating Pathological Inputs. In *SIGSOFT ISSTA*, pages 254–265. ACM, 2018.

[52] D. Letarte, F. Gauthier, E. Merlo, N. Sutyanyong, and C. Zuzarte. Targeted genetic test sql generation for the db2 database. In *DBTEST*. ACM, 2012.

[53] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation via Remote Program Sampling. In *Sigplan Notices*, volume 38, pages 141–154. ACM, 2003.

[54] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. *Sigplan Notices*, 40(6):15–26, 2005.

[55] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[56] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Sigmod Record*, volume 21, pages 39–48. ACM, 1992.

[57] P. Purdom. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[58] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[59] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case Reduction for C Compiler Bugs. In *SIGPLAN Notices*, volume 47, pages 335–346. ACM, 2012.

[60] K.-T. Rehmann, C. Seo, D. Hwang, B. T. Truong, A. Boehm, and D. H. Lee. Performance monitoring in sap hana's continuous integration process. *ACM SIGMETRICS Performance Evaluation Review*, pages 43–52, 2016.

[61] J. Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.

[62] R. L. Sauder. A General Test Data Generator for COBOL. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 317–323. ACM, 1962.

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

[64] L. Song and S. Lu. Statistical Debugging for Real-world Performance Problems. In *SIGPLAN Notices*, volume 49, pages 561–578. ACM, 2014.

[65] M. Stonebraker, S. Madden, and P. Dubey. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *SIGMOD Rec.*, 42(1):44–49, May 2013.

[66] J. Wang, P. Zhang, L. Zhang, H. Zhu, and X. Ye. A Model-based Fuzzing Approach for DBMS. In *CHINACOM*, pages 426–431. IEEE, 2013.

[67] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[68] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's sql performance analyzer. *IEEE Data Eng. Bull.*, pages 51–58, 2008.

[69] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee. Snowtrail: Testing with production queries on a cloud database. In *DBTEST*. ACM, 2018.

[70] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[71] K. Z. Zamli, M. F. Klaib, M. I. Younis, N. A. M. Isa, and R. Abdullah. Design and Implementation of a T-way Test Data Generation Strategy with Automated Execution Tool Support. *Information Sciences*, 181(9):1741–1758, 2011.