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

Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, Woonhak Kang
Georgia Institute of Technology, eBay Inc.

VLDB’20

https://home.cse.ust.hk/~yzhanghw/
Presented by: ZHANG Yushan
Background

- Developers conduct regular regression testing on databases
- **Performance bugs** are typically discovered with complex SQL queries on enormous databases

---

PostgreSQL v11.1 takes 15,800x time compared to v9.5.0

SQLite v3.27.2 2 hours, v3.23.0 within 1 hour

The commit introduces the bug includes 242 lines of additions, 116 lines of deletions, spanning 20 files.
BUG #6275: Horrible performance regression

From: "Finlay Thompson" <finlay(at)dragonfly(dot)co(dot)nz>
To: psql-bugs(at)postgresql(dot)org
Subject: BUG #6275: Horrible performance regression
Date: 2011-10-28 02:42:39
Message-ID: 201110280242.079286R0054867@wwwmaster.postgresql.org

The following bug has been logged online:

Bug reference: 6275
Logged by: Finlay Thompson
Email address: finlay(at)dragonfly(dot)co(dot)nz
PostgreSQL version: 8.4.9
Operating system: Ubuntu 11.04
Description: Horrible performance regression
Details:

After an upgrade from 8.4.8 to 8.4.9 performance of load script went from ~3 hours to not finishing after 24 hours.

Context: We have a continuous build script, that loads, grooms, matches data every few hours. The script has been run over 100 times in the last few weeks, and has steadily been taking around 3 hours. It is really a group of scripts, something like 15000 lines of sql, and some python scripts to run it together.

After upgrading the postgresql*-8.4 packages on ubuntu, to version 8.4.9, the script suddenly stopped working, and consuming all the ram (16GB) on the computer (i7).

Challenging to fix the bug!

Input Minimization:
Which SQL query has bad performance? (?/15000)

Bug Diagnosis:
1. Which commit introduced the bug?
2. Which lines of code in the new version actually triggers the bug?

Besides, it is also challenging to manually write test inputs.
APOLLO

A toolchain for automatically detecting, reporting, and diagnosing performance regressions in DBMSs
Main Contributions

• They introduce a technique to \textit{automatically detect performance regression bug} in DBMSs using domain-specific fuzzing.

• They propose an algorithm to \textit{automatically reduce queries for reporting} regression bugs.

• They formulate a technique to \textit{automatically locate the root cause of regressions} through bisecting and statistical debugging.

• They demonstrate the \textit{utility of our automated techniques for Performance regressions on two popular DBMSs}: SQLite and PostgreSQL.
To guide the fuzzing engine to generate new queries:

- Based on SQL grammar (AST mutation)
- Use Probability for each clause in a SQL query
  - WHERE clause is used in 70% of generated queries
- Monitor Runtime performance
  - JOIN is usually slow on big tables
SQLFuzz

Non-deterministics in query planning, Execution, optimizer, configuration, duplications

Version Differential Testing
- Execute a query on two versions and compare the performance (time)
Query Reduction

Previous research:
• **RAGS** [1]
  • Discard terms in the query and remove WHERE and HAVING clauses
• **Reducer** [2]
  • Delete line by line and remove column name from SELECT or INSERT

• **Problem 1**: data dependency between expressions are ignored
• **Problem 2**: top-down approach removes expression from the entire query, results in a syntax error
• **Problem 3**: tailored for correctness or functionality bugs

1. Extract subquery

2. Remove condition

3. Remove column list

4. Remove subquery

5. Remove clause
SQLDebug

• Identification of Problematic Commit
• Localization of Root Cause
Identify the Commit

• **Binary search** on commits between the two versions

• Compile each commit

• Run the query which triggers performance bug
Locate the Root Cause

- Statistical Debugging on two versions (slow and fast)
- Similar to Automatic Fault Localization with Metrics
- Ranks predictors by importance

2020/11/19

Extra methods!

```c
main()
{
    exif_loader_get_data()
    exif_data_load_data()
        exif_mnote_data_canon_load()
    exif_data_save_data()
        exif_data_save_data_content()
        exif_data_save_data_entry()
    exif_mnote_data_save()
        exif_mnote_data_canon_save()
    memcpy()
}
```

(o + s > buf_size) strong predictor

CRASH HERE SOMETIMES

```c
// snippet of exif_mnote_data_canon_load()
...
n->entries[i].data = malloc(s);
...
```
Implementation

• They wrote 3,054 lines of Python code and 156 lines of C++ code.
• They developed SQLFUZZ based on SQLSmith and Random Query Generator (RQG) [1].
• They used DynamoRIO [2] to collect execution traces in SQLDEBUG.
• During the fuzzing, we use the TPC-C benchmark in our fuzzing and corresponding evaluations [3].
Evaluation

• **Regression Detection**: Is APOLLO effective at finding performance regressions in real-world DBMSs? How effective is SQLFUZZ at removing false positives?

• **Query Reduction**: Can SQLMIN outperform RAGS on reducing discovered queries? How effective are different strategies?

• **Regression Diagnosis**: Can SQLDEBUG localize the root cause of detected performance regressions?
Experiment Setup

• They evaluated 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.

• APOLLO was run on a server with Intel(R) Xeon(R) Gold 6140 CPU (32 processors) and 384 GB of RAM for two months.
Regression Detection (I) finding regressions

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Versions</th>
<th>Perf. Drop</th>
<th>Query Minimization</th>
<th>Commit Bisecting</th>
<th>Statistical Debugging</th>
<th>Bug Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>original</td>
<td>reduced</td>
<td>identifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>3.23.0</td>
<td>&gt; 1000×</td>
<td>3,875</td>
<td>d840e9b</td>
<td>expr.c:impliesNotNullRow</td>
<td>Confirmed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24.5×</td>
<td>1,447</td>
<td>172f5bd</td>
<td>where.c:whereLoopAddBtree</td>
<td>Reported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>51.9×</td>
<td>1,717</td>
<td>57eb2ab</td>
<td>select.c:sqlite3Select</td>
<td>Confirmed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.4×</td>
<td>3,912</td>
<td>7d9072b</td>
<td>expr.c:codeApplyAffinity</td>
<td>Confirmed</td>
</tr>
<tr>
<td></td>
<td>3.27.2</td>
<td>1.6×</td>
<td>923</td>
<td>e130319</td>
<td>expr.c:sqlite3VdbeJumpHere,</td>
<td>Confirmed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>406</td>
<td></td>
<td>expr.c:sqlite3VdbeAdd0p0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>9.5.0</td>
<td>&gt; 1000×</td>
<td>572</td>
<td>5edc63b†</td>
<td>costsize.c:compute_bitmap_pages</td>
<td>Fixed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.2×</td>
<td>767</td>
<td>6b6614</td>
<td>execGrouping.c:BuildTupleHashTable</td>
<td>Fixed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.7×</td>
<td>1,619</td>
<td>77cd477†</td>
<td>costsize.c:max_parallel_degree</td>
<td>Confirmed</td>
</tr>
<tr>
<td></td>
<td>11.1</td>
<td>2.0×</td>
<td>531</td>
<td>0c2070c</td>
<td>costsize.c:cost_seqscan</td>
<td>Reported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.9×</td>
<td>659</td>
<td>7ca25b7</td>
<td>selfuncs.c:neqjoinsel</td>
<td></td>
</tr>
</tbody>
</table>
Regression Detection (II) regression validation

- **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.

24-hour Fuzzing experiment

![Graph showing FP and TP rates for different scenarios](image-url)
Query Reduction

RAGS is the previously developed system. **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.
Regression Diagnosis

Example 8. Existence of Two Problems.
BHS immediately skips the predicate evaluation, while SS evaluates the predicate on all the tuples in the table.

the parallel scan is slower than the sequential scan, and the newer version is slower than the old version

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

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

(a) 24-hour Fuzzing experiment, on PostgreSQL

(b) 1000 normal queries
2268 regression-triggering queries
Summary

• This paper presented APOLLO, a toolchain for automatically detecting and diagnosing performance regressions in DBMSs.
• APOLLO leverages domain-specific fuzzing to detect performance regressions.
• APOLLO discovered 10 previously unknown performance regressions from SQLite and PostgreSQL. (7 confirmed by developers)
My Reflections

• The paper was accepted for:
  • Having **significant and practical impacts** on both research community and developers in database area
  • Providing an **end-to-end solution** for debugging performance regressions
  • **Clearly stating** the challenges and improvement on the previous work
  • **Bringing together** methodology from security and software engineering research
My Reflections

• We have more research opportunities:
  • This papers only provides solutions for performance regressions, there are many other bugs (e.g. executor concurrency).
  • 68% generated queries are discarded due to syntax and semantic errors. If we could do better in generation only, we will gain better results.
  • Unacceptable Memory and Execution overhead by dynamic instrumentation for collecting query trace
  • Relatively low yield after two months evaluation (7/10)
  • Will the current strategies efficient on more complex database? There will be more non-deterministic behaviors. For example MySQL, which has robust inner optimizers and even run on machine clusters.