Georgia Tech

DEBUGGING YOUR DATABASE SYSTEM USING APOLLO

JOY ARULRAJ
GEORGIA TECH

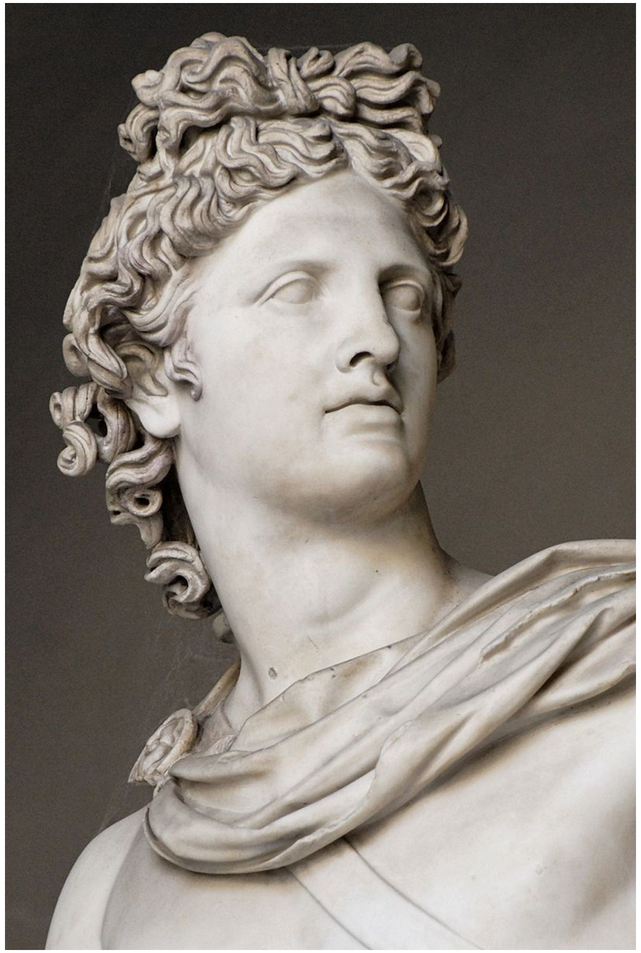
CREATING THE NEXT®

APOLLO

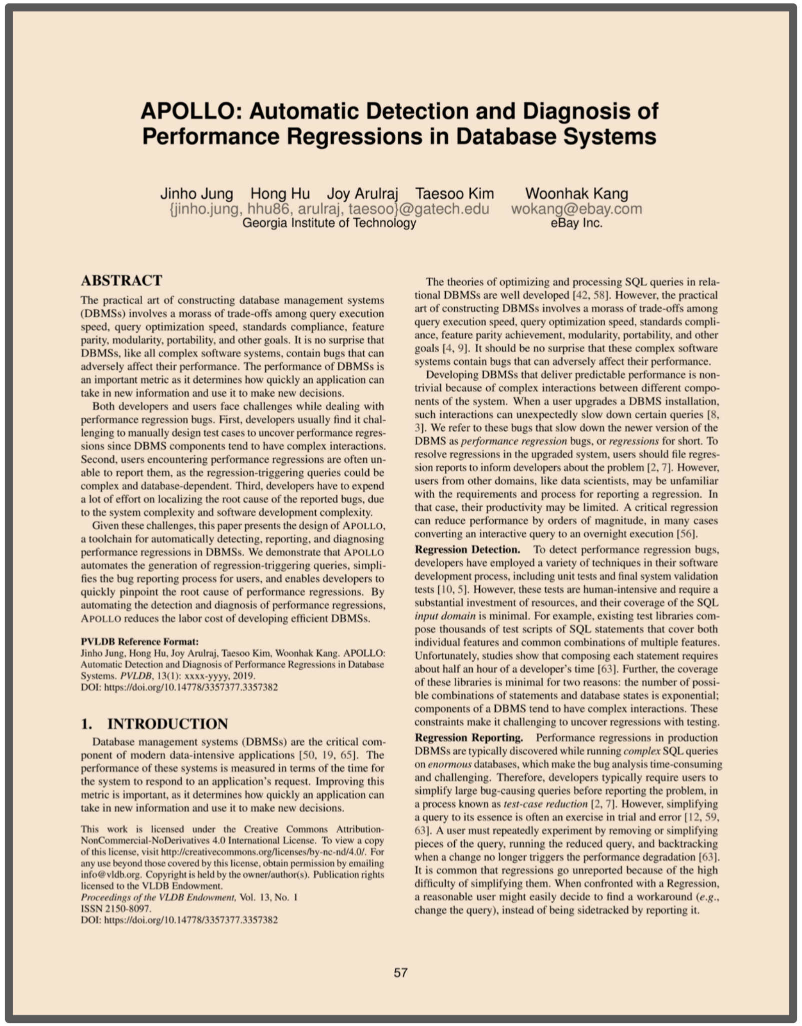
- Holistic toolchain for debugging database systems
  - Inspired by Jepsen

1 AUTOMATICALLY FIND SQL QUERIES EXHIBITING PERFORMANCE REGRESSIONS

2 AUTOMATICALLY DIAGNOSE THE ROOT CAUSE OF PERFORMANCE REGRESSIONS



APOLLO (VLDB 2020)



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): xxxx-yyyy, 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.

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.

57

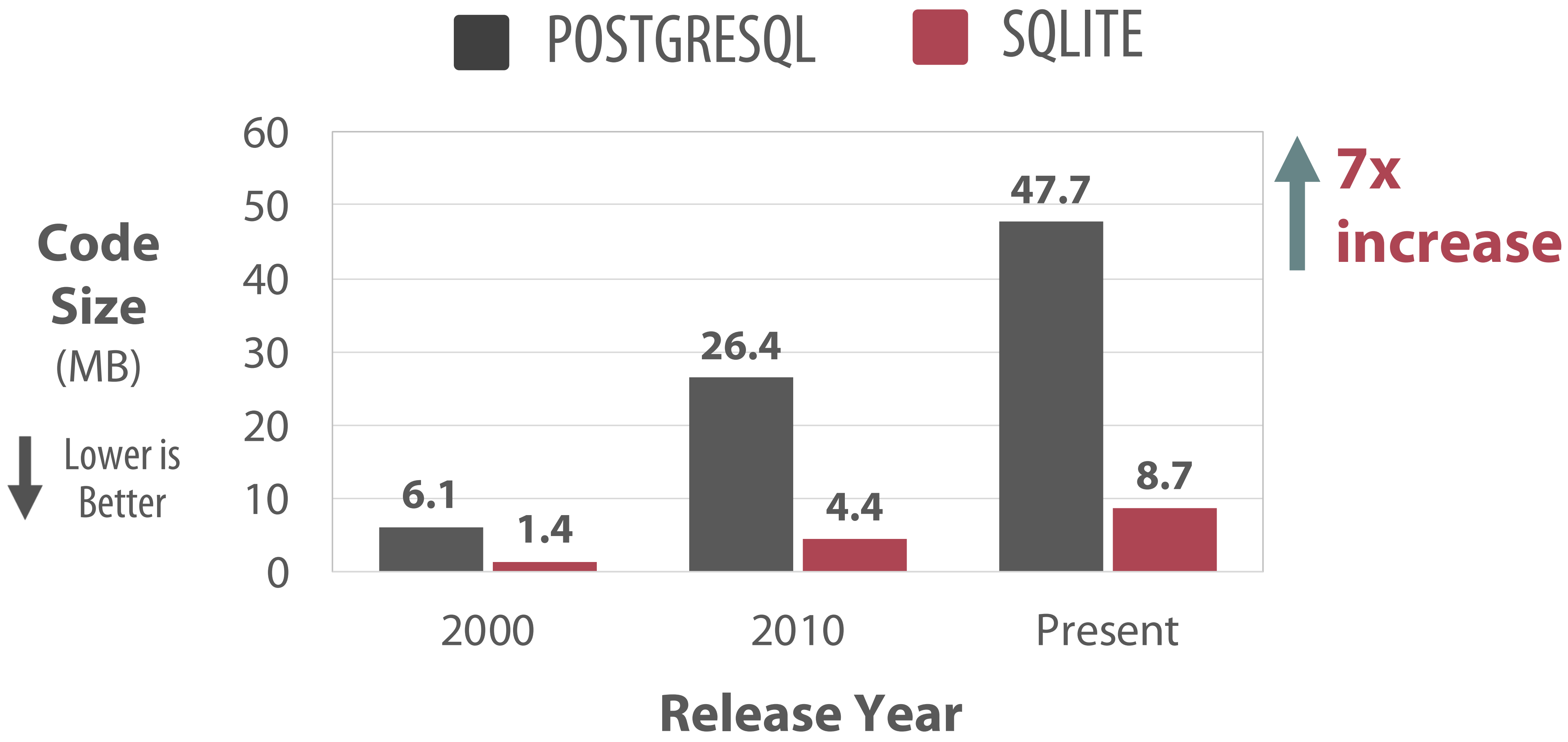
JINHO JUNG

TAESOO KIM

HONG HU

WOONHAK KANG

MOTIVATION: DBMS COMPLEXITY



MOTIVATION: PERFORMANCE REGRESSIONS

- Challenging to build systems with predictable performance
  - Due to complex interactions between different components
- Scenario: User upgrades a DBMS installation
  - Query suddenly takes ten times longer to execute
  - Due to unexpected interactions between different components
  - Refer to this behavior as a <u>performance regression</u>
- Performance regressions can hurt user productivity
  - Can easily covert an interactive query to an overnight one

MOTIVATION: PERFORMANCE REGRESSIONS

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

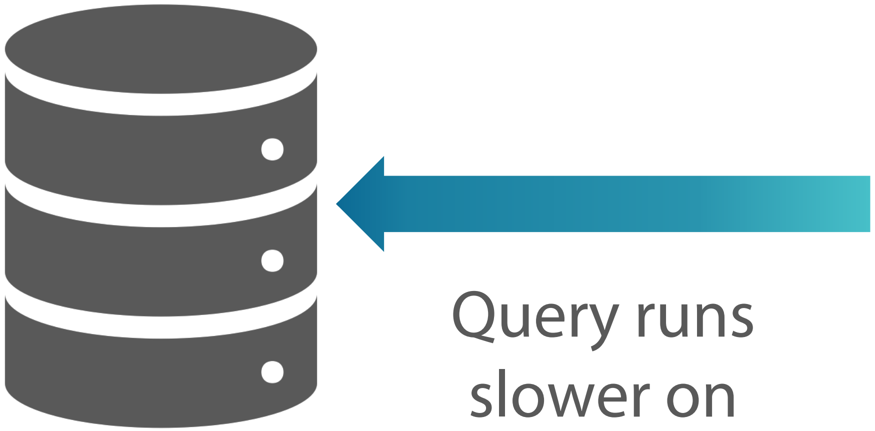
>10,000x slowdown

LATEST VERSION OF POSTGRESQL

- Due to a recent optimizer update
  - New policy for choosing the scan algorithm
  - Resulted in over-estimating the number of rows in the table
  - Earlier version: Fast bitmap scan
  - Latest version: Slow sequential scan

MOTIVATION: DETECTING REGRESSIONS

1 HOW TO DISCOVER QUERIES EXHIBITING REGRESSIONS?

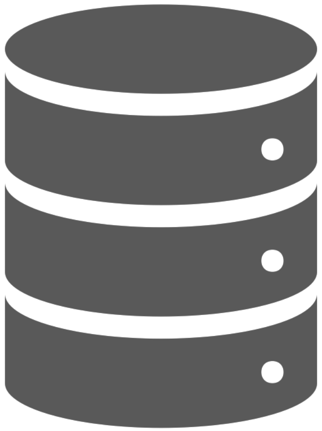


Query runs slower on latest version

```sql
SELECT NO FROM ORDER AS R0
WHERE EXISTS (
  SELECT CNT FROM SALES AS R1
  WHERE EXISTS (
    SELECT ID FROM HISTORY AS R2
    WHERE (R0.INFO IS NOT NULL));
```

MOTIVATION: REPORTING REGRESSIONS

2 HOW TO SIMPLIFY QUERIES FOR REPORTING REGRESSIONS?



Query runs slower on latest version

```
SELECT NO FROM ORDER AS R0
WHERE EXISTS (
 SELECT CNT FROM SALES AS R1
 WHERE EXISTS (
  SELECT ID FROM HISTORY AS R2
  WHERE (R0.INFO IS NOT NULL));
```

MOTIVATION: DIAGNOSING REGRESSIONS

3 HOW TO DIAGNOSE THE ROOT CAUSE OF THE REGRESSION?

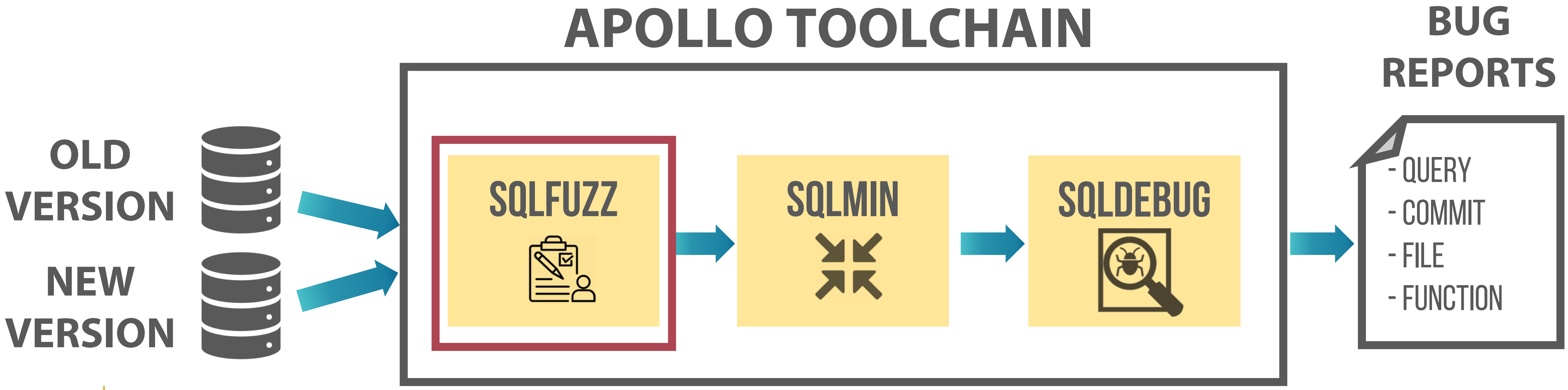
Query runs slower on latest version

```
SELECT NO FROM ORDER AS R0
WHERE EXISTS (
  SELECT CNT FROM SALES AS R1
  WHERE EXISTS (
    SELECT ID FROM HISTORY AS R2
    WHERE (R0.INFO IS NOT NULL));
```

APOLLO TOOLCHAIN

1 HOW TO DISCOVER QUERIES EXHIBITING REGRESSIONS?

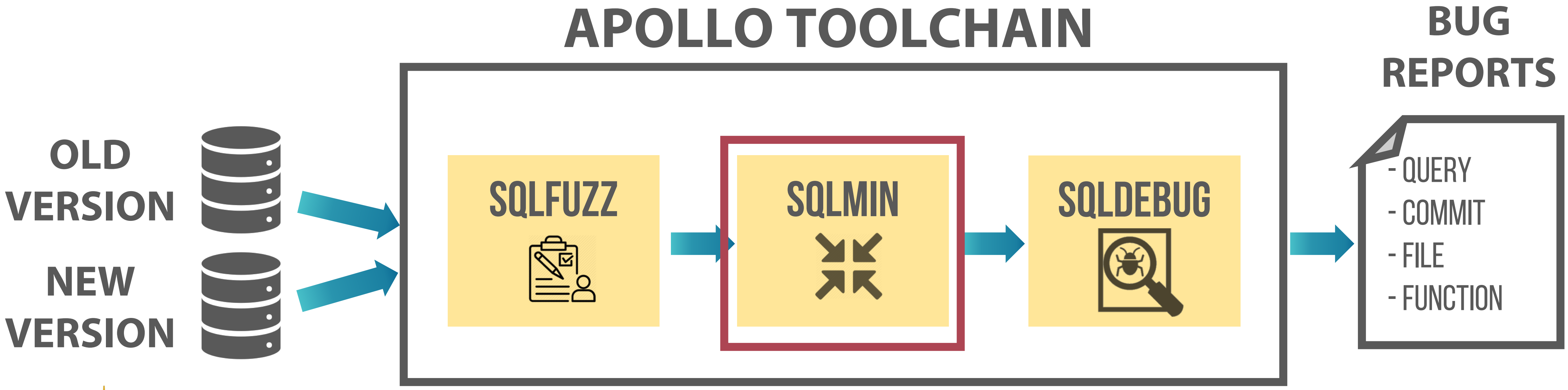
SQLFUZZ: FEEDBACK-DRIVEN FUZZING



APOLLO TOOLCHAIN

2 HOW TO SIMPLIFY QUERIES FOR REPORTING REGRESSIONS?

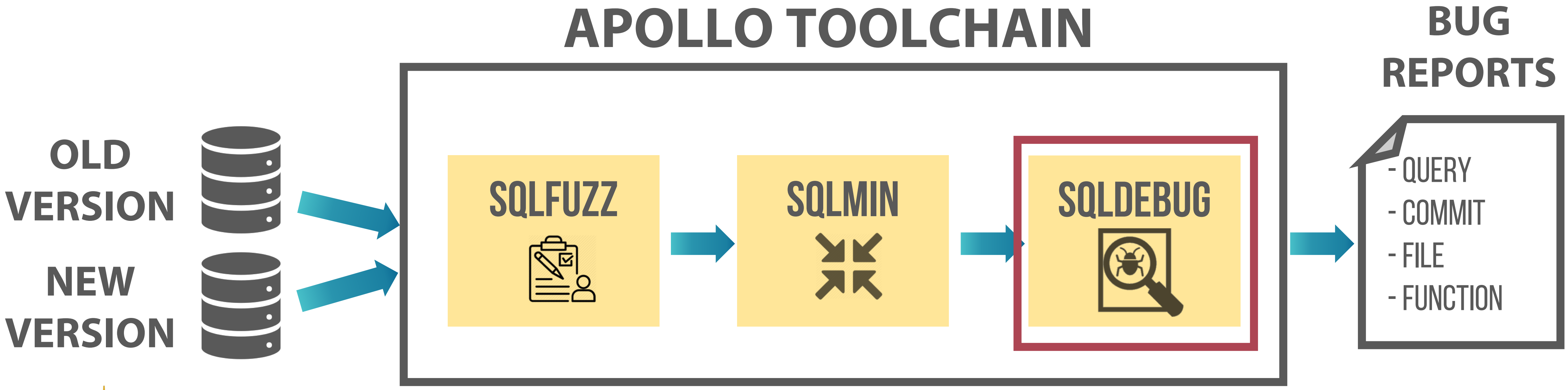
SQLMIN: BI-DIRECTIONAL QUERY REDUCTION ALGORITHMS



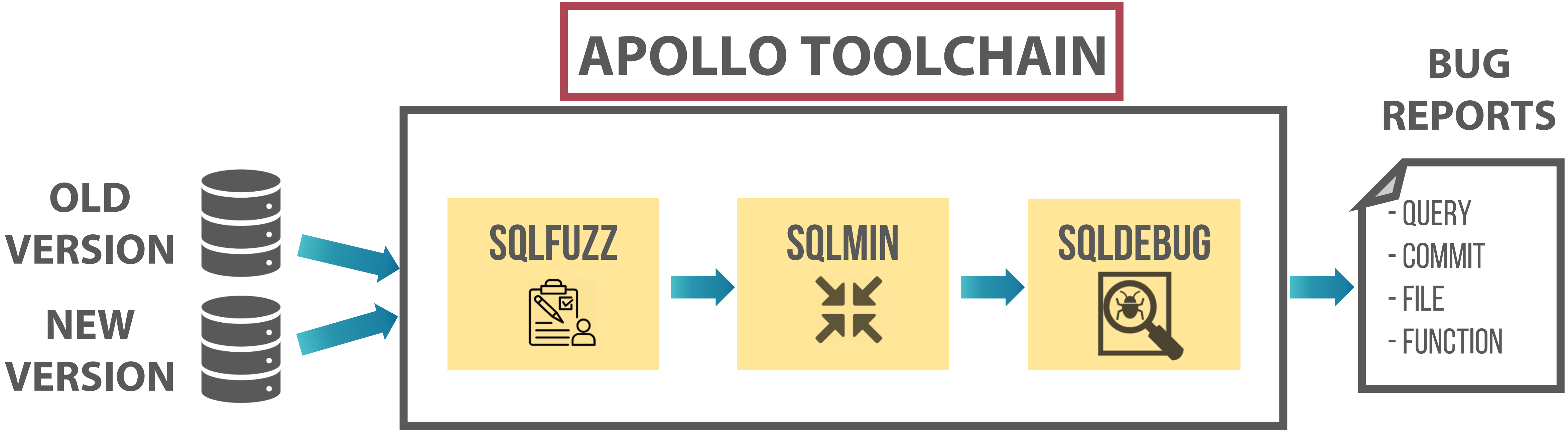
APOLLO TOOLCHAIN

3 HOW TO DIAGNOSE THE ROOT CAUSE OF THE REGRESSION?

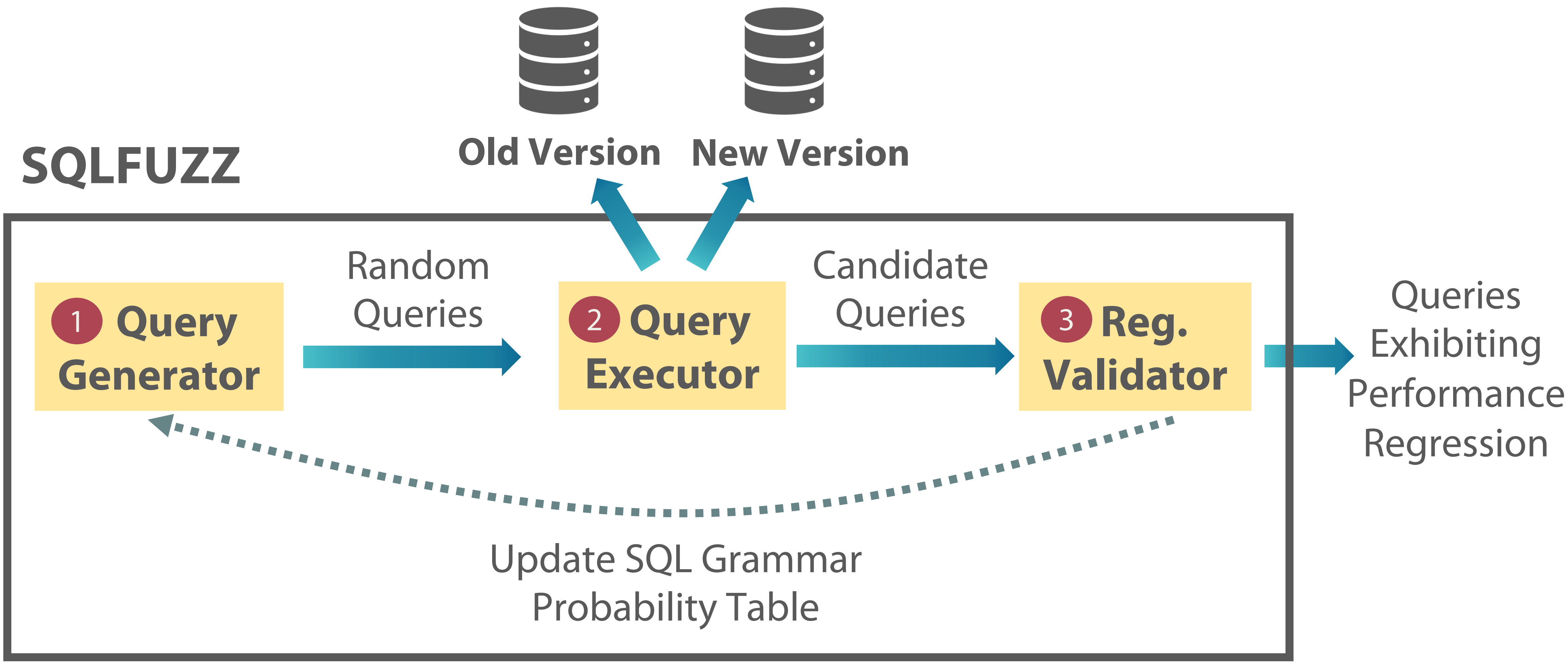
SQLDEBUG: STATISTICAL DEBUGGING + COMMIT BISECTION



TALK OVERVIEW

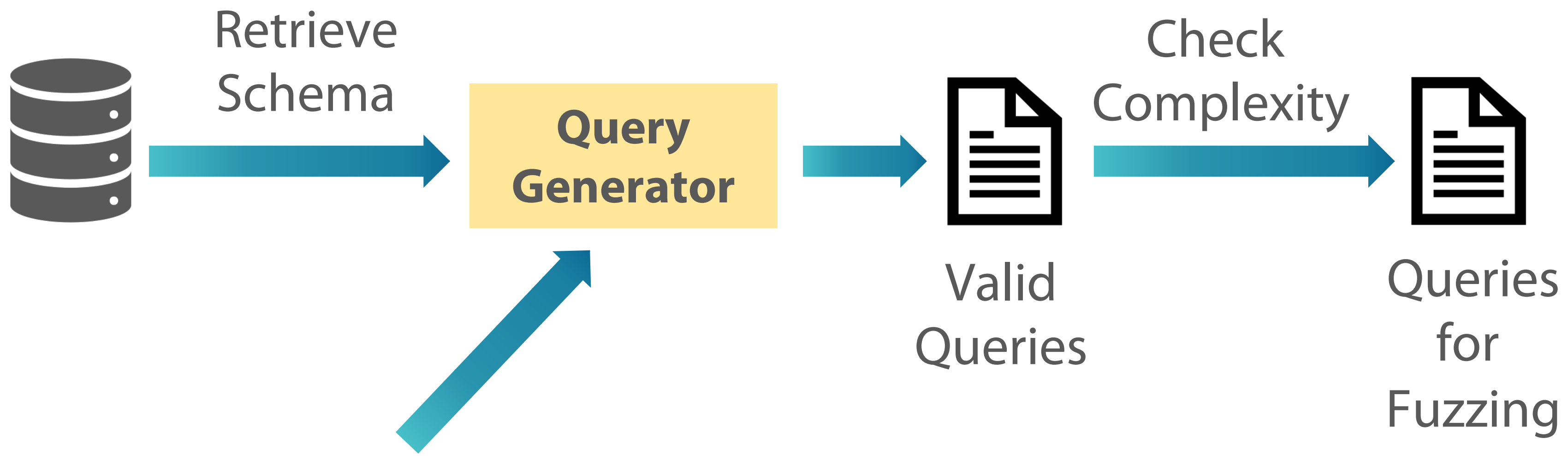


#1: SQLFUZZ — DETECTING REGRESSIONS



#1: SQLFUZZ — DETECTING REGRESSIONS

1 QUERY GENERATOR: RANDOM QUERY GENERATION

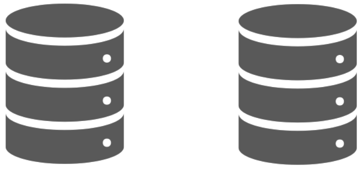


CASE	0.3	LEFT JOIN	0.3
LIMIT	0.2	CAST	0.2

SQL Grammar Probability Table

#1: SQLFUZZ — DETECTING REGRESSIONS

2 QUERY EXECUTOR: FEEDBACK-DRIVEN FUZZING



Old Version New Version

Found Regression?

Query Executor

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

Update Table

CASE		LEFT JOIN	
LIMIT		CAST	+0.1

SQL Grammar Probability Table

#1: SQLFUZZ — DETECTING REGRESSIONS

3 REGRESSION VALIDATOR: REDUCING FALSE POSITIVES

Queries Exhibiting Performance Regression

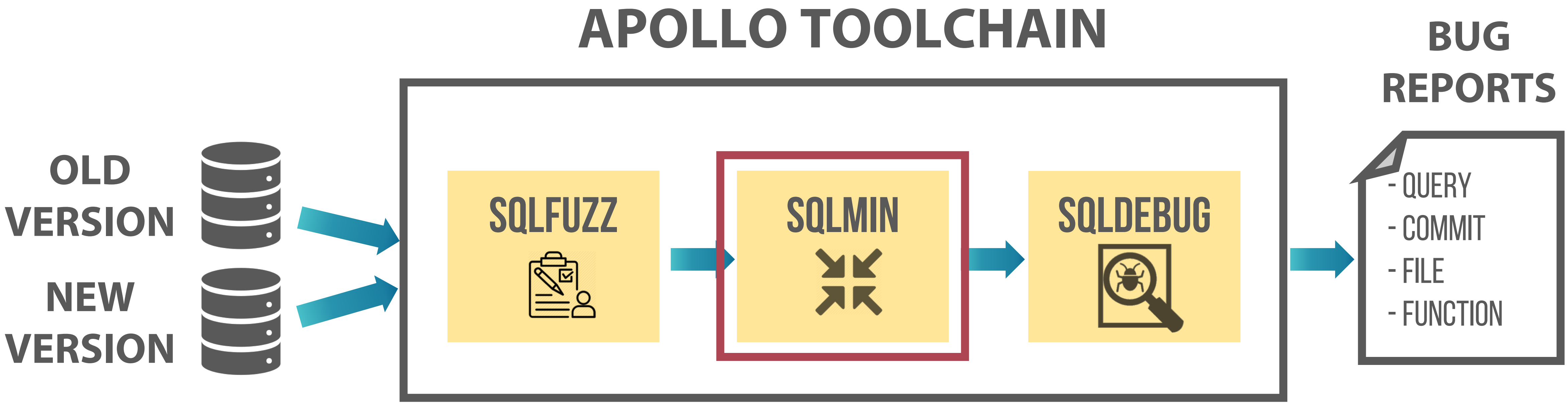
Filtering Rules

1	NON-DETERMINISTIC BEHAVIOR
2	NON-EXECUTED QUERY PLAN?
3	USAGE OF CATALOG STATISTICS?
4	ENOUGH MEMORY?
5	LIMIT STATEMENT?
6	QUERY IS TOO COMPLEX?
7	…

DBMS Developers

Update Filtering Rules

TALK OVERVIEW



#2: SQLMIN — REPORTING REGRESSIONS

- Top-Down Query Reduction
  - Iteratively remove unnecessary query elements
- Bottom-Up Query Reduction
  - Extract valid sub-queries

#2: SQLMIN — REPORTING REGRESSIONS

```
SELECT S1.C2
FROM (
 SELECT
  CASE WHEN EXISTS (
    SELECT S0.C0
    FROM ORDER AS R1
    WHERE ((S0.C0 = 10) AND (S0.C1 IS NULL))
  ) THEN S0.C0 END AS C2,
 FROM (
   SELECT R0.I_PRICE AS C0, R0.I_DATA AS C1,
    (SELECT ID FROM ITEM) AS C2
  FROM ITEM AS R0
  WHERE R0.PRICE IS NOT NULL
    OR (R0.PRICE IS NOT S1.C2)
  LIMIT 1000) AS S0) AS S1;
```

#2: SQLMIN — REPORTING REGRESSIONS

```
SELECT S1.C2
FROM (
 SELECT
  CASE WHEN EXISTS (
      SELECT S0.C0
      FROM ORDER AS R1
      WHERE ((S0.C0 = 10) AND (S0.C1 IS NULL))
  ) THEN S0.C0 END AS C2,
 FROM (
   SELECT R0.I_PRICE AS C0, R0.I_DATA AS C1,
      (SELECT ID FROM ITEM) AS C2
  FROM ITEM AS R0
  WHERE R0.PRICE IS NOT NULL
    OR (R0.PRICE IS NOT S1.C2)
  LIMIT 1000) AS S0) AS S1;
```

BOTTOM-UP REDUCTION

EXTRACT SUB-QUERY

Remove dependencies

#2: SQLMIN — REPORTING REGRESSIONS

```
SELECT S1.C2
FROM (
SELECT
  CASE WHEN EXISTS (
     SELECT S0.C0
     FROM ORDER AS R1
     WHERE ((S0.C0 = 10) AND (S0.C1 IS NULL))
  ) THEN S0.C0 END AS C2,
FROM (
  SELECT R0.I_PRICE AS C0, R0.I_DATA AS C1,
     (SELECT ID FROM ITEM) AS C2
  FROM ITEM AS R0
  WHERE R0.PRICE IS NOT NULL
     OR (R0.PRICE IS NOT S1.C2)
  LIMIT 1000) AS S0) AS S1;
```

TOP-DOWN REDUCTION

REMOVE ELEMENTS

Remove conditions

Remove columns

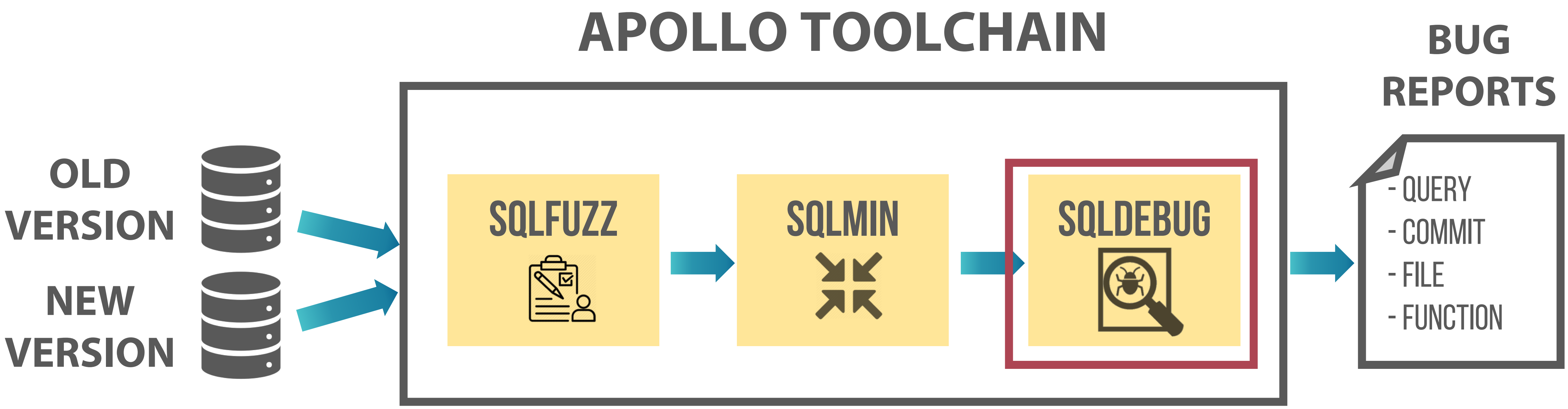
Remove sub-queries

Remove clauses

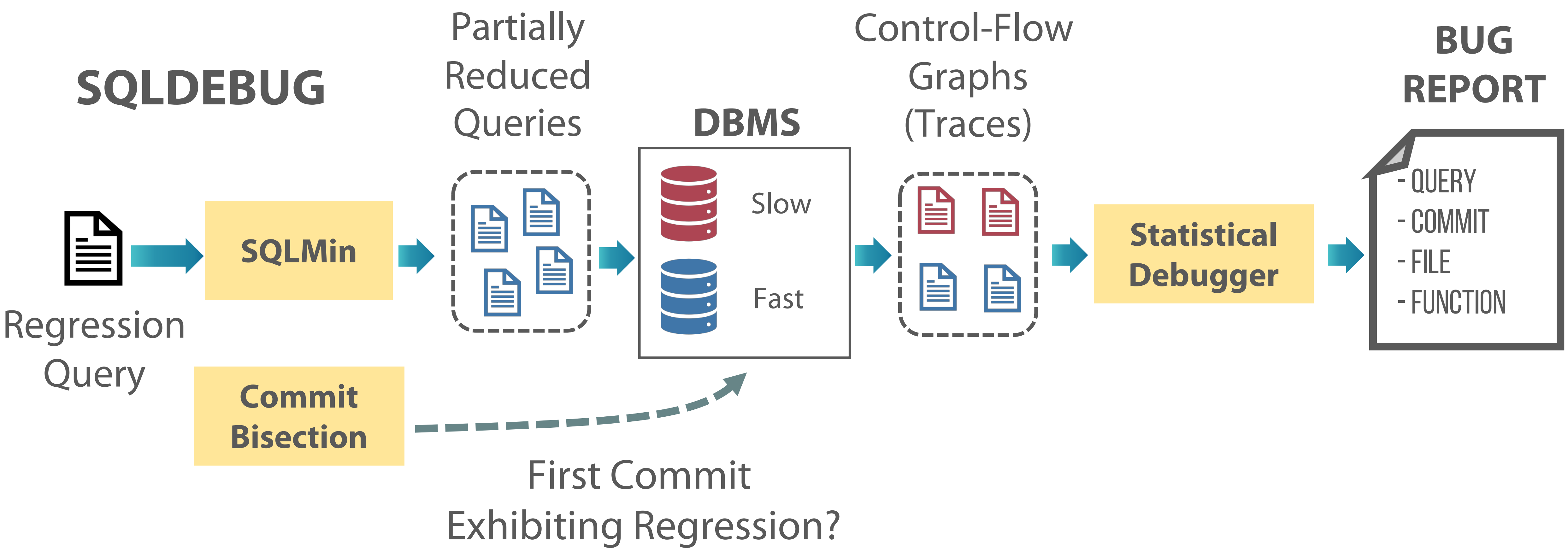
#2: SQLMIN — REPORTING REGRESSIONS

```
SELECT
  CASE WHEN EXISTS (
     SELECT S0.C0
     FROM ORDER AS R1
     WHERE ((S0.C0 = 10))
  ) THEN S0.C0 END AS C2,
FROM (
   SELECT R0.I_PRICE AS C0,
   FROM ITEM AS R0
   WHERE R0.PRICE IS NOT NULL) AS S0)
AS S1;
```

TALK OVERVIEW

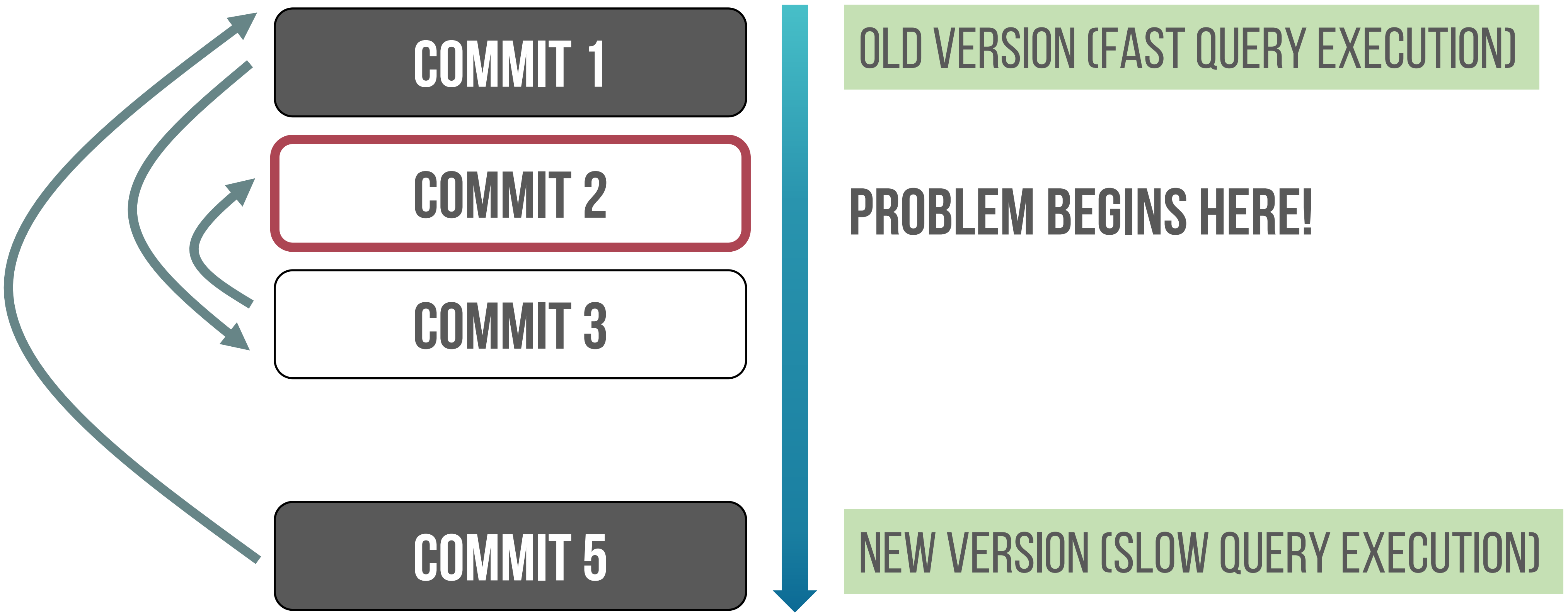


#3: SQLDEBUG — DIAGNOSING REGRESSIONS



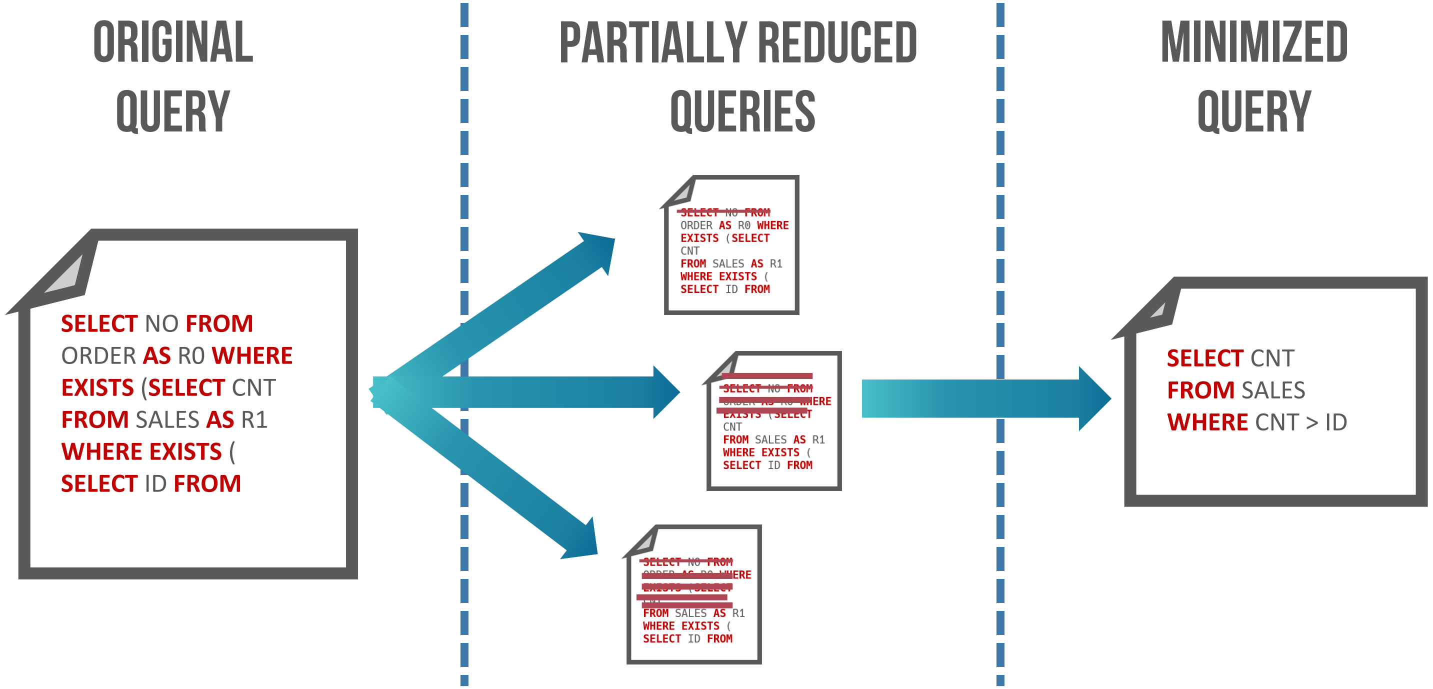
#3: SQLDEBUG — DIAGNOSING REGRESSIONS

1 COMMIT BISECTION: FIND EARLIEST PROBLEMATIC COMMIT



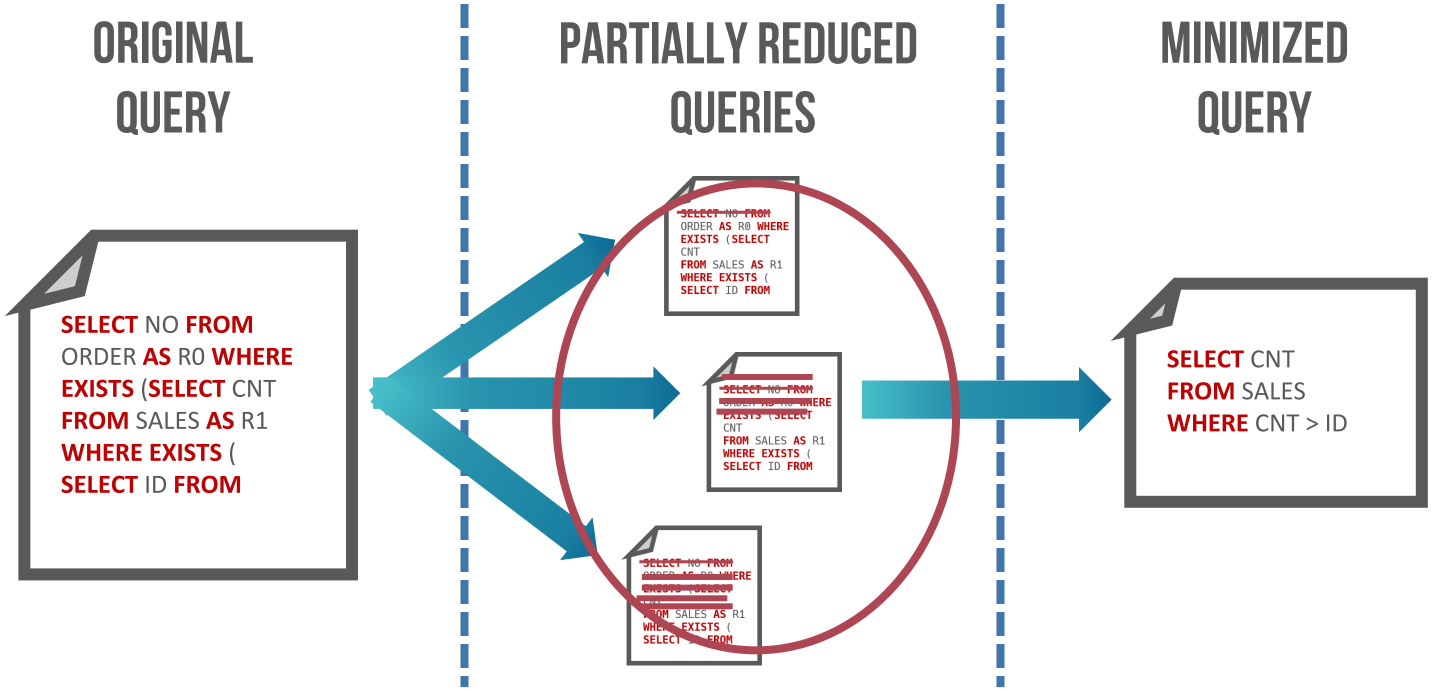
#3: SQLDEBUG — DIAGNOSING REGRESSIONS

2 QUERY REDUCTION: PARTIALLY REDUCED QUERIES



#3: SQLDEBUG — DIAGNOSING REGRESSIONS

2 QUERY REDUCTION: PARTIALLY REDUCED QUERIES



COLLECT SET OF QUERIES

#3: SQLDEBUG — DIAGNOSING REGRESSIONS

3 CONTROL-FLOW GRAPH COMPARISON: ALIGN TRACES

Functions

Old Version

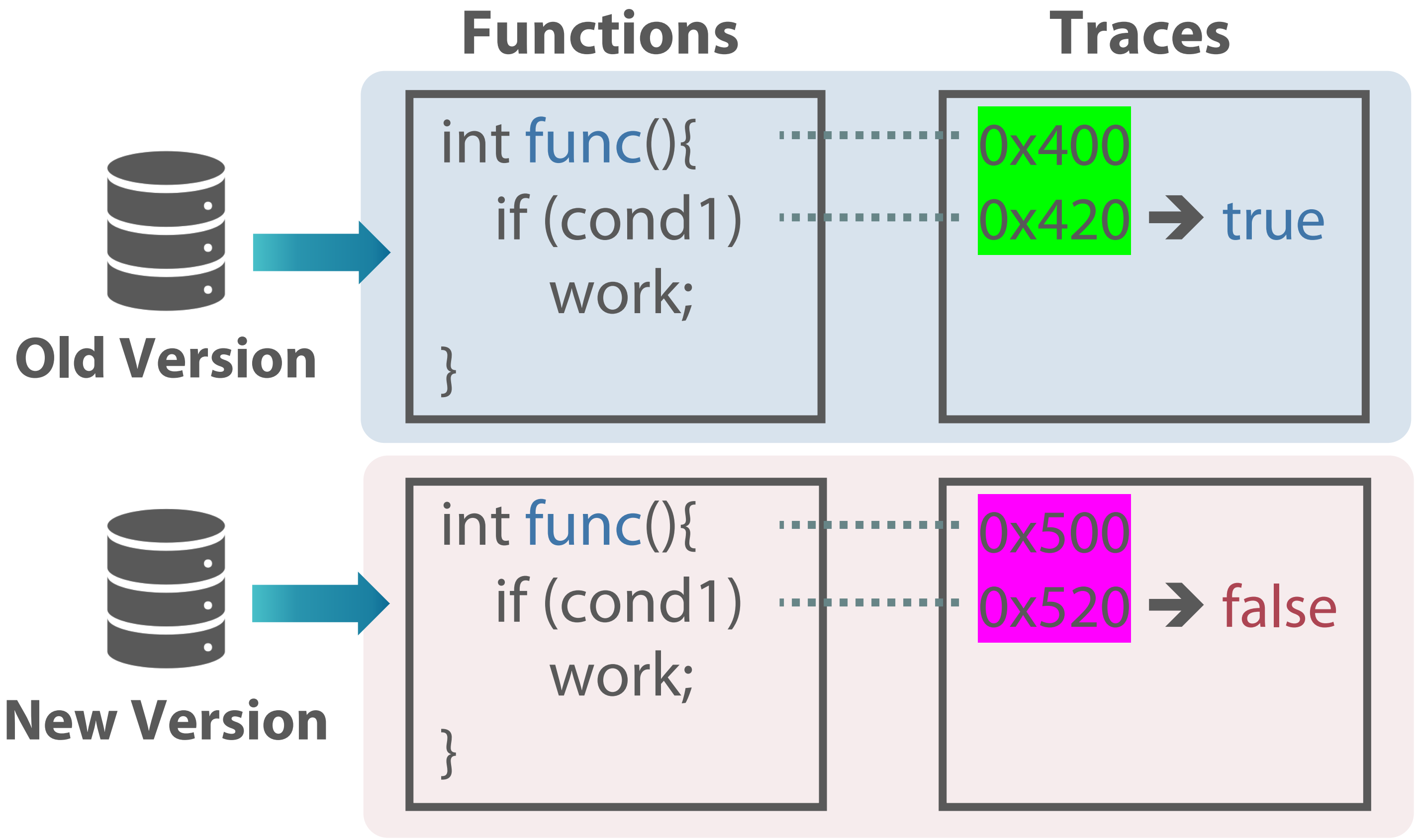
```
int func(){
    if (cond1)
        work;
}
```

New Version

```
int func(){
    if (cond1)
        work;
}
```

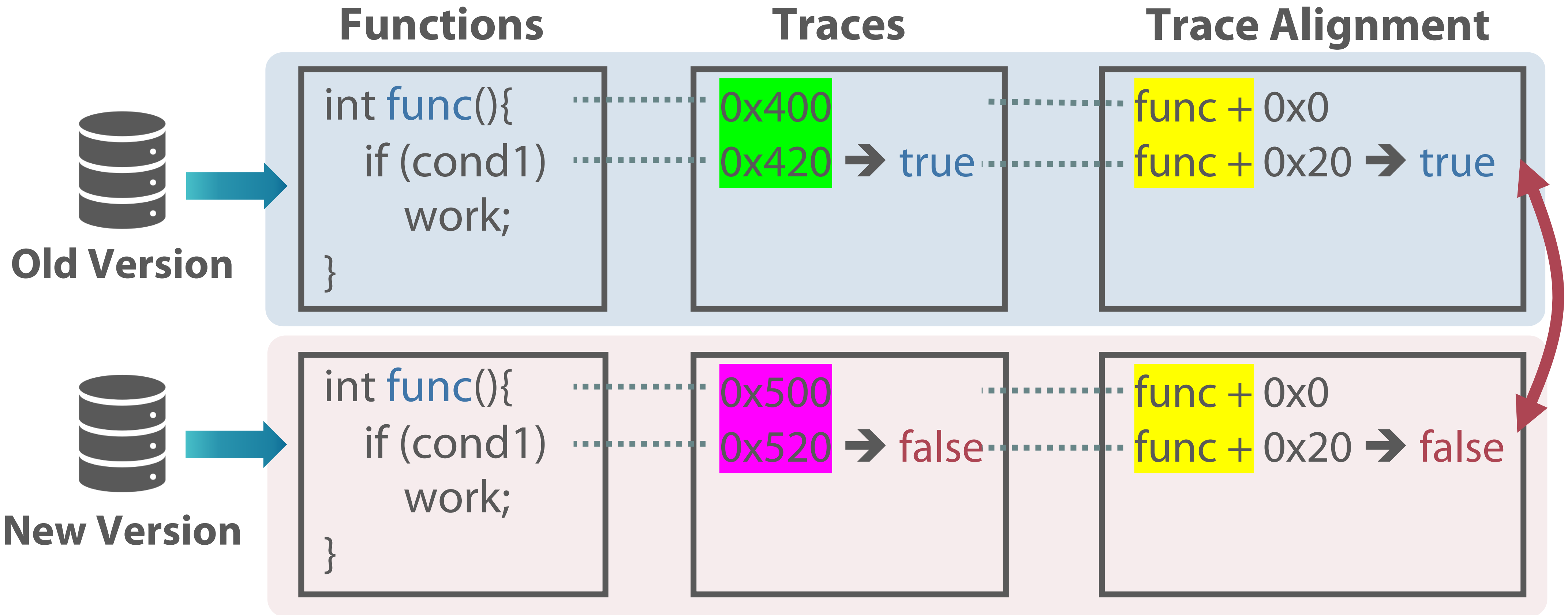
#3: SQLDEBUG — DIAGNOSING REGRESSIONS

3 CONTROL-FLOW GRAPH COMPARISON: ALIGN TRACES



#3: SQLDEBUG — DIAGNOSING REGRESSIONS

3 CONTROL-FLOW GRAPH COMPARISON: ALIGN TRACES



#3: SQLDEBUG — DIAGNOSING REGRESSIONS

4 STATISTICAL DEBUGGING: FAST AND SLOW QUERY TRACES

Fast Query Execution Traces

BRANCH	TRACE
1	TAKEN
2	TAKEN

Slow Query Execution Traces

BRANCH	TRACE
1	TAKEN
2	NOT TAKEN

Statistical Debugging Model

#3: SQLDEBUG — DIAGNOSING REGRESSIONS

4 STATISTICAL DEBUGGING: FAST AND SLOW QUERY TRACES

Fast Query Execution Traces

BRANCH	TRACE
1	TAKEN
2	TAKEN

Slow Query Execution Traces

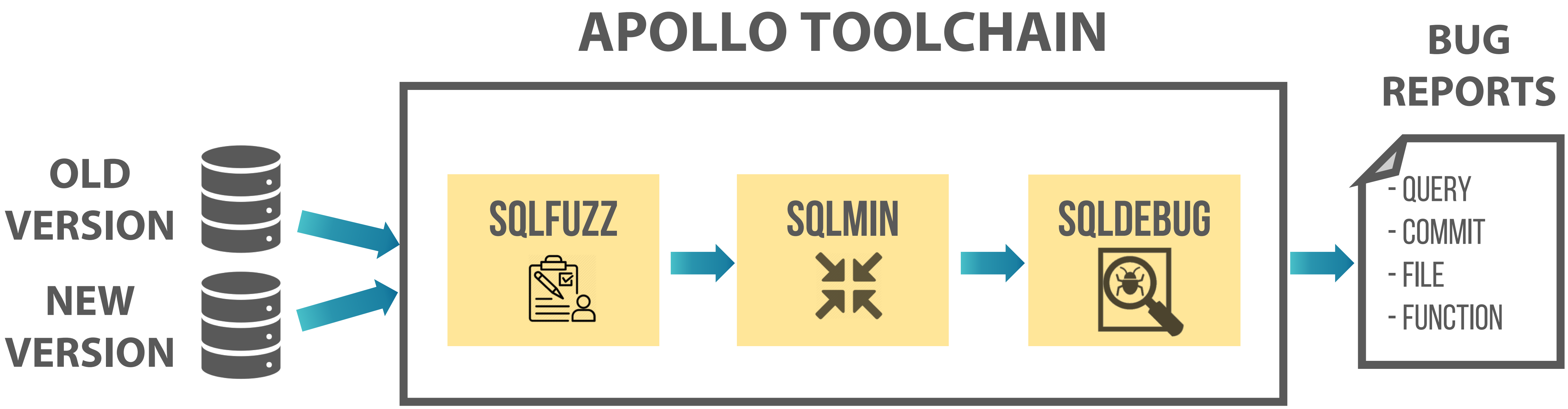
BRANCH	TRACE
1	TAKEN
2	NOT TAKEN

Statistical Debugging Model

RANK	FILE	FUNCTION	LINE
1	foo.c	bar()	2
…	…	…	…

Bug Report

RECAP

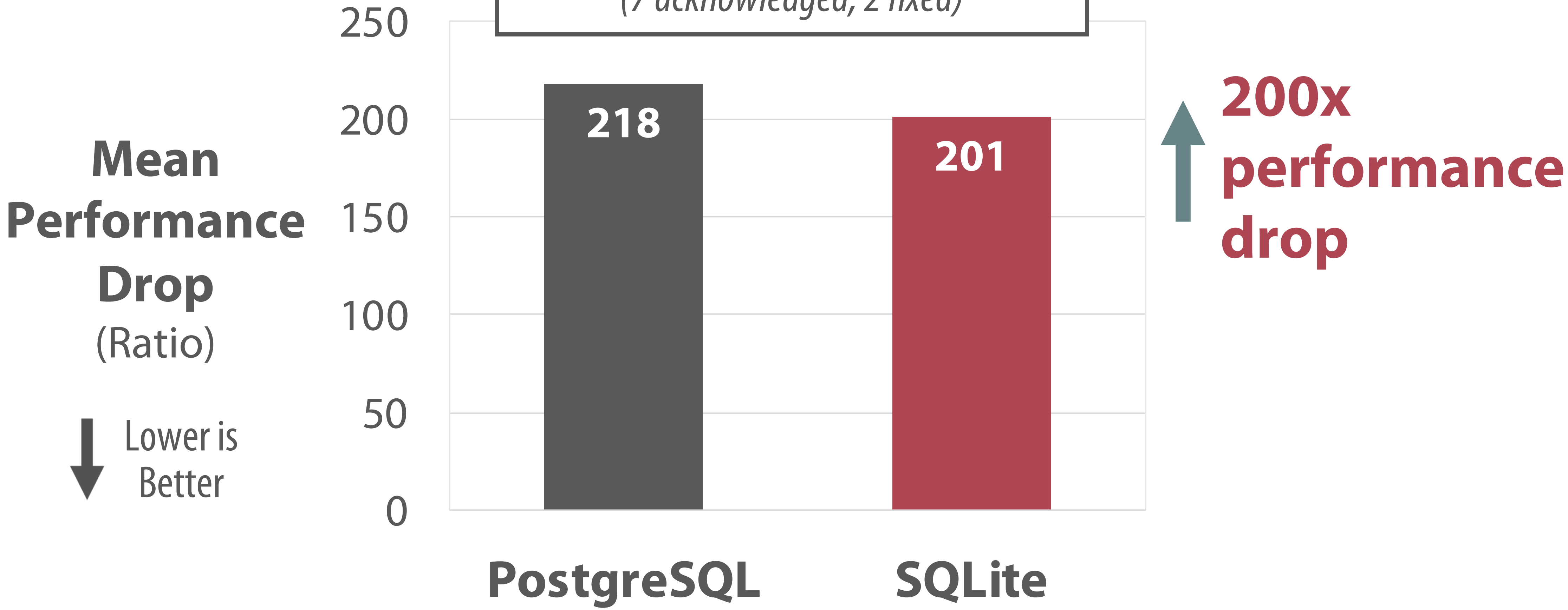

APOLLO TOOLCHAIN
OLD VERSION
NEW VERSION
SQLFUZZ
SQLMIN
SQLDEBUG
BUG REPORTS
- QUERY
- COMMIT
- FILE
- FUNCTION

EVALUATION

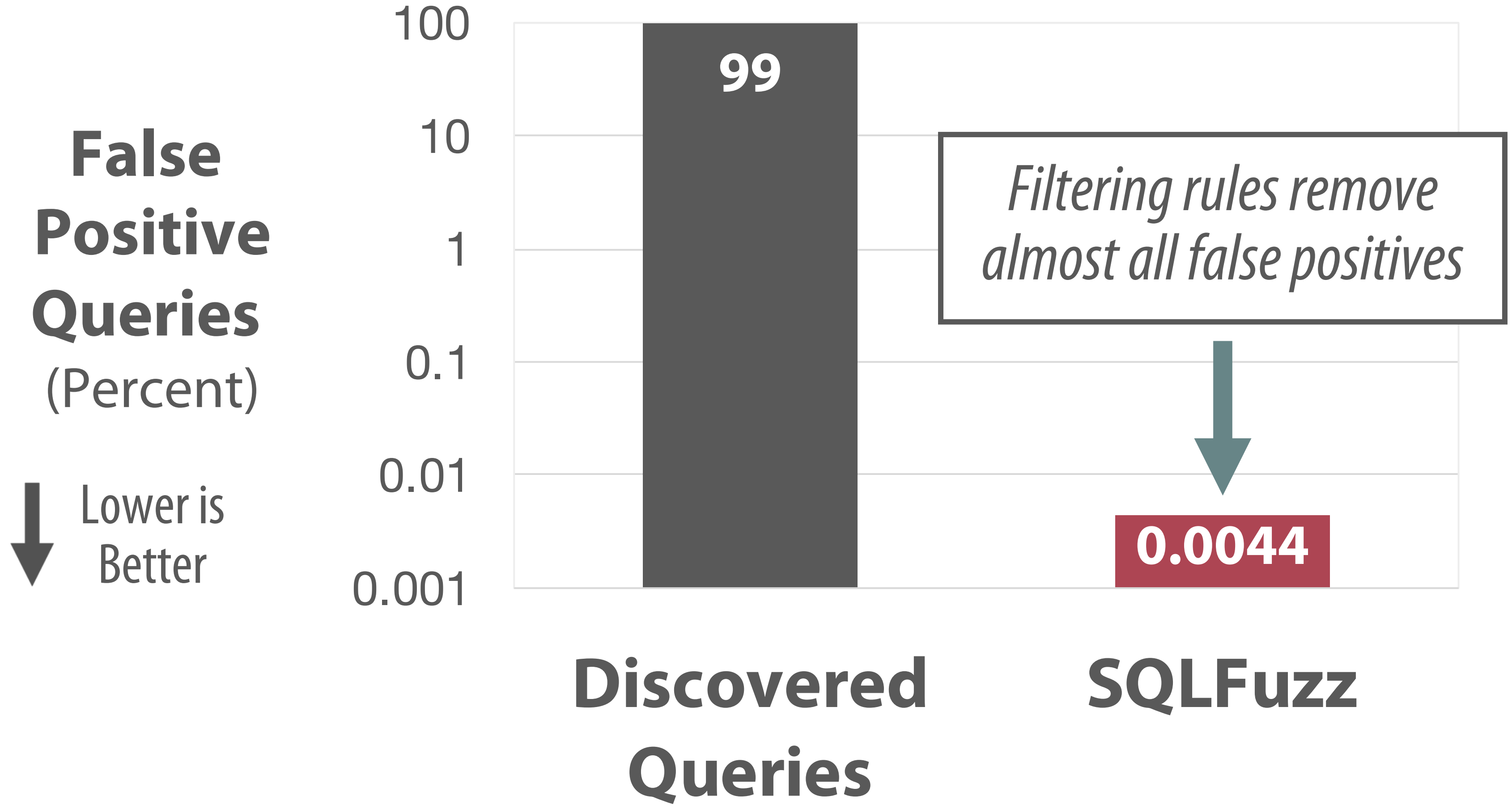
- Tested database systems
  - PostgreSQL, SQLite
- Binary instrumentation to get control flow graphs
  - DynamoRIO instrumentation tool
- Evaluation
  - Efficacy of SQLFuzz in detecting regressions?
  - Efficacy of SQLMin in reducing queries?
  - Accuracy of SQLDebug in diagnosing regressions?

#1: SQLFUZZ — DETECTING REGRESSIONS

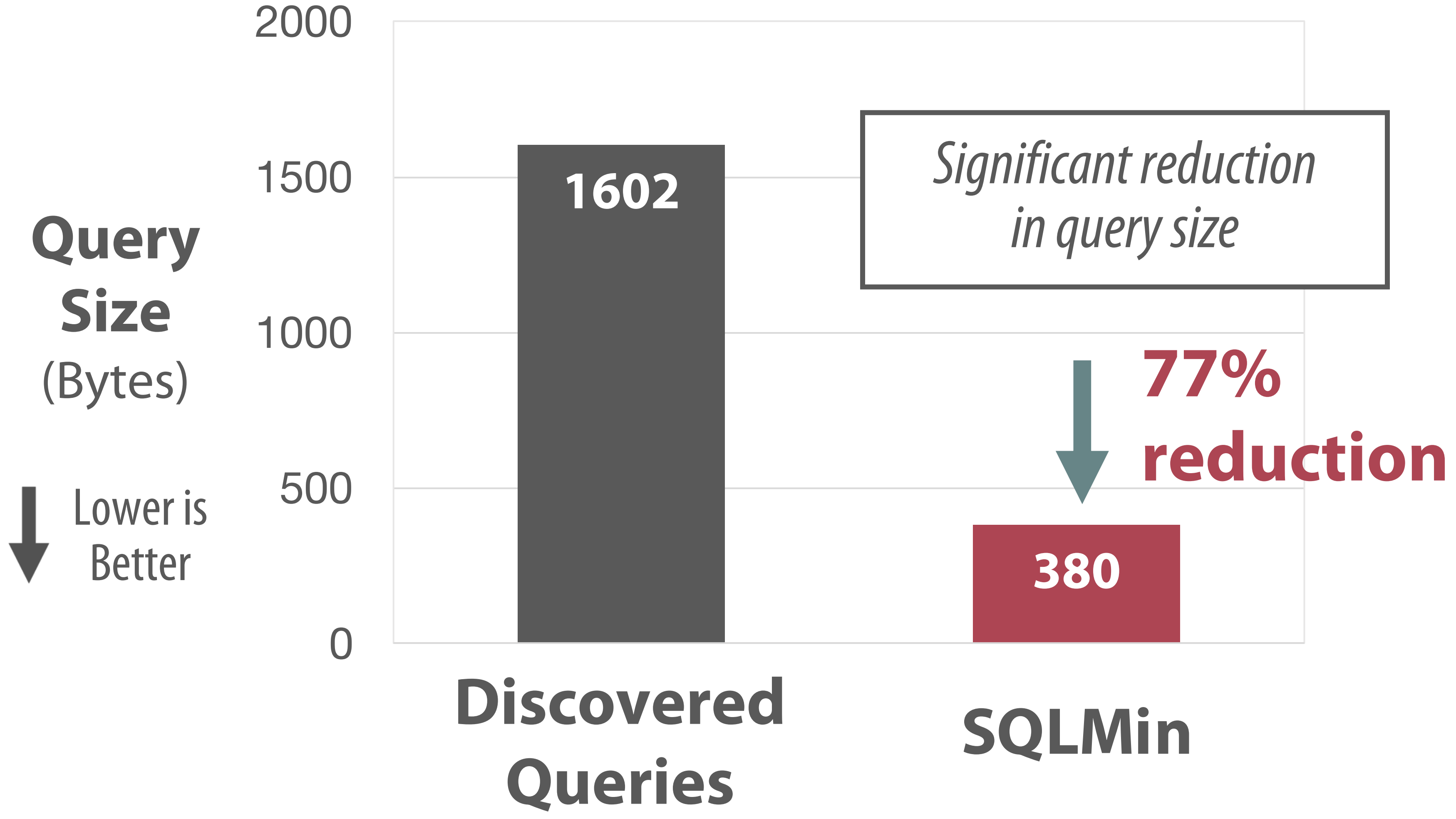
Discovered 10 previously unknown, unique performance regressions.
(7 acknowledged, 2 fixed)



#1: SQLFUZZ — FALSE POSITIVES

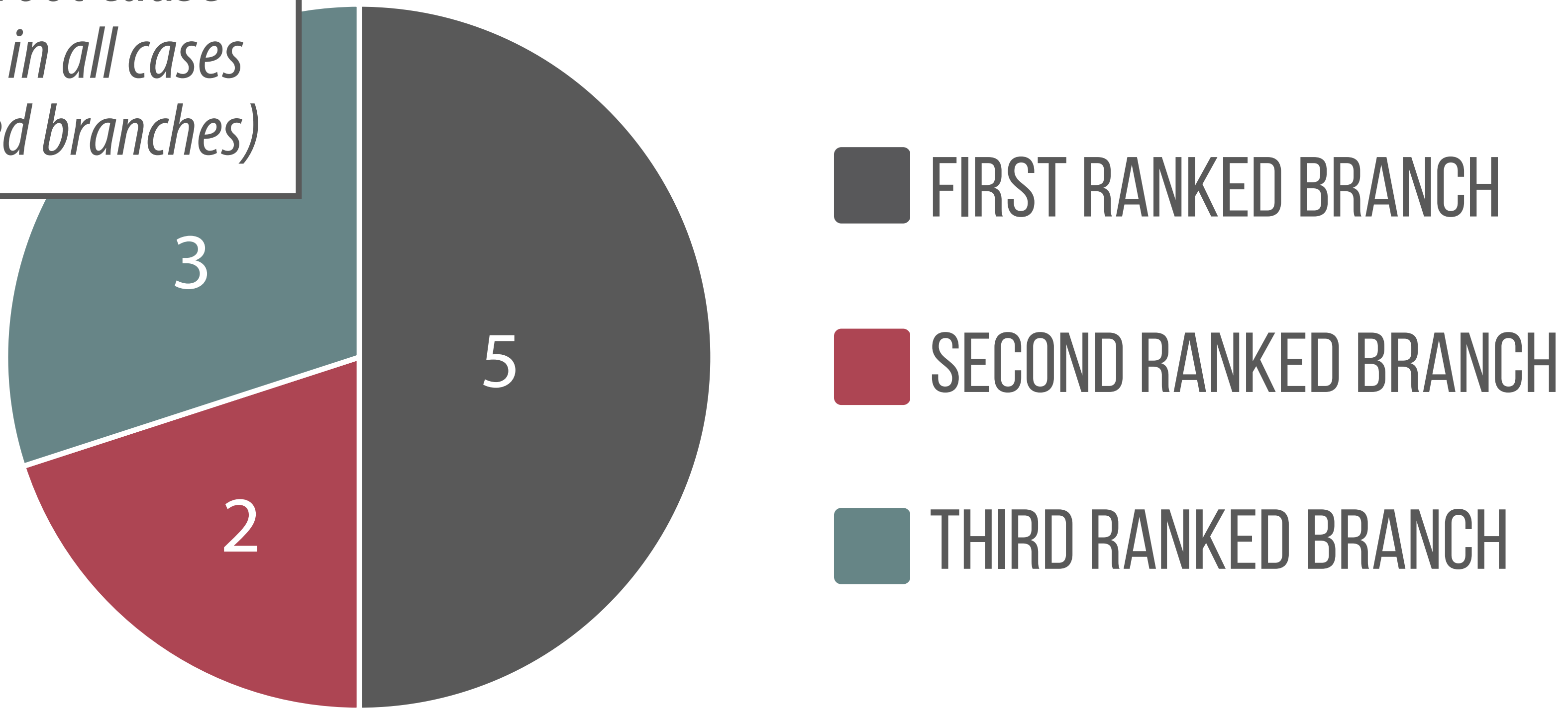


#2: SQLMIN — REPORTING REGRESSIONS



#3: SQLDEBUG — DIAGNOSING REGRESSIONS

Branch related to root cause correctly identified in all cases (within top-3 ranked branches)



10 regressions

CASE STUDY #1: OPTIMIZER UPDATE

```
SELECT COUNT (*)
FROM (SELECT R0.ID
  FROM CUSTOMER AS R0 LEFT JOIN STOCK AS R1
  ON (R0.STREET = R1.DIST)
  WHERE R1.DIST IS NOT NULL) AS S0
WHERE EXISTS (SELECT ID FROM CUSTOMER);
```

>1000x slowdown

LATEST VERSION OF SQLITE

- Due to a bug fix (for a correctness bug)
  - Breaks query optimization
  - Optimizer no longer transforms the LEFT JOIN operator
- Regression status: Not Yet Fixed
  - Searching for a fix that resolves both correctness and performance issues

CASE STUDY #2: EXECUTION ENGINE UPDATE

```
SELECT R0.ID FROM ORDER AS R0
WHERE EXISTS (SELECT COUNT(*)
  FROM (SELECT DISTINCT R0.ENTRY
    FROM CUSTOMER AS R1
    WHERE (FALSE)) AS S1);
```

3x slowdown

LATEST VERSION OF POSTGRESQL

- Hashed aggregation executor update
  - Resulted in redundantly building hash tables
- Regression status: Fixed
  - If hash table already exists, then reuse the table

CONCLUSION

- APOLLO (v1.0)
  - Toolchain for detecting & diagnosing regressions
  - Going to be open-sourced in 2020
- Adding support for other types of bugs (v2.0)
  - Correctness bugs
  - System crashes
  - Database corruption

CONCLUSION

- Interested in integrating APOLLO with more database systems
  - Improve the toolchain based on developer feedback
- Automation will help reduce labor cost of developing DBMSs
  - Developers get to focus on more important problems

END

@joy_arulraj