VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints

YANG HE*, Simon Fraser University, Canada
PINHAN ZHAO*, University of Michigan, USA
XINYU WANG, University of Michigan, USA
YUEPENG WANG, Simon Fraser University, Canada

The task of SQL query equivalence checking is important in various real-world applications (including query rewriting and automated grading) that involve *complex queries with integrity constraints*; yet, state-of-the-art techniques are very limited in their capability of reasoning about complex features (e.g., those that involve sorting, case statement, rich integrity constraints, etc.) in real-life queries. To the best of our knowledge, we propose the first SMT-based approach and its implementation, VeriEQL, capable of proving and disproving bounded equivalence of *complex* SQL queries. VeriEQL is based on a new logical encoding that models query semantics over symbolic tuples using the theory of integers with uninterpreted functions. It is *simple yet highly practical* — our comprehensive evaluation on over 20,000 benchmarks shows that VeriEQL outperforms *all* state-of-the-art techniques by *more than one order of magnitude* in terms of the number of benchmarks that can be proved or disproved. VeriEQL can also generate counterexamples that facilitate many downstream tasks (such as finding serious bugs in systems like MySQL and Apache Calcite).

CCS Concepts: • **Theory of computation** → **Program verification**; • **Software and its engineering** → **Software verification**; **Formal software verification**.

Additional Key Words and Phrases: Program Verification, Equivalence Checking, Relational Databases.

ACM Reference Format:
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.* 8, OOPSLA1, Article 132 (April 2024), 29 pages. https://doi.org/10.1145/3649849

1 INTRODUCTION

Equivalence checking of SQL queries is an important problem with various real-world applications, including validating source-level query rewriting [Chu et al. 2017b; Graefe 1995] and automated grading of SQL queries [Chandra et al. 2019]. A useful SQL equivalence checker should be able to (i) provide formal guarantee on query equivalence (either fully or in a bounded manner), (ii) generate counterexamples to witness query non-equivalence, and (iii) support an expressive query language. For example, in the context of query rewriting where a slow query Q_1 is rewritten to a faster query Q_2 using rewrite rules, one may want to ensure the rewrite is correct by showing Q_1 and Q_2 are semantically equivalent with a certain level of formal guarantee, and in case of non-equivalence, obtain a counterexample input database to help fix the incorrect rule. Equivalence checking is also

*Both authors contributed equally to the paper.

Authors' addresses: Yang He, Simon Fraser University, Burnaby, Canada, yha244@sfu.ca; Pinhan Zhao, University of Michigan, Ann Arbor, USA, pinhan@umich.edu; Xinyu Wang, University of Michigan, Ann Arbor, USA, xwangsd@umich.edu; Yuepeng Wang, Simon Fraser University, Burnaby, Canada, yuepeng@sfu.ca.

ACM 2475-1421/2024/4-ART132
https://doi.org/10.1145/3649849

useful for automated query grading. In particular, it can provide feedback to users by checking their submitted queries against a ground-truth query, where the feedback could be a counterexample (i.e., a concrete database) that illustrates why the user query is wrong. Moreover, these counterexamples can also serve as additional test cases to augment an existing test suite (such as the one maintained by LeetCode [LeetCode 2023], the world's most popular online programming platform).

While prior work [Chu et al. 2017a, 2018, 2017b,c; Veanes et al. 2010; Wang et al. 2018a] has made some advances in both proving and disproving query equivalence, there remain a number of challenges that significantly limit the practical usage of existing techniques in real-world applications. The gap, as we will also show in our evaluation section later, is in fact *extremely large*: for example, existing work supports less than 2% of the SQL queries from LeetCode. The reasons are threefold.

- First, real-world queries are complex. In addition to the simplest select-project-join queries using common aggregate functions such as `SUM`, `COUNT`, `MAX` — that existing techniques support fairly well — queries in the wild frequently use advanced SQL features with much more complex logic (such as sorting, case statement, common table expressions, `IN` and `NOT IN` operators, etc.) which, to our best knowledge, existing SQL equivalence checkers rarely support. For example, among more than 20,000 real-life queries we studied, more than 20% of them involve `ORDER BY`, over 30% have `CASE WHEN`, over 15% require `IN` or `NOT IN`, and 15% use `WITH`, among others. This brings up new challenges in how to precisely model the semantics of these advanced SQL operators.
- Furthermore, the interleaving between these advanced operators and other SQL features (such as three-valued logic, aggregate functions, grouping, etc.) makes the problem even more challenging. For example, the three-valued logic uses a special value `NULL` that many existing techniques (such as HoTTSQL [Chu et al. 2017c]) do not consider. We must take into account all these additional SQL features to properly model the semantics of SQL operators in order to fully support complex real-world queries and reason about query equivalence.
- Finally, most real-world queries involve integrity constraints, but prior work barely supports them. For instance, over 95% of our benchmarks require integrity constraints, yet all existing techniques *cumulatively* support under 2%, to our best knowledge. These constraints are rich: in addition to the primary key and foreign key constraints that stipulate uniqueness and value references, there are many others including `NOT NULL` (used by *all* queries in our `Calcite` benchmark suite) and various constraints that restrict attribute values (>70% across all our benchmarks) such as requiring an attribute to be only positive integers. This richness poses significant challenges: for example, to witness the non-equivalence of two queries, a valid counterexample must not only yield different outputs *but also* meet the integrity constraints.

In this paper, we propose a *simple yet practical* approach to SQL query equivalence checking — we can prove equivalence (in a bounded fashion) and non-equivalence (by generating counterexamples) for *a complex query language with rich integrity constraints.* Our key contribution is a *new SMT encoding* tailored towards bounded equivalence verification, based on a *new semantics formalization* for a practical fragment of SQL (which is *significantly larger* than those considered in prior work). First, we formalize our SQL semantics using list and higher-order functions, different from the K-relations in HoTTSQL [Chu et al. 2017c] or U-semiring in UDP [Chu et al. 2018]. Our formalization is inspired by Mediator [Wang et al. 2018b]. However, Mediator considers unbounded equivalence verification for a small set of SQL queries, while our formalization considers a much larger language. Second, building upon the standard approach of using SMT formulas to relate the program inputs and outputs, we develop a *new SMT encoding*, for our formalized query semantics, in the theory of integers and uninterpreted functions that is based on symbolic tuples. This new SMT encoding allows us to handle complex SQL features (e.g., `ORDER BY`, `NULL`, etc) and rich integrity constraints without unnecessarily heavy theories such as theory of lists in Qex [Veanes et al. 2010]. It also does

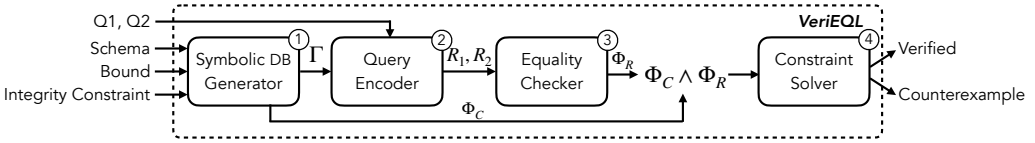


Fig. 1. Schematic workflow of VeriEQL.

not require an indirect encoding by translating the SQL queries to an intermediate representation in solver-aided programming language (e.g., Rosette [Torlak and Bodík 2014]), unlike the previous Cosette line of work [Chu et al. 2017a,b; Wang et al. 2018a]. Last but not least, we provide detailed correctness proofs of our SMT encoding with respect to our formal semantics.

VeriEQL. We have implemented our approach in VeriEQL, which is described schematically in Figure 1. At a high-level, VeriEQL takes as input a pair of queries (Q_1, Q_2), the database schema $\mathcal{S}$ and its integrity constraint C, and a bound N defining the input space. It constructs an SMT formula Φ such that: (i) if Φ is unsatisfiable, then Q_1 and Q_2 are guaranteed to be equivalent for all database relations with at most N tuples, and (ii) if Φ is satisfiable, then Q_1 and Q_2 are provably non-equivalent, and we generate a counterexample (i.e., a database that meets C but leads to different query outputs) from Φ's satisfying assignments. Internally, ① VeriEQL first creates a symbolic representation Γ of all input databases with up to N tuples in their relations, and encodes the integrity constraint C over Γ into an SMT formula Φ_C. Then, we process both queries and encode their equivalence into an SMT formula Φ_R: ② we traverse Q_i in a forward fashion, encode how each operator in Q_i transforms its input to output, obtain the final output R_i of Q_i for all input databases under consideration, and ③ generate an SMT formula Φ_R that asserts $R_1 \neq R_2$ (we support bag and list semantics). While this overall approach is standard, our encoding scheme of each operator's semantics is new and has some important advantages. First, our encoding is based on the theory of integers with uninterpreted functions: it is simple yet sufficient to precisely encode all SQL features in our language (such as complex aggregate functions with grouping and previously unsupported operators like `ORDER BY`). Crucially, our approach can support these advanced SQL features without needing additional axioms, which prior work like Qex would otherwise require. Second, our encoding follows the three-valued logic and supports `NULL` for all of our operators, whereas prior work supports `NULL` for significantly fewer cases. In the final step ④, we construct $\Phi = \Phi_C \wedge \Phi_R$ and use an off-the-shelf SMT solver to solve Φ. Notably, Φ takes into account both query semantics and integrity constraints in a much simpler and more unified manner than some prior work that has separate schemes to handle queries and integrity constraints. For example, SPES [Zhou et al. 2022] performs symbolic encoding for queries but uses rewrite rules to deal with integrity constraints — it is fundamentally hard for such approaches to support complex integrity constraints such as those that restrict the value range.

We have evaluated VeriEQL on an extremely large number of benchmarks, consisting of 24,455 query pairs collected from three different workloads (including all benchmarks from the literature, standard benchmarks from Calcite, and over 20,000 new benchmarks from LeetCode). Our evaluation results show that VeriEQL can prove the bounded equivalence for significantly more benchmarks than all state-of-the-art techniques, disprove and find counterexamples for two orders of magnitude more benchmarks, and uncover serious bugs in real-world codebases (including MySQL and Calcite).

Contributions. This paper makes the following contributions.

- We formulate the problem of bounded SQL equivalence verification *modulo integrity constraints*.
- We formalize the semantics of a practical fragment of SQL queries through list and higher-order functions. Our SQL fragment is significantly larger than those in prior work.

- We propose a novel SMT encoding tailored towards bounded equivalence verification of SQL queries, including previously unsupported SQL operators (e.g., `ORDER BY`, `CASE WHEN`) and rich integrity constraints. To our best knowledge, this is the first approach that supports complex SQL queries with rich integrity constraints.
- We prove our SMT encoding of SQL queries is correct with respect to the formal semantics.
- We implement our approach in VeriEQL. Our comprehensive evaluation on a total of 24,455 benchmarks — including a new benchmark suite with over 20,000 real-life queries — shows that, VeriEQL can solve (i.e., prove or disprove equivalence) 77% of these benchmarks, while state-of-the-art bounded verifier can solve <2% and testing tools can disprove <1%.

2 OVERVIEW

In this section, we further illustrate VeriEQL's workflow using a simple example from LeetCode[1]. Note that for illustration purposes, we significantly simplify the database schema and queries in this task, but VeriEQL can handle the original queries[2] that are much more complex.

Specifically, this task involves a database with two relations, `Friendship` (or F) and `Likes` (or L), where F records users' friends and L stores users' preferred pages. The task is to write a query that, given a user, returns the recommended pages which their friends prefer but are not preferred by the given user. In what follows, we explain this task in more detail.

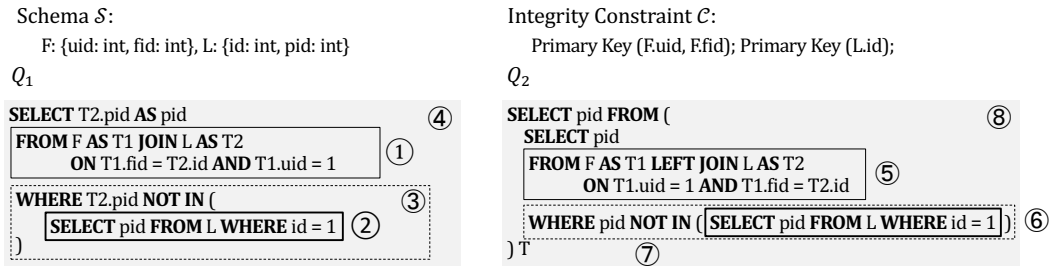
Schema and integrity constraint. Figure 2a shows the database schema $\mathcal{S}$ with the two relations: (i) `Friendship` relation F has two attributes uid and fid, denoting the ID of each user and their friends, and (ii) `Likes` relation L with two attributes id and pid which denote the user ID and the preferred page ID. The integrity constraint C for this task specifies that the pair (uid, fid) is the primary key of F, and id is the primary key of L.

Queries. Consider the user with $id = 1$. Figure 2a shows a query Q_1 that can solve the task for this given user. In particular, Q_1 uses an ① `INNER JOIN` to find the given user's friends, and uses a ② nested query followed by a ③ `NOT IN` filter to rule out the given user's preferred pages. Figure 2a also shows another query Q_2 which is similar to Q_1 but uses a ⑤ `LEFT JOIN` instead. Note that Q_2 is *not* equivalent to Q_1 — consider the case where the given user does not have any friends (i.e., F does not have a tuple with $uid = 1$). In this case, the `INNER JOIN` in Q_1 returns an empty result as the join condition $F.uid = 1$ never holds, whereas the `LEFT JOIN` from Q_2 gives a product tuple t' for each tuple t in L and $t'.pid = \texttt{NULL}$. The `NULL` values on pid will eventually be projected out by Q_2, leading to (incorrect) `NULL` tuples in the result.

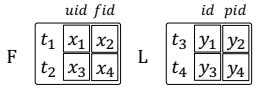
In what follows, let us explain how VeriEQL proves the non-equivalence of (Q_1, Q_2). Specifically, we will explain the key concepts in Figure 1: (1) what does a symbolic database look like, (2) how to encode the integrity constraint, (3) how to encode the query semantics, (4) how to check two queries always produce identical outputs, and (5) how to obtain verification results.

Generating symbolic database. Figure 2b shows the symbolic DB generated by VeriEQL that has *up to* 2 rows in each relation (i.e., the input bound is 2). Each row is called a *symbolic tuple*. For instance, we denote the tuples in F by t_1, t_2 and denote the attribute values by $x_1 = t_1.uid$ and $x_2 = t_1.fid$ and etc. Similarly, the tuples in L are t_3, t_4 and the values for id and pid are y_1, y_2 and y_3, y_4, respectively. In general, we use an *uninterpreted* predicate Del over tuples to indicate whether a tuple is deleted after an operation. Since the value of $\text{Del}(t_i)$ is unspecified, whether t_i is deleted or not is non-deterministic. Thus, the initial symbolic database encodes all databases where each relation has *at most* two tuples.

[1] https://leetcode.com/problems/page-recommendations
[2] The original queries are available in the Appendix of the extended version [He et al. 2024b].



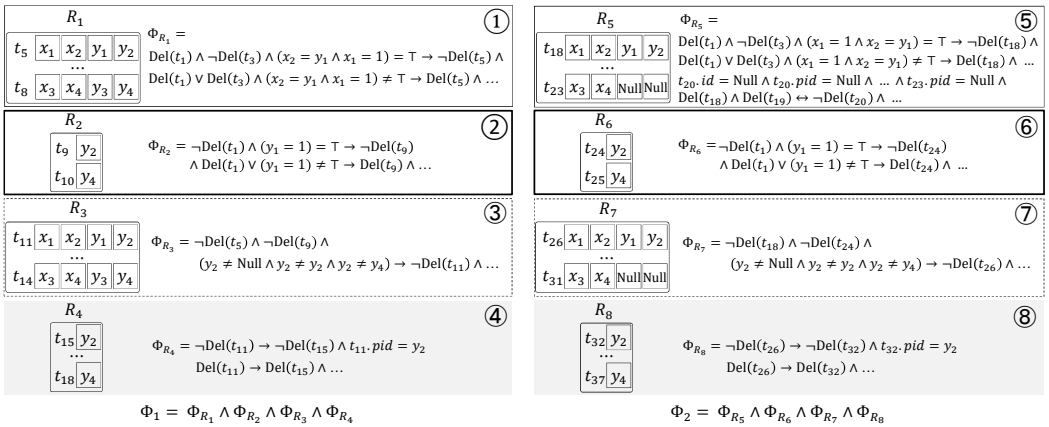
(a) Schema, integrity constraint, and queries.



(b) Generating symbolic database.

$$\Phi_{\mathcal{C}} = x_1 \neq \text{Null} \wedge x_2 \neq \text{Null} \wedge x_3 \neq \text{Null} \wedge x_4 \neq \text{Null} \wedge \neg(x_1 = x_3 \wedge x_2 = x_4) \wedge y_1 \neq \text{Null} \wedge y_3 \neq \text{Null} \wedge y_1 \neq y_3$$

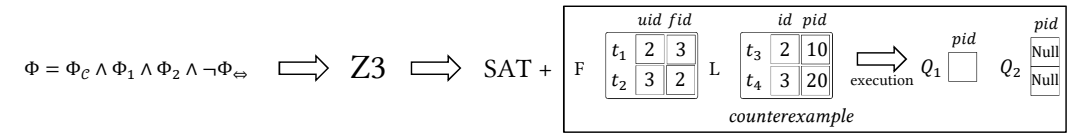
(c) Encoding integrity constraint.



(d) Encoding query semantics.

$$\Phi_{\Leftrightarrow} = \Sigma_{t \in R_4} \mathbb{I}[\neg\text{Del}(t)] = \Sigma_{t' \in R_8} \mathbb{I}[\neg\text{Del}(t')] \wedge \wedge_{t \in R_4} \left(\Sigma_{t' \in R_4} \mathbb{I}[\neg\text{Del}(t) \wedge \neg\text{Del}(t') \wedge t = t'] = \Sigma_{t'' \in R_8} \mathbb{I}[\neg\text{Del}(t) \wedge \neg\text{Del}(t'') \wedge t = t'']\right)$$

(e) Checking equality.



(f) Constraint solving.

Fig. 2. Illustration of how VeriEQL works on a simple LeetCode task.

Encoding integrity constraint. Figure 2c shows VeriEQL's SMT encoding Φ_C of the integrity constraint C. It has two parts. The first part

$$x_1 \neq \text{Null} \wedge x_2 \neq \text{Null} \wedge x_3 \neq \text{Null} \wedge x_4 \neq \text{Null} \wedge \neg(x_1 = x_3 \wedge x_2 = x_4)$$

specifies that tuples t_1, t_2 are unique and all attributes are not null, since (uid, fid) is a primary key. The second part $y_1 \neq \text{Null} \wedge y_3 \neq \text{Null} \wedge y_1 \neq y_3$ encodes that id is a primary key of relation L.

$$\begin{array}{rcl} \text{Schema} & :: & \text{RelName} \to \text{RelSchema} \\ \text{RelSchema} & :: & [(\text{AttrName}, \text{Type})] \\ \text{Type} & \in & \{\text{Int}, \text{Bool}\} \end{array}$$

(a) Schema

$$\begin{array}{rcl} \text{Database} & :: & \text{RelName} \to \text{Relation} \\ \text{Relation} & :: & [\text{Tuple}] \\ \text{Tuple} & :: & [(\text{AttrName}, \text{Value})] \\ \text{Value} & \in & \mathbf{Int} \cup \mathbf{Bool} \cup \{\text{Null}\} \end{array}$$

(b) Database

Fig. 3. Relational schema and database.

Encoding query semantics. To encode the semantics of a query, VeriEQL encodes the semantics of each operator which are then composed to form the encoding of the entire query. For example, the formula Φ_1 to encode Q_1 is the conjunction of Φ_{R_1}, Φ_{R_2}, Φ_{R_3} and Φ_{R_4}. Specifically, to encode the inner join $T_1 \bowtie T_2$ in ①, VeriEQL considers the Cartesian product of tuples in T_1 and T_2, followed by a filter to set all resulted tuples that do not satisfy $T_1.fid = T_2.id$ and $T_1.id = 1$ as deleted. For the `NOT IN` in Q_1, it analyzes the nested query and generates the formula Φ_{R_2} (as shown in ②), and checks the membership of $T_1.pid$ with the obtained pid, which is encoded as Φ_{R_3} in ③. Finally, the pid is projected out by ④, resulting in the output R_4 and a formula Φ_{R_4}. The obtained formula $\Phi_1 = \Phi_{R_1} \wedge \Phi_{R_2} \wedge \Phi_{R_3} \wedge \Phi_{R_4}$ precisely encodes the query semantics of Q_1. Similarly, Q_2's output is R_8 and the corresponding formula Φ_2 consists of Φ_{R_5}, Φ_{R_6}, Φ_{R_7} and Φ_{R_8} from ⑤ ⑥ ⑦ ⑧, respectively.

Checking equality. After encoding the semantics of queries Q_1 and Q_2, VeriEQL needs to compare their outputs R_4 and R_8. Although R_4 and R_8 do not have the same number of symbolic tuples, they can still be equal because some tuples may have been deleted. To check the equality, VeriEQL generates a formula $\Phi_{\Leftrightarrow}$ (in Figure 2e) asserting R_4 is equal to R_8. VeriEQL supports both list and bag semantics. In this example, it uses bag semantics because neither of the queries involves sorting operations. $\Phi_{\Leftrightarrow}$ encodes two properties: (1) R_4 and R_8 have the same number of non-deleted tuples and (2) for each non-deleted tuple in R_4, its multiplicity in R_4 is the same as that in R_8. Since properties (1) and (2) imply that (3) for each non-deleted tuple in R_8, its multiplicity in R_8 is the same as that in R_4, we do not need to include a formula for property (3) in $\Phi_{\Leftrightarrow}$.

Verification result. Finally, VeriEQL builds a formula $\Phi = \Phi_C \wedge \Phi_1 \wedge \Phi_2 \wedge \neg\Phi_{\Leftrightarrow}$ encoding the existence of a database $\mathcal{D}$ such that (1) $\mathcal{D}$ satisfies the integrity constraint C and (2) Q_1 and Q_2 have different outputs on $\mathcal{D}$. VeriEQL invokes Z3 and finds Φ is satisfiable, so it concludes that Q_1 and Q_2 are not equivalent and generates a counterexample database (shown in Figure 2f) where Q_1 and Q_2 indeed yield different results.

3 PROBLEM FORMULATION

We first describe the preliminaries of relational schema and database, and then present our query language and integrity constraints, followed by a formal problem statement.

3.1 Relational Schema and Database

Schema. As shown in Figure 3a, a relational database schema is a mapping from relation names to their relation schemas, where each relation schema is a list of attributes. Each attribute is represented by a pair of attribute name and type. All attribute names are assumed to be globally unique, which can be easily enforced by using fully qualified attribute names that include relation names, e.g., `EMP.age`. We only consider two primitive types, namely Int and Bool, in this paper. Other types (e.g., strings and dates) can be treated as Int and functions involving those types can be treated as uninterpreted functions over Int.

Database. Similarly, as shown in Figure 3b, a relational database is a mapping from relation names to their corresponding relations, where each relation consists of a list of tuples.[3] Each tuple contains a list of values with corresponding attribute names, and a value can be an integer, bool, or `NULL`.

[3] If a relation is created by a non-`ORDER BY` query, then we interpret this list as a bag.

$$
\begin{array}{rcl}
\text{Query } Q_r & ::= & Q \mid \text{OrderBy}(Q, \vec{E}, b) \\
\text{Subquery } Q & ::= & R \mid \Pi_L(Q) \mid \sigma_\phi(Q) \mid \rho_R(Q) \mid Q \oplus Q \mid \text{Distinct}(Q) \mid Q \otimes Q \mid \text{GroupBy}(Q, \vec{E}, L, \phi) \\
& \mid & \text{With}(\vec{Q}, \vec{R}, Q) \\
\text{Attr List } L & ::= & id(A) \mid \rho_a(A) \mid L, L \\
\text{Attr } A & ::= & \text{Cast}(\phi) \mid E \mid \mathcal{G}(E) \mid A \diamond A \\
\text{Pred } \phi & ::= & b \mid \text{Null} \mid A \odot A \mid \text{IsNull}(E) \mid \vec{E} \in \vec{v} \mid \vec{E} \in Q \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
\text{Expr } E & ::= & a \mid v \mid E \diamond E \mid \text{ITE}(\phi, E, E) \mid \text{Case}(\vec{\phi}, \vec{E}, E) \\
\text{Join Op } \otimes & ::= & \times \mid \bowtie_\phi \mid \rightarrowtail\!\!\bowtie_\phi \mid \bowtie\!\!\leftarrowtail_\phi \mid \rightarrowtail\!\!\bowtie\!\!\leftarrowtail_\phi \\
\text{Collection Op } \oplus & ::= & \cup \mid \cap \mid \setminus \mid \uplus \mid \text{⨞} \mid - \\
\text{Arith Op } \diamond & ::= & + \mid - \mid \times \mid / \mid \% \\
\text{Logic Op } \odot & ::= & \leq \mid < \mid = \mid \neq \mid > \mid \geq
\end{array}
$$

$R \in$ **Relation Names** $\quad a \in$ **Attribute Names** $\quad v \in$ **Values** $\quad b \in$ **Bools** $\quad \mathcal{G} \in \{\text{Count}, \text{Min}, \text{Max}, \text{Sum}, \text{Avg}\}$

Fig. 4. Syntax of SQL Queries. **Values** include integers, bools, and Null. $id(A)$ is a construct denoting attribute A itself occurs in the attribute list. If the context is clear, we may also omit the id constructor for brevity. The Cast function takes as input a predicate ϕ and returns Null if ϕ is Null, 1 if ϕ is $\top$, and 0 otherwise.

3.2 Syntax of SQL Queries

The syntax of our query language is shown in Figure 4, which covers various practical SQL operators, including projection Π, selection σ, renaming ρ, set union $\cup$, intersection $\cap$, minus $\setminus$, bag union $\uplus$, intersection ⨞, minus $-$, Distinct, Cartesian product $\times$, inner join $\bowtie_\phi$, left outer join $\rightarrowtail\!\!\bowtie_\phi$, right outer join $\bowtie\!\!\leftarrowtail_\phi$, full outer join $\rightarrowtail\!\!\bowtie\!\!\leftarrowtail_\phi$, GroupBy with Having clauses, With clauses, and OrderBy. In addition, the query language also supports various attribute expressions such as arithmetic expressions $E_1 \diamond E_2$, aggregate functions $\mathcal{G}(E)$, if-then-else expressions $\text{ITE}(\phi, E_1, E_2)$, and case expressions $\text{Case}(\vec{\phi}, \vec{E}, E')$, as well as predicates such as logical comparison $A_1 \odot A_2$, null checks $\text{IsNull}(E)$, and membership check $\vec{E} \in Q$. Many of these constructs are not supported by prior work, such as OrderBy, With, Case, and so on. In what follows, we discuss the syntax of aggregate functions, GroupBy, and OrderBy in more detail.

Aggregate functions and GroupBy. Our language includes five common aggregate functions $\mathcal{G}$, and as is standard, it does not permit nested aggregate functions such as `SUM(SUM(a))`. The language also has a construct $\text{GroupBy}(Q, \vec{E}, L, \phi)$ for grouping tuples. Intuitively, $\text{GroupBy}(Q, \vec{E}, L, \phi)$ groups the tuples of subquery Q based on attribute expressions $\vec{E}$ and for each group satisfying condition ϕ, it computes an aggregate tuple according to the attribute list L.

Example 3.1. The GroupBy operation is closely related to the `GROUP BY` and `HAVING` clauses in standard SQL. For instance, consider a relation `EMP(id, gender, age, sal)` and a SQL query:

```
SELECT AVG(age), AVG(sal) FROM EMP WHERE age > 20 GROUP BY gender HAVING AVG(sal) > 30000
```

It can be represented by the following query in our language

$$\text{GroupBy}(\sigma_{\texttt{age}>20}(\texttt{EMP}), [\texttt{gender}], [\text{Avg}(\texttt{age}), \text{Avg}(\texttt{sal})], \text{Avg}(\texttt{sal}) > 30000)$$

It is worthwhile to point out that the grammar in Figure 4 does not precisely capture all syntactic requirements on valid queries. In particular, a query with certain syntax errors may also be accepted by the language, because those syntactic requirements are difficult to describe at the grammar level. We instead perform static analysis to check if a query is well-formed and throw errors if the query contains invalid expressions. For example, consider the following SQL query

```
SELECT SUM(sal) FROM EMP GROUP BY age HAVING sal > 10000
```

where the `HAVING` clause uses a non-aggregated attribute `sal` that is not in the `GROUP BY` list. Such a query is not permitted in standard SQL, because it may contain ambiguity where a group of tuples with the same `age` may have different `sal` values. Our query language also disallows such

$[\![Q]\!]$ :: Database $\mathcal{D} \to$ Relation

$$
\begin{array}{lcl}
[\![R]\!]_{\mathcal{D}} & = & \mathcal{D}(R) \\
[\![\Pi_L(Q)]\!]_{\mathcal{D}} & = & \mathrm{ite}(\mathrm{hasAgg}(L), [\,[\![L]\!]_{\mathcal{D},[\![Q]\!]_{\mathcal{D}}}\,], \mathrm{map}([\![Q]\!]_{\mathcal{D}}, \lambda x.[\![L]\!]_{\mathcal{D},x})) \\
[\![\sigma_\phi(Q)]\!]_{\mathcal{D}} & = & \mathrm{filter}([\![Q]\!]_{\mathcal{D}}, \lambda x.[\![\phi]\!]_{\mathcal{D},[x]} = \top) \\
[\![\rho_R(Q)]\!]_{\mathcal{D}} & = & \mathrm{map}([\![Q]\!]_{\mathcal{D}}, \lambda x.\mathrm{map}(x, \lambda(n,v).(\mathrm{rename}(R,n),v))) \\
[\![Q_1 \cap Q_2]\!]_{\mathcal{D}} & = & \mathrm{filter}([\![\mathrm{Distinct}(Q_1)]\!]_{\mathcal{D}}, \lambda x.x \in [\![Q_2]\!]_{\mathcal{D}}) \\
[\![Q_1 \cup Q_2]\!]_{\mathcal{D}} & = & [\![\mathrm{Distinct}(Q_1 \uplus Q_2)]\!]_{\mathcal{D}} \\
[\![Q_1 \setminus Q_2]\!]_{\mathcal{D}} & = & \mathrm{filter}([\![\mathrm{Distinct}(Q_1)]\!]_{\mathcal{D}}, \lambda x.x \notin [\![Q_2]\!]_{\mathcal{D}}) \\
[\![Q_1 \intercal Q_2]\!]_{\mathcal{D}} & = & [\![Q_1 - (Q_1 - Q_2)]\!]_{\mathcal{D}} \\
[\![Q_1 \uplus Q_2]\!]_{\mathcal{D}} & = & \mathrm{append}([\![Q_1]\!]_{\mathcal{D}}, [\![Q_2]\!]_{\mathcal{D}}) \\
[\![Q_1 - Q_2]\!]_{\mathcal{D}} & = & \mathrm{foldl}(\lambda xs.\lambda x.\mathrm{ite}(x \in xs, xs - x, xs), [\![Q_1]\!]_{\mathcal{D}}, [\![Q_2]\!]_{\mathcal{D}}) \\
[\![\mathrm{Distinct}(Q)]\!]_{\mathcal{D}} & = & \mathrm{foldr}(\lambda x.\lambda xs.\mathrm{cons}(x, \mathrm{filter}(xs, \lambda y.x \neq y)), [\,], [\![Q]\!]_{\mathcal{D}}) \\
[\![Q_1 \times Q_2]\!]_{\mathcal{D}} & = & \mathrm{foldl}(\lambda xs.\lambda x.\mathrm{append}(xs, \mathrm{map}([\![Q_2]\!]_{\mathcal{D}}, \lambda y.\mathrm{merge}(x,y))), [\,], [\![Q_1]\!]_{\mathcal{D}}) \\
[\![Q_1 \bowtie_\phi Q_2]\!]_{\mathcal{D}} & = & [\![\sigma_\phi(Q_1 \times Q_2)]\!]_{\mathcal{D}} \\
[\![Q_1 \rtimes_\phi Q_2]\!]_{\mathcal{D}} & = & \mathrm{foldl}(\lambda xs.\lambda x.\mathrm{append}(xs, \mathrm{ite}(|v_1(x)| = 0, v_2(x), v_1(x)), [\,], [\![Q_1]\!]_{\mathcal{D}}) \\
 & & \quad \text{where } v_1(x) = [\![[x] \bowtie_\phi Q_2]\!]_{\mathcal{D}} \text{ and } v_2(x) = [\mathrm{merge}(x, T_{\mathrm{Null}})] \\
[\![Q_1 \ltimes_\phi Q_2]\!]_{\mathcal{D}} & = & \mathrm{foldl}(\lambda xs.\lambda x.\mathrm{append}(xs, \mathrm{ite}(|v_1(x)| = 0, v_2(x), v_1(x)), [\,], [\![Q_2]\!]_{\mathcal{D}}) \\
 & & \quad \text{where } v_1(x) = [\![Q_1 \bowtie_\phi [x]]\!]_{\mathcal{D}} \text{ and } v_2(x) = [\mathrm{merge}(T_{\mathrm{Null}}, x)] \\
[\![Q_1 \rtimes\!\ltimes_\phi Q_2]\!]_{\mathcal{D}} & = & \mathrm{append}([\![Q_1 \rtimes_\phi Q_2]\!]_{\mathcal{D}}, \mathrm{map}(xs, \lambda x.\mathrm{merge}(T_{\mathrm{Null}}, x))) \\
 & & \quad \text{where } xs = \mathrm{filter}([\![Q_2]\!]_{\mathcal{D}}, \lambda y.|[\![Q_1 \bowtie_\phi [y]]\!]_{\mathcal{D}}| = 0) \\
[\![\mathrm{GroupBy}(Q, \vec{E}, L, \phi)]\!]_{\mathcal{D}} & = & \mathrm{map}(\mathrm{filter}(Gs, \lambda xs.[\![\phi]\!]_{\mathcal{D},xs} = \top), \lambda xs.[\![L]\!]_{\mathcal{D},xs}) \text{ where} \\
 & & \quad Gs = \mathrm{map}(\mathrm{Dedup}(Q, \vec{E}), \lambda y.\mathrm{filter}([\![Q]\!]_{\mathcal{D}}, \lambda z.\mathrm{Eval}(\vec{E}, [z]) = y)), \\
 & & \quad \mathrm{Dedup}(Q, \vec{E}) = \mathrm{foldr}(\lambda x.\lambda xs.\mathrm{cons}(x, \mathrm{filter}(xs, \lambda y.x \neq y)), [\,], \mathrm{map}([\![Q]\!]_{\mathcal{D}}, \lambda z.\mathrm{Eval}(\vec{E}, [z]))), \\
 & & \quad \mathrm{Eval}(\vec{E}, xs) = \mathrm{map}(\vec{E}, \lambda e.[\![e]\!]_{\mathcal{D},xs}) \\
[\![\mathrm{With}(\vec{Q}, \vec{R}, Q)]\!]_{\mathcal{D}} & = & [\![Q]\!]_{\mathcal{D}'} \text{ where } \mathcal{D}' = \mathcal{D}[R_i \mapsto [\![Q_i]\!]_{\mathcal{D}} \mid R_i \in \vec{R}] \\
[\![\mathrm{OrderBy}(Q, \vec{E}, b)]\!]_{\mathcal{D}} & = & \mathrm{foldl}(\lambda xs.\lambda\_.(\mathrm{append}(xs, [\mathrm{MinTuple}(\vec{E}, b, [\![Q]\!]_{\mathcal{D}} - xs)])), [\,], [\![Q]\!]_{\mathcal{D}}) \text{ where} \\
 & & \quad \mathrm{MinTuple}(\vec{E}, b, xs) = \mathrm{foldl}(\lambda x.\lambda y.\mathrm{ite}(\mathrm{Cmp}(\vec{E}, b, x, y), y, x), \mathrm{head}(xs), xs), \\
 & & \quad \mathrm{Cmp}(\vec{E}, b, x_1, x_2) = b \neq \mathrm{foldr}(\lambda E_i.\lambda y.\mathrm{ite}(v(x_1, E_i) < v(x_2, E_i), \top, \\
 & & \qquad\qquad \mathrm{ite}(v(x_1, E_i) > v(x_2, E_i), \bot, y)), \top, \vec{E}), \\
 & & \quad v(x, E) = \mathrm{ite}([\![E]\!]_{\mathcal{D},[x]} = \mathrm{Null}, -\infty, [\![E]\!]_{\mathcal{D},[x]})
\end{array}
$$

Fig. 5. Formal semantics of SQL queries. ite is the standard if-then-else function. $\mathrm{hasAgg}(L)$ checks if the attribute list L has an attribute computing an aggregation. $\mathrm{merge}(x, y)$ merges two tuples x and y into one tuple. $\mathrm{rename}(R, n)$ replaces all occurrences of the original table name R with the new globally unique name n. T_{Null} represents a tuple of Null's, whose length is determined as appropriate by the context for brevity. Note that $xs - x$ on the right-hand side of $[\![Q_1 - Q_2]\!]_{\mathcal{D}}$ denotes deleting one tuple x in xs iff $xs \in x$. Full formal semantics of all constructs in our query language, including predicates $[\![\phi]\!]_{\mathcal{D},xs}$, attribute lists $[\![L]\!]_{\mathcal{D},xs}$, and expressions $[\![e]\!]_{\mathcal{D},xs}$ is available in the Appendix of the extended version [He et al. 2024b] .

GroupBy queries. Although a simple syntactic check can reveal this error, describing the check in grammar is non-trivial, so we perform static analysis to reject such queries.

Sorting and OrderBy. Our language supports sorting the query result. Specifically, $\mathrm{OrderBy}(Q, \vec{E}, b)$ can sort the result of subquery Q according to a list of attribute expressions $\vec{E}$. The result is in ascending order if $b = \top$ and in descending order otherwise. Note that as shown in Figure 4, OrderBy is only allowed to be used at the topmost level of a query.

3.3 Semantics of SQL Queries

The denotational semantics of our SQL queries is presented in Figure 5, where $[\![Q]\!]$ takes as an input a database $\mathcal{D}$ and produces as output a relation.

Bag and list semantics. As is standard in SQL, we conceptually view a relation as a bag (multiset) of tuples and use the bag semantics for queries. However, our denotational semantics in Figure 5 uses lists to implement bags and define query operators through higher-order combinators such as map, filter, and foldl. In this way, we can easily define complex query operators and switch to list semantics when the query needs to sort the result using `ORDER BY`.

Three-valued semantics. Similar to standard SQL, our semantics also uses three-valued logic. More specifically, a relation may use NULL values to represent unknown information, and all query operators should behave correctly with respect to the NULL value. For example, our semantics considers a predicate can be evaluated to three possible values, namely $\top$ (true), $\bot$ (false), and Null. A notation like $[\![\phi]\!]_{\mathcal{D},xs} = \top$ means the predicate ϕ evaluates to true (not false nor Null) given database $\mathcal{D}$ and tuples xs. [4]

Basics. At a high level, our denotational semantics can be viewed as a function or a functional program $[\![Q]\!]_{\mathcal{D}}$ that pattern matches different constructors of query Q and evaluates them to relations given a concrete database $\mathcal{D}$. If the query is a relation name R, then $[\![R]\!]_{\mathcal{D}}$ simply looks up the name R in $\mathcal{D}$. If the query is a projection, $[\![\Pi_L(Q)]\!]_{\mathcal{D}}$ first checks if the attribute list L includes aggregation functions. If so, it evaluates query Q and invokes $[\![L]\!]_{\mathcal{D},[\![Q]\!]_{\mathcal{D}}}$ to compute the aggregate values. Otherwise, it projects each tuple in the result of $[\![Q]\!]_{\mathcal{D}}$ according to L through a standard map combinator. If the query is renaming, $[\![\rho_R(Q)]\!]_{\mathcal{D}}$ first evaluates $[\![Q]\!]_{\mathcal{D}}$. Then for each tuple in the result, it updates the attribute name n to be a new name $\mathsf{rename}(R, n)$ that is related to relation name R. Following similar ideas of functional programming, we can define semantics for filtering σ, set operations $\cap, \cup, \setminus$, bag operations $\uplus, \biguplus, -$, and Distinct for removing duplicated tuples.

Cartesian product and joins. Apart from the standard Cartesian product $Q_1 \times Q_2$, there are four different joins in our language. An *inner join* $Q_1 \bowtie_\phi Q_2$ can be viewed as a syntactic sugar of $\sigma_\phi(Q_1 \times Q_2)$. A *left outer join* $Q_1 \rtimes_\phi Q_2$ is more involved. $[\![Q_1 \rtimes Q_2]\!]_{\mathcal{D}}$ iterates tuples in the result of $[\![Q_1]\!]_{\mathcal{D}}$ and for each tuple x, it computes the inner join of $[x]$ and Q_2 and denotes the result by $v_1(x)$. If $v_1(x)$ is not empty, it is appended to the final result of the left outer join. Otherwise, $[\![Q_1 \rtimes Q_2]\!]_{\mathcal{D}}$ computes a *null extension* $v_2(x)$ of x by taking the Cartesian product between x and T_{Null} (a tuple of Null values) and appends $v_2(x)$ to the result. Similarly, a *right outer join* $Q_1 \ltimes_\phi Q_2$ can be defined in the same way. For *full outer join*, $[\![Q_1 \rtimes\ltimes_\phi Q_2]\!]_{\mathcal{D}}$ starts with the left outer join $Q_1 \rtimes_\phi Q_2$. Then for each tuple y in $[\![Q_2]\!]_{\mathcal{D}}$, if the inner join between Q_1 and $[y]$ returns empty, $[\![Q_1 \rtimes\ltimes_\phi Q_2]\!]_{\mathcal{D}}$ adds a null extension $\mathsf{merge}(T_{\text{Null}}, y)$ to the result.

Example 3.2. Consider the following two relations EMP and DEPT. What follows illustrates the difference in the results of various joins and their Cartesian product.

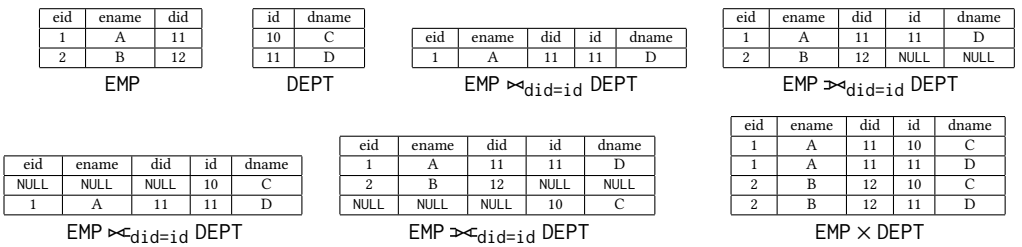


Fig. 6. An example demonstrating different joins in SQL.

GroupBy. To define the semantics of $\text{GroupBy}(Q, \vec{E}, L, \phi)$, we use several auxiliary functions. Specifically, $\text{Eval}(\vec{E}, xs)$ computes the values of expressions $\vec{E}$ to be grouped over tuple list xs, and $\text{Dedup}(Q, \vec{E})$ invokes Eval to retain only unique values of $\vec{E}$. Then GroupBy maps each unique $\vec{E}$ value to a group of tuples sharing the same value over $\vec{E}$ and obtains all groups Gs. It evaluates the HAVING condition ϕ and only retains those groups satisfying ϕ by the filter combinator. Finally, for each retained group xs, GroupBy computes an aggregate value $[\![L]\!]_{\mathcal{D},xs}$ and adds it to the result.

[4]Detailed definition of predicate semantics is available in the Appendix of the extended version [He et al. 2024b].

$$
\begin{array}{rcl}
\text{Constraint } C & ::= & \text{PK}(R, \vec{a}) \mid \text{FK}(R, a, R, a) \mid \text{NotNull}(R, a) \mid \text{Check}(R, \psi) \mid \text{Inc}(R, a, v) \mid C \wedge C \\
\text{Pred } \psi & ::= & a \odot v \mid a \odot a \mid a \in \vec{v} \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \\
\text{Logic Op } \odot & ::= & \leq \mid < \mid = \mid \neq \mid > \mid \geq \\
\multicolumn{3}{c}{R \in \textbf{Relations} \qquad a \in \textbf{Attributes} \qquad v \in \textbf{Values}}
\end{array}
$$

Fig. 7. Syntax of Integrity Constraints.

With clause. To evaluate $\text{With}(\vec{Q}, \vec{R}, Q)$, our semantics first evaluates all subqueries $Q_i \in \vec{Q}$, then creates a new database $\mathcal{D}'$ by adding mappings $R_i \mapsto [\![Q_i]\!]_{\mathcal{D}}$, and finally evaluates Q under $\mathcal{D}'$. Intuitively, the With clause creates local bindings for subqueries Q_i which can be used in query Q.

OrderBy. We define a deterministic semantics for the OrderBy construct in our language, i.e., two runs of OrderBy with the same arguments always return the same result. Specifically, $\text{OrderBy}(Q, \vec{E}, b)$ performs a selection sort on the results of $[\![Q]\!]_{\mathcal{D}}$ by expressions $\vec{E}$. The sorted results are in ascending order if $b = \top$ and in descending order if $b = \bot$. At a high level, OrderBy maintains a list of sorted tuples in ascending (resp. descending) order and repeatedly selects the minimum (resp. maximum) tuple from the unsorted tuples using the MinTuple function. In particular, MinTuple invokes the Cmp function for pair-wise comparison of two tuples x_1 and x_2 over expression list $\vec{E}$. Null is viewed as a special value $-\infty$ that is smaller than any other values, so Null's can also be ordered correctly by our semantics.

Our language vs. prior work's. Our formal semantics is inspired by prior work such as Mediator [Wang et al. 2018b]. While Mediator defines semantics for simple update and query operators, our semantics defines more complex query operators, including outer joins, `GROUP BY`, `WITH`, and `ORDER BY`. Furthermore, to our best knowledge, our semantics considers many operators that are important in practice but not considered in any prior work such as `ORDER BY`, `WITH`, `IF`, `INTERSECT`, `EXCEPT`, etc. In addition, our work supports complex attribute expressions and predicates, such as `Avg(age + 10)`, whereas prior work has very limited support for such features.

3.4 Integrity Constraints

Integrity constraints are fundamental to data integrity and therefore must be followed by a database. In this paper, we support the constraints shown in Figure 7. In particular, an integrity constraint consists of a set of primitive constraints, detailed as follows.

Primary keys. $\text{PK}(R, \vec{a})$ says that a list of attributes $\vec{a}$ is the primary key of relation R. In particular, it requires that all values of any attribute in $\vec{a}$ are not `NULL`. Furthermore, given two different tuples $t_1, t_2 \in R$, t_1 and t_2 must have different values on at least one attribute in $\vec{a}$.

Foreign keys. $\text{FK}(R_1, a_1, R_2, a_2)$ means that the attribute a_1 of relation R_1 is a foreign key referencing the attribute a_2 of relation R_2. Specifically, it requires that, for each tuple $t_1 \in R_1$, there exists a tuple $t_2 \in R_2$ such that $t_1.a_1 = t_2.a_2$.

Not-null constraints. NotNull(R, a) imposes a `NOT NULL` constraint on the attribute a of relation R. It stipulates that for each tuple $t \in R$, $t.a$ is not `NULL`.

Check constraints. $\text{Check}(R, \psi)$ requires that every tuple $t \in R$ must satisfy the predicate ψ, where ψ can be a boolean combination of atomic predicates. Each atomic predicate can be logical comparisons between two attributes of R, between an attribute and a constant value. or of the form $a \in \vec{v}$ meaning the value of attribute a is in a list of provided constants $\vec{v}$.

Auto increment. $\text{Inc}(R, a, v)$ means that the value of attribute a in relation R starts with v and strictly increases by one for each new tuple added to R. We require the value to strictly increase by

one to obtain a deterministic semantics of the auto increment constraint. However, this is not a fundamental limitation. We can easily generalize to the scenarios where values are not continuous.

3.5 Problem Statement

To describe our problem statement, we start with a notion of conformance between a relational database and its schema. Specifically, an attribute value (a, v) conforms to an attribute schema (b, τ), denoted $(a, v) ::_A (b, \tau)$, if $a = b$ and v is of type τ. [5] A tuple t conforms to a relation schema RS, denoted $t ::_T RS$, if (1) they have the same size, i.e. $|t| = |RS|$ and (2) each attribute value (a_i, v_i) in T is conforming to the corresponding attribute schema (b_i, τ_i) in RS.

Definition 3.3 (Conformance between Database and Schema). A database $\mathcal{D}$ conforms to a schema $\mathcal{S}$, denoted $\mathcal{D} :: \mathcal{S}$, if (1) $\mathcal{D}$ and $\mathcal{S}$ have the same domain and (2) for each relation name $R \in \mathsf{Dom}(\mathcal{D})$, all tuples in $\mathcal{D}(R)$ conform to their corresponding relation schema $\mathcal{S}(R)$, i.e.,

$$\mathcal{D} :: \mathcal{S} \stackrel{\text{def}}{=\!=} \mathsf{Dom}(\mathcal{D}) = \mathsf{Dom}(\mathcal{S}) \wedge (\forall R \in \mathsf{Dom}(\mathcal{D}). \forall t \in R.\ t ::_T \mathcal{S}(R))$$

Recall from Figure 3 that we only consider Int and Bool types in this paper. Other types of values (e.g., strings and dates) can be treated as integers and functions involving those types can be treated as uninterpreted functions over integers.

Example 3.4. Let schema $\mathcal{S} = [\texttt{EMP} \mapsto [(\texttt{eid}, \mathit{Int}), (\texttt{cid}, \mathit{Int})], \texttt{CAR} \mapsto [(\texttt{id}, \mathit{Int}), (\texttt{used}, \mathit{Bool})]]$ and database

$$\mathcal{D} = [\quad \texttt{EMP} \mapsto [[(\texttt{eid}, 1), (\texttt{cid}, 10)], [(\texttt{eid}, 2), (\texttt{cid}, \texttt{NULL})]], \\ \texttt{CAR} \mapsto [(\texttt{id}, 10), (\texttt{used}, \top)] \quad]$$

Here, $\mathcal{D} :: \mathcal{S}$ because $[(\texttt{eid}, 1), (\texttt{cid}, 10)] ::_T [(\texttt{eid}, \mathit{Int}), (\texttt{cid}, \mathit{Int})]$ and $[(\texttt{eid}, 2), (\texttt{cid}, \texttt{NULL})] ::_T [(\texttt{eid}, \mathit{Int}), (\texttt{cid}, \mathit{Int})]$ hold for relation name EMP and $[(\texttt{id}, 10), (\texttt{used}, \top)] ::_T [(\texttt{id}, \mathit{Int}), (\texttt{used}, \mathit{Bool})]$ holds for relation name CAR.

Definition 3.5 (Bounded Equivalence modulo Integrity Constraint). Given two queries (Q_1, Q_2) under schema $\mathcal{S}$ and a positive integer bound N, Q_1 and Q_2 are said to be *bounded equivalent modulo integrity constraint* C, denoted by $Q_1 \simeq_{\mathcal{S},C,N} Q_2$, if for any database $\mathcal{D}$ that conforms to schema $\mathcal{S}$ satisfies C and each relation has at most N tuples, the execution result of Q_1 is the same as the execution result of Q_2 on $\mathcal{D}$, i.e.,

$$Q_1 \simeq_{\mathcal{S},C,N} Q_2 \stackrel{\text{def}}{=\!=} \forall \mathcal{D}.\ \mathcal{D} :: \mathcal{S} \wedge (\forall R \in \mathcal{D}.\ |R| \leq N) \wedge C(\mathcal{D}) \Rightarrow [\![Q_1]\!]_{\mathcal{D}} = [\![Q_2]\!]_{\mathcal{D}}$$

4 BOUNDED EQUIVALENCE VERIFICATION MODULO INTEGRITY CONSTRAINTS

This section presents our algorithm for bounded equivalence verification of two queries modulo integrity constraints.

4.1 Algorithm Overview

The top-level algorithm of our verification technique is shown in Algorithm 1. The Verify procedure takes as input two queries Q_1 and Q_2 over schema $\mathcal{S}$, an integrity constraint C, and a bound N on the size of all relations in the database. It returns $\top$ indicating Q_1 and Q_2 are bounded equivalent modulo integrity constraint C, i.e., $Q_1 \simeq_{\mathcal{S},C,N} Q_2$. Otherwise, it returns a counterexample database satisfying C where Q_1 and Q_2 yield different results.

At a high-level, our technique reduces the verification problem into a constraint-solving problem and generates an SMT formula through the encoding of integrity constraints and SQL operations. Specifically, we first create a symbolic representation of the database Γ, where each relation has

[5] Null is viewed as a polymorphic value of both type Int and Bool.

Algorithm 1 Equivalence Verification

1: **procedure** VERIFY($Q_1, Q_2, \mathcal{S}, C, N$)
Input: Queries Q_1, Q_2, schema $\mathcal{S}$, integrity constraint C, and bound size N
Output: $\top$ for equivalence, otherwise a counterexample
2: $\Gamma \leftarrow$ BuildSymbolicDB($\mathcal{S}, N$);
3: $\Phi_C \leftarrow$ EncodeConstraint(Γ, C);
4: $\Phi_{R_1}, R_1 \leftarrow$ EncodeQuery($\mathcal{S}, \Gamma, Q_1$);
5: $\Phi_{R_2}, R_2 \leftarrow$ EncodeQuery($\mathcal{S}, \Gamma, Q_2$);
6: $\Phi \leftarrow \Phi_C \wedge \Phi_{R_1} \wedge \Phi_{R_2} \wedge \neg$Equal($R_1, R_2$);
7: **if** UNSAT(Φ) **then return** $\top$;
8: **else return** BuildExample($\mathcal{S}$, Model(Φ));

at most N tuples (Line 2). Then we encode the integrity constraint C over Γ and obtain an SMT formula Φ_C (Line 3). Next, we analyze queries Q_1, Q_2 to obtain their outputs R_1, R_2 given input Γ and two SMT formulas Φ_{R_1}, Φ_{R_2} encoding how R_1, R_2 are computed from Γ (Lines 4 – 5). Finally, we build a formula Φ asserting the existence of a database satisfying the integrity constraint C such that R_1 is different from R_2 (Line 6). If no such database exists (i.e., formula Φ is unsatisfiable), then Q_1 and Q_2 are bounded equivalent modulo integrity constraint C, i.e., $Q_1 \simeq_{\mathcal{S},C,N} Q_2$ (Line 7). Otherwise, if such a database exists, Q_1 and Q_2 are not equivalent, so we build a counterexample database from a model of Φ to disprove the equivalence (Line 8).

4.2 Schema and Symbolic Database

Since our verification technique is centered around a symbolic representation of the database, we first describe how to build the symbolic database.

Symbolic database. Given a schema $\mathcal{S}$ and a bound N for the size of relations, we build a symbolic database containing all relations in Dom($\mathcal{S}$) and each relation has N symbolic tuples. We denote the symbolic database by Γ where Γ : RelName $\rightarrow$ [SymTuple] maps relation names to their corresponding lists of symbolic tuples. In general, we introduce an uninterpreted predicate Del(t) for each tuple t to indicate whether or not t is deleted by subquery. [6] Since the Del predicates hold non-deterministic values in the symbolic database, we can use Γ to encode all possible databases where each relation has *at most* N tuples.

Example 4.1. Consider again the schema $\mathcal{S}$ in Example 3.4. Given a bound $N = 2$, we can build a symbolic database $\Gamma = \{\texttt{EMP} \mapsto [t_1, t_2], \texttt{CAR} \mapsto [t_3, t_4]\}$, where t_1, t_2, t_3, t_4 are symbolic tuples.

Encoding attributes. As is standard, we use uninterpreted functions to encode attributes. Specifically, for each attribute *attr* in the database schema, we introduce an uninterpreted function called *attr* that takes as input a symbolic tuple and produces as output a symbolic value. For example, $t.name$ for getting the *name* attribute of tuple t should be encoded as $name(t)$ where *name* is an uninterpreted function.

***Encoding* `NULL`.** To support `NULL` and three-valued semantics, we encode each symbolic value as a pair (b, v) in the SMT formula, where b is a boolean variable indicating whether the value is `NULL`, and v is a non-`NULL` value. In particular, if b is $\top$ (i.e., `true`), then the value (b, v) is `NULL`. Otherwise, if b is $\bot$ (i.e., false), the value is v. For example, constant 1 is represented by $(\bot, 1)$. In this way, `NULL` is not equal to any legitimate non-`NULL` values. Furthermore, we consider all the null values

[6] Alternatively, we can introduce a predicate Present(t) to indicate a tuple is indeed present, i.e., Present(t) $\Leftrightarrow \neg$Del(t).

$$\frac{\Gamma(R) = [t_1, \ldots, t_n] \quad \Phi_1 = \wedge_{i=1}^{n} \wedge_{k=1}^{m} t_i.a_k \neq \text{Null} \quad m = |\vec{a}| \quad \Phi_2 = \wedge_{i=1}^{n} \wedge_{j=i+1}^{n} \neg(\wedge_{k=1}^{m} t_i.a_k = t_j.a_k)}{\Gamma \vdash \text{PK}(R, \vec{a}) \rightsquigarrow \Phi_1 \wedge \Phi_2} \text{ (IC-PK)}$$

$$\frac{\Gamma(R_1) = [t_1, \ldots, t_n] \quad \Gamma(R_2) = [t'_1, \ldots, t'_m] \quad \Phi = \wedge_{i=1}^{n} \vee_{j=1}^{m} t_i.a_1 = t'_j.a_2}{\Gamma \vdash \text{FK}(R_1, a_1, R_2, a_2) \rightsquigarrow \Phi} \text{ (IC-FK)}$$

$$\frac{\Gamma(R) = [t_1, \ldots, t_n] \quad \Phi = \wedge_{i=1}^{n} t_i.a \neq \text{Null}}{\Gamma \vdash \text{NotNull}(R, a) \rightsquigarrow \Phi} \text{ (IC-NN)}$$

$$\frac{\Gamma(R) = [t_1, \ldots, t_n] \quad \Phi = \wedge_{i=1}^{n} [\![\psi]\!]_{t_i}}{\Gamma \vdash \text{Check}(R, \psi) \rightsquigarrow \Phi} \text{ (IC-Check)}$$

$$\frac{\Gamma(R) = [t_1, \ldots, t_n] \quad \Phi_1 = \wedge_{i=1}^{n} t_i.a = v + i - 1 \quad \Phi_2 = \wedge_{i=1}^{n} t_i.a \neq \text{Null}}{\Gamma \vdash \text{Inc}(R, a, v) \rightsquigarrow \Phi_1 \wedge \Phi_2} \text{ (IC-Inc)}$$

$$\frac{\Gamma \vdash C_1 \rightsquigarrow \Phi_1 \quad \Gamma \vdash C_2 \rightsquigarrow \Phi_2}{\Gamma \vdash C_1 \wedge C_2 \rightsquigarrow \Phi_1 \wedge \Phi_2} \text{ (IC-Comp)}$$

Fig. 8. Symbolic encoding of integrity constraints.

$$\begin{aligned}
[\![a \odot v]\!]_t &= t.a \neq \text{Null} \wedge t.a \odot v & [\![\psi_1 \wedge \psi_2]\!]_t &= [\![\psi_1]\!]_t \wedge [\![\psi_2]\!]_t \\
[\![a_1 \in \vec{v}]\!]_t &= t.a \neq \text{Null} \wedge \bigvee_{i=1}^{|\vec{v}|} t.a = v_i & [\![\psi_1 \vee \psi_2]\!]_t &= [\![\psi_1]\!]_t \vee [\![\psi_2]\!]_t \\
[\![a_1 \odot a_2]\!]_t &= t.a_1 \odot t.a_2 \text{ where } \odot \in \{=, \neq\} & [\![\neg\psi]\!]_t &= \neg[\![\psi]\!]_t \\
[\![a_1 \odot a_2]\!]_t &= t.a_1 \neq \text{Null} \wedge t.a_2 \neq \text{Null} \wedge t.a_1 \odot t.a_2 \text{ where } \odot \in \{\leq, <, >, \geq\}
\end{aligned}$$

Fig. 9. Symbolic encoding of predicates in integrity constraints.

are equal. Although we allow different representations of null values like $(\top, 1)$ and $(\top, 2)$, they are considered equal because both of them are considered to be NULL.

4.3 Encoding Integrity Constraints

Given a symbolic database Γ and an integrity constraint C, we follow the semantics of C to encode C as an SMT formula over Γ. The encoding procedure is summarized as inference rules in Figure 8, where judgments of the form $\Gamma \vdash C \rightsquigarrow \Phi$ represent that the encoding of integrity constraint C is Φ given a symbolic database Γ.

In a nutshell, we encode each atomic integrity constraint in C and conjoin the formulas together according to the IC-Comp rule. Specifically, the IC-PK rule specifies that the encoding of a primary key constraint $\text{PK}(R, a)$ consists of two parts: Φ_1 asserting all attributes in the primary key have no NULL values, and Φ_2 stating for any pair of tuples t_1 and t_2 where $t_1 \neq t_2$, they do not agree on all attributes in $\vec{a}$. For $\text{FK}(R_1, a_1, R_2, a_2)$ where $R_1.a_1$ is a foreign key referencing $R_2.a_2$, the IC-FK rule looks up the symbolic database Γ and finds the tuples for R_1 are $t_1, \ldots, t_n$ and the tuples for R_2 are $t'_1, \ldots, t'_m$. The formula asserts that for each tuple t_i in R_1, there exists a tuple t'_j in R_2 such that the value $t_i.a_1$ is equal to $t'_j.a_2$. The IC-NN rule simply encodes that $\text{NotNull}(R, a)$ requires that all tuples in R must have a non-NULL value on attribute a. For the constraint $\text{Check}(R, \psi)$ that specifies ranges of values, the IC-Check rule uses an auxiliary function $[\![\psi]\!]_t$ (shown in Figure 9) to compute the formula of predicate ψ on each tuple t_i in R and obtain the formula by conjoining the $[\![\psi]\!]_t$ formulas together. For the auto-increment constraint $\text{Inc}(R, a, v)$, the IC-Inc rule also looks up the symbolic database Γ and finds all tuples $t_1, \ldots, t_n$ of R. It then enforces the value of $t_i.a$ is not NULL and that $t_i.a = v + i - 1$.

Lemma 4.2. [7] *Given a symbolic database Γ and an integrity constraint C, consider a formula Φ such that $\Gamma \vdash C \rightsquigarrow \Phi$. If Φ is satisfiable, then the model of Φ corresponds to a database consistent with Γ that satisfies C. If Φ is unsatisfiable, then no database consistent with Γ satisfies C.*

4.4 Encoding SQL Queries

To encode the semantics of a query, we traverse the query and encode how each operator transforms its input to output. Since this process requires the attributes and symbolic tuples of each subquery,

[7] The proof of all lemmas and theorems can be found in the Appendix of the extended version [He et al. 2024b].

$$\frac{R \in dom(\mathcal{S})}{\mathcal{S} \vdash R : \mathcal{S}(R)}\ \text{(A-Rel)} \qquad \frac{\mathcal{S} \vdash Q : \mathcal{A} \quad \forall e \in L. \forall a \in \text{Attrs}(e).\ a \in \mathcal{A}}{\mathcal{S} \vdash \Pi_L(Q) : L}\ \text{(A-Proj)} \qquad \frac{\mathcal{S} \vdash Q_1 : \mathcal{A}_1 \quad \mathcal{S} \vdash Q_2 : \mathcal{A}_2 \quad \otimes \in \{\times, \bowtie_\phi, \rtimes_\phi, \ltimes_\phi, \rhd\lhd_\phi\}}{\mathcal{S} \vdash Q_1 \otimes Q_2 : \mathcal{A}_1 +\!\!+ \mathcal{A}_2}\ \text{(A-Join)}$$

$$\frac{\mathcal{S} \vdash Q : \mathcal{A} \quad \mathcal{A}' = [\text{rename}(R, a) \mid a \in \mathcal{A}]}{\mathcal{S} \vdash \rho_R(Q) : \mathcal{A}'}\ \text{(A-Rename)} \qquad \frac{\mathcal{S} \vdash Q : \mathcal{A}}{\mathcal{S} \vdash \sigma_\phi(Q) : \mathcal{A}}\ \text{(A-Filter)}$$

$$\frac{\mathcal{S} \vdash Q : \mathcal{A}}{\mathcal{S} \vdash \text{GroupBy}(Q, \vec{E}, L, \phi) : \mathcal{A}}\ \text{(A-Group)} \qquad \frac{\mathcal{S} \vdash Q : \mathcal{A}}{\mathcal{S} \vdash \text{OrderBy}(Q, \vec{E}, b) : \mathcal{A}}\ \text{(A-Order)} \qquad \frac{\mathcal{S} \vdash Q : \mathcal{A}}{\mathcal{S} \vdash \text{Distinct}(Q) : \mathcal{A}}\ \text{(A-Dist)}$$

$$\frac{\oplus \in \{\cup, \cap, \setminus, \uplus, \oplus, -\} \quad \mathcal{S} \vdash Q_1 : \mathcal{A} \quad \mathcal{S} \vdash Q_2 : \mathcal{A}}{\mathcal{S} \vdash Q_1 \oplus Q_2 : \mathcal{A}}\ \text{(A-Coll)} \qquad \frac{\mathcal{S} \vdash Q_i : \mathcal{A}_i \quad 1 \le i \le n \quad n = |\vec{Q}| = |\vec{R}| \quad \mathcal{S}[R_1 \mapsto \mathcal{A}_1, \ldots, R_n \mapsto \mathcal{A}_n] \vdash Q' : \mathcal{A}'}{\mathcal{S} \vdash \text{With}(\vec{Q}, \vec{R}, Q') : \mathcal{A}'}\ \text{(A-With)}$$

Fig. 10. Rules for inferring the attributes of a query result. As is standard in SQL, the name of an attribute expression is assumed to be a function over all attributes involved in the expression. For example, the name of attribute expression Avg(a) is a string Avg_a.

we first describe how to compute attributes and tuples of arbitrary intermediate subqueries, followed by the encoding of all query operators.

4.4.1 Attributes of Intermediate Subqueries. While the database schema describes the attributes of each relation in the *initial* database, it does not directly give the attributes of those intermediate query results. To infer the attributes of each intermediate subquery, we develop an algorithm for all query operations in our SQL language. The algorithm is summarized as a set of inference rules, shown in Figure 10. Intuitively, our rules for inferring attributes are similar to the traditional typing rules. The judgments of the form $\mathcal{S} \vdash Q : \mathcal{A}$ mean the attributes of query Q is $\mathcal{A}$ under schema $\mathcal{S}$.

Specifically, the attributes of a relation in the initial database can be obtained by looking up the schema $\mathcal{S}$ directly (A-Rel). To find all attributes of a projection $\Pi_L(Q)$, the A-Proj rule first computes the attribute list $\mathcal{A}$ of query Q and then checks if all attributes occurring in the attribute expression list L belong to $\mathcal{A}$. If so, the attributes of $\Pi_L(Q)$ are L with all attribute expressions converted to the corresponding attribute names, e.g., Avg(a) to Avg_a. Otherwise, there is a type error in the query. For Cartesian product or any join operator $\otimes$, the attributes of $Q_1 \otimes Q_2$ are obtained by concatenating the attributes of Q_1 and those of Q_2 (A-Join). The renaming operation $\rho_R(Q)$ first computes the attributes of Q and then renames each of them according to the new relation name R in the result (A-Rename). Based on the A-Filter, A-Group, A-Order, and A-Dist rules, the filtering, GroupBy, OrderBy, and Distinct operations do not change the attribute list. The A-Coll rule specifies that for any collection operator $\oplus$, the attributes of its two operands Q_1 and Q_2 must be the same, which are also identical to the attributes of $Q_1 \oplus Q_2$. The inference rule for a `WITH` clause With$(\vec{Q}, \vec{R}, Q')$ is slightly more involved. Since the query Q' can use R_i to refer to the results of a subquery Q_i, the A-With rule first infers the attributes $\mathcal{A}_i$ of each subquery Q_i and then augments the schema $\mathcal{S}$ to a temporary new schema $\mathcal{S}'$ by adding entries $R_i \mapsto \mathcal{A}_i$. Finally, the attributes of query Q' (and the whole `WITH` clause) are inferred based on the schema $\mathcal{S}'$.

4.4.2 Symbolic Tuples of Intermediate Subqueries. Similar to the database schema, a symbolic database Γ only describes those tuples in each relation of the *initial* database, but it does not present what tuples are in the result of intermediate subqueries. To obtain the tuples in each intermediate subquery, we develop a set of inference rules as shown in Figure 11, where judgments of the form $\Gamma \vdash Q \hookrightarrow \mathcal{T}$ mean the result of query Q has a list of symbolic tuples $\mathcal{T}$ given the initial database Γ.

The goal of these rules is to compute the number (at most) of symbolic tuples in the result of each intermediate subquery and what are their names, so we just need to look up the tuples for relations

$$\frac{R \in dom(\Gamma)}{\Gamma \vdash R \hookrightarrow \Gamma(R)} \text{(T-Rel)} \quad \frac{\text{hasAgg}(L)}{\Gamma \vdash \Pi_L(Q) \hookrightarrow \sigma(1)} \text{(T-Agg)} \quad \frac{\neg\text{hasAgg}(L) \quad \Gamma \vdash Q \hookrightarrow \mathcal{T}}{\Gamma \vdash \Pi_L(Q) \hookrightarrow \sigma(|\mathcal{T}|)} \text{(T-Proj)}$$

$$\frac{\Gamma \vdash Q \hookrightarrow \mathcal{T}}{\Gamma \vdash \sigma_\phi(Q) \hookrightarrow \sigma(|\mathcal{T}|)} \text{(T-Filter)} \quad \frac{\Gamma \vdash Q \hookrightarrow \mathcal{T}}{\Gamma \vdash \rho_R(Q) \hookrightarrow \sigma(|\mathcal{T}|)} \text{(T-Rename)} \quad \frac{\Gamma \vdash Q_1 \hookrightarrow \mathcal{T}_1 \quad \Gamma \vdash Q_2 \hookrightarrow \mathcal{T}_2}{\Gamma \vdash Q_1 \times Q_2 \hookrightarrow \sigma(|\mathcal{T}_1| \cdot |\mathcal{T}_2|)} \text{(T-Prod)}$$

$$\frac{\Gamma \vdash \sigma_\phi(Q_1 \times Q_2) \hookrightarrow \mathcal{T}}{\Gamma \vdash Q_1 \bowtie_\phi Q_2 \hookrightarrow \mathcal{T}} \text{(T-IJoin)} \quad \frac{\begin{array}{c}\Gamma \vdash Q_1 \bowtie_\phi Q_2 \hookrightarrow \mathcal{T}_1 \\ \Gamma \vdash Q_1 \hookrightarrow \mathcal{T}_2 \\ \mathcal{T}' = \mathcal{T}_1 +\!\!+\, \sigma(|\mathcal{T}_2|)\end{array}}{\Gamma \vdash Q_1 ⟕_\phi Q_2 \hookrightarrow \mathcal{T}'} \text{(T-LJoin)} \quad \frac{\begin{array}{c}\Gamma \vdash Q_1 \bowtie_\phi Q_2 \hookrightarrow \mathcal{T}_1 \\ \Gamma \vdash Q_2 \hookrightarrow \mathcal{T}_2 \\ \mathcal{T}' = \mathcal{T}_1 +\!\!+\, \sigma(|\mathcal{T}_2|)\end{array}}{\Gamma \vdash Q_1 ⟖_\phi Q_2 \hookrightarrow \mathcal{T}'} \text{(T-RJoin)}$$

$$\frac{\Gamma \vdash Q_1 \bowtie_\phi Q_2 \hookrightarrow \mathcal{T}_1 \quad \Gamma \vdash Q_1 \hookrightarrow \mathcal{T}_2 \quad \Gamma \vdash Q_2 \hookrightarrow \mathcal{T}_3}{\Gamma \vdash Q_1 ⟗_\phi Q_2 \hookrightarrow \mathcal{T}_1 +\!\!+\, \sigma(|\mathcal{T}_2| + |\mathcal{T}_3|)} \text{(T-FJoin)} \quad \frac{\Gamma \vdash Q \hookrightarrow \mathcal{T}}{\Gamma \vdash \text{Distinct}(Q) \hookrightarrow \sigma(|\mathcal{T}|)} \text{(T-Dist)}$$

$$\frac{\Gamma \vdash Q \hookrightarrow \mathcal{T}}{\Gamma \vdash \text{GroupBy}(Q, \vec{E}, L, \phi) \hookrightarrow \sigma(|\mathcal{T}|)} \text{(T-GroupBy)} \quad \frac{\Gamma \vdash Q \hookrightarrow \mathcal{T}}{\Gamma \vdash \text{OrderBy}(Q, \vec{E}, b) \hookrightarrow \sigma(|\mathcal{T}|)} \text{(T-OrderBy)}$$

$$\frac{\oplus \in \{\cap, \Cap\} \quad \Gamma \vdash Q_1 \hookrightarrow \mathcal{T}_1}{\Gamma \vdash Q_1 \oplus Q_2 \hookrightarrow \sigma(|\mathcal{T}_1|)} \text{(T-Intx)} \quad \frac{\oplus \in \{\setminus, -\} \quad \Gamma \vdash Q_1 \hookrightarrow \mathcal{T}_1}{\Gamma \vdash Q_1 \oplus Q_2 \hookrightarrow \sigma(|\mathcal{T}_1|)} \text{(T-Ex)}$$

$$\frac{\begin{array}{c}\oplus \in \{\cup, \uplus\} \\ \Gamma \vdash Q_1 \hookrightarrow \mathcal{T}_1 \quad \Gamma \vdash Q_2 \hookrightarrow \mathcal{T}_2\end{array}}{\Gamma \vdash Q_1 \oplus Q_2 \hookrightarrow \sigma(|\mathcal{T}_1| + |\mathcal{T}_2|)} \text{(T-Union)} \quad \frac{\begin{array}{c}\Gamma \vdash Q_i \hookrightarrow \mathcal{T}_i \quad 1 \le i \le n \quad n = |\vec{Q}| = |\vec{R}| \\ \Gamma[R_1 \mapsto \mathcal{T}_1, \ldots, R_n \mapsto \mathcal{T}_n] \vdash Q' \hookrightarrow \mathcal{T}'\end{array}}{\Gamma \vdash \text{With}(\vec{Q}, \vec{R}, Q') \hookrightarrow \mathcal{T}'} \text{(T-With)}$$

Fig. 11. Rules for inferring the tuples of a query. $\sigma(n)$ generates a list of n fresh tuples.

in Γ (T-Rel) and generate fresh tuples for other queries. Specifically, if a projection $\Pi_L(Q)$ has aggregate functions in the attribute expression list L, there is only one tuple in the result (T-Agg). But if the projection $\Pi_L(Q)$ does not have aggregate functions, the number of tuples in the result is the same as that in Q. For filtering $\sigma_\phi(Q)$, the T-Filter rule generates the same number of tuples as Q, because the predicate ϕ may not filter out any tuple from Q. The T-Rename rule specifies that the renaming operation $\rho_R(Q)$ produces the same number of tuples as Q. Given the tuples of Q_1 are $\mathcal{T}_1$ and the tuples of Q_2 are $\mathcal{T}_2$, the numbers of fresh tuples for the Cartesian product, inner join, left outer join, right outer join, and full outer join are $|\mathcal{T}_1| \cdot |\mathcal{T}_2|$, $|\mathcal{T}_1| \cdot |\mathcal{T}_2|$, $|\mathcal{T}_1| \cdot (|\mathcal{T}_2| + 1)$, $(|\mathcal{T}_1| + 1) \cdot |\mathcal{T}_2|$, and $|\mathcal{T}_1| \cdot |\mathcal{T}_2| + |\mathcal{T}_1| + |\mathcal{T}_2|$, respectively. In addition, the numbers of fresh tuples for union, intersect, and except operations of Q_1 and Q_2 are $|\mathcal{T}_1| + |\mathcal{T}_2|$, $|\mathcal{T}_1|$, and $|\mathcal{T}_1|$, respectively. According to the T-Dist, T-GroupBy, and T-OrderBy rules, the Distinct, GroupBy, and OrderBy operation preserves the number of tuples in their subqueries. Finally, to infer the tuples for With$(\vec{Q}, \vec{R}, Q')$, we first need to obtain the tuples $\mathcal{T}_i$ for its subquery Q_i. Then we create a new symbolic database Γ' by adding mappings from tuples $\mathcal{T}_i$ to relation R_i to Γ. Finally, we infer the tuples $\mathcal{T}'$ for Q' given the new symbolic database Γ' and get the result of With$(\vec{Q}, \vec{R}, Q')$ based on the T-With rule.

4.4.3 Symbolic Encoding of Query Operators. Since our query language supports various complex attribute expressions and predicates, we introduce two auxiliary functions $[\![e]\!]_{\mathcal{S},\Gamma,\mathcal{T}}$ and $[\![\phi]\!]_{\mathcal{S},\Gamma,\mathcal{T}}$ to encode attribute expressions and predicates in a query, respectively[8]. Intuitively, $[\![\cdot]\!]_{\mathcal{S},\Gamma,\mathcal{T}}$ evaluates an attribute expression or a predicate in a recursive fashion given the schema $\mathcal{S}$, database Γ, and a list of tuples $\mathcal{T}$. For example, suppose a is an attribute of an relation and t is a tuple of that relation, $[\![a+1]\!]_{\mathcal{S},\Gamma,[t]} = [\![a]\!]_{\mathcal{S},\Gamma,[t]} + [\![1]\!]_{\mathcal{S},\Gamma,[t]} = t.a + 1$. Furthermore, it is worthwhile to point out that the tuple list $\mathcal{T}$ has more than one tuple in several cases when evaluating `GROUP BY` and aggregate

[8]We precisely define these auxiliary functions in the appendix of the extended version [He et al. 2024b].

$$\frac{R \in \mathrm{Dom}(\Gamma)}{\mathcal{S}, \Gamma \vdash R \rightsquigarrow \top}\ \text{(E-Rel)} \qquad \frac{\begin{array}{c}\neg\mathrm{hasAgg}(L) \quad \mathcal{S}, \Gamma \vdash Q \rightsquigarrow \Phi_1 \\ \Gamma \vdash Q \hookrightarrow [t_1, \ldots, t_n] \\ \mathcal{S} \vdash \Pi_L(Q) : [a'_1, \ldots, a'_l] \quad \Gamma \vdash \Pi_L(Q) \hookrightarrow [t'_1, \ldots, t'_n] \\ \Phi_2 = \wedge_{i=1}^{n}(\wedge_{j=1}^{l}[\![a'_j]\!]_{\mathcal{S},\Gamma,[t'_i]} = [\![a'_j]\!]_{\mathcal{S},\Gamma,[t_i]} \wedge \mathrm{Del}(t'_i) \leftrightarrow \mathrm{Del}(t_i))\end{array}}{\mathcal{S}, \Gamma \vdash \Pi_L(Q) \rightsquigarrow \Phi_1 \wedge \Phi_2}\ \text{(E-Proj)}$$

$$\frac{\begin{array}{c}\Gamma \vdash Q \hookrightarrow [t_1, \ldots, t_n] \quad \Gamma \vdash \sigma_\phi(Q) \hookrightarrow [t'_1, \ldots, t'_n] \quad \mathcal{S}, \Gamma \vdash Q \rightsquigarrow \Phi_1 \\ \Phi_2 = \wedge_{i=1}^{n}((\neg\mathrm{Del}(t_i) \wedge [\![\phi]\!]_{\mathcal{S},\Gamma,[t_i]} = \top) \rightarrow t'_i = t_i \wedge (\mathrm{Del}(t_i) \vee [\![\phi]\!]_{\mathcal{S},\Gamma,[t_i]} \neq \top) \rightarrow \mathrm{Del}(t'_i))\end{array}}{\mathcal{S}, \Gamma \vdash \sigma_\phi(Q) \rightsquigarrow \Phi_1 \wedge \Phi_2}\ \text{(E-Filter)}$$

$$\frac{\begin{array}{l}\mathcal{S} \vdash Q_1 : [a_1, \ldots, a_p] \quad \Gamma \vdash Q_1 \hookrightarrow [t_1, \ldots, t_n] \quad \mathcal{S}, \Gamma \vdash Q_1 \rightsquigarrow \Phi_1 \\ \mathcal{S} \vdash Q_2 : [a'_1, \ldots, a'_q] \quad \Gamma \vdash Q_2 \hookrightarrow [t'_1, \ldots, t'_m] \quad \mathcal{S}, \Gamma \vdash Q_2 \rightsquigarrow \Phi_2 \quad \Gamma \vdash Q_1 \times Q_2 \hookrightarrow [t''_{1,1}, \ldots, t''_{n,m}] \\ \Phi_3 = \wedge_{i=1}^{n} \wedge_{j=1}^{m} (\neg\mathrm{Del}(t_i) \wedge \neg\mathrm{Del}(t'_j)) \rightarrow \wedge_{k=1}^{p}[\![a_k]\!]_{\mathcal{S},\Gamma,[t''_{i,j}]} = [\![a_k]\!]_{\mathcal{S},\Gamma,[t_i]} \wedge \wedge_{k=1}^{q}[\![a'_k]\!]_{\mathcal{S},\Gamma,[t''_{i,j}]} = [\![a'_k]\!]_{\mathcal{S},\Gamma,[t'_j]} \wedge \\ \quad (\mathrm{Del}(t_i) \vee \mathrm{Del}(t'_j)) \rightarrow \mathrm{Del}(t''_{i,j})\end{array}}{\mathcal{S}, \Gamma \vdash Q_1 \times Q_2 \rightsquigarrow \Phi_1 \wedge \Phi_2 \wedge \Phi_3}\ \text{(E-Prod)}$$

Fig. 12. Sample inference rules for encoding SQL queries. $\mathrm{Del}(t)$ indicates that a tuple t is deleted.

queries. For instance, $[\![\mathrm{AVG(EMP.age)}]\!]_{\mathcal{S},\Gamma,[t_1,t_2,\ldots,t_n]}$ is used to compute the average age of EMP where the EMP table has n symbolic tuples.

Our encoding algorithm for query operators is represented as a set of inference rules. Judgments are of the form $\mathcal{S}, \Gamma \vdash Q \rightsquigarrow \Phi$, meaning the encoding of query Q is formula Φ given schema $\mathcal{S}$ and database Γ. Figure 12 presents a sample set of such inference rules.

Relation. The encoding for a simple relation query is trivially $\top$, because the result can be obtained from the database Γ directly, i.e., the output is the same as input.

Filtering. The E-Filter rule specifies how to encode filtering operations. In particular, given a query $\sigma_\phi(Q)$, we can first determine the input (result of Q) is $[t_1, \ldots, t_n]$ and the output should be $[t'_1, \ldots, t'_n]$ based on the rules in Figure 11. Then we can generate a formula Φ_2 to describe the relationship between $[t'_1, \ldots, t'_n]$ and $[t_1, \ldots, t_n]$. Specifically, if a tuple t_i is not deleted and the predicate ϕ evaluates to be $\top$ on t_i, then it is retained in the result, i.e., $t'_i = t_i$. Otherwise, the corresponding output tuple t'_i is deleted. The final formula encoding $\sigma_\phi(Q)$ is the conjunction of Φ_2 and the formula Φ_1 that encodes Q.

Projection. According to the E-Proj rule, to encode $\Pi_L(Q)$ where L does not contain aggregate functions, we can first infer that its input is $[t_1, \ldots, t_n]$ with attributes $[a_1, \ldots, a_m]$ and output is $[t'_1, \ldots, t'_n]$ with attributes $[a'_1, \ldots, a'_l]$ based on the rules in Figure 11 and Figure 10. Then for each a'_k, we find its corresponding index c_k in the attributes of Q and generate a formula Φ_2 that asserts for each tuple and its output (t_i, t'_i), they have the same Del status and they agree on attribute a'_k. The E-Agg rule for encoding projection with aggregate functions is similar to E-Proj. The main difference is that it only generates one tuple t'_1 in the output and sets the aggregated value based on all input tuples, i.e., $[\![a'_j]\!]_{\mathcal{S},\Gamma,[t'_1]} = [\![a'_j]\!]_{\mathcal{S},\Gamma,\vec{t}}$.

Cartesian product. Based on the E-Prod rule to encode $Q_1 \times Q_2$, we just need to generate a formula Φ_3 that encodes an output tuple $t''_{i,j}$ is obtained by concatenating a tuple t_i from Q_1 and a tuple t'_j from Q_2. Specifically, Φ_3 describes the attributes of $t''_{i,j}$ agree with those of t_i and t'_j, and $t''_{i,j}$ is deleted if either t_i or t'_j is deleted.

Left outer join. The E-LJoin rule specifies the encoding of a left outer join $Q_1 \rtimes_\phi Q_2$ is based on the formula Φ_1 of inner join $Q_1 \bowtie_\phi Q_2$ (which is a syntactic sugar of $\sigma_\phi(Q_1 \times Q_2)$). In addition to Φ_1, the encoding also includes Φ_2, which describes the output tuples from Q_1's null extension.

Other operations. The rules for encoding other query operations are in similar flavor of the above rules. Specifically, the high-level idea is to first obtain the schema and tuples for its input and output based on the rules in Figure 10 and Figure 11. Then we can encode the relationship

between the input and output tuples and generate a formula that encodes the query semantics. Due to page limit, the complete set of our inference rules are presented in the Appendix of the extended version [He et al. 2024b]. These rules closely follow the standard three-valued logic when evaluating expressions and predicates that involve NULL's. For example, the predicate NULL = NULL evaluates to NULL, which is consistent with the three-valued logic.

Definition 4.3 (Interpretation). An interpretation of a formula is a mapping from variables, function symbols, and predicate symbols in the formula to values, functions, and predicates, respectively.

Definition 4.4 (Interpretation Extension). Interpretation $\mathcal{I}_1$ is an *extension* of interpretation $\mathcal{I}_2$, denoted $\mathcal{I}_1 \sqsupseteq \mathcal{I}_2$, if (1) $\mathsf{Dom}(\mathcal{I}_1) \supseteq \mathsf{Dom}(\mathcal{I}_2)$, (2) $\forall v \in \mathsf{Vars}(\mathcal{I}_2).\ \mathcal{I}_1(v) = \mathcal{I}_2(v)$, and (3) for the Del predicate, $\forall v \in \mathsf{Vars}(\mathcal{I}_1).\ \mathcal{I}_2(\mathrm{Del})(v) \leftrightarrow \mathcal{I}_1(\mathrm{Del})(v)$.

Intuitively, interpretation $\mathcal{I}_1$ is an extension of $\mathcal{I}_2$ if (1) all variables, functions, and predicates defined by $\mathcal{I}_2$ are also defined by $\mathcal{I}_1$, (2) $\mathcal{I}_1$ preserves the values of variables occurring in $\mathcal{I}_2$ while it assigns values to new variables, and (3) $\mathcal{I}_1$ preserves the definition of Del predicate on variables occurring in $\mathcal{I}_2$.

Theorem 4.5. *Let $\mathcal{D}$ be a database over schema $\mathcal{S}$ and o_R be the output relation of running query Q over $\mathcal{D}$. Consider a symbolic database Γ over $\mathcal{S}$, a symbolic relation R and a formula Φ such that $\Gamma \vdash Q \hookrightarrow R$ and $\mathcal{S}, \Gamma \vdash Q \rightsquigarrow \Phi$. For any interpretation $\mathcal{I}$ such that $\mathcal{I}(\Gamma) = \mathcal{D}$, there exists an extension $\mathcal{I}'$ of $\mathcal{I}$ such that (1) the result of query Q over $\mathcal{D}$ is $\mathcal{I}'(R)$, and (2) the database $\mathcal{D}$ and $\mathcal{I}'$ jointly satisfy Φ, i.e.,* [9]

$$(\Gamma \vdash Q \hookrightarrow R) \wedge (\mathcal{S}, \Gamma \vdash Q \rightsquigarrow \Phi) \wedge (\mathcal{I}(\Gamma) = \mathcal{D}) \Rightarrow \exists \mathcal{I}' \sqsupseteq \mathcal{I}.\ \mathcal{I}'(R) = [\![Q]\!]_{\mathcal{D}} \wedge (\mathcal{I}', \mathcal{D}[o_R \mapsto \mathcal{I}'(R)] \models \Phi)$$

Theorem 4.6. *Let Γ be a symbolic database over schema $\mathcal{S}$ and Q be a query. Consider a symbolic relation R and a formula Φ such that $\Gamma \vdash Q \hookrightarrow R$ and $\mathcal{S}, \Gamma \vdash Q \rightsquigarrow \Phi$. If Φ is satisfiable, then for any satisfying interpretation $\mathcal{I}$ of Φ, running Q over the concrete database $\mathcal{I}(\Gamma)$ yields the relation $\mathcal{I}(R)$, i.e.,*

$$(\Gamma \vdash Q \hookrightarrow R) \wedge (\mathcal{S}, \Gamma \vdash Q \rightsquigarrow \Phi) \wedge (\mathcal{I} \models \Phi) \Rightarrow [\![Q]\!]_{\mathcal{I}(\Gamma)} = \mathcal{I}(R)$$

4.5 Equivalence under Bag and List Semantics

We first discuss how to check the equality of symbolic tuples and then present how to check the equality of query outputs under *both* bag and list semantics. We use bag semantics by default, but for queries that involve sorting, we switch to list semantics to preserve the order of sorted tuples.

Equality of symbolic tuples. Recall from Figure 3b that a tuple is represented by a list of pairs, where each pair consists of an attribute name and its corresponding value. Two symbolic tuples t and t' are considered equal iff (1) they have the same number of pairs and (2) their corresponding symbolic values at the same index are equal. Specifically, suppose $t = [(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)]$ and $t' = [(a'_1, v'_1), (a'_2, v'_2), \ldots, (a'_n, v'_n)]$ where a_i, a'_i are the attribute names and v_i, v'_i are their corresponding values. We say $t = t'$ if $v_i = v'_i$ holds for all $1 \leq i \leq n$. Note that the attribute names are ignored for equality comparison to support the renaming operations in SQL.

Equivalence under bag semantics. Given two query results with symbolic tuples $R_1 = [t_1, \ldots, t_n]$ and $R_2 = [r_1, \ldots, r_m]$, we encode the following two properties to ensure R_1 and R_2 are equivalent under bag semantics:

[9] We slightly abuse the notation $\mathcal{I}(x)$ over lists to apply $\mathcal{I}$ to each element in list x. We also abuse the notation $\mathcal{I}(x)$ over maps to apply $\mathcal{I}$ to each value in the range of map x.

- R_1 and R_2 have the same number of non-deleted tuples.

$$\sum_{i=1}^{n} \mathbb{I}[\neg\text{Del}(t_i)] = \sum_{j=1}^{m} \mathbb{I}[\neg\text{Del}(r_j)] \tag{1}$$

  where $\mathbb{I}[b]$ is an indicator function that evaluates to 1 when b is true or 0 when b is false.
- For each non-deleted symbolic tuple in R_1, its multiplicity in R_1 is equal to its multiplicity in R_2.

$$\bigwedge_{i=1}^{n} \left(\sum_{j=1}^{n} \mathbb{I}[\neg\text{Del}(t_i) \wedge \neg\text{Del}(t_j) \wedge t_i = t_j] = \sum_{k=1}^{m} \mathbb{I}[\neg\text{Del}(t_i) \wedge \neg\text{Del}(r_k) \wedge t_i = r_k] \right) \tag{2}$$

  Here, we include the predicates $\neg\text{Del}(t_i)$, $\neg\text{Del}(t_j)$, and $\neg\text{Del}(r_k)$ to only count those non-deleted symbolic tuples in R_1 and R_2.

Equivalence under list semantics. Since the equivalence of two queries is only checked when they involve sorting operations, the corresponding query results $R_1 = [t_1, \ldots, t_n]$ and $R_2 = [r_1, \ldots, r_m]$ are sorted using the OrderBy operator. Recall from Figure 4 that OrderBy must be the last operation of a query. Furthermore, our encoding of OrderBy ensures that all deleted tuples are placed at the end of the list and non-deleted tuples are sorted in ascending or descending order, so we just need to encode the following two properties to assert R_1 and R_2 are equivalent under list semantics.

- R_1 and R_2 have the same number of non-deleted tuples.

$$\sum_{i=1}^{n} \mathbb{I}[\neg\text{Del}(t_i)] = \sum_{j=1}^{m} \mathbb{I}[\neg\text{Del}(r_j)] \tag{3}$$

- The tuples with the same index in R_1 and R_2 are equal.

$$\bigwedge_{i=1}^{\min\{m,n\}} t_i = r_i \tag{4}$$

Intuitively, we just need to check the pair-wise equality of tuples until the end of R_1 or R_2, because the OrderBy operator has moved the deleted tuples to the end of the list. This OrderBy encoding greatly simplifies the equality check. Alternatively, one can encode the check without the assumption of OrderBy moving deleted tuples to the end. The key idea is to maintain two pointers, one on each list, moving from the beginning to the end of the lists. Every time a pointer moves to a new location, it checks whether or not the corresponding tuple is deleted. If the tuple is not deleted, then it checks if the tuple is equal to the tuple pointed by the other pointer. Otherwise, if the tuple is deleted, it moves to the next location.

Lemma 4.7. *Given two relations $R_1 = [t_1, \ldots, t_n]$ and $R_2 = [r_1, \ldots, r_m]$, if the formula (1) $\wedge$ (2) is valid, then R_1 is equal to R_2 under bag semantics. If the formula (3) $\wedge$ (4) is valid, then R_1 is equal to R_2 under list semantics.*

Theorem 4.8. *Given two queries Q_1, Q_2 under schema $\mathcal{S}$, an integrity constraint C, a bound N, if $\textsc{Verify}(Q_1, Q_2, \mathcal{S}, C, N)$ returns $\top$, then $Q_1 \simeq_{\mathcal{S},C,N} Q_2$. Otherwise, if $\textsc{Verify}(Q_1, Q_2, \mathcal{S}, C, N)$ returns a database $\mathcal{D}$, then $\mathcal{D} :: \mathcal{S}$ and $[\![Q_1]\!]_{\mathcal{D}} \neq [\![Q_2]\!]_{\mathcal{D}}$.*

5 IMPLEMENTATION

Based on the techniques in Section 4, we have implemented a tool called VeriEQL for verifying the bounded equivalence of SQL queries modulo integrity constraints. VeriEQL uses the Z3 SMT solver [de Moura and Bjørner 2008] for constraint solving and model generation. In this section, we discuss several details that are important to the implementation of VeriEQL.

Attribute renaming. We pre-process the queries to resolve attribute renaming issues before starting the verification. Specifically, we design a unique identifier generator and a name cache

for pre-processing. The unique identifier generator allows VeriEQL to efficiently track attributes while the name cache pool stores temporary aliases of an attribute under different scopes and automatically updates them for scope change.

Sorting symbolic tuples. To sort symbolic tuples based on an `ORDER BY` operation, we implement an encoding for the standard bubble sort algorithm. More specifically, we first move those deleted tuples to the end of the list, and then sort non-deleted tuples based on the sorting criteria specified by `ORDER BY`. As part of the bubble sort algorithm, two adjacent tuples t_1, t_2 in the list are swapped under two conditions: (1) $t_1 \neq$ Null whereas $t_2 =$ Null, or (2) t_1, t_2 are not Null but $t_1 < t_2$ for ascending order ($t_1 > t_2$ for descending order).

6 EVALUATION

This section presents a series of experiments that are designed to answer the following questions.

- **RQ1**: How does VeriEQL compare with state-of-the-art SQL equivalence checking techniques in terms of *coverage*? That is, does VeriEQL support more complex queries? (Section 6.2)
- **RQ2:** How effective is VeriEQL at *disproving* query equivalence? Does VeriEQL generate useful counterexamples to facilitate downstream tasks? (Section 6.3)
- **RQ3**: How large input bounds can VeriEQL reach during verification? (Section 6.4)

6.1 Experimental Setup

Benchmarks. We collected benchmarks — each of which is a pair of queries — from three different workloads. To the best of our knowledge, we have incorporated *all* benchmarks publicly available from recent work on query equivalence checking [Chu et al. 2017a, 2018, 2017b,c; Wang et al. 2018a; Zhou et al. 2019, 2022]. Additionally, we also curated a very large collection of new queries that, to the best of our knowledge, has not been used in any prior work.

- `LeetCode`. This is a new dataset curated by us containing a total of 23,994 query pairs (Q_1, Q_2). To curate this dataset, we first crawled *all* publicly available queries accepted by LeetCode [LeetCode 2023] that are syntactically distinct. Then, we grouped them based on the LeetCode problem — queries in the same group are written and submitted by actual users that solve the same problem. For each problem, we have also manually written a "ground-truth" query that is *guaranteed* to be correct. Finally, each *user query* Q_1 is paired with its corresponding *ground-truth query* Q_2, yielding a total of 23,994 query *pairs*. In each pair, Q_1 is supposed to be equivalent to Q_2, which, however, is not always the case due to missing test cases on LeetCode. Indeed, as we will see shortly, VeriEQL has identified many wrong queries confirmed by LeetCode. As a final note: queries in our dataset naturally exhibit diverse patterns since they were written by different people with diverse backgrounds around the world, which makes this particular dataset especially valuable for evaluating query equivalence checkers. We also manually formalize the schema and integrity constraint for each LeetCode problem.
- `Calcite`. Our second benchmark set is constructed from the Calcite's optimization rules test suite [Calcite 2023]. This is a standard workload used extensively in prior work [Chu et al. 2017a, 2018, 2017b,c; Wang et al. 2018a; Zhou et al. 2019, 2022]. In particular, each Calcite test case has a pair of queries (Q_1, Q_2), where Q_1 can be rewritten to Q_2 using the optimization rule under test. In other words, Q_1 and Q_2 should be equivalent to each other, since all optimization rules are supposed to preserve the semantics. We included query pairs from *all* test cases and ended up with a total of 397 query pairs. This number is noticeably higher than 232 pairs reported in prior work [Zhou et al. 2022], because the Calcite project is under active development and prior work

used test cases from the version in year 2018. The Calcite project also includes necessary schema and integrity constraints, which were all incorporated into our benchmark suite.

- `Literature`. Finally, we used *all* other benchmarks from recent work that are publicly available. In particular, the Cosette series [Chu et al. 2017a, 2018, 2017b,c; Cosette 2018] of work has 38 benchmarks — all of them are included in our suite (8 involve integrity constraints). We also used all 26 benchmarks from [Wang et al. 2018a]. In total, our `Literature` dataset has 64 benchmarks.

Across all our workloads, the average query size is 84.54 (measured by the number of AST nodes), where the max, min, and median are 679, 5, and 73, respectively. In addition to the large query size, our benchmarks are complex in terms of query nesting. In particular, 57.7% have sub-queries (e.g., `SELECT * FROM T1 WHERE T1.id IN (SELECT uid FROM T2)`), and a number of them have four levels of query-nesting. In addition, 95.7% of the queries involve joining multiple tables, and 33.2% use the outer-join operation, which is generally quite challenging to verify. Finally, our queries also use other SQL features, such as group-by (58.7%), aggregate functions (65.3%), and order-by (21.6%).

Baselines. We compare VeriEQL against *all* state-of-the-art *bounded* equivalence checking techniques that have corresponding tools publicly available. These tools all accept a smaller subset of SQL than VeriEQL.

- An extended version of Cosette [Chu et al. 2017a,b] which uses the provenance-base pruning technique from [Wang et al. 2018a] to speed up the original Cosette technique. This baseline represents the state-of-the-art along the Cosette line of work. It does not support any integrity constraints.[10] For brevity, we call this baseline Cosette.
- An extension of Qex [Veanes et al. 2010] that incorporates the provenance-base pruning technique from [Wang et al. 2018a] and significantly outperforms the original Qex. For brevity, we call this baseline Qex. This baseline does not consider integrity constraints.

In addition, we compare VeriEQL against two SQL testing tools that cannot verify equivalence of two queries (neither in a bounded nor unbounded way) but can potentially generate counterexamples to disprove their equivalence.

- DataFiller [Coelho 2013; Guagliardo and Libkin 2017] is a random database generator that produces databases based on the schema and integrity constraints. We use DataFiller to generate random databases and run two queries over the databases to check if the outputs are different.
- XData [Chandra et al. 2015] is a constraint-based mutation testing tool that can generate databases to disprove equivalence of two queries.

While not apples-to-apples, we also compare VeriEQL with state-of-the-art *full* (i.e., unbounded) verification techniques that can prove query equivalence. None of these tools are able to generate counterexamples and all of them support a smaller subset of SQL than VeriEQL.

- SPES [Zhou et al. 2022] is a state-of-the-art verifier from the databases community. It does not support query operations beyond select-project-join (e.g., `ORDER BY` and set operations such as `INTERSECT`), and has very limited support for integrity constraints[11].
- HoTTSQL [Chu et al. 2017c] and UDP [Chu et al. 2018] use a proof assistant (in particular, Coq and Lean respectively) to prove query equivalence. They are quite different from the aforementioned techniques that utilize an automated theorem prover (e.g., Z3) and require more manual effort.

[10] Cosette assumes all columns are NOT NULL — which can be viewed as a "built-in integrity constraint" — even if a column can in fact be NULL. In other words, Cosette may not consider all valid inputs and, therefore, may be incomplete.
[11] While the SPES paper claims to accept simple primary key constraints, the artifact (https://github.com/georgia-tech-db/spes) is not parameterized with any constraints. Instead, it is specialized to Calcite's tables, schema, and integrity constraints.

Since UDP does not have a publicly available artifact that is usable, we include HoTTSQL as a baseline to facilitate a complete and thorough evaluation.

6.2 RQ1: Coverage and Comparison against State-of-the-Art Techniques

In this section, we report the number of (1) query pairs whose *bounded* equivalence (against a space of bounded-size inputs) can be successfully verified by VeriEQL and (2) those that can be proved non-equivalent (i.e., counterexamples are generated). We also compare VeriEQL with baselines.

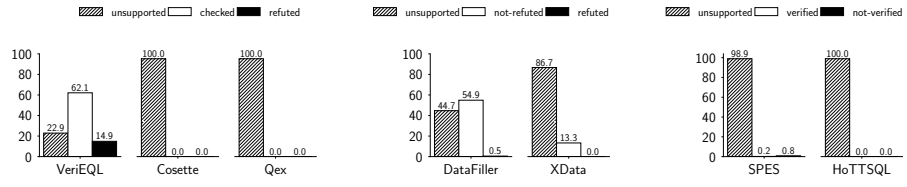
Setup. Given a benchmark consisting of queries (Q_1, Q_2) with their schema and integrity constraint, we run VeriEQL using a 10-minute timeout. There are three possible outcomes for each benchmark: (1) unsupported, (2) checked, or (3) refuted. "unsupported" means VeriEQL is not applicable to the benchmark (e.g., due to unsupported integrity constraints or SQL features). Otherwise, VeriEQL would report either "checked" (meaning bounded equivalence) or "refuted" (with a counterexample witnessing the non-equivalence of Q_1 and Q_2). The bound is incrementally increased from 1 until the timeout is reached or a counterexample is identified. If no counterexample is found before timeout and the bounded equivalence is verified for at least bound 1, we report "checked".

The same setup is used for bounded verification baselines, namely Cosette and Qex. For testing baselines (i.e., DataFiller and XData), there are three possible outcomes: (1) unsupported, (2) not-refuted, and (3) refuted. Since DataFiller is a random database generator, we use it to generate 1,000 random databases where each relation has 100 tuples for each supported benchmark. If any of these databases leads to different execution results on the two queries, we report it as "refuted"; otherwise, we report "not-refuted". By contrast, XData performs mutation testing, so we only run the tool once on each supported benchmark. We report "refuted" if it finds a counterexample of equivalence and report "not-refuted" otherwise. For unbounded verification baselines (i.e., SPES and HoTTSQL), there are three possible outcomes: (1) unsupported, (2) verified, and (3) not-verified. "Verified" means the full equivalence of two queries is verified, and "not-verified" means the result is unknown (as none of them can generate counterexamples). This leads to an apples-to-oranges comparison; nevertheless, we include the results as well for a complete evaluation. All baseline tools use a 10-minute timeout, which is the same as VeriEQL.

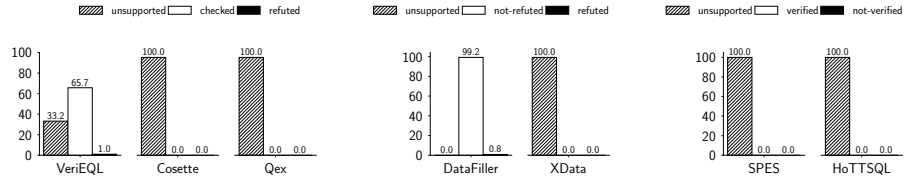
RQ1 take-away: VeriEQL supports over 75% of the benchmarks, which is more than all baselines. VeriEQL also significantly outperforms all baselines across all workloads in terms of both disproving equivalence and proving (bounded) equivalence of query pairs.

Results. Our main result for each workload is shown in Figures 13a, 13b, and 13c. For the `LeetCode` dataset, VeriEQL supports 77.1% of benchmarks, which is higher than all baselines. This very large gap is due to baselines' poor support for integrity constraints and complex SQL features such as `ORDER BY`. Furthermore, VeriEQL is able to disprove equivalence for 14.9% of the `LeetCode` benchmarks, whereas none of the *bounded* verification baselines can disprove any. Unbounded verification tools cannot disprove equivalence by nature, and only the testing tool DataFiller can disprove 0.5%. For `Calcite` (see Figure 13b), the result is similar. The only difference is that DataFiller can support all benchmarks, but it can only refute 0.8% of benchmarks, smaller than the 1.0% of VeriEQL. For `Literature` (see Figure 13c), VeriEQL supports 98.4% of benchmarks, which outperforms most of baselines other than DataFiller. It is comparable with bounded verification baselines, hypothetically because baselines were better-engineered for `Literature` queries, but it still outperforms testing baseline on disproving equivalence.

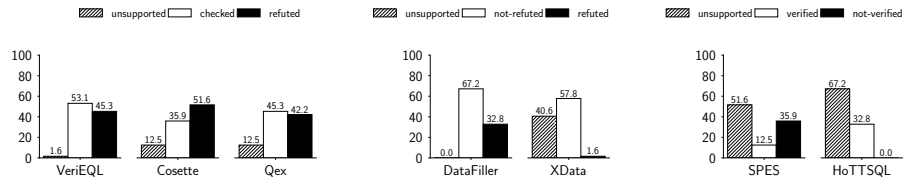
Qualitative Analysis on Failure to Disprove Equivalence. To understand whether two queries are indeed equivalent if VeriEQL reports "checked", we perform manual inspections during the evaluation. Given that we have over 24,000 benchmarks in total and exhaustive inspection is not



(a) *LeetCode.*



(b) *Calcite.*



(c) *Literature.*

Fig. 13. **RQ1**: Each workload has a sub-figure that shows, for each tool, the percentage of benchmarks (i.e., query pairs) in each category: unsupported/checked/refuted for bounded verification, unsupported/not-refuted/refuted for testing, unsupported/verified/not-verified for unbounded verification.

practically feasible, we perform a non-exhaustive manual inspection. Specifically, we sampled 50 benchmarks from `LeetCode` where VeriEQL reported "checked" and confirmed all of them are indeed equivalent. In addition, for the benchmarks that can be proved equivalent by other tools (i.e., SPES), VeriEQL also agrees on the result. This serves as some evidence on which we believe the equivalences checked by VeriEQL are valid. For `Calcite` dataset, we found that one benchmark where VeriEQL reported "checked" but one baseline tool (namely Cosette) reported "refuted". After further inspection, we believe this benchmark should be equivalent but Cosette gave a spurious counterexample input on which both queries produce the same output table. We suspect Cosette may have an implementation bug that led to spurious counterexamples. For `Literature` dataset, we manually inspected all of the 34 "checked" benchmarks and found that 27 are indeed equivalent but 7 are not equivalent. Among these 7 non-equivalent benchmarks, baselines generate spurious counterexamples for 5 of them; in other words, baselines also cannot disprove these 5. The remaining two non-equivalent benchmarks need large input databases (e.g., with >1000 tuples) in order to be differentiated; within our 10-minute timeout, VeriEQL cannot disprove them but Cosette can. The reason is that Cosette has some internal heuristics that consider specific inputs with more tuples before exhausting the space of smaller inputs, whereas VeriEQL simply increments the size of the input database from one.[12]

Small-Scope Hypothesis. Our evaluation echos the small-scope hypothesis discussed in prior work [Miao et al. 2019]: "mistakes in most of the queries may be explained by only a small number of tuples." There are only a small number of sources in SQL queries that could potentially break the

[12]Even with this simple size-increasing strategy, VeriEQL can disprove 5 `Literature` benchmarks for which Cosette reports "checked."

small scope hypothesis, such as `LIMIT` clause [13] or aggregation function `COUNT` with a large constant. Empirically, we observe that the small scope hypothesis holds for most of all our benchmarks. We only find two benchmarks that require an input relation with more than 1000 tuples to disprove equivalence; both of them have a clause like `COUNT(a) > 1000`.

Limitations of Bounded Equivalence Verification. While in principle the bounded equivalence verification approach taken by VeriEQL can disprove or prove equivalence of two SQL queries for all input relations up to a finite size, it may not be able to disprove equivalence within a practical amount of time if the counterexample can only be large input relations. For example, during our evaluation, VeriEQL failed to disprove equivalence for the following two queries from the `Literature` dataset:

```
SELECT C.name FROM Carriers AS C JOIN Flights AS F ON C.cid = F.carrier_id
    GROUP BY C.name, F.year, F.month_id, F.day_of_month HAVING COUNT (F.fid) > 1000
SELECT C.name FROM Carriers AS C JOIN Flights AS F ON C.cid = F.carrier_id
    GROUP BY C.name, F.day_of_month HAVING COUNT (F.fid) > 1000
```

In fact, a counterexample refuting their equivalence has more than 1000 tuples, which VeriEQL cannot find within the 10-minute time limit.

Discussion. Interested readers might wonder why baselines perform poorly on `Calcite` given it was used extensively as a benchmark suite in prior work [Chu et al. 2017b; Wang et al. 2018a]. The reason is that Cosette, Qex, and HoTTSQL do not support the `NOT NULL` constraint (among others) required in all `Calcite` benchmarks. As a result, we directly report "unsupported" in **RQ1** setup. Results reported in prior work were obtained by running the tools *without considering* constraints (like `NOT NULL`). One may also wonder why VeriEQL has 1% (i.e., 4 benchmarks) "not checked" for `Calcite`, where all Calcite's query pairs are supposed to be equivalent. We manually inspected these benchmarks and believe some of Calcite's rewrite rules are not equivalence preserving (i.e., wrong), leading to non-equivalent query pairs — we will expand on this in **RQ2**.

6.3 RQ2: Effectiveness at Generating Counterexamples to Facilitate Downstream Tasks

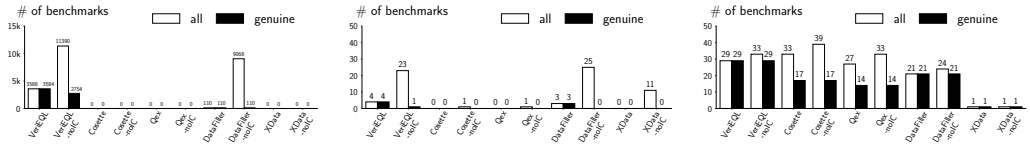
So far, we have seen the overall result of VeriEQL and how that compares against baselines. In **RQ2**, we will concentrate on those "refuted" benchmarks and evaluate VeriEQL's capability at generating counterexamples. Notably, VeriEQL's counterexamples revealed serious bugs in MySQL and Calcite, and suggested new test inputs to augment LeetCode's existing test suite.

Setup. Since the unbounded verifiers do not generate counterexamples, let us answer the following question: given each workload, for each *bounded verification* and *testing* approach (including VeriEQL and baselines), how many benchmarks did the tool identify a counterexample for? How many of these counterexamples are genuine (i.e., not spurious)? Here, a counterexample is genuine if (i) it meets the integrity constraint associated with the benchmark and (ii) the two queries yield different outputs. To study the impact of integrity constraints, we run two versions of each tool: (1) the original version as in **RQ1**, and (2) a version denoted by a suffix "-noIC" that drops all integrity constraints when running the tool (but we still consider integrity constraints when checking if the counterexample is genuine).

> **RQ2 take-away:** VeriEQL can disprove equivalence for significantly more benchmarks than all bounded baselines across all workloads. VeriEQL can also generate counterexamples to help identify bugs in real-world systems and augment existing test suites.

Results. Our main results are presented in Figure 14. In a nutshell, VeriEQL can prove much more non-equivalent benchmarks than all baselines across all workloads. For example, VeriEQL found counterexamples for 3,586 `LeetCode` benchmarks; 3,584 of them are confirmed to be genuine. The

[13] Recall from Figure 4 that we do not support `LIMIT` clauses.



(a) *LeetCode.* (b) *Calcite.* (c) *Literature.*

Fig. 14. **RQ2**: We have a sub-figure for each workload that shows (1) the number of *all* benchmarks for which a tool generates a counterexample and (2) the number of benchmarks where the counterexamples are *genuine*.

counterexample inputs for the remaining two benchmarks conform to the integrity constraints but they produce the same output when executed on the input *using MySQL*. It turns out these two seemingly "spurious" counterexamples are due to a previously unknown bug in MySQL — they should produce different outputs. When dropping integrity constraints, VeriEQL-noIC found 11,390 counterexamples — but only 2,754 are genuine, which is even lower than VeriEQL's. This suggests that the false alarm rate will be significantly higher without considering integrity constraints. For the other two workloads, we observe a very similar trend.

Finding bugs in MySQL. As mentioned earlier, for two `LeetCode` benchmarks, VeriEQL generates "spurious" counterexamples — that are actually genuine — due to a bug in MySQL's latest release version 8.0.32. The MySQL verification team had confirmed and classified this bug with *serious* severity. Details of the bug are provided in Appendix D under supplementary materials.

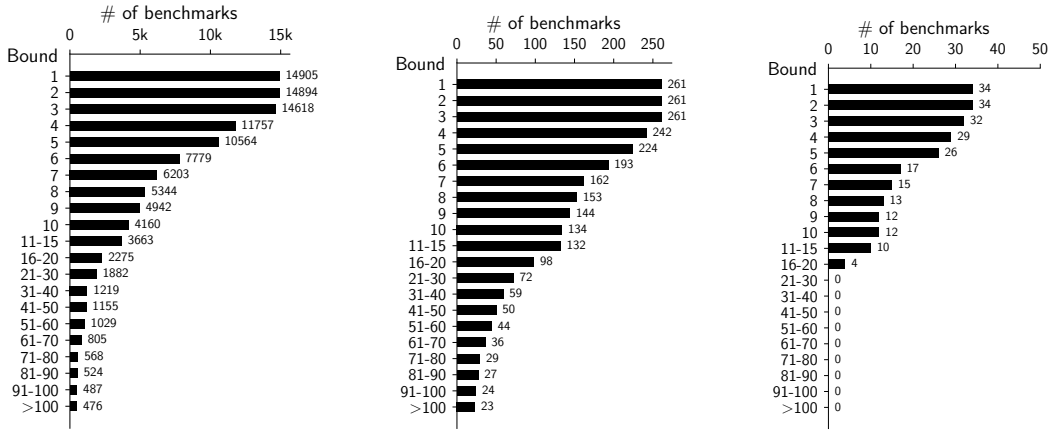
Detecting missing test cases for LeetCode. Recall that some of the user-submitted queries from our `LeetCode` workload may be wrong (i.e., not equivalent to the ground-truth queries), which suggests that new test cases are needed. In particular, as of the submission date (October 20, 2023), we have manually inspected 17 LeetCode problems for which VeriEQL reported non-equivalent benchmarks, and filed issues to the LeetCode team. Notably, our issue reports also have meaningful counterexamples generated by VeriEQL. To date, 13 of them have been confirmed and fixed, while the remaining ones have all been acknowledged. A common pattern is that existing tests miss the `NULL` case — this again highlights the importance of modeling the three-valued logic in SQL.

Identifying buggy Calcite rewrite rules. VeriEQL found 4 "not checked" (i.e., non-equivalent) `Calcite` benchmarks with valid counterexamples. The Calcite team has confirmed that two of them are due to bugs in two rewrite rules: one is a duplicate of an existing bug report, while the other is a new bug. In particular, for the new bug, the rule would rewrite a query Q_1 to a non-equivalent Q_2 when Q_1 has `SUM` applied to an empty table. The third is due to a bug in an internal translation step in Calcite, which has also been confirmed by the Calcite team. The fourth one is still being worked on by the team currently. This result again demonstrates that VeriEQL is able to uncover very tricky bugs from well-maintained open-source projects.

6.4 RQ3: Distribution of Bounds on Checked Benchmarks

As an indicator of VeriEQL's scalability, we study how large input bounds VeriEQL can reach within the 10-minute time limit when it reports a benchmark is checked.

Setup. We collected all 14,905 (62.1% of 23,994) `LeetCode` benchmarks that VeriEQL reports checked in **RQ1**. Similarly, we also collected all 261 (65.7% of 397) checked benchmarks from `Calcite` and all 34 (53.1% of 64) checked benchmarks from `Literature`. For each benchmark, we run VeriEQL by gradually increasing the bound from 1 until it gets timeout and keep records of the bound.



(a) *LeetCode* (b) *Calcite* (c) *Literature*

Fig. 15. **RQ3**: distribution of bounds on checked benchmarks.

RQ3 take-away: VeriEQL can verify over 70% of the checked benchmarks within 10 minutes given an input bound 5. For 3% of the checked benchmarks, VeriEQL can verify the query equivalence for an input bound >100.

Results. The evaluation results are presented in Figure 15, where the bar shows the number of benchmarks VeriEQL can verify for a given input bound. For example, as shown in Figure 15a, VeriEQL can verify equivalence for bound 1 on all 14,905 `LeetCode` benchmarks. Among them, it can verify equivalence for bound 2 on 14,894 benchmarks. In a nutshell, on 71% of checked `LeetCode` benchmarks, VeriEQL can reach bound 5, 28% for bound 10, 8% for bound 50, and 3% for bound 100. It shows similar patterns on `Calcite`. For `Literature`, VeriEQL only reaches bound 20 within the time limit, hypothetically because many benchmarks combine several complex SQL operators (e.g., Distinct and GroupBy) that are challenging for symbolic reasoning.

7 RELATED WORK

Equivalence checking for SQL queries. SQL equivalence checking is an important problem in both programming languages and database communities. There is a line of work on automated verification of query equivalence, including full verification [Aho et al. 1979; Chandra and Merlin 1977; Chu et al. 2018, 2017c; Green 2011; Zhou et al. 2019, 2022] and bounded verification [Chu et al. 2017a,b; Veanes et al. 2010; Wang et al. 2018a]. For example, SPES [Zhou et al. 2022] takes a symbolic approach for proving the existence of a bijective identity map between tuples returned by two queries. HoTTSQL [Chu et al. 2017c] and UDP [Chu et al. 2018] encode SQL queries in an algebraic structure to syntactically canonicalize two queries and then verify the equivalence of corresponding algebraic expressions using interactive theorem provers. Unlike prior work that aims for full equivalence verification, VeriEQL targets to solve the bounded equivalence verification problem modulo integrity constraints and can generate counterexamples for disproving equivalence. Compared to prior work for bounded verification such as Cosette [Chu et al. 2017b; Wang et al. 2018a], VeriEQL supports a larger fragment of SQL queries and a richer set of integrity constraints and significantly outperforms all baselines as demonstrated by our comprehensive evaluation. One limitation of VeriEQL is that it does not support correlated subqueries in its current status, which can be overcome by incorporating unnesting techniques proposed by Seshadri et al. [1996].

Testing SQL queries. Another line of related work is about testing the correctness of SQL queries. For instance, XData [Chandra et al. 2015] uses mutation-based testing to detect common mistake patterns in SQL queries. Shah et al. [2011] also uses mutation testing with hard-coded rules to kill as many query mutants as possible. In addition, EvoSQL [Castelein et al. 2018] uses an evolutionary search algorithm to generate test data in an offline manner, guided by predicate coverage Tuya et al. [2010]. RATest [Miao et al. 2019] uses a provenance-based algorithm to find a minimal counterexample database that can explain incorrect queries. DataFiller [Coelho 2013; Guagliardo and Libkin 2017] uses random database generation to produce test databases based on the schema and integrity constraints. Different from these testing approaches, VeriEQL provides formal guarantees on query equivalence (in a bounded fashion).

Formal methods for databases. Formal methods have been used to facilitate the development and maintenance of database systems and applications, such as optimizing database applications [Cheung et al. 2013; Delaware et al. 2015] and data storage [Feser et al. 2020], model checking database applications [Deutsch et al. 2005, 2007; Gligoric and Majumdar 2013], providing foundations to queries [Cheney and Ricciotti 2021; Ricciotti and Cheney 2019, 2022], verifying correctness of schema refactoring [Wang et al. 2018b], and synthesizing programs for schema evolution [Wang et al. 2019, 2020]. Among various work, Mediator [Wang et al. 2018b] is the most related to our work. However, Mediator studies the full equivalence verification problem of two database applications over different schemas, whereas VeriEQL performs bounded equivalence verification of two SQL queries under the same schema, but it generates counterexamples for disproving query equivalence.

Semantics of SQL queries. Different formal semantics of SQL queries have been proposed in the literature, such as set semantics, bag semantics, list semantics, and their combinations [Benzaken and Contejean 2019; Chu et al. 2018, 2017c; Negri et al. 1991; Ricciotti and Cheney 2019; Wang et al. 2018b]. We base the SQL semantics of VeriEQL on list operations and quotient by bag equivalence at the end mainly because lists, by their ordered nature, facilitate the definition of `ORDER BY` in the semantics. We have not explored the formal difference between our semantics and state-of-the-art semantics proposals. In fact, we believe that proving the equivalence between our semantics and existing semantics or characterizing the differences between semantics with demonstrating examples is an interesting direction for future work.

Bounded verification. Bounded verification has been used to find various kinds of bugs in software systems while providing formal guarantees on the result. Prior work such as CBMC [Clarke et al. 2004], JBMC [Cordeiro et al. 2018], and Corral [Lal et al. 2012] has been quite successful to this end for imperative languages like C, C++, or Java. However, VeriEQL performs bounded verification to check equivalence of declarative SQL queries. Unlike prior work that unrolls loops, VeriEQL bounds the size of relations in the database and can guarantee the equivalence of two queries for all databases where relations are up to a given finite size.

Equivalence verification. Equivalence verification has been studied extensively in the literature. Researchers have developed many approaches for checking equivalence of various applications, such as translation validation for compilers [Necula 2000; Pnueli et al. 1998; Stepp et al. 2011], product program [Barthe et al. 2011; Zaks and Pnueli 2008], relational Hoare logic [Benton 2004], Cartesian Hoare Logic [Sousa and Dillig 2016], and so on. VeriEQL is along the line of equivalence verification, but it is tailored towards a different application, SQL queries, and presents an SMT-based approach for verifying bounded equivalence of queries modulo integrity constraints.

8 CONCLUSION

In this paper, we presented VeriEQL, a simple yet highly practical approach that can both prove and disprove the bounded equivalence of *complex SQL queries with integrity constraints*. For a total of

24,455 benchmarks, VeriEQL can prove and disprove over 70% of them, significantly outperforming all state-of-the-art SQL equivalence checking techniques.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers of OOPSLA for their detailed and helpful comments on an earlier version of this paper. This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant and the National Science Foundation (NSF) Grant No. CCF-2210832.

DATA-AVAILABILITY STATEMENT

The software that implements the techniques described in Section 4 and supports the evaluation results reported in Section 6 is available on Zenodo [He et al. 2024a].

REFERENCES

Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. 1979. Equivalences among Relational Expressions. *SIAM J. Comput.* 8, 2 (1979), 218–246. https://doi.org/10.1137/0208017

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *Proceedings of the International Symposium on Formal Methods (FM) (Lecture Notes in Computer Science, Vol. 6664)*. 200–214. https://doi.org/10.1007/978-3-642-21437-0_17

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 14–25. https://doi.org/10.1145/964001.964003

Véronique Benzaken and Evelyne Contejean. 2019. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 249–261. https://doi.org/10.1145/3293880.3294107

Calcite. 2023. Calcite Optimization Rules Test Suite. https://github.com/apache/calcite/blob/main/core/src/test/java/org/apache/calcite/test/RelOptRulesTest.java.

Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-based test data generation for SQL queries. In *Proceedings of the 40th international conference on software engineering*. 1220–1230. https://doi.org/10.1145/3180155.3180202

Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. ACM, 77–90. https://doi.org/10.1145/800105.803397

Bikash Chandra, Ananyo Banerjee, Udbhas Hazra, Mathew Joseph, and S Sudarshan. 2019. Automated grading of sql queries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1630–1633. https://doi.org/10.1109/ICDE.2019.00159

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. https://doi.org/10.1007/s00778-015-0395-0

James Cheney and Wilmer Ricciotti. 2021. Comprehending nulls. In *Proceedings of the International Symposium on Database Programming Languages (DBPL)*. ACM, 3–6. https://doi.org/10.1145/3475726.3475730

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*. ACM, 3–14. https://doi.org/10.1145/2491956.2462180

Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. 2017a. Demonstration of the cosette automated sql prover. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1591–1594. https://doi.org/10.1145/3035918.3058728

Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 11, 11 (2018), 1482–1495. https://doi.org/10.14778/3236187.3236200

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

Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017c. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (2017), 510–524. https://doi.org/10.1145/3062341.3062348

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European*

Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10. Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Fabien Coelho. 2013. DataFiller – generate random data from database schema. https://www.cri.ensmp.fr/people/coelho/datafiller.html.

Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A bounded model checking tool for verifying Java bytecode. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Springer, 183–190. https://doi.org/10.1007/978-3-319-96145-3_10

Cosette. 2018. Cosette website. https://cosette.cs.washington.edu/.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 4963)*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 689–700. https://doi.org/10.1145/2676726.2677006

Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. 2005. A Verifier for Interactive, Data-Driven Web Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 539–550. https://doi.org/10.1145/1066157.1066219

Alin Deutsch, Liying Sui, and Victor Vianu. 2007. Specification and verification of data-driven Web applications. *J. Comput. Syst. Sci.* 73, 3 (2007), 442–474. https://doi.org/10.1016/j.jcss.2006.10.006

John K. Feser, Sam Madden, Nan Tang, and Armando Solar-Lezama. 2020. Deductive optimization of relational data storage. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 170:1–170:30. https://doi.org/10.1145/3428238

Milos Gligoric and Rupak Majumdar. 2013. Model checking database applications. In *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*. Springer, 549–564. https://doi.org/10.1007/978-3-642-36742-7_40

Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

Todd J. Green. 2011. Containment of Conjunctive Queries on Annotated Relations. *Theory Comput. Syst.* 49, 2 (2011), 429–459. https://doi.org/10.1007/s00224-011-9327-6

Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proc. VLDB Endow.* 11, 1 (sep 2017), 27–39. https://doi.org/10.14778/3151113.3151116

Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024a. Artifact Evaluation VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. https://doi.org/10.5281/zenodo.10795614

Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024b. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. arXiv:2403.03193

Akash Lal, Shaz Qadeer, and Shuvendu K Lahiri. 2012. A solver for reachability modulo theories. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*. Springer, 427–443. https://doi.org/10.1007/978-3-642-31424-7_32

LeetCode. 2023. LeetCode website. https://leetcode.com/.

Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining wrong queries using small examples. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 503–520. https://doi.org/10.1145/3299869.3319866

George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 83–94. https://doi.org/10.1145/349299.349314

Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. 1991. Formal semantics of SQL queries. *ACM Transactions on Database Systems (TODS)* 16, 3 (1991), 513–534. https://doi.org/10.1145/111197.111212

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 1384)*. 151–166. https://doi.org/10.1007/BFb0054170

Wilmer Ricciotti and James Cheney. 2019. Mixing set and bag semantics. In *Proceedings of the ACM SIGPLAN International Symposium on Database Programming Languages (DBPL)*. ACM, 70–73. https://doi.org/10.1145/3315507.3330202

Wilmer Ricciotti and James Cheney. 2022. A Formalization of SQL with Nulls. *J. Autom. Reason.* 66, 4 (2022), 989–1030. https://doi.org/10.1007/s10817-022-09632-4

Praveen Seshadri, Hamid Pirahesh, and TY Cliff Leung. 1996. Complex query decorrelation. In *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE, 450–458. https://doi.org/10.1109/ICDE.1996.492194

Shetal Shah, S Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. 2011. Generating test data for killing SQL mutants: A constraint-based approach. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1175–1186. https://doi.org/10.1109/ICDE.2011.5767876

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 57–69. https://doi.org/10.1145/2908080.2908092

Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Proceedings of the International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 6806)*. 737–742. https://doi.org/10.1007/978-3-642-22110-1_59

Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 530–541. https://doi.org/10.1145/2594291.2594340

Javier Tuya, María José Suárez-Cabal, and Claudio De La Riva. 2010. Full predicate coverage for testing SQL database queries. *Software Testing, Verification and Reliability* 20, 3 (2010), 237–288. https://doi.org/10.1002/stvr.424

Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL query explorer. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 425–446. https://doi.org/10.1007/978-3-642-17511-4_24

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2018a. Speeding up symbolic reasoning for relational queries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25. https://doi.org/10.1145/3276527

Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018b. Verifying Equivalence of Database-Driven Applications. *Proc. ACM Program. Lang.* 2, POPL, Article 56 (2018), 29 pages. https://doi.org/10.1145/3158144

Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 286–300. https://doi.org/10.1145/3314221.3314588

Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (2020), 1006–1019. https://doi.org/10.14778/3384345.3384350

Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *Proceedings of the International Symposium on Formal Methods (FM) (Lecture Notes in Computer Science, Vol. 5014)*. 35–51. https://doi.org/10.1007/978-3-540-68237-0_5

Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288. https://doi.org/10.14778/3342263.3342267

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 *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2735–2748. https://doi.org/10.1109/ICDE53745.2022.00250

Received 2023-10-21; accepted 2024-02-24