Proving Query Equivalence Using Linear Integer Arithmetic

HAORAN DING, Shanghai Jiao Tong University, China
ZHAOGUO WANG*, Shanghai Jiao Tong University, China
YICUN YANG, Shanghai Jiao Tong University, China
DEXIN ZHANG, Shanghai Jiao Tong University, China
ZHENGLIN XU, Shanghai Jiao Tong University, China
HAIBO CHEN, Shanghai Jiao Tong University, China
RUZICA PISKAC, Yale University, USA
JINYANG LI, New York University, USA

Proving the equivalence between SQL queries is a fundamental problem in database research. Existing solvers model queries using algebraic representations and convert such representations into first-order logic formulas so that query equivalence can be verified by solving a satisfiability problem. The main challenge lies in "unbounded summations", which appear commonly in a query's algebraic representation in order to model common SQL features, such as Projection and aggregate functions. Unfortunately, existing solvers handle unbounded summations in an ad-hoc manner based on heuristics or syntax comparison, which severely limits the set of queries that can be supported.

This paper develops a new SQL equivalence prover called SQLSolver, which can handle unbounded summations in a principled way. Our key insight is to use the theory of LIA*, which extends linear integer arithmetic formulas with unbounded sums and provides algorithms to translate a LIA* formula to a LIA formula that can be decided using existing SMT solvers. We augment the basic LIA* theory to handle several complex scenarios (such as nested unbounded summations) that arise from modeling real-world queries. We evaluate SQLSolver with 359 equivalent query pairs derived from the SQL rewrite rules in Calcite and Spark SQL. SQLSolver successfully proves 346 pairs of them, which significantly outperforms existing provers.

CCS Concepts: • **Information systems → Query optimization**; • **Theory of computation → Automated reasoning**; **Program verification**; **Logic and databases**.

Additional Key Words and Phrases: SQL query equivalence; SQL solver; linear integer arithmetic; LIA; linear integer arithmetic with stars; LIA*

ACM Reference Format:
Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 227 (December 2023), 26 pages. https://doi.org/10.1145/3626768

*Corresponding author (zhaoguowang@sjtu.edu.cn)

Authors' addresses: Haoran Ding, nhaorand@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China, 200240; Zhaoguo Wang, zhaoguowang@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China, 200240; Yicun Yang, yangyicun@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China, 200240; Dexin Zhang, zhangdexin@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China, 200240; Zhenglin Xu, kevinsouth@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China, 200240; Haibo Chen, haibochen@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China, 200240; Ruzica Piskac, ruzica.piskac@yale.edu, Yale University, New Haven, Connecticut, USA, 06511; Jinyang Li, jinyang@cs.nyu.edu, New York University, New York, New York, USA, 10003.

2836-6573/2023/12-ART227 $15.00
https://doi.org/10.1145/3626768

```
Q1: select a, count(b) from R group by a
Q2: select S.a, sum(S.cnt) from
    (select a, count(b) as cnt from R group by a) as S
    group by S.a
```

Fig. 1. A pair of equivalent queries that cannot be proven by existing solvers.

1 INTRODUCTION

In recent years, we have witnessed the increasing importance of developing automatic tools to prove the equivalence of two SQL queries. Such a SQL equivalence solver can be used in many important scenarios, e.g., proving or refuting certain existing query rewrite rules [2, 9, 11, 18, 41], discovering novel rewrite rules [43], identifying any overlap in computation in a data processing pipeline [46].

The objective of a SQL solver, which is to determine whether two queries are semantically equivalent, is unfortunately undecidable for general SQL queries [1]. Despite this theoretical limitation, recent works have developed several solvers [9, 11, 46, 47] that aim for practical use by handling an increasing set of SQL queries. These solvers all model a given SQL query as an algebraic representation. Syntax-based checkers, such as UDP [9] and SPES [47], normalize the algebraic representations of a pair of queries and check for equivalence by determining whether the normalized representations are isomorphic. Such syntax-based checking has trouble proving queries that differ significantly in their syntax structures. Semantics-based checkers, such as WeTune [43], are introduced recently to address this limitation. These solvers translate a query's algebraic representation into a first-order logic formula and thus reduce the problem of proving equivalence to deciding the satisfiability of a first-order logic formula.

Unfortunately, existing semantics-based solvers, such as WeTune, are still very limited in the kind of queries that can be handled. WeTune uses U-expressions to model queries under bag semantics. Specifically, the U-expression of a query is an algebraic expression $f(t)$, which returns the multiplicity of the tuple t in the query result. Two queries are equivalent if their U-expressions always return the same result. In order to model certain essential SQL features, such as Projection and aggregate functions, a query's algebraic expression ends up containing summations ($\sum$) over the infinite domain of all possible tuples. We refer to these terms as *unbounded summations*. For example, the projection query "`select X from R`" has this corresponding U-expression: $f(t) = \sum_{t'}([t = X(t')] \times R(t'))$, where the unbounded summation ($\sum'_t$) adds up the multiplicity of each tuple in relation R (represented by $R(t')$) if its X attribute value is t (represented by $[t = X(t')]$).

However, WeTune lacks a principled way to translate U-expressions with unbounded summations to first-order logic formulas. Using heuristics, WeTune can only handle a few simple cases involving unbounded summations. As a result, WeTune can only prove about a third of the equivalent queries (78 out of 232) derived from Calcite test suites [5] and about a quarter of those (31 out of 127) derived from Spark SQL. For example, the pair of queries in Figure 1 are equivalent. However, they cannot be proven by WeTune because the corresponding algebraic expressions involve unbounded summations in complex forms. Neither can the pair be proven by syntax-based provers like UDP and SPES. This is because the numbers of aggregate functions in the two queries are different, leading to different syntax structures that cannot be aligned even after normalization.

This paper presents a new semantics-based prover called SQLSolver that can handle unbounded summations in U-expressions in a principled way. We take inspiration from existing literature on decision procedures and use the theory of linear integer arithmetic with stars (LIA*), which extends

linear integer arithmetic formulas with unbounded summations [36, 37]. Existing works on LIA* have shown how to convert a LIA* formula to a LIA formula [30]. Thus, the basic approach of SQLSolver is to first translate U-expressions into a LIA* formula and then further convert the LIA* formula into a LIA formula before solving them using a SMT solver. While this basic approach is promising, applying LIA* in our setting encounters additional challenges not considered by the existing theory of LIA*. In particular, the U-expressions of real-world SQL queries can often contain nested unbounded summations, parameterized summations, or summations involving non-linear operations. We augment the original LIA* theory to support each of these cases. As a result of our principled way of handling unbounded summations, SQLSolver is able to support many more complex SQL queries and common features, such as the example in Figure 1.

As U-expressions model SQL queries under bag semantics, it cannot handle ORDER BY clauses, which sort tuples in the query result. To solve this issue, we develop a "divide and conquer" strategy, which enables SQLSolver to prove the equivalence between two queries with ORDER BY clauses by proving the equivalence between queries without ORDER BY clauses.

We evaluate SQLSolver using equivalent query pairs derived from Calcite [5], Spark SQL [17], TPC-C, and TPC-H. They contain 232, 127, 19, and 22 pairs of equivalent queries, respectively. SQLSolver can prove all 232 query pairs derived from Calcite test suites, 114 (out of 127) pairs of queries derived from Spark SQL, all 19 query pairs derived from TPC-C, and 19 (out of 22) query pairs derived from TPC-H. By contrast, existing solvers can only prove 121 pairs in Calcite test suites, 71 pairs derived from Spark SQL, 15 pairs derived from TPC-C, and 0 pairs derived from TPC-H.

In summary, our work makes the following contributions:

- We propose a principled way to translate U-expressions with unbounded summations into first-order logic formulas by applying the theory of LIA*. Unbounded summations are essential for modeling common SQL queries involving Projection and aggregate functions.
- We augment the original LIA* theory to handle nested unbounded summations, parameterized summations, or summations involving non-linear operations, all of which occur in common SQL queries.
- We additionally handle ORDER BY clauses, which cannot be modeled by U-expressions under bag semantics. Our method based on "divide-and-conquer" can be used in conjunction with our main LIA*-based solver algorithm.
- Based on our design, we implement a new SQL equivalence solver, which can prove many more queries than existing solvers. On the Calcite test suite, existing solvers can prove approximately half of the 232 queries, while our solver SQLSolver can prove all of them. When using SQLSolver to discover SQL rewrite rules, we find 42 new rewrite rules beyond the 35 rules found by using the existing solver in WeTune.

2 BACKGROUND AND MOTIVATION

In this section, we discuss the evolution of existing SQL query equivalence solvers and describe the challenges that state-of-the-art solvers face today.

2.1 Syntax vs. Semantics-based Checking

To prove equivalence, we must first model the semantics of SQL queries. Existing solvers model queries under bag semantics (i.e., the query result may contain duplicate tuples [12]) and transform queries into algebraic representations. Next, we need to check the equivalence of a pair of algebraic representations. Existing solvers differ in how they perform such checks. There are two main approaches: *syntax-based* and *semantics-based* checking.

Table 1. The comparison of automated provers. Methodology indicates their method of modeling SQL queries and checking query equivalence. Capability shows the number of equivalent query pairs that can be proved by each prover, which are derived from Calcite and Spark SQL.

		UDP	SPES	WeTune	SQLSolver
Methodology	Query Modeling	U-expression	Tree-based Algebraic Representation (AR)	U-expression	Extended U-expression
	Equivalence Checking	Syntax Structure	Syntax Structure + SMT Solver	Trans. Rule + SMT Solver	LIA* Solver
Capability	Calcite (232 pairs)	33	99	78	232
	Spark SQL (127 pairs)	20	56	31	114

With **syntax-based approaches**, solvers attempt to normalize the algebraic representations using pre-defined rules and then compare their syntax structures [8, 9, 11, 47]. Two representative syntax-based solvers are UDP [9] and SPES [47]. UDP transforms each SQL query into an algebraic representation called a U-expression, which is a function that takes any tuple t as its input and returns the corresponding multiplicity of t in the query result. UDP applies a set of manually crafted rules to normalize and simplify U-expressions. Finally, UDP determines the equivalence of two SQL queries by checking whether the normal forms of their U-expressions are isomorphic in terms of their syntax structures. SPES works in a way similar to UDP but differs in some aspects. In particular, SPES converts SQL queries into a tree-structured algebraic representation instead of U-expressions. Similar to UDP, before checking the isomorphism between two queries in terms of syntax structures, SPES applies manually crafted rules to normalize and simplify tree-based representations into normal forms. Unlike UDP, SPES leverages SMT solvers to check the equivalence of predicates in different queries so that it is capable of verifying concrete predicates, such as the equivalence between $t > 10$ and $t + 10 > 20$. Due to its integration with SMT solvers, SPES can support certain features that UDP cannot, such as arithmetic operators.

Limitation of syntax-based equivalence checking. Syntax-based solvers determine the equivalence of two SQL queries by verifying the isomorphism of their normalized algebraic representations. Thus, it is difficult for them to prove the equivalence of two SQL queries that differ significantly in their syntax structures despite introducing many normalization rules. For example, it is difficult for SPES to prove the equivalence of SQL queries with differing numbers of input relations, UNION ALL operators, or aggregate functions. Additionally, although UDP and SPES can handle UNION and EXISTS operators, they cannot reason the following equivalent queries. Each of the queries returns all unique tuples in relation R.

```
Q1: (select * from R) union (select * from R)
Q2: select distinct * from R
    where exists (select * from R)
```

Neither UDP nor SPES can verify the isomorphism of the above queries' algebraic representations in normal forms because the first query lacks EXISTS operators and the second query lacks UNION operators. As a result, for 232 equivalent query pairs collected from the Calcite test suite [5], UDP can only prove 33 pairs and SPES can prove 99 pairs. Meanwhile, for 127 equivalent query pairs derived from the Spark SQL rewrite rules [17], UDP can prove 20 pairs and SPES can prove 56 pairs. Additionally, there are 117 pairs of queries among both test suites that neither UDP nor SPES can prove due to the limitation mentioned above.

With **semantics-based approaches**, solvers determine the equivalence of two algebraic expressions by transforming the equivalence problem into the satisfiability problem of a first-order logic formula [43, 46]. A recent semantics-based solver is WeTune [43]. WeTune models SQL queries using U-expressions, like UDP. However, instead of comparing normalized U-expressions based on syntax structures, WeTune transforms queries' U-expressions into a first-order logic formula. Thus, proving the equivalence of SQL queries is reduced to reasoning about a first-order logic formula, which can be processed by SMT solvers. As a result, WeTune is capable of proving equivalent queries even when they vary significantly in terms of syntax structures.

Limitation of semantics-based equivalence checking. Existing semantics-based solvers such as WeTune [43] are heavily reliant on their abilities to translate queries' algebraic representations into a first-order logic formula. In particular, WeTune can only translate simple U-expressions with limited expressiveness. As a result, queries that entail U-expressions with complicated syntax structures or specific aggregate functions fall beyond WeTune's capabilities. We will elaborate on the challenges in converting U-expressions to a first-order logic formula in the next subsection (§ 2.2). For the 232 pairs of equivalent queries in the Calcite test suites, WeTune successfully proves 78 of them. For the 127 equivalent queries derived from Spark SQL rewrite rules [17], WeTune can prove 31 pairs. Despite the limitation, semantics-based solvers are quite promising as they can prove queries beyond what can be proven by syntax-based solvers. For example, 20 pairs of queries derived from Calcite and 14 pairs derived from Spark SQL that can be proven by WeTune cannot be proven by UDP or SPES.

Table 1 compares the methodologies and capabilities of existing solvers. Our goal is to address the limitations of semantics-based equivalence checking to expand the capabilities of these solvers.

2.2 Challenge of Semantics-based Checking

Let us now examine the challenges a semantics-based solver like WeTune faces. WeTune models queries using U-expressions and then translates queries' U-expressions into a first-order logic formula. We first give a brief overview of U-expression-based query modeling. Then, we will investigate the key difficulties in translating U-expressions to a first-order logic formula.

U-expression. U-expression is a common way to model queries with algebraic expressions [9, 43]. Given a query Q, its U-expression is a function $f(t)$, which returns the multiplicity of the tuple t in the query result of Q. SQLSolver translates each SQL query into a U-expression by composing a number of pre-defined terms summarized in Table 2. Each term returns a non-negative integer value and is connected by "×" or "+" in the U-expression. Specifically, for each relation name R, there is a pre-defined function $[\![R]\!](t)$ that returns the multiplicity of tuple t in the relation R. For each SQL predicate b, the U-expression $[b]$ is either 1 (b is true) or 0 (b is false). Table 2 also defines the operators of squash ($\|\cdot\|$) and negation ($not(\cdot)$) to model the semantics of DISTINCT and NOT accordingly. For brevity, we omit $[\![\ ]\!]$ in U-expressions if there is no ambiguity. For example, given a query of "`select * from R where R.x > 0`", its U-expression is $f(t) := R(t) \times [x(t) > 0]$, where $x(t)$ returns the attribute x of tuple t.

To model the semantics of Projection, a U-expression introduces summations $\sum$ over the infinite domain of all possible tuples. We refer to such summation as an *unbounded summation*. Unbounded summation is commonly used for calculating the total number of tuples that meet certain conditions. Take an example query with Projection, "`select R.x from R where R.x > 0`", its U-expression is $f(t) := \sum_{t_1}(R(t_1) \times [x(t_1) > 0] \times [t = x(t_1)])$. The unbounded summation enumerates all possible tuples t_1 over unspecified relation R whose attribute x is more than 0 and equal to t. Unbounded summation is crucial not only for modeling Projection but also for other common SQL features, such as aggregate functions. We will discuss it deeply in Section 4.

Table 2. The definition of terms / operators in U-expressions.

SQL Feature	Term / Operator	Concept	Example Query	Example U-expression $f(t)$
Relation	$[\![R]\!](t)$	$[\![R]\!](t)$ is a function that returns the multiplicity of the tuple t in the relation R.	`select * from R`	$[\![R]\!](t)$
DISTINCT	$\|\|e\|\|$	$\|\|e\|\|$ returns 0 if the U-expression e is 0. Otherwise, it returns 1.	`select distinct * from R`	$\|\|[\![R]\!](t)\|\|$
Predicate	[b]	[b] returns 1 if the predicate b is true. Otherwise, [b] returns 0.	`select * from R where R.x < 500`	$[\![R]\!](t) \times [[\![x]\!](t) < 500]$
NOT	$not(e)$	$not(e)$ returns 1 if the U-expression e is 0. Otherwise, $not(e)$ returns 0.	`select * from R where not R.x = 0`	$[\![R]\!](t) \times not([[\![x]\!](t) = 0])$
Projection	$\sum_t f(t)$	$\sum_t f(t)$ returns the sum of $f(t_i)$ for each possible tuple t_i, where f is a function in the type of $tuple \rightarrow \mathbb{N}$. Since there may be infinite t_i, $\sum_t f(t)$ is called an unbounded summation.	`select R.x from R where R.x > 0`	$\sum_{t_1}([t = [\![x]\!](t_1)] \times [\![R]\!](t_1) \times [[\![x]\!](t_1) > 0])$
OUTER JOIN / UNION ALL / UNION	$e_1 + e_2$	$e_1 + e_2$ returns the sum of the two U-expressions e_1 and e_2.	`select * from R union all select * from S`	$[\![R]\!](t) + [\![S]\!](t)$

Trouble with unbounded summations. With U-expressions, the equivalence between Q_1 and Q_2 can be reduced to the unsatisfiability of the formula $\exists t. f_1(t) \neq f_2(t)$, where $f_1(t)$ and $f_2(t)$ are the U-expressions of Q_1 and Q_2. However, the primary obstacle to reasoning the formula $\exists t. f_1(t) \neq f_2(t)$ lies in the presence of unbounded summations, which renders the problem undecidable [9].

Through heuristics, existing semantics-based solvers like WeTune can only handle certain simple forms of U-expressions involving unbounded summations and translate them into a first-order logic formula. For instance, WeTune can handle the simple case when the U-expressions of two queries both contain only a single unbounded summation. For this case, WeTune would determine the equivalence of the U-expressions by determining whether the inner expressions of the two summations are equivalent for all tuples. Beyond U-expressions with only a single unbounded summation, there are only a few other limited forms of U-expressions that can be handled by WeTune. Syntax-based solvers that use U-expressions for query modeling face this difficulty and require the U-expressions of equivalent queries to contain unbounded summations with syntactically identical structures [9].

```
Q1: select x from R where y<=1000
    union all
    select x from R where y>1000 and z<500
Q2: select x from R where y<=1000 or z<500
```

Their corresponding U-expressions are

$$
\begin{aligned}
& f_1(t) \coloneqq \sum_{t_1} E_1 + \sum_{t_1} E_2 \text{ and } f_2(t) \coloneqq \sum_{t_1} E_3 \\
& \quad \textbf{where} \\
& E_1 \coloneqq [t = x(t_1)] \times R(t_1) \times [y(t_1) <= 1000] \\
& E_2 \coloneqq [t = x(t_1)] \times R(t_1) \times [y(t_1) > 1000] \times [z(t_1) < 500] \\
& E_3 \coloneqq [t = x(t_1)] \times R(t_1) \times [y(t_1) <= 1000 \vee z(t_1) < 500]
\end{aligned} \tag{1}
$$

Fig. 2. The example of a pair of equivalent queries and their U-expressions.

The lack of a principled way to handle unbounded summations is the main reason why many practical SQL queries cannot be proven equivalent. For example, for the Calcite test suites, there are 180 query pairs whose U-expressions have unbounded summations. UDP can reason about 33 of them, while WeTune can reason about 71. Spark SQL's rewrite suite contains 69 query pairs whose U-expressions contain unbounded summations. UDP and WeTune can only reason about 6 and 16 pairs of them, respectively. The queries that cannot be proven contain complex unbounded summations or features. For example, the equivalent queries in Figure 2 cannot be handled by UDP or WeTune. For UDP, the U-expressions of two queries have different numbers of unbounded summations. For WeTune, UNION ALL in the first query poses the addition of two unbounded summations. The rules in WeTune cannot address this syntax structure.

3 OVERVIEW

We have developed a new prover called SQLSolver to overcome the limitations of existing solvers by handling unbounded summations in a principled manner. Compared to existing solvers, SQLSolver can handle more complex SQL queries and ubiquitous features such as aggregate functions. Like UDP and WeTune, SQLSolver models each SQL feature using U-expressions [9] under bag semantics. Instead of relying on heuristics to handle unbounded summations in the resulting U-expressions, SQLSolver translates U-expressions into LIA* logic [30, 38, 39]. It has been shown that every LIA* formula can be converted into an equisatisfiable linear integer arithmetic (LIA) formula [38], which can be solved by SMT solvers. Once U-expressions are converted into a LIA* formula, SQLSolver translates the LIA* formula further into a LIA formula and solves it via SMT solvers. For Calcite test suites [5], SQLSolver can prove all 232 equivalent query pairs, while the combined efforts of UDP, SPES, and WeTune could only prove 121. For Spark SQL test suites with 127 equivalent SQL pairs [17], SQLSolver could successfully prove 114, while the other three solvers could prove 71 in total.

Initial inspiration: LIA*. The theory of linear integer arithmetic with stars (LIA*) provides a methodology to reason a formula having unbounded summations [30, 39]. In general, the LIA* formula is an extension of the linear integer arithmetic formula with unbounded summations, and each LIA* formula has an equisatisfiable LIA formula [39]. Thus, reasoning a LIA* formula can be reduced to reasoning a LIA formula that does not contain unbounded summations.

Before formally defining a LIA* formula, we need to introduce the additive closure operator $*$. It is an operator defined over a set of integer vectors S as follows: [1]

$$S^* = \left\{ \vec{v} \;\middle|\; \vec{v} = \sum_{i=1}^{n} \lambda_i \vec{v}_i \;\wedge\; \forall i.(\vec{v}_i \in S \;\wedge\; \lambda_i \geq 0) \right\}$$

In other words, S^* is the set of all possible summations of the elements in S.

A LIA* formula has the following form, where $\vec{u}, \vec{v}$ and $\vec{x}$ are integer vectors. F_1 and F_2 are linear integer arithmetic (LIA) formulas.

$$\exists \vec{u}, \vec{v}.F_1(\vec{u}, \vec{v}) \wedge \vec{v} \in \{\vec{x} \mid F_2(\vec{x})\}^* \tag{2}$$

In the above formula, $\{\vec{x} \mid F_2(\vec{x})\}$ represents the set of integer vectors that satisfy F_2, and the additive closure operator $*$ generates its additive closure. Thus, each vector $\vec{v}$ is a sum of vectors in $\{\vec{x} \mid F_2(\vec{x})\}$. Namely, $\vec{v} = \sum_{i=1}^{n} \vec{x}_i$, where $\vec{x}_i$ satisfies F_2 and n is an arbitrary integer. The fact that n can be an arbitrary integer allows LIA* to model a summation with an infinite domain.

We use a simple example to illustrate how to leverage the LIA* theory to solve a formula with unbounded summations. The formal details can be found in Section 4.1. The example is reasoning the distributive law on unbounded summations as below, where y is an integer. f_1 and f_2 are functions. Each function's parameter and return value are both integers.

$$\left(\sum_y f_1(y) + \sum_y f_2(y) \right) = \sum_y (f_1(y) + f_2(y)) \tag{3}$$

The basic idea is to convert the above formula into a LIA* formula and then invoke an SMT solver to check the satisfiability of the derived LIA formula.

First, we introduce integer variables v_1, v_2 and v_3 to represent $\sum f_1(y)$, $\sum f_2(y)$ and $\sum(f_1(y) + f_2(y))$ in Equation (3) respectively. This way, checking the validity of Equation (3) is equivalent to checking if the following formula is unsatisfiable.

$$\exists v_1, v_2, v_3.(v_1 + v_2 \neq v_3) \wedge \left((v_1, v_2, v_3) = \sum_y (f_1(y), f_2(y), f_1(y) + f_2(y)) \right)$$

Next, we translate the derived formula into an equisatisfiable LIA* formula, where "_" denotes an omitted integer variable.

$$\exists v_1, v_2, v_3.(v_1 + v_2 \neq v_3) \wedge (v_1, v_2, v_3, \_) \in \{(x_1, x_2, x_3, y) \mid (x_1 = f_1(y)) \wedge (x_2 = f_2(y)) \wedge (x_3 = f_1(y) + f_2(y))\}^*.$$

Two formulas are equisatisfiable iff one formula is satisfiable whenever the other one is also satisfiable, and vice versa.

Finally, we apply the procedure described in [30] to generate an equisatisfiable LIA formula:

$$\exists v_1, v_2, v_3, \lambda_1, \lambda_2.(v_1 + v_2 \neq v_3) \wedge ((v_1, v_2, v_3) = \lambda_1(1, 0, 1) + \lambda_2(0, 1, 1))$$

The LIA formula is unsatisfiable, which can be solved by SMT solvers. This unsatisfiability implies the correctness of the distributive law (i.e., the originally given formula is valid).

Basic approach. The above example illustrates how SQLSolver uses LIA* to check the equivalence of two SQL queries. Its basic algorithm can be described through the following steps. Our running example will be the example given in Figure 2.

Step 1. Given two SQL queries, Q_1 and Q_2, we generate their corresponding U-expressions $f_1(t)$ and $f_2(t)$. Proving the equivalence between Q_1 and Q_2 becomes proving that $\exists t.f_1(t) \neq f_2(t)$ is unsatisfiable. In our running example, $f_1(t) = \Sigma E_1 + \Sigma E_2$, while $f_2(t) = \Sigma E_3$, as shown in Equation (1).

[1] The variables n, λ_i, and $\vec{v}_i$ are bounded by existential quantifiers, which are omitted.

Step 2. For each unbounded summation $\sum E_i$, we introduce a new integer variable v_i with the meaning that $v_i = \sum_t E_i$. This way, we can rewrite the formula from Step 1 into the conjunction of two formulas: one is a LIA formula, while the other is the definition of all the v_i variables in the vector form. The resulting formula for our example looks as follows:

$$\exists t, v_1, v_2, v_3 . v_1 + v_2 \neq v_3 \wedge (v_1, v_2, v_3) = \sum_{t_1} (E_1, E_2, E_3)$$

Step 3. We eliminate unbounded summations by translating the formula into an equisatisfiable LIA* formula. This translation further introduces new integer variables that denote the multiplicities of tuples. In our example, by introducing x_1, x_2 and x_3 to represent E_1, E_2, and E_3 accordingly, the formula becomes:

$$\begin{aligned} &\exists t, v_1, v_2, v_3 . v_1 + v_2 \neq v_3 \\ &\quad \wedge (v_1, v_2, v_3) \in \{(x_1, x_2, x_3) \mid x_1 = E_1 \wedge x_2 = E_2 \wedge x_3 = E_3\}^* \end{aligned} \tag{4}$$

Step 4. We replace U-expression terms with LIA terms to construct a standard LIA* formula. In our example, this step replaces $[b]$ with $ite(b, 1, 0)$. Besides, the step replaces each tuple variable t and each function $R(t)$ with an integer variable x_j. Finally, it generates the following LIA* formula, where "_" represents omitted variables:

$$\begin{aligned} &\exists v_1, v_2, v_3 . v_1 + v_2 \neq v_3 \wedge (v_1, v_2, v_3, \_) \in \{(x_1, x_2, x_3, \_) \mid \\ &\quad x_1 = ite(x_4 = x_5 \wedge x_7 <= 1000, x_6, 0) \wedge \\ &\quad x_2 = ite(x_4 = x_5 \wedge x_7 > 1000 \wedge x_8 < 500, x_6, 0) \wedge \\ &\quad x_3 = ite(x_4 = x_5 \wedge (x_7 <= 1000 \vee x_8 < 500), x_6, 0)\}^*. \\ &\quad \textbf{where } x_4 \coloneqq t,\ x_5 \coloneqq x(t_1),\ x_6 \coloneqq R(t_1), x_7 \coloneqq y(t_1),\ x_8 \coloneqq z(t_1) \end{aligned}$$

Step 5. By applying the algorithm described in [30], we can find an equisatisfiable LIA formula, which can be directly solved by SMT solvers (Section 4.1). For our example, the algorithm could infer that $x_1 + x_2 = x_3$ according to the conditions on these variables and reduce the LIA* formula to the following LIA formula:

$$\exists v_1, v_2, v_3, \lambda_1, \lambda_2 . v_1 + v_2 \neq v_3 \wedge (v_1, v_2, v_3) = \lambda_1(1, 0, 1) + \lambda_2(0, 1, 1)$$

$(v_1, v_2, v_3) = \lambda_1(1, 0, 1) + \lambda_2(0, 1, 1)$ can be simplified to $(v_1, v_2, v_3) = (\lambda_1, \lambda_2, \lambda_1 + \lambda_2)$, which requires that v_3 equals $v_1 + v_2$. However, this conclusion conflicts with the first condition that $v_1 + v_2 \neq v_3$. Thus, the whole formula is unsatisfiable, which indicates that Q_1 is equivalent to Q_2.

Challenge in using the original LIA* theory. The complicated semantics of SQL queries makes applying LIA* theory non-trivial. In the following three cases, U-expressions cannot be translated into LIA* formulas by applying only the basic approach. First, the formula contains a nested summation, which is usually used to model SQL features in subqueries. Second, the formula contains terms not supported by the LIA* theory, such as strings or real numbers. This case is typically caused by non-integer attributes or constants in queries. Third, the formula may contain the multiplication between variables, which is not allowed in a standard LIA* formula. This case is usually caused by joined tables and INTERSECT table operators in queries. Note that none of existing works [30, 39] has considered the above cases. Thus, we devise and implement new decision procedures and extensions for LIA* to address these challenges.

4 DECISION PROCEDURE WITH LIA*

This section provides detailed procedures for proving query equivalence based on the LIA* theory by addressing the challenges noted previously. First, we introduce the fundamental concepts of LIA* theory (Section 4.1). Second, SQLSolver reasons the equivalence problem between U-expressions based on the theory of LIA* (Section 4.2). Recognizing the semantic disparities between

U-expressions and LIA* formulas, SQLSolver proposes a novel decision procedure that reduces the equivalence problem of any two U-expressions to solving a LIA* formula. Third, compared to existing works using U-expressions [9, 43], SQLSolver is able to encode more SQL features under bag semantics, such as concrete aggregate functions (Section 4.3). Last, SQLSolver makes a further extension to support ordered bag semantics, which enables it to handle SQL queries with ORDER BY clauses (Section 4.4).

4.1 Basic Concept of LIA*

In a standard LIA* formula (Equation (2)), F_1 and F_2 are required to be linear integer arithmetic (LIA) formulas. Specifically, a LIA formula F is a first-order logic formula defined as follows:

$$F \coloneqq A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F_1 \mid \exists x.F \mid \forall x.F$$
$$A \coloneqq T_1 \leq T_2 \mid T_1 = T_2$$
$$T \coloneqq x \mid c \mid T_1 + T_2 \mid c \times T_1 \mid ite(F, T_1, T_2)$$

T denotes a linear integer expression that yields an integer result, while x represents an integer variable, and c is an integer constant. As T is linear, T can only be multiplied by a constant c. It also allows the expression of $ite(F, T_1, T_2)$, which returns T_1 if F is true, otherwise returns T_2. A is a first-order logic formula with the predicates of "$\leq$" and "=". The LIA formula F is either an atomic first-order logic formula (A) or multiple LIA formulas connected with logic connectors and quantifiers.

Previous work [39] has demonstrated that for every LIA* formula, there exists an equisatisfiable LIA formula (Theorem 4.1).

Theorem 4.1. *Given a LIA* formula $\exists \vec{u}, \vec{v}.F_1(\vec{u}, \vec{v}) \wedge \vec{v} \in \{\vec{x} \mid F_2(\vec{x})\}^*$, there exists an integer k such that the LIA* formula and the following formula are equisatisfiable:*

$$\exists \vec{u}, \vec{v}, \vec{x}_1, \ldots, \vec{x}_k, \lambda_1, \ldots, \lambda_k.F_1(\vec{u}, \vec{v}) \wedge \vec{v} = \sum_{i=1}^{k} \lambda_i \vec{x}_i \wedge F_2(\vec{x}_1) \wedge \ldots \wedge F_2(\vec{x}_k)$$

Previous work [30] has also proved that $\vec{x}_1, \ldots, \vec{x}_k$ serve as the generators of the LIA formula F_2. Given a LIA formula F_2, its generators consist of integer vectors $(\vec{y}_i)$ that fulfill the following two conditions: 1) $F_2(\vec{y}_1) \wedge \ldots \wedge F_2(\vec{y}_i) \wedge \ldots$; 2) $\forall \vec{x}.(F_2(\vec{x}) \rightarrow (\exists \vec{\lambda}.(\vec{x} = \sum \lambda_i \vec{y}_i)))$. Specifically, each $\vec{y}_i$ satisfies F_2. Each vector fulfilling F_2 can be represented as a linear combination of $\vec{y}_i$. According to the definition of a LIA* formula and generators, each vector $\vec{v}$ in $\{\vec{x} \mid F_2(\vec{x})\}^*$ is the linear combination of vectors satisfying F_2 and each vector satisfying F_2 is the linear combination of the generators of F_2. Thus, each vector $\vec{v}$ in $\{\vec{x} \mid F_2(\vec{x})\}^*$ can be represented as a linear combination of the generators of F_2. Notably, $\vec{x}_1, \ldots, \vec{x}_k$ in Theorem 4.1 correspond to the generators of F_2.

According to this conclusion, SQLSolver adopts an approximation-based method [30], which is a normal method of incrementally calculating the generators of a LIA formula. For example, to reason about the LIA* formula $\exists v_1, v_2, v_3.v_1 + v_2 = v_3 \wedge (v_1, v_2, v_3) \in \{(x_1, x_2, x_3) \mid x_3 = x_1 + x_2\}^*$, SQLSolver calculates the generators of its LIA formula $x_3 = x_1 + x_2$ with the approximation-based method. The resulting generators include $(1, 0, 1)$ and $(0, 1, 1)$. Apparently, each vector in the set $\{(x_1, x_2, x_3) \mid x_3 = x_1 + x_2\}^*$ can be represented as the linear combination of $(1, 0, 1)$ and $(0, 1, 1)$. Then, $(v_1, v_2, v_3) \in \{(x_1, x_2, x_3) \mid x_3 = x_1 + x_2\}^*$ is equisatisfiable to $\exists \lambda_1, \lambda_2.(v_1, v_2, v_3) = \lambda_1(1, 0, 1) + \lambda_2(0, 1, 1)$. The LIA* formula is reduced to $\exists v_1, v_2, v_3, \lambda_1, \lambda_2.v_1 + v_2 = v_3 \wedge (v_1, v_2, v_3) = \lambda_1(1, 0, 1) + \lambda_2(0, 1, 1)$.

4.2 Reasoning Equivalence of U-expressions

To prove the equivalence of Q_1 and Q_2, SQLSolver reasons the unsatisfiability of the formula $\exists t.f_1(t) \neq f_2(t)$, where $f_1(t)$ and $f_2(t)$ are the U-expressions of Q_1 and Q_2 accordingly. The basic idea

is converting the formula into a LIA* formula, which can be further reduced to a LIA formula and solved by SMT solvers.

First, SQLSolver uses new integer variables to replace unbounded summations in the formula of $\exists t.f_1(t) \neq f_2(t)$ and introduces an additive closure operator. Assume that the formula contains n unbounded summations $\sum E_1, \ldots$, and $\sum E_n$, SQLSolver first uses a new integer variable x_i to replace each expression E_i ($1 \leq i \leq n$). Then, it uses a variable v_i to represent each summation $\sum x_i$. As a result, the formula of $\exists t.f_1(t) \neq f_2(t)$ is rewritten into the following formula:

$$\begin{aligned} &\exists t, v_1, \ldots, v_n.f_1'(t) \neq f_2'(t) \\ &\quad \wedge (v_1, \ldots, v_n) \in \{(x_1, \ldots, x_n) \mid x_1 = E_1 \ldots \wedge x_n = E_n\}^*. \end{aligned} \tag{5}$$

Second, to conform to the stipulations of a standard LIA* formula, SQLSolver needs to convert each U-expression E_i in Equation (5) into a LIA formula. A straightforward method is to replace each U-expression term within E_i with a corresponding LIA term. For example, in Equation (4), SQLSolver achieves this transformation by replacing integer tuples and $R(t_1)$ with integer variables while replacing $[b]$ with $ite(b, 1, 0)$, respectively. Unfortunately, such a method fails to handle three categories of U-expressions:

(1) Nested summation. An unbounded summation is nested if some terms in the expression E_i are also unbounded summations. For example, the U-expression $\sum_{t_1}([t = x(t_1)] \times \sum_{t_2}(R(t_2) \times [x(t_2) = 1]))$ is a nested unbounded summation. Such summations are usually used to model SQL features in subqueries, such as Projection, EXISTS, and aggregate functions.
(2) Parameterized summation. An unbounded summation is parameterized if its expression contains free variables or non-integer variables. Given an unbounded summation, a variable is free if it is defined and used outside the summation expression. In $\sum_{t_1}([a(t_1) > 3] \times \sum_{t_2}([a(t_1) = a(t_2)] \times R(t_2)))$, the U-expression of $\sum_{t_2}([a(t_1) = a(t_2)] \times R(t_2))$ is a parameterized summation as t_1 is defined and used in the outer summation. A summation is also considered parameterized if it has variables in other types, such as strings. Such variables are unsupported by LIA* formulas.
(3) Non-linear summation. An unbounded summation is non-linear if the expression contains the multiplication between variables or functions. For example, $\sum_{t_1,t_2}(R(t_1) \times S(t_2))$ is a non-linear summation. When translating into a LIA* formula, this summation will introduce the multiplication between variables, which is non-linear. The U-expressions in the above example will be translated into $\sum(x_1 \times x_2)$, which is not allowed in LIA* formulas.

While the above cases are normal when utilizing U-expressions to model SQL queries, none of the existing works on LIA* reasoning [30, 38, 39] have considered these cases. Therefore, SQLSolver proposes new decision procedures by extending the basic approach in Section 3 to address these scenarios.

Case-1. The formula has nested summations. When some U-expression E_i in the formula of Equation (5) is also an unbounded summation, SQLSolver adopts the following reasoning procedure. For simplicity, we use $P(\vec{x})$ to represent $(x_1 = E_1 \wedge \ldots x_n = E_n)$ in Equation (5).

First, assume there are m unbounded summations in $P(\vec{x})$, which are $\sum e_1, \ldots, \sum e_m$. SQLSolver replaces each unbounded summation expression $\sum e_j$ with a new variable y_j and uses z_j to represent e_j, where $1 \leq j \leq m$. Then, it can apply the additive closure operator $*$, and Equation (5) is converted to:

$$\begin{aligned} &\exists t, \vec{v}, \vec{r}.f_1'(t) \neq f_2'(t) \\ &\quad \wedge (\vec{v}, \vec{r}) \in \left\{(\vec{x}, \vec{y}) \;\middle|\; P'(\vec{x}) \wedge \vec{y} \in \{\vec{z} \mid (z_1 = e_1) \wedge \ldots (z_m = e_m)\}^*\right\}^* \end{aligned} \tag{6}$$

Second, SQLSolver replaces each U-expression term with a LIA term. Specifically, it uses integer variables to replace the tuple variables t or $a(t)$. Each function $R(t)$ is also replaced with an integer

variable. Besides, it uses $ite(b, 1, 0)$ to replace $[b]$, uses $ite(E = 0, 0, 1)$ to replace $\|E\|$, and uses $ite(E = 0, 1, 0)$ to replace $not(E)$. Such replacement converts the $P'(\vec{x})$ into a LIA formula.

SQLSolver recursively performs the above two steps until all summations are replaced by formulas with additive closure operators (a.k.a., star formulas). Then, SQLSolver constructs a nested LIA* formula ℓ, which can be defined inductively:

$$\ell := \exists \vec{u}, \vec{v}.F(\vec{u}, \vec{v}) \wedge \vec{v} \in \{\vec{x} \mid \ell'(\vec{x})\}^* \tag{7}$$

Specifically, F is a LIA formula. $\ell'(\vec{x})$ can be either a LIA formula or a nested LIA* formula. Then, we have the following theorem.

Theorem 4.2. *For any nested LIA* formula $\ell := \exists \vec{u}, \vec{v}.F_1(\vec{u}, \vec{v}) \wedge \vec{v} \in \{\vec{x} \mid \ell'(\vec{x})\}^*$, if $\ell'(\vec{x})$ is equivalent to a LIA formula F_2, then ℓ is equivalent to $\exists \vec{u}, \vec{v}.F_1(\vec{u}, \vec{v}) \wedge \vec{v} \in \{\vec{x} \mid F_2(\vec{x})\}^*$.*

Based on the theorem, SQLSolver reasons the nested LIA* formula by recursively replacing each LIA* formula with another LIA formula.

For example, consider the following equivalent queries that other existing provers cannot prove:

```
Q1: select y from
    (select count(x) as y from R where x = 1) as S
Q2: select y from
    (select sum(x) as y from R where x + 1 = 2) as S
```

First, SQLSolver models them via U-expressions. However, existing methods [9, 43] cannot model concrete aggregate functions. Thus, SQLSolver extends existing encoding methods to support concrete aggregate functions, which can be found in Section 4.3. For this example, assume that the attribute x in R is unique and not NULL. "`select count(x) from R`" is encoded into $f_1(t) := [t = \sum_{t_1} \|R(t_1)\|]$, while "`select sum(x) from R`" is encoded into $f_2(t) := [t = \sum_{t_1}(x(t_1) \times \|R(t_1)\|)]$. Thus, the corresponding U-expressions of Q_1 and Q_2 are:

$$f_1(t) := \sum_{t_1} \left([t = y(t_1)] \times [y(t_1) = \sum_{t_2} (\|R(t_2)\| \times [x(t_2) = 1])] \right)$$

$$f_2(t) := \sum_{t_1} \left([t = y(t_1)] \times [y(t_1) = \sum_{t_2} (\|R(t_2)\| \times [x(t_2) + 1 = 2] \times x(t_2))] \right)$$

Second, SQLSolver constructs the following formula by applying the additive closure operator $*$:

$$\exists \vec{v}.(v_1 \neq v_2) \wedge (v_1, v_2, \ldots) \in \left\{ (x_1, x_2, \ldots) \;\middle|\; \begin{array}{l} x_1 = ite\left((x_3 = x_4) \wedge (x_3 = \sum_{t_2} (\|R(t_2)\| \times [x(t_2) = 1])), 1, 0 \right) \\ \wedge\ x_2 = ite\left((x_3 = x_4) \wedge (x_3 = \sum_{t_2} (\|R(t_2)\| \times [x(t_2) + 1 = 2] \times x(t_2))), 1, 0 \right) \\ \textbf{where } x_3 := y(t_1), x_4 := t \end{array} \right\}^*$$

Then, SQLSolver recursively applies the additive closure operator by using integer variables x_5 and x_6 to replace the inner two unbounded summations, which generates the following formula:

$$\begin{aligned}
&\exists \vec{v}.(v_1 \neq v_2) \wedge (v_1, v_2, \ldots) \in \{(x_1, x_2, \ldots) \mid \\
&\quad (x_1 = ite((x_3 = x_4) \wedge (x_3 = x_5), 1, 0)) \\
&\quad \wedge (x_2 = ite((x_3 = x_4) \wedge (x_3 = x_6), 1, 0)) \\
&\quad \wedge (x_5, x_6) \in \{(y_1, y_2) \mid y_1 = \|R(t_2)\| \times [x(t_2) = 1] \\
&\qquad \wedge y_2 = \|R(t_2)\| \times [x(t_2) + 1 = 2] \times x(t_2)\}^*\}^* \\
&\quad \textbf{where } x_3 \coloneqq y(t_1), x_4 \coloneqq t
\end{aligned}$$

This formula can be further changed by replacing U-expression terms with LIA terms, which results in a nested LIA* formula:

$$\begin{aligned}
&\exists \vec{v}.((v_1 \neq v_2) \wedge (v_1, v_2, \ldots) \in \{(x_1, x_2, \ldots) \mid \\
&\quad (x_1 = ite((x_3 = x_4) \wedge (x_3 = x_5), 1, 0)) \\
&\quad \wedge (x_2 = ite((x_3 = x_4) \wedge (x_3 = x_6), 1, 0)) \\
&\quad \wedge (x_5, x_6) \in \{(y_1, y_2) \mid (y_1 = ite(y_3 \neq 0 \wedge y_4 = 1, 1, 0)) \\
&\qquad \wedge (y_2 = ite(y_3 \neq 0 \wedge y_4 + 1 = 2, y_4, 0))\}^*\}^*) \\
&\quad \textbf{where } x_3 \coloneqq y(t_1), x_4 \coloneqq t, y_3 = R(t_2), y_4 = x(t_2)
\end{aligned}$$

To reason the formula, SQLSolver first converts the star formula $(x_5, x_6) \in \{(y_1, y_2) \mid (y_1 = ite(y_3 \neq 0 \wedge y_4 = 1, 1, 0)) \wedge (y_2 = ite(y_3 \neq 0 \wedge y_4 + 1 = 2, y_4, 0))\}^*$ into a LIA formula with the algorithm in Section 4.1.

$$\begin{aligned}
&\exists \vec{v}.(v_1 \neq v_2) \wedge (v_1, v_2, \ldots) \in \{(x_1, x_2, \ldots, \lambda) \mid \\
&\quad (x_1 = ite((x_3 = x_4) \wedge (x_3 = x_5), 1, 0)) \\
&\quad \wedge (x_2 = ite((x_3 = x_4) \wedge (x_3 = x_6), 1, 0)) \\
&\quad \wedge ((x_5, x_6) = \lambda(1, 1))\}^*
\end{aligned}$$

After that, it converts the above LIA* formula to the following LIA formula, which is apparently unsatisfiable:

$$\exists v_1, v_2, \lambda.(v_1 \neq v_2) \wedge (v_1, v_2) = \lambda(1, 1).$$

Case-2. The formula has parameterized summations. A LIA* formula with a free or non-integer variable is considered to have an external parameter. A variable is free if it is defined and used outside of the formula. Before going into the details, we first consider the following two SQL queries that involve string tuples.

```
Q1: select x from R where y = 'string'
Q2: select x from R where y like 'string'
```

As the non-integer variable y only exists in the predicate, one intuition is to separate the predicate reasoning from the SQL reasoning. Based on this intuition, SQLSolver addresses parameterized summations by decoupling the formula $\exists t.\ f_1(t) \neq f_2(t)$ into two parts: a standard LIA* formula and a first-order logic formula including the parameter. Then, it can leverage different decision procedures for reasoning. Specifically, it performs the following steps to decouple the formula:

First, for the formula with an additive closure operator $\vec{v} \in \{\vec{x} \mid P\}^*$ (a.k.a., star formula) in Equation (5) or Equation (6), SQLSolver converts P into an equivalent formula in disjunctive normal form (DNF) as below. Each P_{ij} is an atomic logic formula or its negation.

$$\vec{v} \in \{\vec{x} \mid (P_{11} \wedge P_{12} \wedge \ldots) \vee \ldots \vee (P_{n1} \wedge P_{n2} \wedge \ldots)\}^* \tag{8}$$

Given a first-order logic formula, SQLSolver converts it into the DNF by repeatedly applying distributive laws, De Morgan laws, etc. [4]. Furthermore, SQLSolver has to expand each $v = ite(c, t, e)$ in the formula with $(c \wedge (v = t)) \vee (\neg c \wedge (v = e))$, as the DNF does not accept if-then-else (ite) expressions.

Second, SQLSolver converts Equation (8) into a conjunction of multiple star formulas. Existing work [36] has shown that the formula $\vec{v} \in \{\vec{x} \mid P_1 \vee P_2\}^*$ is equivalent to $\exists v_1, v_2.(\vec{v} = \vec{v}_1 + \vec{v}_2) \wedge \vec{v}_1 \in \{\vec{x} \mid P_1\}^* \wedge \vec{v}_2 \in \{\vec{x} \mid P_2\}^*$. Thus, Equation (8) can be converted to the following formula:

$$\begin{aligned} &\exists \vec{v}_1, \ldots, \vec{v}_n.(\vec{v} = \vec{v}_1 + \ldots + \vec{v}_n) \wedge \vec{v}_1 \in \{\vec{x} \mid P_{11} \wedge P_{12} \wedge \ldots\}^* \\ &\quad \wedge \ldots \wedge \vec{v}_n \in \{\vec{x} \mid P_{n1} \wedge P_{n2} \wedge \ldots\}^* \end{aligned} \tag{9}$$

Third, SQLSolver identifies each predicate P_{ij} involving external parameters and decouples these predicates from Equation (9) with the following theorem:

Theorem 4.3. *Assume that $\vec{v}$ and $\vec{x}$ are integer vectors with the same size, while $\vec{y}$ is a vector that is independent of $\vec{v}$ and $\vec{x}$. A vector $\vec{a} \coloneqq (a_1, \ldots a_n)$ is independent of $\vec{b} \coloneqq (b_1, \ldots b_m)$ if and only if for any a_i and b_j, $a_i \neq b_j$. Then, for any first-order logic formula P, P_1, and P_2, the following formula*

$$\exists \vec{u}, \vec{v}.P(\vec{u}, \vec{v}) \wedge (\vec{v}, \_) \in \{(\vec{x}, \vec{y}) \mid P_1(\vec{x}) \wedge P_2(\vec{y})\}^* \tag{10}$$

is equisatisfiable with

$$\exists \vec{u}, \vec{v}.P(\vec{u}, \vec{v}) \wedge \left((\vec{v} \in \{\vec{x} \mid P_1(\vec{x})\}^* \wedge \exists \vec{y}.P_2(\vec{y})) \vee (\vec{v} = \vec{0})\right). \tag{11}$$

Note that since $\vec{y}$ is independent of $\vec{x}$ and $\vec{v}$, its variables could be of any type, and the formula of P_2 is allowed to contain free variables. To apply the theorem, SQLSolver needs to check the LIA* formulas in Equation (9) and identify those predicates P_{ij} that may contain parameters. Then, each LIA* formula can be converted into Equation (10), where P_1 is the conjunction of predicates only including integer variables, and P_2 is the conjunction of predicates having parameters. SQLSolver needs to further check whether $\vec{y}$ is independent of $\vec{x}$ and $\vec{v}$. If this criterion is satisfied, it can decouple the predicate P_2 with parameters from star formulas by altering Equation (10) to Equation (11).

Last, in Equation (11), $\exists \vec{u}, \vec{v}.P(\vec{u}, \vec{v}) \wedge \vec{v} \in \{\vec{x} \mid P_1(\vec{x})\}^*$ is a standard LIA* formula that typical LIA* solvers [30, 39] can solve. $\exists \vec{y}.P_2(\vec{y})$ is a first-order logic formula that SMT solvers can solve.

For example, consider the following equivalent queries:

```
Q1: select x2
    from R0 join (select distinct x0 from R1) as S on R0.x1 = S.x0
Q2: select x2
    from R0
    where R0.x1 in (select x0 from R1)
```

Their corresponding U-expressions are

$$f_1(t) \coloneqq \sum_{t_0} \left([t = x_2(t_0)] \times R_0(t_0) \times not([IsNull(x_1(t_0))]) \times \left\| \sum_{t_1} (R_1(t_1) \times [x_1(t_0) = x_0(t_1)]) \right\| \right)$$

$$f_2(t) \coloneqq \sum_{t_0} \left([t = x_2(t_0)] \times R_0(t_0) \times \left\| \sum_{t_1} (R_1(t_1) \times [x_1(t_0) = x_0(t_1)] \times not([IsNull(x_1(t_0))])) \right\| \right)$$

Apparently, they include nested summations. Thus, SQLSolver applies the procedure for case-1 and translates the formula $\exists t.f_1(t) \neq f_2(t)$ into the following nested LIA* formula:

$$\begin{aligned}
&\exists v_1,v_2.v_1 \neq v_2 \wedge (v_1, v_2, \ldots) \in \{(x_1, x_2, \ldots) \mid \\
&\quad x_1 = ite(x_4 = 0 \wedge x_5 \neq 0 \wedge x_7 = x_8, x_3, 0) \\
&\quad \wedge x_2 = ite(x_6 \neq 0 \wedge x_7 = x_8, x_3, 0) \\
&\quad \wedge (x_5, x_6, _) \in \{(y_1, y_2, _) \mid y_1 = ite(y_4 = y_5, y_3, 0) \\
&\qquad \wedge y_2 = ite(x_4 = 0 \wedge y_4 = y_5, y_3, 0)\}^*\}^* \\
&\quad \textbf{where} \\
&\quad v_1 \coloneqq f_1(t),\ v_2 \coloneqq f_2(t),\ x_3 \coloneqq R_0(t_0),\ x_4 \coloneqq [IsNull(x_1(t_0))] \\
&\quad x_7 \coloneqq t,\ x_8 \coloneqq x_2(t_0),\ y_3 \coloneqq R_1(t_1),\ y_4 \coloneqq x_1(t_0),\ y_5 \coloneqq x_0(t_1)
\end{aligned}$$

According to the procedure for case-1, SQLSolver needs to recursively translate the nested LIA* formula into a LIA formula. However, the inner LIA* formula $(x_5, x_6, \_) \in \{(y_1, y_2, \_) \mid y_1 = \ldots \wedge y_2 = ite(x_4 = 0 \wedge \ldots)\}^*$ is not in the standard form because x_4 is a free variable defined and used by the outer formula.

To handle the inner LIA* formula with the free variable x_4, SQLSolver first converts the formula $y_1 = \ldots \wedge y_2 = ite(x_4 = 0 \wedge \ldots)$ into a DNF formula. As DNF does not allow ite operators, SQLSolver expands ite expressions. The inner LIA* formula is translated into the following formula:

$$\begin{aligned}
(x_5, x_6, _) \in \{(y_1, y_2, _) \mid &((y_4 = y_5) \wedge (x_4 = 0) \wedge (y_1 = y_3) \wedge (y_2 = y_3)) \\
&\vee ((y_4 = y_5) \wedge (x_4 \neq 0) \wedge (y_1 = y_3) \wedge (y_2 = 0)) \\
&\vee ((y_4 \neq y_5) \wedge (y_1 = 0) \wedge (y_2 = 0))\}^*
\end{aligned}$$

This formula can be further translated as a conjunction of three star formulas:

$$\begin{aligned}
&\exists \vec{a}, \vec{b}, \vec{c}.((x_5, x_6) = (a_1, a_2) + (b_1, b_2) + (c_1, c_2) \\
&\quad \wedge (a_1, a_2, _) \in \{(y_1, y_2, _) \mid (y_4 = y_5) \wedge (x_4 = 0) \wedge (y_1 = y_3) \wedge (y_2 = y_3)\}^* \\
&\quad \wedge (b_1, b_2, _) \in \{(y_1, y_2, _) \mid (y_4 = y_5) \wedge (x_4 \neq 0) \wedge (y_1 = y_3) \wedge (y_2 = 0)\}^* \\
&\quad \wedge (c_1, c_2, _) \in \{(y_1, y_2, _) \mid (y_4 \neq y_5) \wedge (y_1 = 0) \wedge (y_2 = 0)\}^*
\end{aligned}$$

To apply Theorem 4.3 to the formula $(a_1, a_2) \in \{\vec{y} \mid (y_4 = y_5) \wedge (x_4 = 0) \wedge (y_1 = y_3) \wedge (y_2 = y_3)\}^*$, SQLSolver identifies that $x_4 = 0$ is a parameterized predicate. According to Theorem 4.3, the above star formula can be converted into the following formula without parameters:

$$\left((a_1, a_2, \_) \in \{(y_1, y_2, \_) \mid (y_1 = y_3) \wedge (y_2 = y_3) \wedge (y_4 = y_5)\}^* \wedge (x_4 = 0)\right) \vee \left((a_1, a_2, \_) = \vec{0}\right)$$

Similarly, SQLSolver shifts the parameter x_4 outside $(b_1, b_2) \in \{\vec{y} \mid (y_4 = y_5) \wedge (x_4 \neq 0) \wedge (y_1 = y_3) \wedge (y_2 = 0)\}^*$. Following the algorithm for case-1, SQLSolver further converts each inner star formula to a LIA formula and generates a LIA* formula. Finally, SQLSolver further generates an unsatisfiable LIA formula $\exists v_1, v_2, \lambda.(v_1 \neq v_2) \wedge ((v_1, v_2) = \lambda(1, 1))$.

Case-3. The formula has non-linear summations. A standard LIA* formula only allows multiplications between a constant and a variable. Non-linear operations, such as multiplications between variables, cannot be resolved by currently available decision procedures [30, 38, 39]. Unfortunately, these non-linear operations are frequently encountered when modeling SQL features, such as INTERSECT, joined tables, and aggregate functions.

SQLSolver employs the concept of over-approximation [30] to handle non-linear summations. An over-approximation of a logic formula P is a logic formula Q if and only if $\neg Q \rightarrow \neg P$ is true. The definition implies that if Q is unsatisfiable, then P is also unsatisfiable. Therefore, when the formula $\exists t.f_1(t) \neq f_2(t)$ comprises non-linear summations, SQLSolver can construct its over-approximation

that only includes linear operations to reason about the unsatisfiability. Specifically, given the formula $\exists t.f_1(t) \neq f_2(t)$ with non-linear summations, SQLSolver first constructs Equation (5) and replaces each U-expression term with a LIA term. Then, the formula can be represented as:

$$\exists \vec{u}, \vec{v}.F(\vec{u}, \vec{v}) \wedge \vec{v} \in \{\vec{x} \mid (x_1 = (x_2 \times \ldots \times x_n)) \wedge P(\vec{x})\}^* \tag{12}$$

Then, SQLSolver replaces the multiplication expression $(x_2 \times \ldots \times x_n)$ with a new variable y and generates the following formula:

$$\exists \vec{u}, \vec{v}.F(\vec{u}, \vec{v}) \wedge (\vec{v}, \_) \in \{(\vec{x}, y) \mid (x_1 = y) \wedge P(\vec{x})\}^*$$

Note that if two multiplication expressions involve the same group of variables, they will be substituted with the same variable.

Finally, SQLSolver imposes an additional linear constraint on the value of the new variable. Specifically, the constraint necessitates that the new variable y is zero if and only if the multiplication expression $(x_2 \times \ldots \times x_n)$ is zero:

$$\begin{aligned} &\exists \vec{u}, \vec{v}.F(\vec{u}, \vec{v}) \wedge (\vec{v}, \_) \in \{(\vec{x}, y) \mid (x_1 = y) \wedge P(\vec{x}) \\ &\quad \wedge ((x_2 = 0 \vee \ldots \vee x_n = 0) \leftrightarrow (y = 0))\}^* \end{aligned} \tag{13}$$

Equation (13) is an over-approximation of Equation (12). Thus, instead of reasoning the Equation (12), SQLSolver directly proves Equation (13).

For example, consider the following equivalent SQL queries, which are also unable to be reasoned by other provers:

```
Q1: select distinct R.x from R
Q2: select distinct S.x from
    (select * from R intersect select * from R) as S
```

Their corresponding U-expressions are

$$f_1(t) \coloneqq \left\| \sum_{t_1} ([t = x(t_1)] \times R(t_1)) \right\|$$

$$f_2(t) \coloneqq \left\| \sum_{t_1} ([t = x(t_1)] \times \|R(t_1) \times R(t_1)\|) \right\|$$

For these two U-expressions, SQLSolver constructs the following formula:

$$\begin{aligned} &\exists \vec{v}.ite(v_1 = 0, 0, 1) \neq ite(v_2 = 0, 0, 1) \\ &\quad \wedge (v_1, v_2, \_) \in \{(x_1, x_2, \_) \mid x_1 = ite(x_3 = x_4, x_5, 0) \\ &\qquad \wedge x_2 = ite(x_3 = x_4 \wedge x_5 \times x_5 \neq 0, 1, 0)\}^* \\ &\quad \textbf{where } x_3 \coloneqq t,\ x_4 \coloneqq x(t_1),\ x_5 \coloneqq R(t_1) \end{aligned} \tag{14}$$

SQLSolver constructs an over-approximation of the above formula by replacing $x_5 \times x_5$ with x_6.

$$\begin{aligned} &\exists \vec{v}.ite(v_1 = 0, 0, 1) \neq ite(v_2 = 0, 0, 1) \\ &\quad \wedge (v_1, v_2, \_) \in \{(x_1, x_2, \_) \mid x_1 = ite(x_3 = x_4, x_5, 0) \\ &\qquad \wedge x_2 = ite(x_3 = x_4 \wedge x_6 \neq 0, 1, 0) \\ &\qquad \wedge ((x_5 \neq 0) \leftrightarrow (x_6 \neq 0))\}^* \end{aligned}$$

This rewrite enables SQLSolver to use LIA* theory to convert the formula to the following unsatisfiable LIA formula, which implies the equivalence between two queries.

$$\exists v_1, v_2, \lambda.(ite(v_1 = 0, 0, 1) \neq ite(v_2 = 0, 0, 1)) \wedge ((v_1, v_2) = \lambda(1, 1))$$

All the above decision procedures provide soundness. The detailed discussion can be found in Section 5.

Table 3. Main extensions of U-expressions to support more SQL features.

SQL Feature	U-expression $f(t)$	Example Query	Example U-expression $f(t)$
Aggregate Function	Table 4	/	/
INTERSECT	$\|\|f_l(t) \times f_r(t)\|\|$	(select * from R) intersect (select * from S)	$\|\|R(t) \times S(t)\|\|$
INTERSECT ALL	$E \times f_l(t) + not(E) \times f_r(t)$, $E \coloneqq [f_l(t) \leq f_r(t)]$	(select * from R) intersect all (select * from S)	$E \times R(t) + not(E) \times S(t)$, $E \coloneqq [R(t) \leq S(t)]$
values $(t_1) \cdots$	$[t = t_1] + \cdots$	select * from (values (1,2), (3, 4))	$[(x(t) = 1) \wedge (y(t) = 2)]$ $+[(x(t) = 3) \wedge (y(t) = 4)]$
(Scalar Subquery) select x from R where p	$[1 = \sum_{t_1} E]$ $\times\|\|\sum_{t_1}(E \times [t = a(t_1)])\|\|$ $+IsNULL(t) \times not(\sum_{t_1} E)$, $E \coloneqq R(t_1) \times [p(t_1)]$	select (select x from R)	$[1 = \sum_{t_1} R(t_1)]$ $\times\|\|\sum_{t_1}(R(t_1) \times [t = x(t_1)])\|\|$ $+IsNull(t) \times not(\sum_{t_1} R(t_1))$
a = case when p_0 then a_0 $\cdots$ else a_n end	$[p_0] \times [a = a_0] + \cdots$ $+not([p_0]) \times \cdots$ $\times not([p_{n-1}]) \times [a = a_n]$	select (case when R.x = 0 then 1 else 0 end) from R	$\sum_{t_1}(R(t_1) \times [x(t_1) = 0]$ $\times[t = 1] + R(t_1)$ $\times not([x(t_1) = 0]) \times [t = 0])$
a in $(v_1, \ldots)$	$\|\|[a(t) = v_1] + \ldots\|\|$	select * from R where R.x in (1, 2)	$R(t) \times \|\|[x(t) = 1] + [x(t) = 2]\|\|$

4.3 Extension of U-expressions

Compared with previous provers [9, 43] based on U-expressions, SQLSolver additionally extends the translation method from SQL queries to U-expressions and thus allows modeling more SQL features, such as concrete aggregate functions, INTERSECT, scalar subqueries, and VALUES. Table 3 shows our main translation methods for these SQL features.

One of the main extensions is the translation of queries with aggregate functions, including all kinds of aggregate functions specified as mandatory SQL features in the SQL standard [24]. Previous approaches treat aggregate functions as uninterpreted functions, neglecting their concrete semantics. A query with aggregate functions typically takes the following structure, where R denotes a relation and p denotes a predicate:

```
/* agg_func can be max, min, sum, count, and avg */
select R.x, agg_func([distinct] R.y)
from R group by R.x having p
```

Table 4 shows how to model the concrete semantics of aggregate functions via U-expressions. Each tuple in the query result is the concatenation of t_l and t_r, which is denoted by $t_l \cdot t_r$. The tuple t_l represents the value of $R.x$, while t_r represents the value of agg_func(R.y). The basic idea of generating U-expressions is to encode the requirement t_l and t_r should satisfy. Although different aggregate functions have different semantics, there are two common requirements for t_l and t_r. First, the value of agg_func(R.y) should satisfy the predicate p of the HAVING clause, which is denoted by $[p(t_r)]$. The second requirement is that there should exist at least one tuple in the relation R

Table 4. The translation method from a query with aggregate functions, GROUP BY, and HAVING into a U-expression.

Function Name	U-expression
SUM	$f(t_l \cdot t_r) \coloneqq [p(t_r)] \times \|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)])\| \times$ $[t_r = \sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times$ $not([IsNull(y(t_1))]) \times y(t_1))]$
SUM DISTINCT	$f(t_l \cdot t_r) \coloneqq [p(t_r)] \times \|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)])\| \times$ $[t_r = \sum_{t_1}(t_1 \times not([IsNull(t_1)]) \times$ $\|\sum_{t_2}(R(t_2) \times [t_l = x(t_2)] \times [y(t_2) = t_1])\|)]$
COUNT	$f(t_l \cdot t_r) \coloneqq [p(t_r)] \times \|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)])\| \times$ $[t_r = \sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times$ $not([IsNull(y(t_1))]))]$
COUNT DISTINCT	$f(t_l \cdot t_r) \coloneqq [p(t_r)] \times \|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)])\| \times$ $[t_r = \sum_{t_1}(\|\sum_{t_2}(R(t_2) \times [t_l = x(t_2)] \times [t_1 = y(t_2)] \times$ $not([IsNull(y(t_2))]))\|)]$
MAX	$f(t_l \cdot t_r) \coloneqq [p(t_r)] \times \|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)])\| \times$ $not(\sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times [y(t_1) > t_r])) \times$ $\|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times [y(t_1) = t_r]\|$
MIN	$f(t_l \cdot t_r) \coloneqq [p(t_r)] \times \|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)])\| \times$ $not(\sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times [y(t_1) < t_r])) \times$ $\|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times [y(t_1) = t_r]\|$
AVG	$f(t_l \cdot t_r) \coloneqq [p(t_r)] \times \|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)])\| \times$ $[t_r \times (\text{COUNT Expr}) = (\text{SUM Expr})]$

such that its attribute x equals t_l. Otherwise, the query result has no tuples. This requirement is represented by $\|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)])\|$.

Then, we introduce how to model different semantics for different aggregate functions in U-expressions. The straightforward idea is to represent that t_r is the result of agg_func(R.y) in U-expressions. When the aggregate function is COUNT, the summations in U-expressions can be used to calculate the result. Specifically, $[t_r = \sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times not([IsNull(y(t_1))]))]$ means that t_r equals the sum of the attribute y from every possible tuple t_1 satisfying: 1) t_1 is in the relation R; 2) the attribute x of t_1 equals to t_l; 3) the attribute y of t_1 is not NULL because COUNT ignores NULL. Similarly, we can leverage summations to calculate the result of SUM, SUM DISTINCT, and COUNT DISTINCT.

Different from previous aggregate functions, MAX compares the attribute y of tuples and finds the maximum value to be the result, which cannot be directly calculated by summations. Thus, SQLSolver models two properties of the maximum value rather than calculating the exact value. These properties can imply that t_r is the result of max(R.y). First, no tuple in the relation R can satisfy 1) its attribute x equals t_l, and 2) its attribute y is greater than t_r. This property is denoted as $not(\sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times [y(t_1) > t_r]))$. Second, there exists a tuple in relation R such that its attribute x is t_l and its attribute y equals t_r. The property is denoted by $\|\sum_{t_1}(R(t_1) \times [t_l = x(t_1)] \times [y(t_1) = t_r])\|$. Similarly, we can generate U-expressions for queries with MIN, which can be found in Table 4.

Finally, we explain how to model queries with AVG functions. Since SQLSolver can represent the result of sum(R.y) and count(R.y) in U-expressions, the basic idea is to calculate the result of avg(R.y) via dividing the result of sum(R.y) by the result of count(R.y). Although there is no division operator in U-expressions, we observe that the result of sum(R.y) equals the multiplication between the result of count(R.y) and avg(R.y). Thus, SQLSolver models the semantics of avg(R.y)

by $[t_r \times (\text{COUNT } Expr) = (\text{SUM } Expr)]$, where COUNT *Expr* and SUM *Expr* are U-expressions representing the result of count(R.y) and sum(R.y), respectively.

4.4 Support of Ordered Bag Semantics

Existing works focus on proving the equivalence of queries under bag or set semantics. In particular, two queries are equivalent if their results consist of the same tuples. However, SQL queries may return ordered tuples (i.e., ordered bag semantics), such as queries with ORDER BY clauses.

To support ordered bag semantics, SQLSolver adopts a "divide and conquer" strategy. The equivalence of queries with ORDER BY clauses can be proved by the equivalence of their sub-queries without ORDER BY clauses. Specifically, if two queries sort the tuples returned by sub-queries according to the value of the same attribute, the equivalence of the two queries can be implied by the equivalence of their sub-queries. For example, we can prove the equivalence between "`select * from R order by id`" and "`select * from S order by id`" by proving the equivalence between "`select * from R`" and "`select * from S`".

Before discussing details, we need to revise the definition of query equivalence under ordered bag semantics.

Definition 1. Given two SQL queries Q_1 and Q_2, if the result of Q_1 is ordered, then Q_2 is equivalent to Q_1 if and only if: 1) the results of Q_1 and Q_2 contain the same multiset of tuples; 2) for any two tuples $t1$ and $t2$, they are in the same order in both results of Q_1 and Q_2.

SQLSolver first simplifies queries by eliminating and merging ORDER BY clauses. Then, it checks the equivalence between two queries by recursively checking their sub-queries based on the algorithm in Section 4.2.

Step 1. Eliminate and merge ORDER BY clauses. SQLSolver eliminates and merges ORDER BY clauses in the following three cases. First, an ORDER BY clause is in a sub-query and is not followed by LIMIT or OFFSET clauses. This is because an ORDER BY clause affects the result of queries only if it appears in the top-most query. Second, a query in the form of "`R order by expression limit 0`" can be replaced by "`select * from (values) as R`", where "values" represents an empty table. Third, a query in the form of the following Q_1 can be rewritten into Q_2.

```
Q1: (R1 order by expression limit number_rows offset offset_value)
    [union all|full join|left join] R2
    order by expression limit number_rows offset offset_value
Q2: (R1 [union all|full join|left join] R2)
    order by expression limit number_rows offset offset_value
```

SQLSolver can also merge multiple ORDER BY clauses into one clause. Specifically, for a sub-query in the form of the following Q_1, SQLSolver will merge its ORDER BY/ LIMIT/ OFFSET clauses and generate Q_2.

```
Q1: (R order by expression limit number_rows1 offset offset_value1)
    order by expression limit number_rows2 offset offset_value2
Q2: R order by expression limit number_rows3 offset offset_value3
```

The constant offset_value3 equals (offset_value1 + offset_value2). When (offset_value2 + number_rows2) ≤ number_rows1, number_rows3 equals to number_rows2. Otherwise, number_rows3 is the *MAX*((number_rows1 - offset_value2), 0).

Step 2. Check the equivalence recursively. Assume two queries after simplification are Q_1 and Q_2. First, SQLSolver checks whether Q_1 and Q_2 have ORDER BY clauses. If no ORDER BY clauses exist, SQLSolver invokes the algorithm based on LIA to check their equivalence and finishes. Second, SQLSolver randomly selects an ORDER BY clause in Q_1. If it fails to find ORDER BY

clauses, the verification fails. The two queries are not equivalent. Otherwise, it further finds the set of all ORDER BY clauses in Q_2 that match the selected ORDER BY clause in Q_1, which is S. Specifically, two ORDER BY clauses are matching means: 1) they sort tuples by the same attributes; 2) they are followed by the same LIMIT/ OFFSET clauses. Third, SQLSolver returns true when any ORDER BY clause $\in S$ passes the following check, which means Q_1 and Q_2 are equivalent. Assume that two ORDER BY clauses in Q_1 and Q_2 are performed on the tuples returned by the sub-query Q_3 and Q_4, respectively. SQLSolver recursively checks the equivalence between Q_3 and Q_4. If they are proved to be equivalent, SQLSolver replaces Q_3 and Q_4 along with their following ORDER BY/ LIMIT/ OFFSET clauses in Q_1 and Q_2 by the same arbitrary relation, generating two new queries Q_5 and Q_6. Then, SQLSolver recursively checks the equivalence between Q_5 and Q_6. If they are proved to be equivalent, the check succeeds.

The following theorem shows the soundness of the above algorithm.

Theorem 4.4. *Given two queries with ORDER BY clauses, if they pass the check of the step 2 after simplification, they are equivalent under ordered bag semantics.*

The theorem can be proved by two lemmas about the above two steps. First, eliminating unnecessary ORDER BY clauses from a query always produces another equivalent query. Second, if two queries can pass the check of step 2, then they are equivalent under ordered bag semantics.

5 SOUNDNESS AND COMPLETENESS

Verifying the equivalence of two SQL queries is an undecidable problem [1]. Thus, it is impossible to have a verification algorithm that is both sound and complete. The decision procedure proposed by SQLSolver is sound but incomplete. This section provides a proof sketch of the soundness and a short discussion about the completeness.

Theorem 5.1. *SQLSolver ensures soundness under both bag semantics and ordered bag semantics. Specifically, given two queries Q_1 and Q_2, if SQLSolver proves they are equivalent, they must be equivalent under bag semantics or ordered bag semantics.*

Proof sketch. When two queries Q_1 and Q_2 do not have ORDER BY clauses, we need to prove that the unsatisfiability of the LIA formula generated by SQLSolver implies the equivalence between two queries. UDP [9] has proved that the equivalence of two queries' U-expressions $f_1(t)$ and $f_2(t)$ implies the equivalence of Q_1 and Q_2. Thus, we only need to prove that if the generated LIA formula is unsatisfiable, then $f_1(t)$ and $f_2(t)$ are equivalent. When two queries have ORDER BY clauses, the soundness can be proved by Theorem 4.4. □

On the aspect of completeness, SQLSolver cannot ensure completeness because of the following reasons: 1) it does not model all SQL features, such as lateral sub-queries. This is also the major reason why there are 13 query pairs derived from Spark SQL that cannot be proved by SQLSolver; 2) Translating U-expressions into a LIA* formula and the algorithm of handling non-linear operations can also introduce completeness issues. Specifically, the FOL formula generated by SQLSolver is the over-approximation [30] of the original formula $\exists x.f_1(x) \neq f_2(x)$. They are not equisatisfiable. If the FOL formula is unsatisfiable, $\exists x.f_1(x) \neq f_2(x)$ is unsatisfiable. The two queries are equivalent. However, if the FOL formula is satisfiable, SQLSolver cannot ensure that $\exists x.f_1(x) \neq f_2(x)$ is satisfiable. Two queries may still be equivalent. 3). Our method of supporting ordered bag semantics relies on syntax structures and cannot handle any queries with ORDER BY clauses. However, in our evaluation, the last two reasons do not incur any false positives. The equivalent query pairs that fail to be proved are due to the lack of semantics modeling.

Besides, SQLSolver ensures the completeness for conjunctive queries (CQ) and unions of conjunctive queries (UCQ), which is similar to existing works [9, 47]. However, the real power of

SQLSolver lies in other kinds of queries. To illustrate it, we present another kind of SQL query such that SQLSolver guarantees completeness, whereas other provers cannot.

Theorem 5.2. *SQLSolver is complete if the input SQL queries have the following form*

Q_1: Q'_1 `union all` ... `union all` Q'_n
Q_2: Q'_{n+1} `union all` ... `union all` Q'_m

and each Q'_i has either one of the following forms.

Q'_i: `select` $R_i.a_1$, ..., $R_i.a_j$, $agg_1(R_i.b_1)$, ..., $agg_k(R_i.b_k)$
`from` R_i `where` p_i
`group by` $R_i.a_1$, ..., $R_i.a_j$ `having` q_i
Q'_i: `select [distinct]` $R_i.a_1$, ..., $R_i.a_j$ `from` R_i `where` p_i

The aggregate functions are limited to count, max or min, and these queries do not include any predicates whose satisfiability cannot be determined by SMT solvers.

6 EVALUATION

In evaluation, we want to answer the following questions.

Q1. Does SQLSolver have better verification capability than existing provers?
Q2. What is the main reason for its advantage?
Q3. Can SQLSolver help find more rewrite rules?

6.1 Experimental Setup

SQLSolver is built using Java 17.0.4 and employs Z3 4.8.9 [15] as the SMT solver. It implements a parser for queries based on the parser generator ANTLR 4.8 [35].

Baseline. To answer the first two questions, we compare the verification capability of SQLSolver with other provers based on their source code, including UDP [10], SPES [45], and WeTune [44].

Benchmark. The verification capability is evaluated using 232 equivalent query pairs derived from Calcite test suites and 127 pairs of equivalent queries derived from the rewrite rules in the optimization engine of Spark SQL. Additionally, we construct 19 pairs of equivalent queries from TPC-C [21] and 22 pairs of equivalent queries from TPC-H [22]. Specifically, TPC-C and TPC-H are used to evaluate the performance of database systems. We use the query optimizer of Spark SQL to generate an equivalent query for each query in TPC-C and TPC-H. Then, we use SQLSolver to verify the equivalence of each pair of queries. Since Spark SQL generates a logical plan rather than a SQL query, we manually convert a plan into a SQL query while preserving its semantics. For the third question, we integrate SQLSolver into WeTune, which discovers new rewrite rules based on query equivalence verification. Then, we compare the rules discovered based on SQLSolver with those in WeTune's paper.

Testbed. Each prover is executed on an AWS EC2 c5a.8xlarge machine. To discover new rules, we integrate SQLSolver into WeTune and run the rule discovery program on m4.10xlarge AWS EC2 instances. When assessing the performance benefits posed by rules discovered by SQLSolver, we use them to rewrite queries and measure latency on Microsoft SQL Server 2019. The initial tables with 1M rows are populated with randomly generated data derived from the uniform distribution. Each query is recurrently executed 200 times within a closed-loop framework.

6.2 Verification Capability

Among all 400 equivalent query pairs, UDP, SPES, and WeTune are able to prove the equivalence of 207 pairs in total. Compared with them, SQLSolver can prove 384 pairs, 177 of which cannot be proved by any of these existing provers. Notably, every equivalent query that can be proved by

Table 5. Reasons of verification failure for each prover and the number of failed query pairs due to each reason.

	Unsupported Cases	Reason		
		Unsupported SQL Feature	Checking Algorithm	Ordered Bag Semantics
UDP	331	132	160	39
SPES	214	103	72	39
WeTune	275	63	181	31

existing provers can also be proved by SQLSolver. Additionally, SQLSolver could prove all 232 query pairs derived from Calcite and all 19 query pairs constructed based on TPC-C. There remain a total of 16 query pairs that could not be verified by SQLSolver or any other existing provers. The main reason is due to SQL features unsupported by SQLSolver, such as lateral subqueries and window functions.

To address the second question, we analyze all failed query pairs for each existing provers out of the query pairs that SQLSolver can prove, as shown in Table 5. Note that WeTune does not support ordered bag semantics, but it can reason some equivalent queries with ORDER BY clauses. Because it performs verification by repeatedly rewriting their subqueries without ORDER BY clauses until the two queries with ORDER BY become the same. Thus, we still treat it to be a prover that does not consider ordered bag semantics. The algebraic representation defined by SPES cannot represent scalar subqueries, but it can prove a query pair with scalar subqueries. Because it uses Calcite to transform SQL queries into logical plans, which eliminates scalar subqueries via rewrite. Thus, we still treat SPES to be a prover that does not support scalar subqueries. The first reason is that SQLSolver supports more SQL features under bag semantics, such as VALUES and scalar sub-queries. Among query pairs that can be proved by SQLSolver, 132, 103, and 63 query pairs cannot be proved by UDP, SPES and WeTune due to this reason, respectively. In addition, SQLSolver's LIA-based verification algorithm also contributes a lot to its verification capability. Even if the given queries do not have SQL features unsupported by other provers or other provers adopt our algorithm to convert more SQL features to U-expressions, they still cannot prove many pairs due to this reason. Among the query pairs proved by SQLSolver under bag semantics, UDP and SPES fail to prove 160, 72 pairs of them due to the limitations of their syntax-based algorithm. They cannot handle queries whose algebraic representations vary a lot. WeTune fails to prove 181 pairs of them because its semantics-based algorithm cannot handle complicated unbounded summations in U-expressions via its rules. Last, existing provers target bag semantics rather than ordered bag semantics. This also causes verification failure, as shown in Table 5.

The algorithm SQLSolver employs to address ORDER BY clauses is decoupled with the algorithm used to establish the equivalence of queries under bag semantics. We combine other provers with it to address queries with ORDER BY clauses. Among the total of 400 query pairs, SQLSolver relies on the algorithm in Section 4.4 to prove 39 of them under ordered bag semantics. Combined with our algorithm, UDP, SPES, and WeTune can prove the equivalence of 22 query pairs in total. The other 17 query pairs cannot be proved by them because of SQL features unsupported by these provers and the limitations of their checking algorithms. This evaluation result demonstrates that SQLSolver's algorithm of addressing ORDER BY clauses and its algorithm of proving query equivalence under bag semantics are both essential to handling these queries with ORDER BY clauses.

Table 6 presents a comparison of the average verification latency between SQLSolver and other existing provers, including UDP, SPES, and WeTune. To ensure fairness in our comparison, we compare each pair of provers on a set of query pairs that both provers can prove. Note that every pair

Table 6. Average verification latency (ms) for each prover. UDP vs. SQLSolver means the average latency of proving query pairs that can be proved by both of them. We use "/" to indicate that no query pair can be proved by both provers.

	Calcite	Spark SQL	TPC-C	TPC-H
UDP vs. SQLSolver	3178 vs. 28	3039 vs. 12	/	/
SPES vs. SQLSolver	45 vs. 122	17 vs. 35	38 vs. 39	/
WeTune vs. SQLSolver	8 vs. 30	2 vs. 10	/	/

of queries that can be proved by UDP, SPES, and WeTune can also be proved by SQLSolver. Since UDP and WeTune cannot handle queries derived from TPC-C and TPC-H, we do not compare their verification latencies with SQLSolver on these queries. We also do not compare the verification latencies of SPES and SQLSolver on query pairs derived from TPC-H because SPES cannot prove any of these query pairs. As shown in Table 6, WeTune exhibits the fastest verification speed owing to its distinct verification approach from other existing provers. It targets the verification of rewrite rules rather than concrete SQL queries. Thus, to prove the equivalence between two concrete queries, WeTune rewrites each of them using a set of rules it discovers and checks whether the two queries can be rewritten to the same query. Thus, WeTune achieves the fastest verification speed by not invoking SMT solvers for verification. In contrast, SQLSolver employs SMT solvers several times, resulting in a comparatively longer verification time than WeTune and SPES. Despite this, SQLSolver stands out by proving more queries than other provers, including queries with complicated syntax structures. Furthermore, the algorithm of SQLSolver supports more SQL features, enabling it to prove the equivalence of more query pairs. Note that verification latency does not affect query execution latency because verification is performed offline.

6.3 Discovery of Useful Rewrite Rules

To answer the third question, we integrate SQLSolver into the WeTune framework's rewrite rule discovery module to discover additional useful rules. Our integration of SQLSolver reveals all 35 useful rules previously found by WeTune. Furthermore, SQLSolver also identifies novel rules intended to eliminate unnecessary aggregate functions and UNION operators. For quantitative analysis, we evaluate the performance benefit of these discovered rules with a few manually crafted queries. On these generated queries, the new rules induce a latency reduction of up to 99.70% compared to queries without rewrite. The decreased latency can be attributed to removing certain relational operations, demonstrating the usefulness of the newly discovered rules.

7 RELATED WORK

Formalization of semantics for SQL queries. There are several different methods to formalize SQL semantics in recent works. UDP [9] defines an algebra called U-expression to model SQL queries under bag semantics as a function, which returns the multiplicity of a tuple in the query result. WeTune extends U-expressions in UDP to represent NULL. Compared with U-expression, the algebra representation defined in SPES [47] can encode concrete predicates. However, it cannot encode integrity constraints. Inspired by UDP and WeTune, SQLSolver proposes new methods to encode more SQL features.

Automated verification for equivalence between SQL queries. Proving the equivalence of queries is an important problem [6, 7, 13, 14, 23, 25, 40]. Recent works have proposed multiple methods to prove the equivalence between queries automatically. Some recent work [46] targets set semantics, while our work targets bag semantics. The other work supporting bag semantics can be classified into two classes. The first type of work[8, 9, 11, 47] translates SQL queries into

algebra representations, rewrites the algebra via manually crafted rules, and compares them based on syntax structure. The second type of work [43] translates the equivalence problem of two SQL queries into a FOL formula and uses SMT solvers to perform the verification, which captures the concrete semantics of SQL queries. The verification capabilities of these works are limited by their manually crafted rules.

Decision procedure for LIA* formulas. SQLSolver is inspired and based on existing works about LIA* [28–30, 37, 39]. Piskac et al. [39] proves that the satisfiability problem of a LIA* formula is NP-complete and gives an algorithm to reduce it to the satisfiability problem of a LIA formula. Recent work [30] further designs a more efficient decision procedure for a LIA* formula based on approximation, which SQLSolver adopts to solve the generated LIA* formula.

Query optimization. There is a long line of work targeting the optimization of queries [3, 16, 19, 20, 26, 27, 31–34, 42, 48]. SQLSolver is orthogonal to these works. We will explore how to combine SQLSolver with them to further optimize the performance of queries, which will be an interesting topic in the future.

8 CONCLUSION

This paper presents SQLSolver, which is a new prover for SQL equivalence. The evaluation shows that SQLSolver can prove more equivalent pairs of queries than existing provers and help discover more useful rewrite rules.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable comments. We appreciate Zhuoran Wei for developing some of the code in SQLSolver. This work is supported by the National Natural Science Foundation of China (No. 62272304 and 62132014) and the Fundamental Research Funds for the Central Universities. Jinyang Li and Ruzica Piskac are partially supported by NSF grant FMitF-2220407. Ruzica Piskac is also partially supported by NSF grant CCF-2131476.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.
[2] amcintosh. 2013. Query featuring outer joins behaves differently in oracle 12c. http://stackoverflow.com/questions/19686262.
[3] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662
[4] Aaron R Bradley and Zohar Manna. 2007. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media.
[5] Apache Calcite. 2021. Calcite Test Suite. https://ipads.se.sjtu.edu.cn:1312/opensource/wetune/-/blob/main/wtune_data/calcite/calcite_tests.
[6] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 77–90. https://doi.org/10.1145/800105.803397
[7] Surajit Chaudhuri and Moshe Y Vardi. 1993. Optimization of real conjunctive queries. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 59–70.
[8] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. 2017. Demonstration of the cosette automated sql prover. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1591–1594.
[9] 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 (jul 2018), 1482–1495. https://doi.org/10.14778/3236187.3236200
[10] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. UDP source code. https://github.com/uwdb/Cosette/tree/master/uexp.

[11] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research* (Chaminade, California, USA) *(CIDR '17)*.

[12] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. *SIGPLAN Not.* 52, 6 (June 2017), 510–524. https://doi.org/10.1145/3140587.3062348

[13] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2007. Deciding equivalences among conjunctive aggregate queries. *Journal of the ACM (JACM)* 54, 2 (2007), 5–es.

[14] Sara Cohen, Werner Nutt, and Alexander Serebrenik. 1999. Rewriting aggregate queries using views. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 155–166.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS '08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[16] Visweswara Sai Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. to appear. SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns. (to appear).

[17] The Apache Software Foundation. 2023. Spark SQL. https://github.com/apache/spark/tree/master/sql.

[18] Richard A Ganski and Harry KT Wong. 1987. Optimization of nested SQL queries revisited. *ACM SIGMOD Record* 16, 3 (1987), 23–33.

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

[20] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE, 209–218.

[21] Carnegie Mellon Database Research Group. 2023. Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase/tree/main/src/main/java/com/oltpbenchmark/benchmarks/tpcc.

[22] Carnegie Mellon Database Research Group. 2023. Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase/tree/main/src/main/java/com/oltpbenchmark/benchmarks/tpch.

[23] Yannis E. Ioannidis and Raghu Ramakrishnan. 1995. Containment of Conjunctive Queries: Beyond Relations as Sets. *ACM Trans. Database Syst.* 20, 3 (sep 1995), 288–324. https://doi.org/10.1145/211414.211419

[24] ISO. 2016. ISO/IEC 9075-2:2016. https://www.iso.org/standard/63556.html.

[25] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. 2006. The Containment Problem for <bi>Real</bi> Conjunctive Queries with Inequalities. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Chicago, IL, USA) *(PODS '06)*. Association for Computing Machinery, New York, NY, USA, 80–89. https://doi.org/10.1145/1142351.1142363

[26] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research* (Asilomar, California, USA) *(CIDR '19)*.

[27] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[28] Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. 2005. An algorithm for deciding BAPA: Boolean algebra with Presburger arithmetic. In *International Conference on Automated Deduction*. Springer, 260–277.

[29] Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. 2006. Deciding Boolean algebra with Presburger arithmetic. *Journal of Automated Reasoning* 36, 3 (2006), 213–239.

[30] Maxwell Levatich, Nikolaj Bjørner, Ruzica Piskac, and Sharon Shoham. 2020. Solving LIA* Using Approximations. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 360–378.

[31] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[32] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.

[33] Ryan Marcus and Olga Papaemmanouil. 2019. Towards a Hands-Free Query Optimizer through Deep Learning. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research* (Asilomar, California, USA) *(CIDR '19)*.

[34] M. Muralikrishna. 1992. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 91–102.

[35] Terence Parr. 2020. ANTLR v4. https://github.com/antlr/antlr4.

[36] Ruzica Piskac. 2011. Decision Procedures for Program Synthesis and Verification. (2011), 200. https://doi.org/10.5075/epfl-thesis-5220

[37] Ruzica Piskac and Viktor Kuncak. 2008. Decision procedures for multisets with cardinality constraints. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 218–232.

[38] Ruzica Piskac and Viktor Kuncak. 2008. Decision Procedures for Multisets with Cardinality Constraints *(VMCAI'08)*. Springer-Verlag, Berlin, Heidelberg, 218–232.
[39] Ruzica Piskac and Viktor Kuncak. 2008. Linear Arithmetic with Stars. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 268–280.
[40] Yehoshua Sagiv and Mihalis Yannakakis. 1980. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)* 27, 4 (1980), 633–655.
[41] David Schmitt. 2010. Optimizer creates strange execution plan leading to wrong results. http://tinyurl.com/hwwn53r.
[42] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 19–28.
[43] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 94–107. https://doi.org/10.1145/3514221.3526125
[44] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune source code. https://ipads.se.sjtu.edu.cn:1312/opensource/wetune.
[45] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2020. SPES source code. https://github.com/georgia-tech-db/spes.
[46] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability modulo Theories. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1276–1288. https://doi.org/10.14778/3342263.3342267
[47] 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)*. 2735–2748. https://doi.org/10.1109/ICDE53745.2022.00250
[48] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System Using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (sep 2021), 46–58. https://doi.org/10.14778/3485450.3485456

Received April 2023; accepted June 2023