Automated DBMS Fuzzing Framework

Surendar Kumar (C00577830)
Department of Computer Science
University of Louisiana at Lafayette
Lafayette, LA 70504
`surendar.kumar1@louisiana.edu`

Rakshya Pandey (C00580017)
Department of Computer Science
University of Louisiana at Lafayette
Lafayette, LA 70504
`rakshya.pandey1@louisiana.edu`

March 2025

1 Introduction

Relational Database Management Systems (RDBMS) like PostgreSQL are critical components in numerous applications, managing data efficiently across various industries. Despite their robustness and widespread use, these systems are susceptible to logic bugs that can compromise data integrity and performance [2]. Traditional testing methods often fail to detect such subtle yet critical bugs due to the complexity of SQL queries and the database's internal optimizations. The Automated DBMS Fuzzing Framework is introduced as an innovative solution designed to intensify the detection of these bugs by implementing advanced fuzzing techniques directly tailored to PostgreSQL. This approach aims not only to enhance the security and reliability of PostgreSQL but also to streamline the process of testing and maintenance.

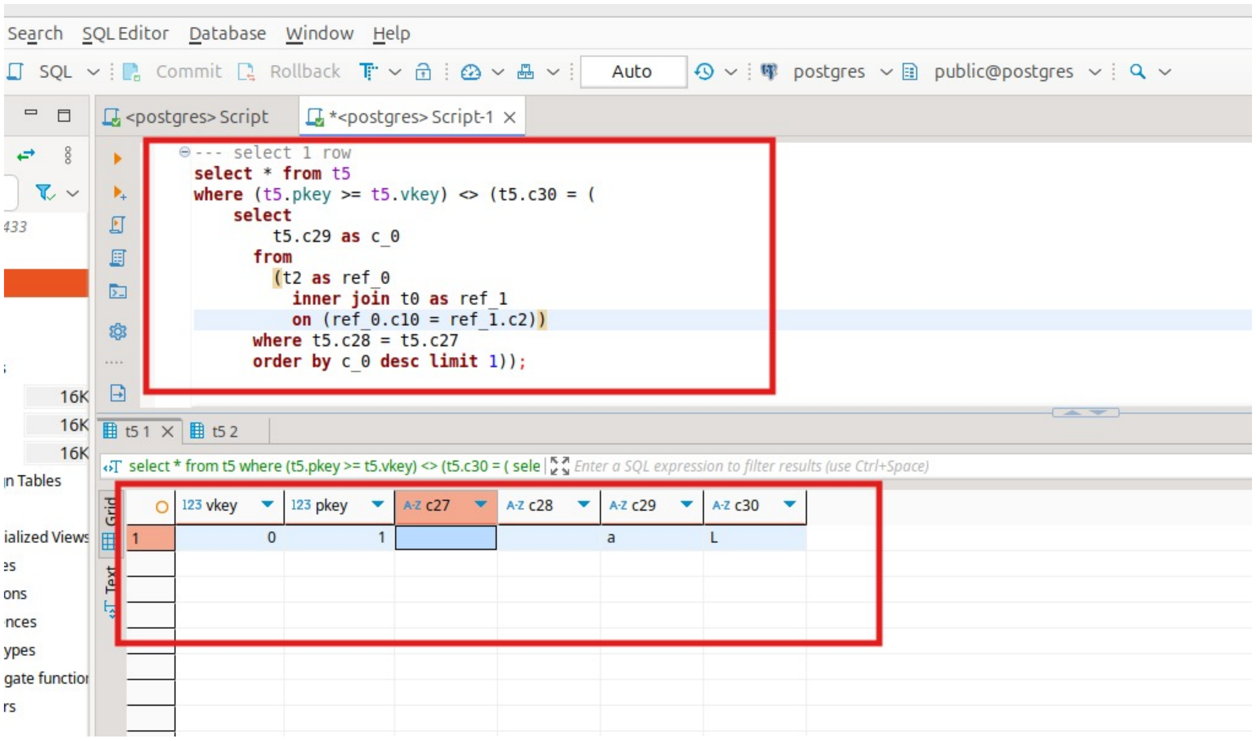
2 Related Works

The concept of testing database systems using synthetic SQL queries or fuzzing mechanisms isn't novel. Prior works such as SQLancer have made significant strides in this field by generating random SQL queries to test the logic correctness of RDBMS like SQLite, MySQL, and PostgreSQL [3]. More recently, the research presented in "Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation" extended this approach through the Equivalent Expression Transformation (EET) methodology, which systematically mutates expressions within SQL queries to test the underlying logic of database engines [1]. While these frameworks provided a foundation, they typically lacked deep integration with specific database optimizations and configurations, particularly those unique to PostgreSQL. Our framework builds upon these methodologies, introducing deeper integration and customized fuzzing strategies designed to exploit PostgreSQL's unique features and common use cases.

3 Problem and Solution

The complexity of PostgreSQL's query optimizer and execution engines introduces numerous challenges in bug detection. The Automated DBMS Fuzzing Framework addresses these by embedding customized fuzzing logic within the query processing modules, allowing for real-time manipulation and evaluation of SQL queries. This approach targets specific components most susceptible to bugs, significantly improving the detection rate and accuracy.

As we can see in Figure 1 and Figure 2, two queries are semantically equivalent and should be expected to generate the same result. But the second query in Figure 2, generates no results, whereas it should have returned one row similar to the query in Figure 1.



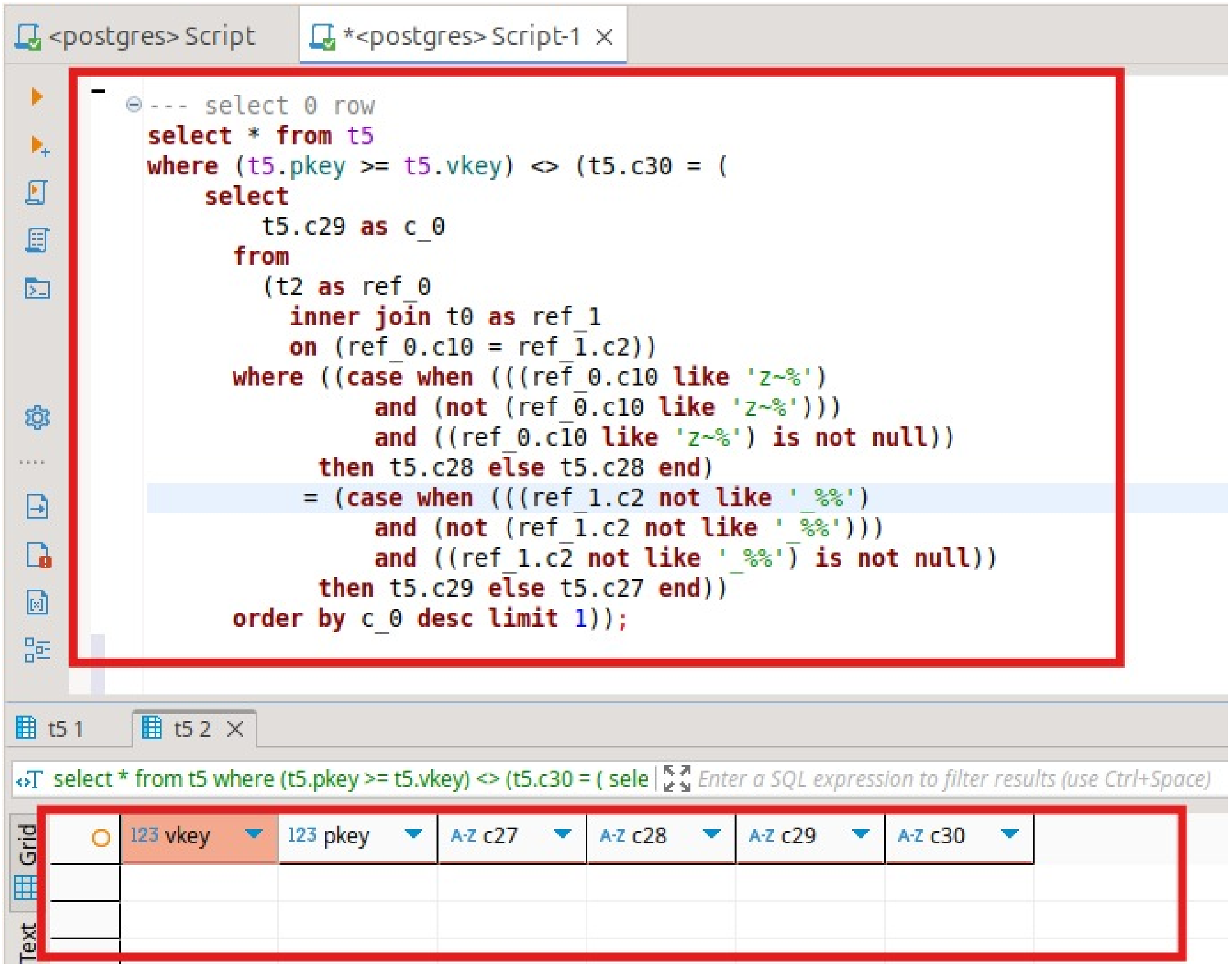
```
--- select 1 row
select * from t5
where (t5.pkey >= t5.vkey) <> (t5.c30 = (
    select
        t5.c29 as c_0
      from
        (t2 as ref_0
          inner join t0 as ref_1
          on (ref_0.c10 = ref_1.c2))
      where t5.c28 = t5.c27
      order by c_0 desc limit 1));
```

select * from t5 where (t5.pkey >= t5.vkey) <> (t5.c30 = (sele

Enter a SQL expression to filter results (use Ctrl+Space)

	vkey	pkey	c27	c28	c29	c30
1	0	1			a	L

Figure 1: Original Query



```sql
--- select 0 row
select * from t5
where (t5.pkey >= t5.vkey) <> (t5.c30 = (
    select
        t5.c29 as c_0
      from
        (t2 as ref_0
          inner join t0 as ref_1
          on (ref_0.c10 = ref_1.c2))
      where ((case when (((ref_0.c10 like 'z~%')
                and (not (ref_0.c10 like 'z~%')))
                and ((ref_0.c10 like 'z~%') is not null))
            then t5.c28 else t5.c28 end)
           = (case when (((ref_1.c2 not like '_%%')
                and (not (ref_1.c2 not like '_%%')))
                and ((ref_1.c2 not like '_%%') is not null))
            then t5.c29 else t5.c27 end))
      order by c_0 desc limit 1));
```

```
select * from t5 where (t5.pkey >= t5.vkey) <> (t5.c30 = ( sele
```

	vkey	pkey	c27	c28	c29	c30

Figure 2: Mutated Query

4 Summary of the Project

PostgreSQL's sophisticated query optimizer and execution engines, while powerful, introduce complexities that obscure potential logic bugs. These bugs might not cause immediate failures but can lead to incorrect query results or performance degradation under specific conditions. The Automated DBMS Fuzzing Framework for Postgres addresses this challenge by embedding itself within the query processing modules of PostgreSQL. This integration allows the framework to perform real-time mutation of SQL queries based on their execution plans, directly targeting the components that are most likely to harbor logic errors. By leveraging knowledge about PostgreSQL's behavior, the framework can generate more effective fuzzing scenarios, thereby improving the likelihood of uncovering significant bugs.

5 Methodology

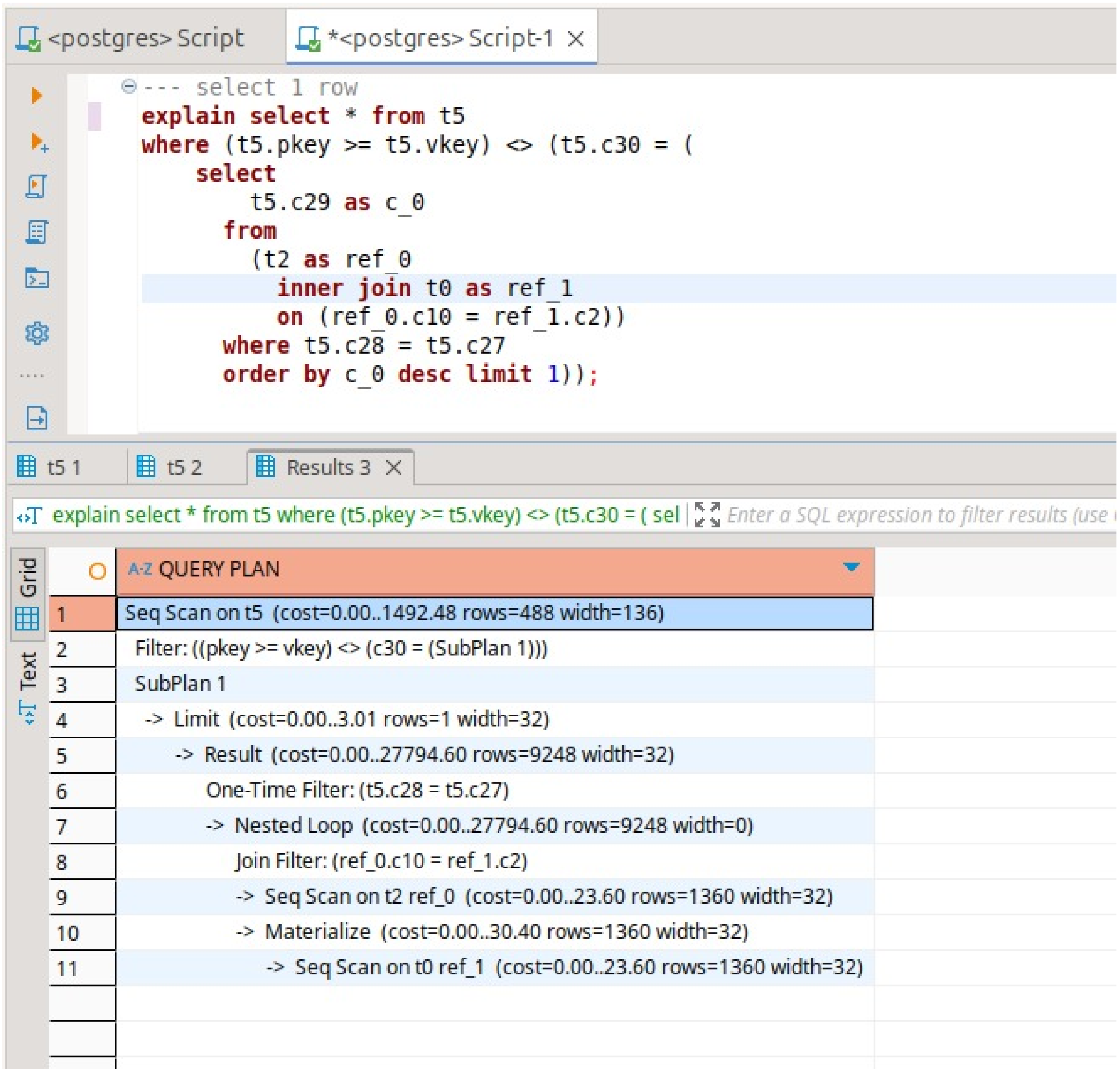
Our methodology involves several key components:

5.1 Query Capture and Transformation:

SQL queries are captured before execution and subjected to transformations that should not alter the expected results.

5.2 Execution Plan Analysis:

The execution plans of both original and transformed queries are analyzed. Differences in execution plans provide insights into potential optimizer bugs.

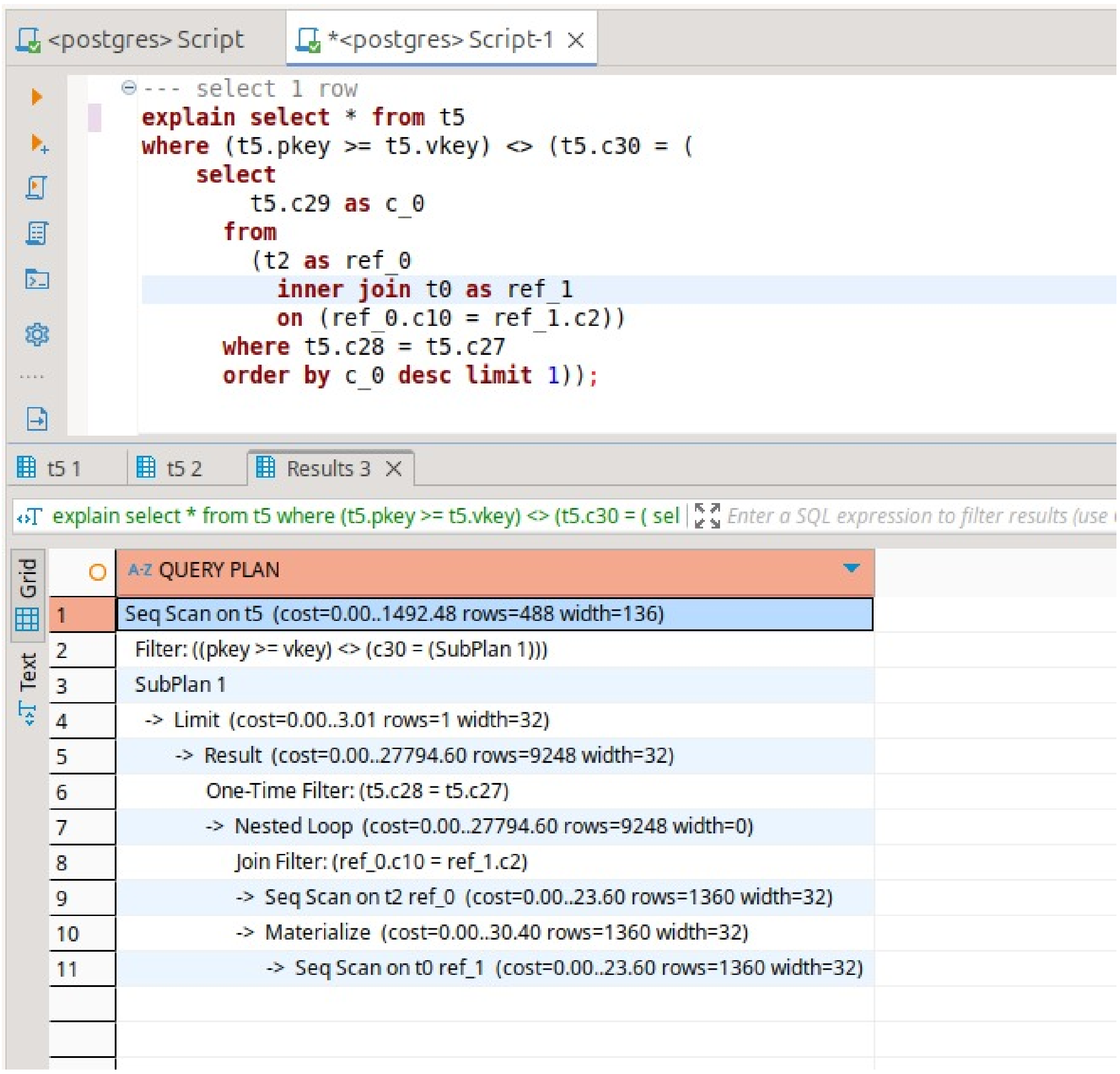


```
--- select 1 row
explain select * from t5
where (t5.pkey >= t5.vkey) <> (t5.c30 = (
    select
        t5.c29 as c_0
      from
        (t2 as ref_0
          inner join t0 as ref_1
          on (ref_0.c10 = ref_1.c2))
      where t5.c28 = t5.c27
      order by c_0 desc limit 1));
```

```
explain select * from t5 where (t5.pkey >= t5.vkey) <> (t5.c30 = ( sel
```

	QUERY PLAN
1	Seq Scan on t5 (cost=0.00..1492.48 rows=488 width=136)
2	Filter: ((pkey >= vkey) <> (c30 = (SubPlan 1)))
3	SubPlan 1
4	-> Limit (cost=0.00..3.01 rows=1 width=32)
5	-> Result (cost=0.00..27794.60 rows=9248 width=32)
6	One-Time Filter: (t5.c28 = t5.c27)
7	-> Nested Loop (cost=0.00..27794.60 rows=9248 width=0)
8	Join Filter: (ref_0.c10 = ref_1.c2)
9	-> Seq Scan on t2 ref_0 (cost=0.00..23.60 rows=1360 width=32)
10	-> Materialize (cost=0.00..30.40 rows=1360 width=32)
11	-> Seq Scan on t0 ref_1 (cost=0.00..23.60 rows=1360 width=32)

Figure 3: Execution Plan for Original Query



```
--- select 1 row
explain select * from t5
where (t5.pkey >= t5.vkey) <> (t5.c30 = (
    select
        t5.c29 as c_0
      from
        (t2 as ref_0
          inner join t0 as ref_1
          on (ref_0.c10 = ref_1.c2))
      where t5.c28 = t5.c27
      order by c_0 desc limit 1));
```

explain select * from t5 where (t5.pkey >= t5.vkey) <> (t5.c30 = (sel

```
QUERY PLAN
1   Seq Scan on t5  (cost=0.00..1492.48 rows=488 width=136)
2     Filter: ((pkey >= vkey) <> (c30 = (SubPlan 1)))
3     SubPlan 1
4       ->  Limit  (cost=0.00..3.01 rows=1 width=32)
5             ->  Result  (cost=0.00..27794.60 rows=9248 width=32)
6                   One-Time Filter: (t5.c28 = t5.c27)
7                   ->  Nested Loop  (cost=0.00..27794.60 rows=9248 width=0)
8                         Join Filter: (ref_0.c10 = ref_1.c2)
9                         ->  Seq Scan on t2 ref_0  (cost=0.00..23.60 rows=1360 width=32)
10                        ->  Materialize  (cost=0.00..30.40 rows=1360 width=32)
11                              ->  Seq Scan on t0 ref_1  (cost=0.00..23.60 rows=1360 width=32)
```

Figure 4: Execution Plan for Mutated Query

5.3 Result Comparison:

Results from the original and transformed queries are compared. Inconsistencies trigger alerts for potential bugs.

5.4 Automated Bug Reporting:

Bugs are automatically reported with detailed logs including the query, the transformation applied, the expected results, and the actual results.

5.5 System Performance Analysis:

The psutil library will be used to retrieve information on system utilization (like CPU and memory usage). In the framework, it will be used to monitor and log the resource usage of the host machine while the fuzzing operations are being performed.

6 Current Progress

We started with validating if the bugs existed, so we regenerated a prevailing bug in the PostgresSQL engine. Since we needed linux environment to test the previous version of postgresSQL that contained the bug, we used VirtualBox as a hypervisor to host the latest Ubuntu 24.04.2 LTS. After necessary git installation, docker initialization and various packages installations, we cloned the opensource PostgreSQL code and switched to a bit older and a stable version, specifically at commit 3f1aaaa. On top of the commit, we built a containerized PostgreSQL application and successfully validated the existence of a bug in the code, as evident in Figure 1 and Figure 2.

After validating our hypothesis, we are implementing a robust framework that will automatically detect those kinds of bugs and report them so they can be tackled. Here is the link to the framework repository: Git Repository

7 Expected Delivery

We aim to deliver a robust automated DBMS fuzzing framework designed to detect logic errors and inconsistencies in query execution. The final deliverables will include automated query mutation and execution, detailed performance and bug reports, a containerized testing environment, and a system performance analysis.

References

1. Jiang, Z.-M., & Su, Z. (2024). Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation* (OSDI '24), Santa Clara, CA, USA.

2. Tschudin, M. F., & Braendli, M. (2021). SQLancer: Detecting Logic Bugs in DBMS. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA '21).

3. PostgreSQL Global Development Group. (n.d.). PostgreSQL 9.6 Documentation. Retrieved from `https://www.postgresql.org/docs/9.6/static/`.