Demonstration of the VeriEQL Equivalence Checker for Complex SQL Queries

Pinhan Zhao[*]
pinhan@umich.edu
University of Michigan
Ann Arbor, MI, USA

Yang He[*]
yha244@sfu.ca
Simon Fraser University
Burnaby, BC, Canada

Xinyu Wang
xwangsd@umich.edu
University of Michigan
Ann Arbor, MI, USA

Yuepeng Wang
yuepeng@sfu.ca
Simon Fraser University
Burnaby, BC, Canada

ABSTRACT

Equivalence checking for SQL queries has many real-world applications but typically requires supporting an expressive SQL language in order to be practical. We develop VeriEQL, a system that can prove and disprove equivalence of *complex* SQL queries. Specifically, given two SQL queries under a database schema, VeriEQL can verify whether these two queries always produce identical results on all possible input databases up to a bounded size that conform to the schema. This paper demonstrates VeriEQL in three scenarios, including validating the correctness of query optimizations, grading SQL queries on online coding platforms, and finding implementation bugs in database management systems.

PVLDB Reference Format:
Pinhan Zhao, Yang He, Xinyu Wang, and Yuepeng Wang. Demonstration of the VeriEQL Equivalence Checker for Complex SQL Queries. PVLDB, 17(12): 4437 - 4440, 2024.
doi:10.14778/3685800.3685894

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/whatsmyname/VeriEQL-demo.

1 INTRODUCTION

SQL, the de facto standard query language for relational databases, has been broadly studied in the databases community and is well-supported by relational database engines [6]. Equivalence checking of SQL queries has many real-world applications, such as validating rewrites for query optimization [13], finding bugs in database management systems [10], and grading SQL queries automatically [2].

Motivated by these real-world applications, we developed a tool called VeriEQL that aims to prove and disprove equivalence of SQL queries automatically. At a high level, VeriEQL takes as input two SQL queries Q_1, Q_2 over schema $\mathcal{S}$, and checks if Q_1 and Q_2 are semantically equivalent for a space of inputs. If VeriEQL identifies an input database where Q_1 and Q_2 produce different results, we can safely conclude that Q_1 and Q_2 are not equivalent with this input being a counterexample. Otherwise, we prove that Q_1 is equivalent to Q_2 for any input database in the entire space. Internally, VeriEQL utilizes a symbolic reasoning approach [9]. It constructs symbolic input databases and computes the symbolic outputs of Q_1 and Q_2, through a rigorous encoding of query semantics using satisfiability modulo theories (SMT). This enables us to reduce the equivalence checking problem into an SMT problem and resort to off-the-shelf constraint solvers to determine the satisfiability of SMT formulas.

Expressive query language. VeriEQL supports a wide variety of SQL operations. In addition to selection, projection, inner join, outer joins, `GROUP BY`, and aggregate functions, VeriEQL also supports `WITH` clauses, `IF`, `CASE WHEN`, `ORDER BY`, `LIMIT`, set/bag union, intersection, minus, and three-valued semantics involving `NULL`'s. Many of these operations, such as `WITH`, `ORDER BY`, `LIMIT`, or their realistic combinations, are not supported by existing equivalence checkers. To the best of our knowledge, VeriEQL supports the most expressive query language compared to all prior work such as Cosette [3, 4], SPES [13], and SQLSolver [8]. Neither Cosette nor SPES can reason equivalence involving `ORDER BY`. SQLSolver cannot support conditional statements such as `IF` and `CASE WHEN`. Our experimental results [9] showed that VeriEQL can prove or disprove over 75% of a large benchmark suite with more than 24,000 query pairs. It significantly outperforms prior work such as Cosette and SPES, which support 0.2% and 1.2% of the SQL queries from the benchmark suite, respectively.

Genuine counterexample. To provide firm evidence when two SQL queries are not equivalent, VeriEQL can generate a counterexample disproving the equivalence based on the result of its symbolic reasoning. The counterexample consists of concrete input tables under the given schema and is guaranteed to be *genuine*. In other words, (1) the tables always satisfy the integrity constraints defined in the schema, and (2) executing two provided SQL queries on those input tables guarantees to produce different results.

Scalability and Small-Scope Hypothesis. Thanks to the symbolic reasoning techniques that only involve simple and decidable first-order theories such as theory of integers and uninterpreted functions, VeriEQL can scale to a moderate-size symbolic input database for equivalence checking. Specifically, for 70% of the 15,200

[*]Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685894

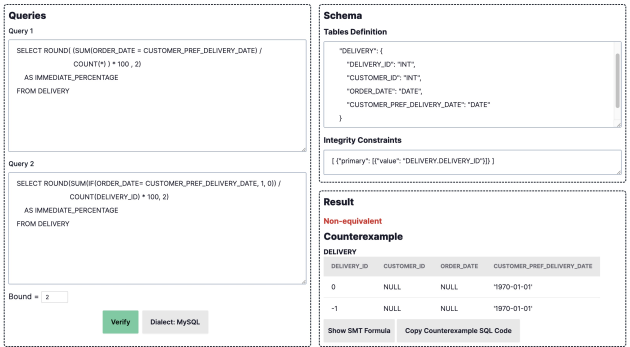

Figure 1: Graphical interface of the VeriEQL tool.

benchmarks where counterexamples are not identified, VeriEQL can check equivalence on databases with each table containing 5 symbolic tuples within 10 minutes [9]. According to the small-scope hypothesis reported in prior work [11], "mistakes in most of the queries may be explained by only a small number of tuples." Our experience with VeriEQL is consistent with the small-scope hypothesis. In particular, among 3,619 non-equivalent benchmarks, 96% of them have counterexample tables with less than 3 tuples.

Graphical interface. For better usability, VeriEQL provides a graphical interface, illustrated in Figure 1. On the left, the user can provide two SQL queries Q_1 and Q_2 for equivalence checking in a standard SQL language. The interface also provides a text box for users to specify the maximum size of input tables they want VeriEQL to check against. On the right, the user can specify the schema $\mathcal{S}$, including a complete description of table definitions and integrity constraints (such as primary keys, foreign keys, nullability, and value ranges). After clicking the Verify button, VeriEQL performs symbolic reasoning (explained in more detail in Section 2) to determine if Q_1 and Q_2 are equivalent given the maximum table size and shows the result to the user. If Q_1, Q_2 are not equivalent, VeriEQL also displays a set of concrete tables under schema $\mathcal{S}$ as the counterexample that refutes the equivalence.

Supporting Different SQL Dialects. To be compatible with a wide variety of academic and industrial settings, VeriEQL considers its SQL semantics based on four popular dialects, namely MySQL, MariaDB, Oracle, and PostgreSQL. It also features a switch for users to specify the database management system and select the corresponding SQL dialect.

Demonstration details. We demonstrate the VeriEQL tool in three scenarios: (1) validating the correctness of query rewrites, (2) automated grading of SQL queries from online coding platforms, and (3) identifying bugs in database management systems. These scenarios highlight VeriEQL's capability of proving equivalence of complex SQL queries and disproving the equivalence with genuine counterexamples. During the demonstration, we will present how to use the graphical interface of VeriEQL, illustrate the workflow of symbolic reasoning with concrete SQL queries, and explain how to interpret the verification results in different usage scenarios.

2 SYSTEM OVERVIEW

The schematic workflow of VeriEQL is shown in Figure 2. At a high level, the VeriEQL system consists of four modules: input analyzer, semantics encoder, equality checker, and counterexample generator. In what follows, we describe these modules in more detail.

Input analyzer. Given two SQL queries Q_1, Q_2 and their database schema $\mathcal{S}$ with the corresponding integrity constraint C, VeriEQL first checks that all these inputs are well-formed and there is no syntactic error. Then it performs a conformance check to validate that Q_1 and Q_2 are consistent with the schema $\mathcal{S}$, i.e., all tables and columns used in Q_1, Q_2 exist in $\mathcal{S}$ and their types match the declarations in $\mathcal{S}$. In case of errors, VeriEQL terminates and presents the error message to the user.

Semantics encoder. To reason about the equivalence between Q_1 and Q_2, VeriEQL first builds a *symbolic database* Γ that satisfies the integrity constraint C based on schema $\mathcal{S}$, and then it performs symbolic execution to obtain the symbolic results R_1, R_2 of Q_1, Q_2 over Γ. The symbolic database is a set of symbolic tables. Each table consists of a list of N symbolic tuples, where a variable represents each symbolic tuple. To ensure that Γ satisfies the integrity constraint C, VeriEQL encodes C as an SMT formula Φ_C over variables in Γ. For example, consider a database that only contains one table `EMP(id, age)` and $N = 2$, we can create two symbolic tuples t_1 and t_2 for `EMP` in the symbolic database Γ, where t_1, t_2 are variables. If the integrity constraint requires the age to be positive, we can encode it and obtain $\Phi_C : t_1.\texttt{age} > 0 \land t_2.\texttt{age} > 0$.

To perform symbolic execution, VeriEQL faithfully encodes the relations between inputs and outputs of each SQL operation as an SMT formula. By composing the formulas of all operations in a query Q_i, we can obtain the formula Φ_{R_i} describing the result R_i. To understand the encoding, let us continue with the `EMP` example and consider the following query:

```
SELECT id FROM EMP WHERE age > 30
```

Suppose the result R has tuples t'_1 and t'_2, the formula Φ_R is

$$(t_1.\texttt{age} > 30 \rightarrow \neg\mathrm{Del}(t'_1) \land t'_1.\texttt{id} = t_1.\texttt{id}) \land (t_1.\texttt{age} \leq 30 \rightarrow \mathrm{Del}(t'_1)) \\ \land (t_2.\texttt{age} > 30 \rightarrow \neg\mathrm{Del}(t'_2) \land t'_2.\texttt{id} = t_2.\texttt{id}) \land (t_2.\texttt{age} \leq 30 \rightarrow \mathrm{Del}(t'_2))$$

where $\mathrm{Del}(t)$ is an uninterpreted function denoting whether the tuple t is deleted or not.

Equality checker. After obtaining the query results and formulas from symbolic execution, VeriEQL needs to decide if the two query results R_1, R_2 are always equal for any symbolic database Γ conforming to the schema $\mathcal{S}$ (and the integrity constraint C). The key idea is to perform a symbolic search to find a concrete database D such that the query results R_1 and R_2 are unequal under the bag semantics. In particular, VeriEQL builds an SMT formula $\Phi_C \land \Phi_{R_1} \land \Phi_{R_2} \land R_1 \neq R_2$ and checks its satisfiability using the Z3 SMT solver [7]. Intuitively, the formula asserts that the database D satisfies the integrity constraint encoded by Φ_C, but the result R_1 is not equal to R_2. If the formula is unsatisfiable, such a database D does not exist under the given size bound, so Q_1 and Q_2 are (bounded) equivalent. Otherwise, if the formula is satisfiable, Q_1 is not equivalent to Q_2.

Counterexample generator. Where Q_1 and Q_2 are not equivalent, VeriEQL generates a counterexample database as the evidence. Specifically, VeriEQL obtains a model of the formula $\Phi_C \land \Phi_{R_1} \land \Phi_{R_2} \land R_1 \neq R_2$ that maps each variable in the formula to a concrete

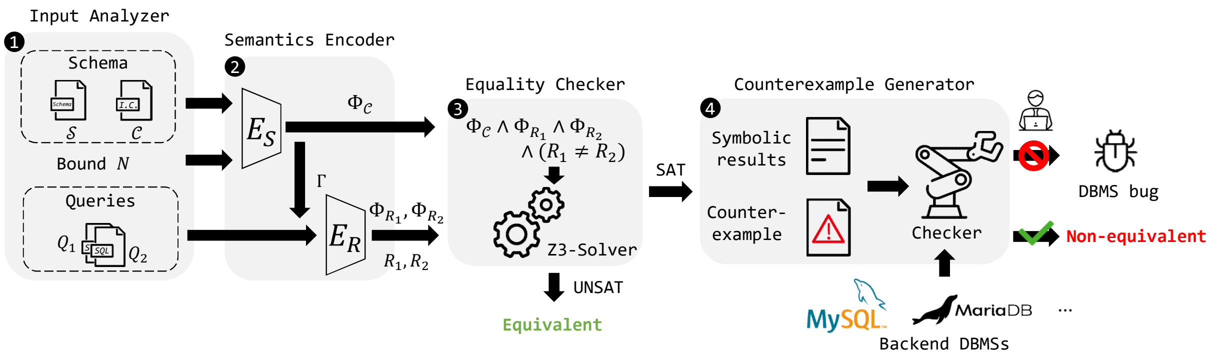

Figure 2: The schematic workflow of VeriEQL.

Q_1	SELECT DEPTNO, COUNT (*) FILTER (WHERE JOB = 'CLERK') FROM (SELECT * FROM EMP WHERE DEPTNO = 10 UNION ALL SELECT * FROM EMP WHERE DEPTNO > 20) AS t3 GROUP BY DEPTNO
Q_2	SELECT DEPTNO, COALESCE (SUM (EXPR$1), 0) FROM (SELECT DEPTNO, COUNT (*) FILTER (WHERE JOB = 'CLERK') AS EXPR$1 FROM EMP WHERE DEPTNO = 10 GROUP BY DEPTNO UNION ALL SELECT DEPTNO, COUNT (*) FILTER (WHERE JOB = 'CLERK') AS EXPR$1 FROM EMP WHERE DEPTNO > 20 GROUP BY DEPTNO) AS t12 GROUP BY DEPTNO

Figure 3: The optimized and original queries adapted from the `testPushCountFilterThroughUnion` test case of Calcite.

value. Based on the values in the model, VeriEQL can simply follow the schema to build the counterexample database. To further confirm that the counterexample is genuine, VeriEQL also executes queries Q_1 and Q_2 on the counterexample using different DBMSs, such as MySQL and MariaDB. If the query results are indeed different, VeriEQL returns the counterexample database to the user. Otherwise, it sends an alert to the user and requests manual inspection because such a case is a good indication of implementation bugs in the DBMS.

3 DEMONSTRATION SCENARIOS

We demonstrate three real-world scenarios where VeriEQL is useful for proving and disproving equivalence of SQL queries. In general, users can interact with VeriEQL in the following steps: (1) fill in the text boxes with SQL queries and database schemas, (2) specify a bound size for symbolic tables, (3) select a SQL dialect in the drop-down menu to which the queries conform, and (4) click on the Verify button to obtain the equivalence checking result.

3.1 Validating Query Optimizations

Query optimization constantly happens in relational database management systems. One essential and central requirement of query optimization is to ensure that the optimized query is equivalent to the original query. We demonstrate that VeriEQL can help validate the correctness of query optimizations by checking the equivalence between optimized and original queries.

Specifically, the user can provide a SQL query and its optimized version as input to VeriEQL and ask to check their equivalence. As a demonstration, we have collected a pair of queries from the test suite of Apache Calcite [1], which is a framework for query optimization. As shown in Figure 3, the queries Q_1, Q_2 are complex and use many operations such as `COALESCE`, `GROUP BY`, `UNION ALL`,

Table 1: Time to check query equivalence on different input sizes for validating query optimizations.

Size	1	2	3	4	5	6	7	8	9
Time (s)	0.2	0.4	0.6	1.0	2.4	6.6	19.7	98.5	118.2

aggregate functions, etc. Furthermore, query Q_2 is significantly different from Q_1 syntactically, because it pushes the count filter through the `UNION ALL`. Despite the challenges, VeriEQL can prove that Q_2 is equivalent to Q_1 for any input tables up to size 4 within one second. Furthermore, as shown in Table 1, VeriEQL can prove equivalence of Q_1 and Q_2 up to input size 9 in 2 minutes.

3.2 Grading Queries on Coding Platforms

Existing online coding platforms such as `LeetCode` grade the submission through a small number of test cases, which typically suffers from the low coverage issue. We demonstrate that VeriEQL can be used to check the correctness of a submission to SQL programming questions. Specifically, a submission is considered correct if VeriEQL concludes that the submission and a standard solution are equivalent. If they are not equivalent, VeriEQL returns a counterexample to help the programmer understand the mistake.

In general, the coding platform can provide the standard solution and a user-submitted query of a SQL programming question as input and ask VeriEQL to check their equivalence. As a concrete example to demonstrate this usage scenario, we crawled user-submitted queries and the ground-truth solution of `LeetCode` problems. Figure 4a shows such a pair of queries, where Q_1 is a query submitted by a user and Q_2 is the solution. The corresponding `LeetCode` problem asks to write a query that finds all customers who bought *both* products A and B but did not buy product C. Here, the solution Q_2 uses two `IN` subqueries to find customers who bought A and B and uses a `NOT IN` subquery to ensure the customer did not buy C. However, the user-submitted query Q_1 uses aggregation functions to compute the number of products A *or* B and the number of product C bought by each customer, and then returns the customers accordingly. Q_1 is not correct, because it may return a customer who bought two A's or two B's but not both.

VeriEQL can conclude query Q_1 is incorrect by disproving the equivalence between Q_1, Q_2 and provide a counterexample database as shown in Figure 4b. Running Q_1 on the database produces an output in Figure 4c, but running Q_2 returns an empty table.

Q_1:

```
WITH temp AS (SELECT DISTINCT A.customer_id, B.customer_name,
            SUM (CASE WHEN A.product_name IN ('A', 'B') THEN 1 ELSE 0 END ) AS AB,
            SUM (CASE WHEN A.product_name = 'C' THEN 1 ELSE 0 END ) AS C,
      FROM orders A JOIN customers B ON A.customer_id = B.customer_id
      GROUP BYA.customer_id )
SELECT customer_id, customer_name FROM temp WHERE AB >= 2 AND C = 0
```

Q_2:

```
SELECT customer_id, customer_name FROM customers
WHERE customer_id IN (
            SELECT DISTINCT customer_id FROM orders WHERE product_name = 'A'
      ) AND customer_id IN (
            SELECT DISTINCT customer_id FROM orders WHERE product_name = 'B'
      ) AND customer_id NOT IN (
            SELECT DISTINCT customer_id FROM orders WHERE product_name = 'C'
) ORDER BY customer_id
```

(a) User-submitted and solution queries from a `LeetCode`.

`orders`

order_id	customer_id	product_name
0	0	B
1	0	B

`customers`

customer_id	customer_name
0	Alice
1	Bob

(b) Counterexample database.

id	name
0	Alice

(c) Q_1's output.

Figure 4: Two non-equivalent queries from `LeetCode`.

Although VeriEQL can check equivalence of expressive queries, it cannot partially grade queries or pinpoint incorrect statements. We leave the extension to support such features as future work.

3.3 Finding Bugs in DBMSs

We also demonstrate that VeriEQL can reveal bugs in the optimizer of DBMSs. Figure 5a shows a pair of user-submitted queries and the solution collected from another `LeetCode` problem[1]. Here, Q_1 and Q_2 are not equivalent because the `LEFT JOIN` in Q_2 preserves all tuples from the `friendship` table, whereas Q_1 uses `JOIN`, which only retains those tuples that satisfy the join predicate. VeriEQL, correspondingly, disproves the equivalence between Q_1, Q_2 and thus concludes that the user-submitted query is incorrect. As evidence, it also generates a counterexample database D, as shown in Figure 5b. Running the query Q_1 on D yields an empty table, but running Q_2 on D should return the table shown in Figure 5c.

However, the counterexample generator of VeriEQL alerts that Q_2 produces an empty table on D in MySQL when it tries to validate that D is a genuine counterexample. The result is different from the expected output from symbolic reasoning. After manual inspection, we found an implementation bug in MySQL's latest release version (v8.0.32). The MySQL maintenance team has confirmed this is a bug at the *serious* severity level. This example also demonstrates VeriEQL's capability of finding previously unknown bugs in database management systems.

4 RELATED WORK

Various approaches, such as formal methods, testing, and neural reasoning, have been proposed to validate SQL equivalence. VeriEQL takes a formal method approach. Compared to testing [2, 5] and neural reasoning [12] techniques that cannot provide formal guarantees, VeriEQL can guarantee that no tables up to a certain size can distinguish the input queries if they are checked to be equivalent. Other formal methods [3, 4, 8, 13] can provide similarly bounded or full correctness guarantees. However, full-fledged

[1]https://leetcode.com/problems/page-recommendations/

Q_1:

```
SELECT DISTINCT page_id AS recommended_page
FROM (SELECT CASE WHEN user1_id = 1 THEN user2_id WHEN user2_id = 1
                  THEN user1_id ELSE NULLEND AS user_id FROM friendship)
      AS tb1 JOIN likes AS tb2 ON tb1.user_id = tb2.user_id
WHERE page_id NOT IN (SELECT page_id FROM likes WHERE user_id = 1)
```

Q_2:

```
SELECT DISTINCT page_id AS recommended_page
FROM ( SELECT b.user_id, b.page_id FROM friendship a LEFT JOIN likes b
      ON (a.user2_id = b.user_id OR a.user1_id=b.user_id)
         AND (a.user1_id = 1 OR a.user2_id = 1)
      WHERE b.page_id NOT IN (
                  SELECT DISTINCT (page_id) FROM likes WHERE user_id=1) ) T
```

(a) User-submitted and solution queries from `LeetCode`.

`friendship`

user1_id	user2_id
0	1

`likes`

user_id	page_id
-1	0

(b) Counterexample database.

page_id
NULL

(c) Q_2's expected output.

Figure 5: Queries from `LeetCode` that reveal the MySQL bug.

equivalence verification tools such as SPES [13] cannot provide counterexamples to refute the equivalence. Among all prior work, the most related is the Cosette [3, 4] bounded equivalence checker. However, Cosette works on none of the three examples in Section 3, because it lacks support for conditional expressions such as `COALESCE` and `CASE WHEN`. To the best of our knowledge, VeriEQL supports the most expressive query language among all bounded equivalence checking tools for SQL queries.

ACKNOWLEDGMENT

We thank reviewers for the insightful comments and feedback. This research is supported by the NSERC Discovery Grant and the NSF Grant No. CCF-2210832.

REFERENCES

[1] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*. 221–230.

[2] Bikash Chandra, Bhupesh Chawda, Biplab Kar, KV Maheshwara Reddy, Shetal Shah, and S Sudarshan. 2015. Data generation for testing and grading SQL queries. *The VLDB Journal* 24, 6 (2015), 731–755.

[3] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. 2017. Demonstration of the Cosette Automated SQL Prover. In *SIGMOD*. 1591–1594.

[4] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.

[5] F Coelho. 2014. DataFiller–generate random data from database schema.

[6] Chris J Date. 1989. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc.

[7] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340.

[8] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *SIGMOD* 1, 4 (2023), 1–26.

[9] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proc. ACM Program. Lang.* OOPSLA (2024).

[10] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *ICSE*. 225–236.

[11] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining wrong queries using small examples. In *SIGMOD*. 503–520.

[12] Fuheng Zhao, Lawrence Lim, Ishtiyaque Ahmad, Divyakant Agrawal, and Amr El Abbadi. 2023. LLM-SQL-Solver: Can LLMs Determine SQL Equivalence? *arXiv preprint arXiv:2312.10321* (2023).

[13] Qi Zhou, Joy Arulraj, Shamkant B Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A symbolic approach to proving query equivalence under bag semantics. In *ICDE*. 2735–2748.