A Demonstration of DLBD: Database Logic Bug Detection System

Xiu Tang
Zhejiang University, China
tangxiu@zju.edu.cn

Sai Wu*
Zhejiang University, China
wusai@zju.edu.cn

Dongxiang Zhang
Zhejiang University, China
zhangdongxiang@zju.edu.cn

Ziyue Wang
Zhejiang University, China
ragesi_wang@163.com

Gongsheng Yuan
Zhejiang University, China
ygs@zju.edu.cn

Gang Chen
Zhejiang University, China
cg@zju.edu.cn

ABSTRACT

Database management systems (DBMSs) are prone to logic bugs that can result in incorrect query results. Current debugging tools are limited to single table queries and struggle with issues like lack of ground-truth results and repetitive query space exploration. In this paper, we demonstrate DLBD, a system that automatically detects logic bugs in databases. DLBD offers holistic logic bug detection by providing automatic schema and query generation and ground-truth query result retrieval. Additionally, DLBD provides minimal test cases and root cause analysis for each bug to aid developers in reproducing and fixing detected bugs. DLBD incorporates heuristics and domain-specific knowledge to efficiently prune the search space and employs query space exploration mechanisms to avoid the repetitive search. Finally, DLBD utilizes a distributed processing framework to test database logic bugs in a scalable and efficient manner. Our system offers developers a reliable and effective way to detect and fix logic bugs in DBMSs.

PVLDB Reference Format:
Xiu Tang, Sai Wu, Dongxiang Zhang, Ziyue Wang, Gongsheng Yuan, and Gang Chen. A Demonstration of DLBD: Database Logic Bug Detection System. PVLDB, 16(12): 3914 - 3917, 2023.
doi:10.14778/3611540.3611584

PVLDB Artifact Availability:
The source code, data, and other artifacts have been made available at https://github.com/xiutangzju/dlbd.

1 INTRODUCTION

Database Management Systems (DBMSs) play an essential role in today's software ecosystem, as they provide efficient and reliable data storage and retrieval mechanisms. However, despite the extensive testing and optimization efforts, query processing in DBMSs can still suffer from implementation errors, resulting in bugs that range from crashes to logic bugs. Crash bugs are raised either by the operating system, or by the process of DBMS. They cause the process of DBMS to be forcefully killed, due to limited resources (e.g., out of memory) or access to an invalid memory address, etc. However, the logic bugs are different from crashes, because the query processing algorithm still runs normally, and the consequence is that the DBMS fetches incorrect result sets. Crashes are easier to detect as they typically halt the system, whereas logic bugs can go unnoticed, leading to incorrect results. In this paper, we focus on the task of detecting these hidden bombs in DBMSs.

```
mysql> SET optimizer_switch='semijoin=off';

mysql> SELECT  t1.c0 FROM t1 WHERE (t1.c0 NOT IN (SELECT
t2.c0 FROM t1 as t2 WHERE t2.c0 )) = (t1.c0);
Empty set (0.00 sec)

mysql> SET optimizer_switch='semijoin=on';

mysql> SELECT  t1.c0 FROM t1 WHERE (t1.c0 NOT IN (SELECT
t2.c0 FROM t1 as t2 WHERE t2.c0 )) = (t1.c0);
Return 3 rows (0.00 sec)
```

Figure 1: MySQL's incorrect semi-join execution.

In Figure 1, we illustrate a logic bug of MySQL for join queries. The bug was first detected by our proposed tool in this paper. Figure 1 demonstrates a logic bug of semi-join in MySQL 8.0.28. In this example, the first query returns the correct result set, as it is executed using the inner hash join algorithm. However, the second query produces incorrect results due to a logic bug in the implementation of the semi-join algorithm. Specifically, this is because the equality was neither pushed down to the materialized sub-query, nor evaluated as part of the semi-join.

SQLancer [5] is a well-known tool in the field of testing DBMS for logic bugs. The tool employs several approaches to detect logic bugs, such as Pivoted Query Synthesis (PQS), Ternary Logic Partitioning (TLP), and Non-optimizing Reference Engine Construction (NoREC). PQS [6] constructs queries to fetch a randomly selected tuple from a table, while NoREC [4] compares the results of randomly generated optimized queries and rewritten queries that DBMS cannot optimize. On the other hand, QPG [1] steers testing towards exploring a variety of unique query plans. Despite the effectiveness of these approaches, they still have two limitations when it comes to a holistic logic bug detection system for databases.

- Previous approaches to verifying the correctness of query results have adopted the differential testing strategy. This involves processing a query using different physical plans within the same database or using different databases. If the queries return inconsistent result sets, a possible logic bug is detected. However, this strategy has two major drawbacks. First, some logic bugs affect multiple physical plans, resulting in all of them generating the same incorrect result. Second, when inconsistent result sets are observed, we must manually check which plan generates the correct

*Sai Wu is the corresponding author.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. 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. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611584

one, incurring high overheads. A possible solution to these issues is obtaining the ground-truth results for an arbitrary testing query, which existing tools do not support.
- The number of queries that can be generated from a given database schema is exponential to the number of tables and columns. Enumerating all possible queries for verification is infeasible. Thus, an effective query space exploration mechanism is necessary to automatically generate diversified and complex queries.

To address these challenges, we propose a **d**atabase **l**ogic **b**ug **d**etection system DLBD based on our previous work [8] as a remedy for database implementations. The system integrates all our proposed techniques and has the following features:

Holistic logic bug detection. DLBD provides automatic schema generation based on the input dataset or schema. To facilitate bug discovery, DLBD also inserts some artificial noise data into the generated database. DLBD first converts the database schema into a graph, then performs automatic query generation by adopting random walking on the schema graph to select tables for queries. For a specific query spanning over multiple tables, DLBD can easily identify its ground-truth results from the input data.

In this way, DLBD can effectively generate (query, result) pairs for database verification. To reduce developers' time required to identify and fix the bugs, DLBD provides minimal test case and primary root cause analysis. With pause and continue capabilities, DLBD can be paused during testing and resumed at a later time, providing a flexible and reliable bug detection process. Overall, DLBD's holistic logic bug detection approach is a powerful tool for identifying and fixing logic bugs in DBMS.

Query space exploration. DLBD incorporates an efficient and effective query space exploration mechanism that enables comprehensive testing of DBMS and early detection of logic bugs. By leveraging heuristics and domain-specific knowledge, DLBD significantly reduces the search space. The novel pruning strategy employed by DLBD plays a significant role in reducing the number of redundant queries generated during testing. This strategy identifies and prunes queries that are structurally similar to previously tested ones, thereby reducing the amount of time and resources needed for testing. The real-time statistics provided by DLBD are also instrumental in helping users monitor the progress of testing, providing valuable insights into the efficacy of the testing process.

2 SYSTEM OVERVIEW

In this section, we provide an overview of DLBD's architecture. Figure 2 depicts the architecture of DLBD that consists of the frontend and the backend. In particular, the frontend creates a web interface and data visualizations for the database schema graph, query space exploration, and detected bug details and statistics. The backend is responsible for schema and query generation, ground-truth generation, logic bug auto-detection, and query space exploration.

Schema and query generation. To generate a random schema for database testing, DLBD first applies schema normalization techniques on the inputted wide table to minimize data redundancy. DLBD leverages data-driven algorithms such as FD discovery [2] and schema normalization [3] to generate a testing database schema.

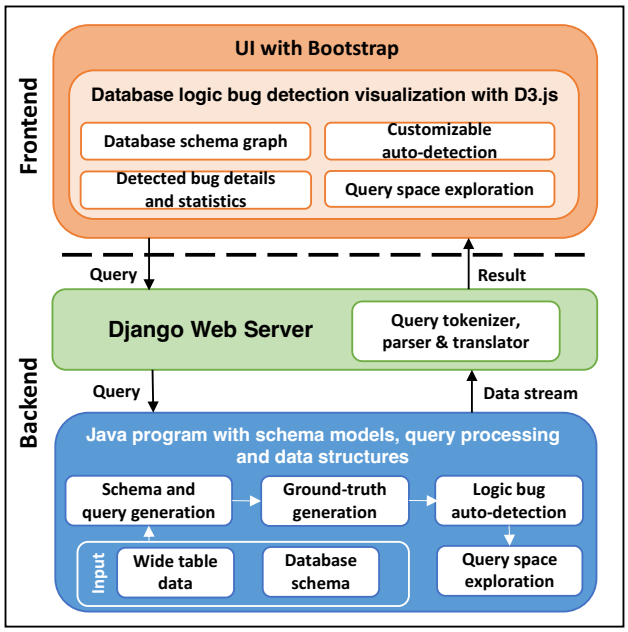

Figure 2: The architecture of DLBD system.

Additionally, DLBD incorporates a noise injection technique that increases the probability of detecting logic bugs.

To model the database schema information, DLBD adopts a graph model where nodes represent tables and columns, and edges represent the relationships between them. DLBD generates queries using abstract syntax trees (ASTs) and employs a random walk algorithm on the schema graph to select tables to include in the query. By randomly traversing the schema graph and selecting tables, DLBD can generate diverse queries that effectively test the database.

Ground-truth generation. Given a query, its ground-truth result is generated by mapping back all involved tables into a wide table where a rewritten query is executed to retrieve all results. During schema generation, the input wide table is split into multiple smaller tables, and each table is indexed by its primary key. The RowID mapping table records how each row in the wide table maps to the corresponding primary key value in the smaller tables. To further optimize the process, DLBD uses the bitmap index and WAH encoding [9] to quickly retrieve the required rows from the original wide table, ensuring that the ground-truth result for any query is accurate and efficiently generated.

Logic bug auto-detection. DLBD employs comparison algorithms that compare the query results to their expected ground-truth values to automatically detect logic bugs in the DBMSs. First, it retrieves the ground-truth result from the input data for a given query. Then, it materializes the logic query into physical plans and transforms the query with different hints to enable the DBMS to execute multiple physical plans for bug searching. Finally, the result set of the query is compared with the ground truth, and if there is a difference, a logic bug is detected.

Query space exploration. To efficiently explore the space of queries, DLBD can avoid generating similar queries repeatedly. First, DLBD extends the schema graph to a plan-iterative graph and maps each query to a sub-graph in this graph. It then builds an

Figure 3: A screenshot of DLBD (Database Logic Bug Detection).

embedding-based graph index that stores the historical embeddings of the query graph and indexes the nearest neighbor. By generating query graphs that are far away from their nearest neighbors in the graph index, DLBD can explore new query graphs while avoiding the repetition of existing ones. This is achieved by adjusting the weights of edges in the plan-iterative graph, which is guided by the graph index. By reducing the walk probability of already covered paths, DLBD can generate novel and diverse queries efficiently.

3 IMPLEMENTATION DETAILS

Root cause analysis. The basic idea of root cause analysis is to identify the physical operators that cause the logic bugs. For this purpose, DLDB provides a relevant portion of the schema graph that highlights the schema involved in the query, which can provide useful context for debugging the issue. This context is particularly important when dealing with complex databases with numerous interdependent tables and columns.

DLBD also provides detailed bug reports to developers which try to identify and fix bugs in databases. The detailed bug reports include the query that triggered the bug, and the expected and actual query results. Furthermore, DLBD compares the execution plan with the correct result to the execution plan with the wrong result to provide a root cause analysis. This detailed analysis enables developers to quickly pinpoint the root cause of the problem.

Pause and continue testing. The pause and continue capabilities provided by the DLBD are key features in achieving flexible and reliable DBMS testing. By identifying appropriate checkpoints at the end of each query, DLBD ensures that the test results are not compromised when the system is paused and resumed. When the system is paused, DLBD saves the current state of the system so that it can be resumed from where it left off. This includes saving the search space that has been explored, the detected logic bugs, and any intermediate state required to execute the remaining queries.

Search space estimation. The search space of potential queries in a database can be vast, particularly in a large and complex database schema, due to the exponential growth of the number of possible combinations of tables and columns. To address this challenge, we incorporate heuristics and domain-specific knowledge to efficiently reduce the search space.

We roughly partition the query space into several sub-spaces based on their graph embeddings, which can be further partitioned into more sub-spaces in a hierarchical way. When at least K queries are tested for a leaf-level sub-space, we say the space has been covered. In this way, we can estimate the process of our exploration.

Additionally, we continuously monitor the test results to refine the search space estimation based on the observed bugs and their underlying causes. By analyzing the root causes of detected bugs, we can adjust our heuristics to improve the coverage and effectiveness of the search space. For example, if many bugs are found in queries involving a particular table or join condition, this may indicate that more testing is needed to cover that category.

Distributed processing framework. DLBD utilizes a distributed processing framework to test database logic bugs in a scalable and efficient manner. By breaking down the testing workload and distributing it across multiple processing nodes, DLBD can assign different groups of queries to different nodes for simultaneous testing. This reduces the time required for testing and improves the overall efficiency of the testing process.

DLBD has chosen Apache Spark [7] as its distributed processing framework due to its ease of use, scalability, and ability to handle large data sets. Spark allows DLBD to distribute the testing workload across multiple nodes, and it provides fault tolerance and recovery mechanisms to ensure that the testing is not affected by hardware or software failures.

During the testing process, DLBD monitors the progress of the processing nodes to ensure that they are working correctly and that the testing is progressing as expected. It also collects and aggregates

the results from multiple nodes once the testing is complete. This enables DLBD to identify and analyze the database logic bugs more quickly and effectively, allowing developers to find and fix the bugs in a timely manner.

4 DEMONSTRATION SCENARIOS

Figure 3 is a screenshot of the frontend of DLBD. The user can observe the logic bug detection and analysis pipeline with the following steps:

Step 1 (Database schema graph.) The user first needs to connect our tool to a specific DBMS. Currently, we support MySQL, MariaDB, TiDB, and PolarDB. The tool has some specific considerations for different DBMSs, due to their features and SQL supports.

The user can upload a wide table or an existing database schema using the "Dataset" or "Schema" button, respectively (see Figure 3-1). After clicking the "Generate" button, DLBD generates a database with schema normalization. The database schema is modeled by a graph, which is shown in the "Database Schema Graph" panel. And the table information is displayed in the top bar of the panel (see Figure 3-2). The user can click on a node in the schema graph to see its detailed information in the right sidebar.

DLBD also provides a "Check Bugs" panel that allows users to select the types of bugs to be detected. The "Noise Injection" button controls whether DLBD injects noise data for boundary testing, while the "Table Join" button controls whether DLBD tests the joins. The "Aggregation" button controls whether DLBD tests the aggregations, and the "Column Sort" button controls whether DLBD tests the sorting of columns. Finally, the user can click the "Start" button to initiate bug detection for the selected bug types.

Step 2 (Customizable auto-detection.) When a user clicks the "Start" button, DLBD will automatically execute generated queries and verify the query results are correct. And the interface also provides a "Pause" button for pause and continue.

DLBD lists all detected bugs at the right of interface ("Detected Bugs" panel, see Figure 3-3). In DLBD, we design three levels, Critical, Serious, and Non-critical, to evaluate the bug severity, which takes into account both the number of bugs and the bug type with a weighted score based on the importance of the query optimizer. The darker the color of the severity, the more serious the bug. If the user wants to further observe one specific bug, she just needs to click the "Detail" button to see the detailed information.

The quantitative statistics of detected bugs are shown in Figure 3-5. It shows the total number of generated queries, the query types, and the bugs in real-time. The proportion of different bugs is shown by a pie chart based on the operator type of the detected bugs.

Step 3 (Detected bug details and statistics.) Once finding a possible bug, DLBD adds it to the "Detected Bugs" panel. When the user clicks one specific bug, DLDB provides a relevant portion of the schema graph and query space that highlights the schema involved in the query. When the user clicks the "Detail" button to show the root cause analysis of the bug (see Figure 4). It shows the severity, database, category, and operator type of the bug. DLBD produces a minimal test case to save the user's time and effort to repeat the bug. The interface provides the "Original Query" button and "Minimal Query" to show the original query that triggered the bug and its minimal test case. The interface shows the query,

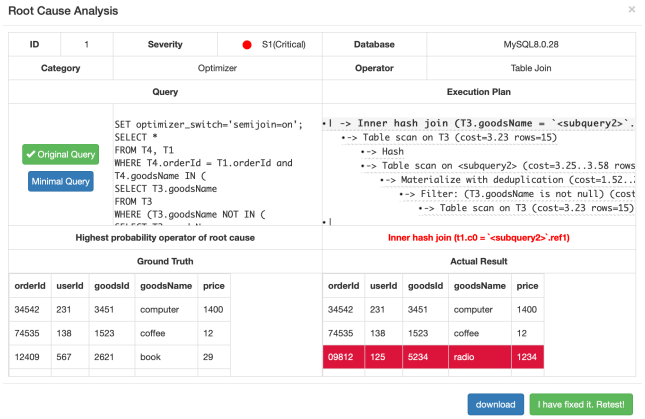

Figure 4: An example of root cause analysis of a bug.

the execution plan, the ground-truth and the actual result of the query. DLBD calculates the highest probability operator for the root cause based on the comparison, which is highlighted in red. And the incorrect tuple in the query result is also labeled in red.

DLBD also provides "download" button that allows the user to download test cases to reproduce the bug in the local database. When the user clicks "I have fixed it. Retest!" button, DLBD retests the query to verify the correctness of query results. If the query is executed correctly, the bug turns gray to indicate that it is fixed.

Step 4 (Query space exploration.) The total search space of query space exploration is shown in "Query Space Exploration" panel (see Figure 3-4). Because of the huge search space, only the search space of the tables selected by the user is represented here. Color blocks represent different edge connections, which can be clicked to reveal more information. When the user clicks one specific bug, the panel shows a relevant portion of the query space that is involved in the query.

ACKNOWLEDGMENT

This work was supported by the Key Research Program of Zhejiang Province (Grant No. 2023C01037).

REFERENCES

[1] Jinsheng Ba and Manuel Rigger. 2023. Testing database engines via query plan guidance. In *Proceedings of International Conference on Software Engineering (ICSE)*.
[2] Yka Huhtala, Juha Karkkainen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal* 42, 2 (1999), 100–111.
[3] Thorsten Papenbrock and Felix Naumann. 2017. Data-driven Schema Normalization. In *EDBT*. OpenProceedings.org, 342–353.
[4] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *ACM Joint Meeting on ESEC and FSE*. 1140–1152.
[5] Manuel Rigger and Zhendong Su. 2020. SQLancer. [EB/OL]. https://github.com/sqlancer/sqlancer.
[6] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *OSDI 20*. 667–682.
[7] Apache Spark. 2020. Apache Spark. [EB/OL]. https://spark.apache.org.
[8] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proc. ACM Manag. Data* 1, 1 (2023), 55:1–55:26.
[9] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2002. Compressing Bitmap Indexes for Faster Search Operations. In *SSDBM*. IEEE Computer Society, 99–108.