Towards Practical SQL Equivalence Reasoning

Shuxian Wang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-188
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-188.html

May 22, 2023

Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Towards Practical SQL Equivalence Reasoning

by Shuxian Wang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Alvin Cheung
Research Advisor

May 10, 2023

(Date)

* * * * * * *

Professor Sanjit A. Seshia
Second Reader

5/21/23

(Date)

Towards Practical SQL Equivalence Reasoning

Shuxian Wang
University of California, Berkeley
Berkeley, California, USA
wsx@berkeley.edu

ABSTRACT

The notion of query equivalence is of great significance in the theory and practise of database systems. Prior work in automated query equivalence checking under bag semantics set the first steps in formally modeling and reasoning about query optimization rules. However, many SQL features such as integrity constraints and NULL semantics are avoided or handled in ad hoc ways in prior attempts, resulting in many unsupported use cases in real world usages. Here we present a new framework for query equivalence checking, based on the U-semiring semantics, that leverage SMT solvers as the reasoning engine with improved modeling of SQL features. Empirically, our implementation can verify 235 out of the 415 query pairs extracted from the latest test suite of the Calcite data management framework, which is 2 times over the state-of-the-art tool (104/415). On the theoretical side, we show that our formalism admits a large query fragment in which query equivalence is decidable relative to an SMT solver, generalizing prior decidability results further.

1 INTRODUCTION

The idea of query equivalence is being widely used in modern database, where the use of semantic-equivalent query rewriting is essential to the correctness and performance of the query execution engine. There are many previous attempts to automatically decide query equivalence albeit its nature of undecidability, essentially forming two lines of work. The first is started by Chu et al. in the series of work [4] and [3], and axiomatized as the theory of U-semiring in [2], where the bag semantics of SQL queries is captured by a symbolic arithmetic expression representing the multiplicities of elements in a bag. The check of semantic equivalence is then performed on the formal expressions of multiplicity without any further knowledge of SQL, and producing a machine-checkable proof of equivalence in the end. The other line of work by Zhou et al. in [7] and [8] reason directly on the query expressions, by first repetitively applying a few selected query rewrite rules to put SQL queries into a normal form, and then us an SMT solver to check the equivalence of queries in normal form.

On one hand, we find the use of SMT solvers enables more powerful semantics reasoning compared to the syntactic approaches used in the first line of work. On the other hand, many SQL features are modelled as seemingly ad hoc and incomplete set of query rewrite rules that are hard to compose in the second approach, while the same can be expressed when lowered to the theory of U-semiring as a relatively minimal group of equational axioms and achieve higher fidelity. Moreover, we notice that more subtle aspects of SQL queries, such as the integrity constraints and NULL semantics, remains unsupported or supported in ad hoc ways by both prior works, making them not capable of showing equivalence when such features are used in a rewrite rules.

Contributions. We believe a more complete account for SQL features is essential for query equivalence checker to scale to more real world cases, and we propose a new approach based on the theory of U-semiring [2] and leverage SMT solvers as oracles to gain richer expressively when modeling SQL features, combining the separated approaches (U-semiring and SMT solvers) in prior arts for the first time. This has the following implications:

(1) The solver can handle a wider range of SQL queries, since we can now encode NULL semantics, have a better formulation of integrity constraints, and allow queries to contain uninterpreted symbols. We use our implementation to prove the validity of many query rewrite pairs used in modern database engine, doubling the number of cases proved compared to the previous state-of-the-art tool in the test set extracted from the Calcite data management framework.
(2) Additionally, we can identify a fragment of SQL queries relative to some first-order theory, demonstrate a decision procedure, and prove its soundness and completeness over the specified fragment. This results generalize over prior decidability results of query equivalence with bag semantics.

Example. To give a better motivation, we discuss a concrete example in more detail. Consider the table $R(k, r)$ with two columns, where k is a primary key, and r is a foreign key that self-reference to the column k of R. A valid rewrite to prove is that performing an left outer join on the foreign key is equivalent to performing an inner join instead. In SQL syntax, we have the queries

```
R AS R1 INNER JOIN R AS R2 ON R1.r = R2.k;
R AS R1 LEFT OUTER JOIN R AS R2 ON R1.r = R2.k;
```

Do note that this query pair uses features such as primary key, foreign key, and NULL semantics together. The current state-of-the-art, SPES [8], handles integrity constraints by ad hoc preprocessing rules on the query syntax, and the above case is not handled as discussed in their work. For UDP [2] the current best equivalence prover that is also based on the U-semiring semantics, encodes foreign key as a rewrite rule. But the rewrite rule they proposed fails to terminate for the self-referencing situation we have here.

Our approach, which will be explained in details later, can prove this rewrite just fine. First, the queries are interpreted into the U-semiring semantics as

$$Q_1(a,b,c,d) = R(a,b) \times R(c,d) \times [b=c],$$
$$Q_2(a,b,c,d) = R(a,b) \times R(c,d) \times [b=c] + R(a,b) \times [c = \text{NULL} \land d = \text{NULL} \land \neg \| \textstyle\sum_{c',d'} [b=c'] \times R(c',d') \|].$$

The key here is to show the second term in Q_2 is empty, which we focus on for now. With the semantics we provide for primary key (8) and further normalization, the second term becomes

$$[\|R'(a)\| \land b = f_R(a) \land c = \text{NULL} \land d = \text{NULL} \land \neg \|R'(b)\|].$$

Now using the semantics of foreign key (12), we may assert

$$\forall k.\ \|R'(k)\| \to \|R'(f_R(k))\|, \tag{1}$$

to an SMT solver. The solver will then recognize the term in (1) is trivially empty since $\|R'(a)\| \wedge b = f_R(a) \wedge \neg\|R'(b)\|$ contradicts the above. We highlight using this example that prior attempts to model integrity constraints are limited and ad hoc, while our formalism for various SQL features are composable and applicable, making it scalable to more complicated cases.

Outline of the paper. We start in Section 2 to discuss the formal semantics we defined for SQL queries, as well as the semantics of additional integrity constraints and NULL values. Then in Section 3 we present a general algorithm to check for query equivalence within the given formal semantics. This algorithm, albeit incomplete, covers all of the SQL features we covered in Section 2, and are shown to be very effective in practise. In Section 4, we show that our semantics admits a fragment of SQL queries in which semantic equivalence can be fully decidable relative to some SMT solver, by constructing another specialized equivalence checking algorithm, which is then shown to be sound and complete. We then evaluate our first algorithm in Section 5 by running it over query optimization pairs present in real world databases, and compare the result with similar tools. Finally, we make a more detailed comparison with other related work and discuss future improvements in the remaining sections.

2 SEMANTICS

Definition 1. In general, we allow SQL queries to reference external table and undetermined variables. We use $\Gamma \mid \Delta \mid \Phi \vdash Q$ to denote the query Q referencing table variables from the context Γ and uninterpreted symbols from the context Δ, with the symbols in Δ constrained by a logical formula Φ. Using Query$S$ to denote a query of schema S, we give the syntax inductively using the following rules (omitting repetitive ambient context annotations).

$$\frac{P : S \to \mathsf{Bool} \quad Q : \mathsf{Query}\ S}{\mathsf{Filter}(P,Q) : \mathsf{Query}\ S} \qquad \frac{f : S \to T \quad Q : \mathsf{Query}\ S}{\mathsf{Proj}(f,Q) : \mathsf{Query}\ T}$$
$$\frac{Q_1 : \mathsf{Query}\ S_1 \quad x : S \vdash Q_2 : \mathsf{Query}\ S_2}{\mathsf{CorrJoin}(Q_1,Q_2) : \mathsf{Query}\ (S_1 \times S_2)}$$
$$\frac{Q_1 : \mathsf{Query}\ S \quad Q_2 : \mathsf{Query}\ S}{\mathsf{Union}(Q_1,Q_2) : \mathsf{Query}\ S} \qquad \frac{Q : \mathsf{Query}\ S}{\mathsf{Distinct}(Q) : \mathsf{Query}\ S}$$
$$\frac{v_1 : S \quad \cdots \quad v_n : S}{\mathsf{Values}(v_1,\dots,v_n) : \mathsf{Query}\ S} \tag{2}$$

The use of constraint Φ might be apparent in later sections. The notation here uses a multi-segment context of the form $\Gamma \mid \Delta \mid \Phi$ to annotate the query expression, keeping track of the variables used and additional assumptions made about them. The turnstile $\vdash$ annotates an expression on the right by the context on the left, but for brevity, we often partially or fully omit context annotations when they can be inferred.

The semantics we choose to interpret the SQL query expressions mostly follows the U-semiring formalism [2]. To recap, the intuition is to capture the well-received bag semantics of SQL query, where a table of schema S is a multiset of rows of type S. Such multiset can be instead perceived as a function $S \to U$ taking in some $s \in S$ and return the multiplicity of s in the multiset, where U the type of U-semiring is intend to serve as the domain of multiplicity. Additional operations on U, namely $+$, $\times$, $\sum$, and $\|\cdot\|$, are axiomatized with equational laws as in [2]. This foundation allows us to interpret SQL queries formally as follows.

Definition 2. The semantics of a query $\Gamma \mid \Delta \mid \Phi \vdash Q$, denoted as $\Gamma \mid \Delta \mid \Phi \vdash [\![Q]\!]$, is given inductively as follows :

$$\begin{aligned}
[\![R \vdash R]\!] &= R \vdash \lambda x.\ R(x) \\
[\![\mathsf{Filter}(P,Q)]\!] &= \lambda x.\ [P(x)] \times [\![Q]\!](x) \\
[\![\mathsf{Proj}(f,Q)]\!] &= \lambda x.\ \sum_s [x = f(s)] \times [\![Q]\!](s) \\
[\![\mathsf{CorrJoin}(Q_1,Q_2)]\!] &= \lambda x_1, x_2.\ [\![Q_1]\!](x_1) \times [\![x \vdash Q_2]\!][x_1/x](x_2) \\
[\![\mathsf{Union}(Q_1,Q_2)]\!] &= \lambda x.\ [\![Q_1]\!](x) + [\![Q_2]\!](x) \\
[\![\mathsf{Distinct}(Q)]\!] &= \lambda x.\ \|[\![Q]\!](x)\| \\
[\![\mathsf{Values}(v_1,\dots,v_n)]\!] &= \lambda x.\ [x = v_1] + \cdots + [x = v_n]
\end{aligned} \tag{3}$$

Moreover, we give the semantics of the Exists predicate which takes in a sub-query as input

$$[\![\mathsf{Exists}(Q)]\!] = \left\|\sum_s [\![Q]\!](s)\right\|. \tag{4}$$

2.1 The domain of multiplicity

It is very tempting to use the usual set of natural numbers $\mathbb{N}$ as the domain of multiplicity, instead of the axiomatized theory of U-semiring, and we discuss some of the subtleties there. The main reason that we choose the present the formal semantics in expressions of U-semiring, or U-expressions for short, is that we use the unbounded summation $\sum$ to characterize the projection operator as in Eq. (3) and existence predicate as in Eq. (4). The unbounded summation *a priori* does not ensure producing a finite result, hence interpreting it directly as an operation on natural number is ill-defined. As pointed out by Chu et al. [2], the natural number can be viewed as a model of U-semiring only if all summations are of finite support. That is, for $\sum_s U(s)$, there are only finitely many s such that $U(s) \neq 0$.

However, taking a pure U-expression approach also has its own challenge. As we will show later, it is convenient to delegate some multiplicity equality check to an SMT solver, given that most have substantial support for integer arithmetics. But to express equality of U-expressions, one has to define the theory of U-semiring within the SMT solver, and it is not clear how to even define the summation operator $\sum_s U(s)$ within a first-order theory. In fact, a core contribution of our work is reducing the equality between U-expressions down to some SMT formulas, without directly defining the notion of unbounded summation.

Another issue is that U-semiring can be overly general to serve as the domain of multiplicity for real-world databases and queries. As pointed out by Chu et al. [2], there are many models for U-semiring, including the set of natural number extended with an infinity element $\bar{\mathbb{N}} \coloneqq \mathbb{N} \cup \{\infty\}$, univalent types, cardinal numbers, etc. This allows us to make unrealistic constructions such as a table that contains all integers, and each integer occurs infinitely many times.

But database developers only care about whether a query rewrite rule preserve its result for all finite instantiation of databases.

The perspective that we take to balance between practicality and formal rigor is as follows. We pick the actual domain of multiplicity to be the extended natural number $\bar{\mathbb{N}}$, which is a model of the theory of U-semiring, and it allows us to perform rewrites based on the equational laws of U-expressions without worrying about the finiteness of multiplicity along the way. But for any table variable R occurred within the U-expressions, we restrict them to be an arbitrary table where each distinct row may only occur finitely many time, and thus essentially regarding R (of schema S) as a function $S \to \mathbb{N}$. This restriction, justified by the finiteness of real world database, allows us to check equalities of U-expressions that only contains table variables and the $+$ and $\times$ operator, by translating the U-expressions into the obvious arithmetic expressions and invoking an SMT solver to check their equality.

An observation that would be useful later is that U-expressions under the squash operator $\|\cdot\|$ can be equivalently written as a logical formula without the squash operator, using the rules

$$\begin{aligned} \|U_1 + U_2\| &\rightsquigarrow \|U_1\| \vee \|U_2\|, & \|\|U\|\| &\rightsquigarrow \|U\|, \\ \|U_1 \times U_2\| &\rightsquigarrow \|U_1\| \wedge \|U_2\|, & \|[P]\| &\rightsquigarrow P, \\ \Big\|\sum_s U\Big\| &\rightsquigarrow \exists s.\ \|s \vdash U\|, & \|R \vdash R(a)\| &\rightsquigarrow R \vdash R(a) \neq 0. \end{aligned} \tag{5}$$

And we hence adopt the practise of regarding squashed U-expressions as boolean expressions under the appropriate context.

2.2 Integrity constraints

In addition to the expression of the query one is executing, SQL allows user to define additional integrity constrains on the tables involved in the query, which also bear semantics significance that we wish to capture. We focus on three main types of constraints that are definable by SQL users in the CREATE TABLE command, namely the primary key (uniqueness) constraint, the foreign key constraint, and the user-defined check constraint.

Since all such constraints are declared in table schema definition and not reflected directly in the syntax of the query expression, we handle all the integrity constraints in a post-processing step after translating SQL queries into U-expressions. Specifically, all integrity constraints will be captured by rewrite rules on U-expressions of the form

$$\Gamma \mid \Delta \mid \Phi \vdash U \quad \rightsquigarrow \quad \Gamma' \mid \Delta' \mid \Phi' \vdash U'.$$

Do notice that the rules may rewrite the U-expression as well as its surrounding context, and we will provide justifications in the following parts.

2.2.1 Primary key constraint. Suppose we have a table R with the schema $K \times S$, where a column of type K being the primary key of R. Intuitively, this means for any certain $k \in K$, there are at most one $s \in S$ such that the tuple (k, s) occurs in R, and we additionally require (k, s) to occur exactly once. In the language of U-semiring, an attempting way to capture this notion is to require for all $k \in K$, $\sum_{s \in S} R(k, s) \leq 1$. However, we do not allow the use of inequality in the U-semiring formalism to avoid extra complexity. Another attempt is the following equivalent formalism

$$\forall k \in K.\ \sum_{s \in S} R(k, s) = \Big\|\sum_{s \in S} R(k, s)\Big\|, \tag{6}$$

which is fully expressible with the U-semiring operators.

The above characterization of primary key, while correct, impose the question of how can it be properly incorporated into the pipeline of the solver, as an extra axiom on the table R. One may suggest to encode Eq. (6) into a first-order logic formula and assert the rule in an SMT solver to further offload the work, but it is not clear how can one even define the summation operator $\sum$, which is inherently second-order and occur in both sides of Eq. (6), in a first-order formula. Another approach is to use the equation as a rewrite rule during normalisation. But neither the left nor right side of Eq. (6) is an easy pattern to match on, making the rule hard to integrate in a rewriting pipeline.

The observation we make here is that for any table R of schema $K \times S$ with the primary key on K, there exists a function $f : K \to S$ that represents the functional dependency of the non-key columns over the key columns, over the subset of K that is contained in R. Conversely, any subset of K combined with a function $f : K \to S$ induces a table R with primary key on K. In fact, for any logical predicate P over a table, we have that

$$\begin{aligned} &\forall R \text{ with primary key. } P(R) \\ &\iff \forall R', f.\ P(\lambda k, s.\ \|R'(k)\| \times [s = f(k)]). \end{aligned} \tag{7}$$

This gives justifications to the following rule that for any table symbol R in the context with a primary key constraint, we may change the context and apply the rewrite

$$R \vdash R \quad \rightsquigarrow \quad R' \mid f_R \vdash \lambda k, s.\ \|R'(k)\| \times [s = f_R(k)] \tag{8}$$

2.2.2 Foreign key constraint. We may now proceed to discuss the modeling of foreign key constraint, with the generic example of R being a table of schema $K \times A$ with primary key on K, and S being a table of schema $B \times K$ with a foreign key with K referencing R. Intuitively, this means that for any (b, k) in S, there must exists exactly one $a \in A$ such that (k, a) occurs exactly once in R, which is captured by the following equality.

$$\forall b, k.\ S(b, k) = S(b, k) \times \sum_{a \in A} R(k, a) = S(b, k) \times \|R'(k)\|. \tag{9}$$

This works well as a rewrite rule, but if R additionally has a foreign key constraint referring to other tables (or even back to S), the rule may need to be applied repetitively, with the risk of non-termination.

One way to resolve the non-termination of rewrites caused by cyclical foreign keys is to encode the foreign key constraint as a single global logical formula, and hence avoiding modifications of U-expressions. Concretely, we first notice that the above intuition can be precisely translated as

$$\forall b, k.\ \|S(b, k)\| \to \sum_{a \in A} R(k, a) = 1. \tag{10}$$

A minor technical point of Eq. (10) is that it is a logical formula that contains U-expressions not under the squash operator $\|\cdot\|$, which makes it hard to assert as a logical formula expressible in the framework of SMT solvers. But we may overcome this issue by further

simplifying Eq. (10) using the fact that R has primary key, and obtain the rewrite rule

$$S \vdash S \quad \rightsquigarrow \quad S, R' \mid \forall b, k.\ \|S(b,k)\| \to \|R'(k)\| \vdash S. \tag{11}$$

Notice that as a derived rule, when S has both the primary key constraint and a foreign key constraint, we should apply both Eq. (8) and (11) and obtain

$$S \vdash S \quad \rightsquigarrow \quad S', R' \mid \forall b, k.\ \|S'(b)\| \to \|R'(f_S(b))\| \vdash S. \tag{12}$$

2.2.3 Check constraint. Finally, SQL allows user to specify custom constraints with the CHECK command on the columns of a specific tables. Suppose on a table R of schema S, we have a check constraint C which should be a logical predicate over S. The check constraint ensures all rows in R satisfy C, namely

$$\forall s.\ \|R(s)\| \to C(s).$$

And this gives the rewrite rule

$$S \vdash S \quad \rightsquigarrow \quad S \mid \forall s.\ \|R(s)\| \to C(s) \vdash S. \tag{13}$$

2.3 NULL semantics

Up till now we have yet covered how NULL values are handled, which we proceed now, but do note that all the formalisms above require no changes in the presence of NULL values. Although NULL values and their interactions with all the other SQL features are subtle and complicated to model, the flexibility and generality of our formal setup enable us to precisely express all those complications on a unified foundation.

First, since NULL values are allowed to occur under any typing context in SQL, we lift every type T to its option type $\mathsf{Option}(T)$ in the semantics, effectively introducing a distinguished NULL value in each elementary type. Such translation is supported by modern SMT solvers as the option types can be defined as custom algebraic data types. We also notice that as a general rule, for an n-ary operation f that is well defined on non-NULL values, the behaviour of $f(a_1, \dots, a_n)$ is extended to return NULL exactly when some a_i is NULL. Hence we can systematically lift many operations such as $+$, $\times$, etc. that occurs in the surface syntax into the corresponding operations that also operates on nullable/optional values.

The above rule, however, has a few exceptions that we need to handle as special cases. For the boolean operations $\wedge$ and $\vee$, we have the definition as in three-value logic:

$$\begin{aligned} a \wedge b &:= \mathsf{ite}(a = \textsc{null}, \mathsf{ite}(b = \mathsf{True}, \textsc{null}, b), \mathsf{ite}(a = \mathsf{True}, b, a)),\\ a \vee b &:= \mathsf{ite}(a = \textsc{null}, \mathsf{ite}(b = \mathsf{False}, \textsc{null}, b), \mathsf{ite}(a = \mathsf{False}, b, a)), \end{aligned} \tag{14}$$

where ite denotes the if-then-else construct. Another special case is the element membership predicate, namely $\mathsf{In}(a, R)$ that checks if the row a is contained in table R. Complications arise when a itself is NULL or when R contains NULL values, and we use the following definition.

$$\begin{aligned} &\mathsf{In}(a, R) := \\ &\mathsf{ite}(a = \textsc{null}, \textsc{null}, \mathsf{ite}(\|R(a)\|, \mathsf{True}, \mathsf{ite}(\|R(\textsc{null})\|, \textsc{null}, \mathsf{False}))). \end{aligned} \tag{15}$$

This concludes the additional notions we introduce for modelling the semantics of NULL, and the other important NULL-aware operation such as the different flavors of outer join can be defined using a combinations of NULL, Union, and Join, as a derived notion.

3 EQUIVALENCE CHECKING

Now we can present an equivalence checking algorithm based on the formalism, namely U-expressions, we discussed in earlier sections. As hinted earlier, the algorithm is mainly two-staged, by first normalizing arbitrarily formed U-expressions into a normal form, and then trying to unify the two normal forms in a unification algorithm. We will be using an SMT solver along the way, idealized as an oracle $\mathcal{O}$ with the power to decide the satisfiability problem of the logical theories in interest. We use the notation

$$\Gamma \mid \Delta \mid \Phi \vdash_{\mathcal{O}} P \tag{16}$$

to denote checking the validity of the first-order logical formula P with the oracle $\mathcal{O}$ under the context $\Gamma \mid \Delta \mid \Phi$, which is equivalent to checking the satisfiability of $\Phi \nrightarrow P$ with uninterpreted symbols from Γ and Δ.

3.1 Normalization

All U-expressions can be organized into the following normal form, and we demonstrate the procedure in Algorithm 1.

Definition 3. An U-expression U under some context $\Gamma \mid \Delta \mid \Phi$ is in sum–product normal form (SPNF) when it is the sum of n sum–product normal terms

$$\Gamma \mid \Delta \mid \Phi \vdash U = T_1 + \cdots + T_n, \tag{17}$$

where a sum–product normal term T is in the form of some m nested summations

$$\Gamma \mid \Delta \mid \Phi \vdash T = \sum_{s_1, \dots, s_m} \|L\| \times V. \tag{18}$$

Here L is any U-expression, and U is the product of some k applications of some table variable with some expression $R_i(e_i)$,

$$\Gamma \mid \Delta, s_1, \dots, s_m \mid \Phi \vdash V = R_1(e_1) \times \cdots \times R_k(e_k). \tag{19}$$

Since an U-expression in SPNF is always the finite sum of normal terms, we use the convention of regarding SPNF as lists of normal terms in the presentation of algorithms.

Algorithm 1 Obtaining sum–product normal form

-- *Given a U-expression* $\Gamma \mid \Delta \mid \Phi \vdash U$,
-- *return an equivalent U-expression in SPNF.*
$\mathsf{spnf} : \mathsf{UExpr} \to [\mathsf{SPNTerm}]$
$\mathsf{spnf}\ (R \vdash R(a)) = R \vdash R(a)$
$\mathsf{spnf}\ (U_1 + U_2) = \mathsf{spnf}\ U_1 \mathbin{+\!\!+} \mathsf{spnf}\ U_2$
$\mathsf{spnf}\ (U_1 \times U_2) = [\mathsf{mult}\ T_1\ T_2 \mid T_1 \leftarrow \mathsf{spnf}\ U_1, T_2 \leftarrow \mathsf{spnf}\ U_2]$ **where**
  $\mathsf{mult}\ (\sum_{\overline{s}} \|L\| \times V)\ (\sum_{\overline{s'}} \|L'\| \times V') = \sum_{\overline{s}, \overline{s'}} \|L \times L'\| \times V \times V'$
$\mathsf{spnf}\ (\sum_s U) = [\sum_s T \mid T \leftarrow \mathsf{spnf}\ (s \vdash U)]$
$\mathsf{spnf}\ \|U\| = \mathbf{let}\ U' = \mathsf{spnf}\ U\ \mathbf{in}\ \|U'\|$
$\mathsf{spnf}\ [P] = [P]$

Applying spnf does encompass many equivalent U-expressions into one, but an important class of rewrites are not yet covered by spnf. Consider the rule of composing two consecutive projections

into one, which may result in the following pair of U-expressions when normalized.

$$R \mid x \vdash \sum_y \sum_z \|[x = f(y)] \times [y = g(z)]\| \times R(z),$$
$$R \mid x \vdash \sum_z \|[x = f(g(z))]\| \times R(z). \tag{20}$$

The key here is to observe the summation variable y can be expressed in terms of other variables as $g(z)$ and thus the summation over y is unnecessary. Our solution here is to introduce an extra step to eliminate as much redundant summation variable as possible to the point where every remaining summation variable does not have functional dependency over any other variables. We call such process the stabilization of SPNF and is demonstrated in Algorithm 2.

Algorithm 2 Stabilize SPNF

−− Stabilize a U–expression $\Gamma \mid \Delta \mid \Phi \vdash U$.
stable : [SPNTerm] ⟶ [SPNTerm]
stable U = **map** stableTerm U
stableTerm : SPNTerm ⟶ SPNTerm
stableTerm $(\sum_{\bar{s}} \|L\| \times U)$ = **case mapMaybe** (depend $\|L\|$ $\bar{s}$) $\bar{s}$ **of**
  $[] \Longrightarrow \sum_{\bar{s}} \|L\| \times U$
  $(s, \overline{s'}, e) :: _ \Longrightarrow$ stableTerm $(\sum_{\overline{s'}} \|L\|[e/s] \times U[e/s])$ **where**
    depend $\|L\|$ $\bar{s}$ s_i = **let** $\overline{s'} = \bar{s} \setminus s_i$ **in**
−− Check if exists some functional dependency between $\overline{s'}$ *and* s.
      if $\exists(\Delta \vdash f).\ \Gamma \mid \Delta \mid \Phi \vdash_{\mathcal{O}} \|L\| \rightarrow f(\overline{s'}) = s$
      then Just $(s, \overline{s'}, f(\overline{s'}))$ **else Nothing**

An important implementation note is that when searching for possible functional dependency between $\overline{s'}$ and s, we need to perform the task of finding a function f under the context Δ, such that we have $\Gamma \mid \Delta \mid \Phi \vdash_{\mathcal{O}} \|L\| \rightarrow f(\overline{s'}) = s$. Such task is not possible to perform using SMT solvers in general, and there are two implementation strategies we can take. First is to use synthesis tools such as Syntax-Guided Synthesis (SyGuS) solvers to directly synthesis the required function f. Another strategy is to first obtain the congruence group information of all the expressions involved in the term, using calls such as `Z3_get_implied_equalities` exposed by the Z3 solver. And then we can analyze the congruence classes to find expressions that are equal to s and get a suitable f.

The second strategy theoretically can miss some cases if the solution is not present as an expression already in the term, but we find it equally powerful in practise and choose it over the first to eliminate the extra dependency on a SyGuS solver.

3.2 Unification

Once we normalize and stabilize the U-expressions to obtain them in SPNF, checking equivalence can be relatively easy. In Algorithm 3, we accommodate for the commutativity of $+$ and $\sum$ before comparing the body of the terms by checking it with an SMT solver.

Algorithm 3 Unifying SPNFs

−− Unify two SPNFs under the same context $\Gamma \mid \Delta \mid \Phi \vdash U_1, U_2$.
unify : [SPNTerm] → [SPNTerm] → **Bool**
unify U_1 U_2 = bagEq termEq (prune U_1) (prune U_2) **where**
  prune = **filter** $(\lambda\ (\sum_{\bar{s}} \|L\| \times U) \Rightarrow \Gamma \mid \Delta, \bar{s} \mid \Phi \nvdash_{\mathcal{O}} \neg\|L\|)$
  termEq $(\sum_{\bar{s}} \|L\| \times U)$ $(\sum_{\overline{s'}} \|L'\| \times U')$ =
    any $(\lambda\ \sigma \Longrightarrow \Gamma \mid \Delta, \bar{s} \mid \Phi \vdash_{\mathcal{O}} \|L\| \times U = \|L'\|[\sigma] \times U'[\sigma])$ (perms $\bar{s}$ $\overline{s'}$)
  −− Enumerates all valid bijective substitutions
  −− between two lists of variables.
  perms : [Var] → [Var] → [(Var, Var)]
−− Compare two lists up to reordering of elements.
bagEq : (A → A → **Bool**) → [A] → [A] → **Bool**
bagEq (~) [] [] = **True**
bagEq (~) [] (_ :: _) = **False**
bagEq (~) (x :: xs) ys = **case** dropEl x ys **of**
  Nothing ⇒ **False**
  Just ys ⇒ bagEq (~) xs ys **where**
    dropEl a [] = **Nothing**
    dropEl a (x :: xs) | a ~ x = **Just** xs
    dropEl a (x :: xs) = fmap (x ::) (dropEl a xs)

4 COMPLETE FRAGMENT

The previous algorithm is designed to cover a large range of SQL features, but it is also incomplete in the sense that the equivalence checking procedure returning false does not imply the queries are not equivalent. And the incompleteness seems unavoidable even in the presence of the perfect oracle $\mathcal{O}$ deciding any first-order satisfiability problem: Since the algorithm takes in arbitrary U-expressions as inputs (not necessarily those that come from the interpretation of queries), for arbitrary first-order predicate P and Q, we can have the problem

$$\sum_s [P(s)] \stackrel{?}{=} \sum_s [Q(s)] \tag{21}$$

that encodes the notion of equicardinality (Hartig's) quantifier, which is known to be strictly more expressive than quantifiers in first-order logic [6]. Therefore, to obtain a complete algorithm, we restrict our scope to a fragment of SQL queries such that their semantics in U-expressions contain extra structures that we can capture and exploit during equivalence checking.

Real-world SQL queries can be an open ended domain to characterize, as endless data types and operations may occur in database schema, filter conditions, projection expressions, etc. It is more accurate to capture a query fragment relative to some (first-order) theory T which restricts the available data types and operations.

Definition 4. For some first-order theory T, we have the query fragment $\mathscr{F}_T$ generated by:

- Ground tables as distinct variables $R_1, R_2, \ldots$, with optionally imposed primary key constraints;
- Raw tables defined by Values$(v_1, \ldots, v_n)$.
- The filter operation Filter(P, Q);
- The projection operation Project(f, Q);
- The correlated join operation CorrJoin(Q_1, Q_2);

- The union operation $\text{Union}(Q_1, Q_2)$;

And all the involved values v_i, projection expressions f, and filter conditions P are definable in the theory T.

Moreover, we have an accompanying oracle $\mathcal{O}_T$ for $\mathcal{F}_T$ when $\mathcal{O}_T$ can decide any satisfiability problem in T. Equivalently, for any context Γ and formula φ in T under the context Γ, we can check if φ holds under all possible instantiation of the context Γ, denoted $\Gamma \vdash_{\mathcal{O}_T} \varphi$, by querying the oracle $\mathcal{O}_T$ whether $\neg\varphi$ is unsatisfiable with uninterpreted symbols from Γ.

The significance of defining the query fragment relative to some theory T is in the same spirit of that in satisfiability modulo theories (SMT). Since our method is generic over the underlying theory T, the user has the freedom to pick and choose the theory T and its solver according to their specific problems in hand. We hereby fix an arbitrary theory T and proceed to demonstrate another algorithm which completely decides the query equivalence problem in $\mathcal{F}$ with the oracle $\mathcal{O}$, and with two technical assumptions:

1. The theory T contains at least the equality logic with uninterpreted function and predicate.
2. For every sort S in the theory T, a total order $<$ on S is definable.

Our algorithm first translates the queries in $\mathcal{F}$ directly into U-expressions of a certain normal form, which are then further normalized to obtain stronger normal forms, and are finally unified to obtain the decision. We will then prove the soundness and completeness of the procedure.

4.1 Normalization

The queries in the fragment $\mathcal{F}$ can be structured, which we wish to capture and exploit during equivalence check. As we demonstrate later, any U-expression $[\![Q]\!]$ obtained from some query $Q \in \mathcal{F}$ can be rewritten into the following normal form.

Definition 5. An U-expression U under some context $\Gamma \mid \Delta$ is in normal form when it is the sum of n normal terms

$$\Gamma \mid \Delta \vdash U = T_1 + \cdots + T_n, \tag{22}$$

where a normal term T is in the form of some m nested summations

$$\Gamma \mid \Delta \vdash T = \sum_{R_1^{k_1}(s_1)} \cdots \sum_{R_m^{k_m}(s_m)} [P], \tag{23}$$

where P, called the body of T and denoted $\text{Bdy}\, T$, is a predicate under the context $\Delta, s_1, \ldots, s_m$, i.e., not containing any relational variables. We use the notation

$$\sum_{R^0(s)} f(s) := \sum_s \|R(s)\| \times f(s), \qquad \sum_{R^k(s)} f(s) := \sum_s \underbrace{R(s) \times \cdots \times R(s)}_{k \text{ times}} \times f(s)$$

to signify every variable introduced by a summation is always applied to some relational variable R for some k number of times, with the special case of having $\|R(s)\|$ when $k = 0$.

Additionally, we may choose to globally fix a certain enumeration of the relational variables in Γ, which gives $\Gamma = R_1, R_2, \ldots$, and require the summations in T introduce variables applied to R_i before those applied to R_j whenever $i < j$.

Since the normal form is highly structured, we define some additional operations and notation on it that would be helpful later.

Definition 6. For a normal term T and a relational variable R that occurred in T, we let $\text{Scp}_R\, T$ be the list of variables introduced in T that is also applied to R. Furthermore, we use $\text{Scp}\, T$ to denote the entire scope of T, namely the list of all variables introduced by the summations in T. In the case of Eq. (23), $\text{Scp}\, T = s_1, \ldots, s_n$. Similarly, we define $\text{Rel}\, T$ to be the list of relational variables (including their power) introduced by the summations in T, and $\text{Rel}'\, T$ to be the list of relational variables without the power. In the case of (23), $\text{Rel}\, T = R_1^{k_1}, \ldots, R_m^{k_m}$ and $\text{Rel}'\, T = R_1, \ldots, R_m$.

Definition 7. For some R and distinct $s, s' \in \text{Scp}_R\, T$, the notation $T[s'/s]$ represents the new normal term based on T but with:

- All occurrences of s in $\text{Bdy}\, T$ substituted with s'.
- The summation introducing s by $R^k(s)$ merged with that introducing s' by $R^l(s')$ to form one summation introducing s' by $R^{k+l}(s')$.

And for a normal term $\Gamma \mid \Delta \vdash T$, the notation $T[P]$ with $\Delta, \text{Scp}\, T \vdash P$ being a predicate, represents the new normal term based on T but with the body changed to P.

Now we show that for all $Q \in \mathcal{F}$, the U-expression $[\![Q]\!]$ can always be rewritten into the normal form. In fact, we proceed by an inductive argument with a strengthened inductive hypothesis: For $\Gamma \mid \Delta, x \vdash [\![Q]\!](x)$, we can write it as the sum of normal terms where each is of form

$$\Gamma \mid \Delta, x \vdash T = \sum_{R_1^{k_1}(s_1)} \cdots \sum_{R_m^{k_m}(s_m)} [x = f(s_1, \ldots, s_m) \wedge P], \tag{24}$$

where P does not contain x. The proof is essentially given by the construction of Algorithm 4.

Algorithm 4 Normalizing queries to normal form

-- Given a query Q producing a result with schema S,
-- return a normal U-expression under context $s \in S \vdash U$
-- that is equivalent to $s \vdash [\![Q]\!](s)$

norm : Query $\rightarrow$ [NTerm]
norm (Table R) = $[R \mid x \vdash \sum_{R(s)} [x = s]]$
norm (Values vals) = **map** $(\lambda v \Rightarrow x \vdash [x = v])$ vals
norm (Filter P Q) = **map** $(\lambda T \Rightarrow T[\text{Bdy}\, T \wedge P])$ (norm Q)
norm (Proj f Q) = **map** projTerm (norm Q) **where**
  projTerm $(y \vdash \sum_{\overline{R^k(s)}} [y = g(\bar{s}) \wedge P]) =$
    $x \vdash \sum_{\overline{R^k(s)}} [x = f(g(\bar{s})) \wedge P]$
norm (CorrJoin Q_1 Q_2) = [**join** T_1 T_2 | $T_1 \leftarrow U_1, T_2 \leftarrow U_2$] **where**
  (U_1, U_2) = (norm Q_1, norm Q_2)
  join $(x \vdash \sum_{\overline{R^k(s)}} [x = f(\bar{s}) \wedge P])$ $(y, x' \vdash \sum_{\overline{R'^{k'}(s')}} [x' = f'(\overline{s'}) \wedge P']) =$
    $x, x' \vdash \sum_{\overline{R^k(s)}, \overline{R'^{k'}(s')}} [(x, x') = (f(\bar{s}), f'[f(\bar{s})/y](\overline{s'})) \wedge P \wedge P'[f(\bar{s})/y]]$
norm (Union Q_1 Q_2) = norm Q_1 ++ norm Q_2

4.1.1 Linearisation. Once we obtain the normal form above, comparing them in a term-by-term manner seems to be straightforward. However, the problem of variable ordering occurs when deciding term equivalence. For example, consider the terms

$$R \mid x \vdash T_1 = \sum_{R(s_1)} \sum_{R(s_2)} [x = s_1], \qquad R \mid x \vdash T_2 = \sum_{R(s_1)} \sum_{R(s_2)} [x = s_2],$$

which are identical if we (rightfully) swap the summation variables s_1 and s_2 in T_2 before unification. But T_1 and T_2 are not the same in the sense of checking

$$R \mid x \vdash \forall s_1, s_2.\ [x = s_1] \leftrightarrow [x = s_2].$$

The issue comes from the fact that variables in $S_R(T)$ can be equivalently introduced in any order, while when comparing normal terms by their body, we enforce a certain ordering of scope by putting both sides under the same universal quantifier.

Another problematic case comes from redundant variables:

$$R \mid x \vdash \sum_{R(s_1)} \sum_{R(s_2)} [(x = s_2) \wedge (s_1 = s_2)], \qquad R \mid x \vdash \sum_{R^2(s_1)} [x = s_1],$$

where in T_1 we can avoid s_2 by removing the scope and substituting all occurrences of s_2 with s_1 to obtain an identical term to T_2. But it is hard to check the equivalence of T_1 and T_2 *as is* due to their mismatching summation scope.

Our solution is to fix variable ordering and eliminate redundancy in normal terms by the process of linearization.

Definition 8. A U-expression U is in linearized normal form (LNF) if it is in normal form and additionally satisfies the property that: for all normal term T in U and distinct relational variable R in T, $\mathrm{Bdy}\,T$ implies the variables $\mathrm{Scp}_R\,T$ are pairwise-distinct and form a linearly ordered chain.

Such property effectively fix the ordering of $\mathrm{Scp}_R\,T$ for any R in T, and since we have already fixed the ordering of distinct relational variables, the entire scope $\mathrm{Scp}\,T$ now have a well defined order of variables.

The procedure to rewrite a normal form U into an equivalent and linearized normal form (LNF) is presented in Algorithm 5. We define $\mathsf{Lin}(X)$ to be the procedure that enumerates all possible relations between the variables in set X using the total order $<$ or equality. Namely for $X = \{a, b, c\}$, we have

$$\begin{aligned} &\mathsf{Lin}(\{a,b,c\}) \\ &= \{a<b<c, a<c<b, b<a<c, b<c<a, c<a<b, c<b<a\} \\ &\cup \{a<b=c, b=c<a, a=b<c, c<a=b, a=c<b, b<a=c\} \\ &\cup \{a=b=c\}. \end{aligned}$$

4.1.2 Partition. The LNF intuitively ensures terms can be equivalent only if they have a matching scope. But we need some more processing of LNF to take into account the fact that some terms can be combined into/split from a single term. For example, consider the following two LNFs,

$$R \mid P, x \vdash \sum_{R(s)} [(x = s) \wedge P(s)] + \sum_{R(s)} [(x = s) \wedge \neg P(s)],$$
$$R \mid P, x \vdash \sum_{R(s)} [x = s].$$

Algorithm 5 Linearising the normal form

$\mathsf{linear} : [\mathsf{NTerm}] \to [\mathsf{LNTerm}]$
$\mathsf{linear}\ U = \textbf{concatMap}\ \mathsf{linExpand}\ U\ \textbf{where}$
$\quad \mathsf{clauses} = \textbf{sequence}\ [\ \mathsf{Lin}(\mathrm{Scp}_R\,T) \mid R \leftarrow \Gamma\ ]$
$\quad \mathsf{linExpand}\ T = \textbf{map}\ (\mathsf{cleanUp} \circ \textbf{foldl} \wedge \top)\ \mathsf{clauses}$
$\quad \mathsf{cleanUp}\ T = \textbf{foldl}\ \mathsf{elimVar}\ T\ \mathsf{varPairs}$
$\quad \mathsf{varPairs} = [\ (s_i, s_j) \mid R \leftarrow \Gamma, s_i \leftarrow \mathrm{Scp}_R\,T, s_j \leftarrow \mathrm{Scp}_R\,T, i < j\ ]$
$\quad \mathsf{elimVar}\ T\ (s, s') =$
$\qquad \textbf{if}\ \Delta, \mathrm{Scp}\,T \mid \Phi \vdash_{\mathscr{O}} \mathrm{Bdy}\,T \to s = s'\ \textbf{then}\ T[s'/s]\ \textbf{else}\ T$

The two terms are equivalent since we can break down the second term by the equality

$$\begin{aligned} [x = s] &= [(x = s) \wedge (P(s) \vee \neg P(s))] \\ &= [(x = s) \wedge P(s)] + [(x = s) \wedge \neg P(s)]. \end{aligned}$$

To accommodate, we propose to further normalize LNFs by partitioning the terms into the finest possible disjoint fragments, which we make precise in the following definition.

Definition 9. A LNF $\Gamma \mid \Delta \vdash U$ is *partitioned* if for any normal terms T, T' of U such that $\mathrm{Scp}\,T = \mathrm{Scp}\,T'$, we have exactly one of

$$\Delta, \mathrm{Scp}\,T \vdash \mathrm{Bdy}\,T \leftrightarrow \mathrm{Bdy}\,T', \qquad \Delta, \mathrm{Scp}\,T \vdash \mathrm{Bdy}\,T \not\leftrightarrow \mathrm{Bdy}\,T'. \tag{25}$$

Moreover, we call a pair of LNF under the same context, say U_1 and U_2, *fully partitioned* when $U_1 + U_2$ is partitioned.

We present in Algorithm 6 a procedure to produce fully partitioned LNFs. Notice that the algorithm takes both U-expressions U_1 and U_2 that we wish to perform equivalence check on as input, and produce fully partitioned U_1' and U_2'. This agrees with the observation from the above example that partitioning the terms in one side requires knowledge of the terms in the other.

4.2 Unification

With a fully partitioned pair U_1 and U_2, unification can be done in a truly term-by-term fashion. Additionally, with the guarantee that U_1 and U_2 are both in LNF, term comparison can also be trivial. Hence we can finish the equivalence check with a relatively simple unification procedure as in Algorithm 7.

And we finally have the equivalence checking procedure

Algorithm 8 Equivalence check of SQL queries

$\mathsf{equiv} : \mathsf{Query} \to \mathsf{Query} \to \textbf{Bool}$
$\mathsf{equiv}\ Q_1\ Q_2 = \mathsf{unify}\ U_1\ U_2\ \textbf{where}$
$\quad (U_1, U_2) = \mathsf{part}\ (\mathsf{linear}\ (\mathsf{norm}\ [\![Q_1]\!])\ (\mathsf{linear}\ (\mathsf{norm}\ [\![Q_2]\!]))$

4.3 Soundness

We show that our approach is sound with respect to the U-semiring semantics in the following sense.

Theorem 1. *For a pair of queries under the same context* $\Gamma \mid \Delta \mid \Phi \vdash Q_1, Q_2$*, we have*

$$\mathsf{equiv}(Q_1, Q_2) \to [\![Q_1]\!] = [\![Q_2]\!]. \tag{26}$$

Algorithm 6 Partitioning LNF

```
part : [LNTerm] → [LNTerm] → ([LNTerm], [LNTerm])
part U1 U2 = separate (parts U1) (parts U2)
-- Given expression U in LNF, return a partitioned LNF expression
-- that is equivalent to U.
parts U = concatMap (foldl cut [] U′) (groupWith Scp U) where
  cut U T = T′ :: concatMap pieces U where
    P = Bdy T
    pieces T1 = [T1[P1 ∧ ¬P], T1[P1 ∧ P], T[P ∧ P1]] where
      (P, P1) = (Bdy T, BdyT1)
    T′ = foldl (λT2 T1 ⇒ T2[Bdy T2 ∧ Bdy T1]) T U
-- Given U1 and U2 both partitioned, return a fully partitioned pair
-- that is equivalent to (U1, U2).
separate U1 U2 = unzip (concatMap sepPair pairs) where
  -- Obtain the distinct scopes among all terms.
  S = Data.List.nub (map Scp (U1 ++ U2))
  pairs = [(T1, T2) | s ← S, T1 ← U1, T2 ← U2, Scp T1 = Scp T2 = s]
  sepPair (T1, T2) = (U1′, U2′) where
    (P1, P2) = (Bdy T1, Bdy T2)
    U1′ = [T1[P1 ∧ ¬P2], T1[P1 ∧ P2]]
    U2′ = [T2[P2 ∧ ¬P1], T2[P2 ∧ P1]]
```

Algorithm 7 Unifying LNFs with partitioned terms

```
unify : [LNTerm] → [LNTerm] → Bool
unify U1 U2 = bagEq termEq (prune U1) (prune U2) where
  prune = filter (λT ⇒ Δ, Scp T ⊬_O ¬ Bdy T)
  termEq T1 T2 = if Rel T1 ≠ Rel T2 then False else
    Δ, Scp T1 ⊢_O Bdy T1 ↔ Bdy T2
```

Proof. During each stage of normalization we are only using the axioms of U-semiring for rewriting, and during unification the equality check is also based on those axioms. □

4.4 Completeness

Theorem 2. *For a pair of queries under the same context* $\Gamma \mid \Delta \mid \Phi \vdash Q_1, Q_2$*, we have*

$$[\![Q_1]\!] = [\![Q_2]\!] \rightarrow \mathsf{equiv}(Q_1, Q_2). \tag{27}$$

Proof. We prove the contrapositive by first assuming the procedure $\mathsf{Equiv}(Q_1, Q_2)$ returns false. Given that our normalization procedures preserve U-semiring identity, and let $s \vdash U_1$ and $s \vdash U_2$ be the fully partitioned LNFs of $s \vdash [\![Q_1]\!](s)$ and $s \vdash [\![Q_2]\!](s)$ respectively, we wish to show $U_1 \neq U_2$.

By the assumption, during unification, we have some term T that occurs in an unbalanced manner between U_1 and U_2, as in

$$C(T, U_1) \neq C(T, U_2),$$

where $C(T, U)$ denotes the number of terms that are equal to T in the sense of termEq in Algorithm 7. Among all such unbalanced term T, we pick one with the minimal length of scope and denote it with T^*. Additionally, we may prune away balanced terms from U_1 and U_2 at this point as they have no effect on the equality of U_1 and U_2.

We now construct an instance of Δ and a family of instances of Γ based on T^* and later show at least one such instantiation leads to the desired conclusion $U_1 \neq U_2$. First, since we have already pruned trivially empty terms during unification, there must exists some $\delta \in \Delta$ and $\sigma \in \mathrm{Scp}\, T^*$ under which $\mathrm{Bdy}\, T^*$ is true. The instantiation $\sigma \in \mathrm{Scp}\, T^*$ can be regarded as a sequence of values, where the i-th value is the instantiation of the i-th variable in $\mathrm{Scp}\, T^*$. Moreover, we use σ_R to denote the subsequence of σ which form the instantiation of $\mathrm{Scp}_R\, T^*$.

Now suppose there are m values in the sequence σ, and we will construct a family of instances of Γ over the set $N = \mathbb{N}^m_{>0}$. Concretely, over the index $n \in N$ we construct an instance $\gamma \in \Gamma$, namely for each relational variable $R \in \Gamma$, let

$$\gamma_R(s) = \begin{cases} n_i & \text{if } s \in \sigma_R \text{ and } s \text{ is the } i\text{-th element of } \sigma \\ 0 & \text{otherwise} \end{cases}, \tag{28}$$

where n_i denotes the i-th component of n. Such instance of R is well defined, since the LNF already ensures the values in σ_R to be pairwise distinct, which guarantees γ_R to be a well-formed function.

Under the constructed context instantiations δ and γ, we may evaluate the fully partitioned LNF of both sides, U_1 and U_2. Many terms in both sides will be evaluated to zero, and we claim that the remaining terms that are non-zero will have a form similar to T^*, in the sense that for any $T' \neq 0$, we have

$$\mathrm{Rel}'\, T^* = \mathrm{Rel}'\, T', \qquad \Delta, \mathrm{Scp}\, T^* \vdash \mathrm{Bdy}\, T^* \leftrightarrow \mathrm{Bdy}\, T'.$$

This holds for the non-zero terms since we can prove the contrapositive by cases.

1. Suppose $\mathrm{Rel}'\, T^* \neq \mathrm{Rel}'\, T'$. Since we have chosen T^* to be an unbalanced term with the minimal length in scope, their must exists some relational variable R such that $\mathrm{Scp}_R\, T'$ is longer than $\mathrm{Scp}_R\, T^*$. With the fact that T' is in LNF, there must be some $s \in \mathrm{Scp}_R\, T'$ where $\mathrm{Bdy}\, T'$ implies s is always distinct from the values in σ_R. But since s is applied to R in the term, and the instantiation γ_R vanishes at all points beyond those of σ_R, the term T' must also vanish under the evaluation.
2. Otherwise we have $\mathrm{Rel}'\, T^* = \mathrm{Rel}'\, T'$ but the body of T^* and T' are not equivalent. Using the fact that the terms are fully partitioned, we then know

   $$\Delta, \mathrm{Scp}\, T^* \vdash \mathrm{Bdy}\, T^* \not\leftrightarrow \mathrm{Bdy}\, T'.$$

   Hence under the evaluation with γ and δ, the value of T' vanishes, as the summations in T' can only be non-zero on σ, which does not satisfy $\mathrm{Bdy}\, T'$.

Finally, we can inspect the remaining non-zero terms, which are all in a form similar to T^* as shown. For any such term T' in U_1 (or similarly in U_2), the variable and relational variable introduced by the i-th summation $R_i^{k_i}(s_i) \in \mathrm{Rel}\, T'$ evaluates to $\gamma_{R_i}^{k_i}(\sigma_i) = n_i^{k_i}$. And thus $T' = n_1^{k_1} \cdots n_m^{k_m}$. Therefore, the remaining terms in U_1 (and similarly in U_2) evaluates to a polynomial P_1 over the variables $n_1, \ldots, n_m$, where each term in P_1 originates from a unique term T' in U_1 with the coefficient being $C(T', U_1)$ and the powers of variables given by $\mathrm{Rel}\, T'$.

Two polynomials over variables of positive integers are equal if and only if all terms of a certain power have the same coefficient at both sides. However, since T^* occurs in an unbalanced manner, the term originated from T^* will have different coefficients in P_1 and P_2. Thus in the family of instantiation γ over N, there must be some instantiation for which $P_1 \neq P_2$, and hence $U_1 \neq U_2$. □

5 EVALUATION

We implement the algorithm presented in Section 3 in 2,520 lines of Rust, with CVC5 and Z3 as our backend SMT solver. To demonstrate a comparison with similar tools, we first run our prover on a common benchmark suite initially compiled to evaluate the UDP implementation [2], and later also run by EQUITAS [7] and SPES [8]. This benchmark suite consists of 232 query pairs that are extracted from a previous version the Apache Calcite data management framework [1]. Each of these query pairs comes from a specific test case for the query rewrite engine in Calcite, which has an input query and a corresponding expected rewritten query, all sharing the same table schema information.

However, we notice that among the 232 collected query pairs, 23 pairs are already identical and hence trivial to prove. We believe this is due to the fact that certain test cases in Calcite is setup such that the expected output query should not be rewritten, and those cases were still accidentally collected in the benchmark suite. To obtain query pairs of higher quality, we prepare another suite extracted from a newer version of Calcite (1.32.0) and take care to filter out all the trivial pairs. This new benchmark suite contains 416 query pairs, and we are able to rerun the previous state-of-the-art, SPES on this new suite for direct comparison.

Tool	Semantics	Old suite (232 total) Proved	Avg. time	New suite (416 total) Proved	Avg. time
UDP	Bag	34	4.16 s	–	–
EQUITAS	Set	67	0.19 s	–	–
SPES	Bag	95	0.08 s	107	0.08 s
Our tool	Bag	148	1.18 s	235	1.67 s

Table 1: Comparison of automated SQL query checkers. We collect the number and average runtime of provable cases using the original and new benchmark suite extracted from Calcite. The data for UDP and EQUITAS are respectively taken from [2] and [7].

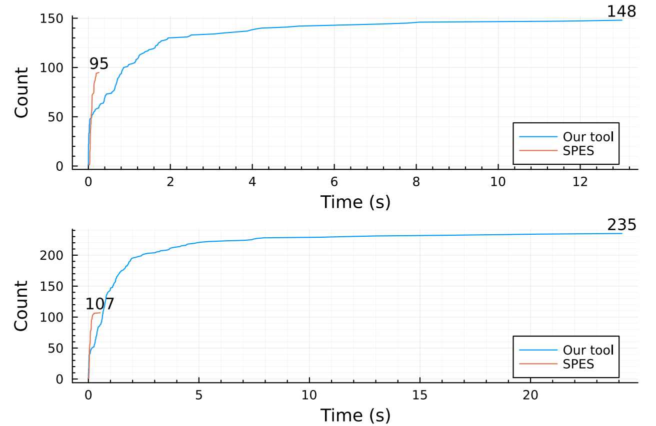

Figure 1: The distribution of runtime for provable cases by SPES and our tool. For a given time t, the corresponding case count is the number of cases proved under t by each tool. The upper plot contains the data collected when running both tools on the old Calcite suite, and the lower plot contains those from the new Calcite suite.

In Table 1 we report the number of provable cases and average execution time for the relevant tools running on the two Calcite benchmark suite prepared as above. Our tool can successfully prove substantially more cases compared to the next best tool, SPES in both the old and new benchmark suite. And compared to UDP, the only other implementation that is also based on U-semiring, we can see both a great increase in proof count as well as much faster execution time. To gain more insight on the difference in runtime compared to SPES, we record the runtime of both tools on a per case basis as in Figure 1. We can see that our tool has comparable performance to SPES for the fastest 50 cases, and our runtime distribution is greatly skewed by a few slowest cases. We do note that the slowest cases arise when the normalization algorithm (Algorithm 1) hits the exponential worst case of distributing + over × and $\sum$, producing a large term for later parts to process. But since in such cases the produced terms are very similar, we expect some form of caching can be helpful in avoiding repetitive computation as a future performance improvement.

We would also like to comment a few points on our implementation strategy. First, we only implement the general algorithm and avoid implementing the more specialized algorithm in Section 4. Even though the specialized algorithm is proven to be complete, we regard it to be only of theoretical interest since its time complexity makes it perform poorly in practise, and its specialized nature makes it cover less query pairs that occur in real world use. Another notable design choice is that we choose to implement the decision procedure in Rust, rather than using a proof assistant such as Coq or Lean as in prior work [2] that also based on the U-semiring semantics. The advantage of using a more general purpose programming language is that we can get good language binding support to SMT solvers and avoid doing tactic-level meta-programming common in proof assistants. This leads to better runtime performance and easier extensibility in the codebase. But we do believe implementing our algorithm within a proof assistant would be beneficial in the future, when better integration with SMT solver is possible. A current issue is that one should regard our current Rust implementation as a trusted codebase, while a proof tactic in Coq or Lean can not only provide an answer to the user, but also generate a corresponding proof script that can be independently checked. Moreover, since our general algorithm are always incomplete, some user intervention at the proof searching stage can be helpful or sometimes necessary, and interactive proof assistants are the ideal user interface for such experience.

6 RELATED WORK

Reasoning about query equivalence is a long standing central topic in the field of database systems. Due to its undecidable nature in general, many works have devoted to characterizing special fragments of SQL that admits decidable query equivalence. More specifically for bag semantics, Cohen [5] has shown the decidability of equivalence for union of conjuncted queries (UCQ). Our work can be seen as an extension to Cohen's work in the sense that we a) generalizes the proof techniques to work on more general constructs, b) additionally include correlated joins and primary keys in the decidable fragment, and c) relativize the decidability to the decidability of some underlying logical theory, capturing the use of SMT solvers as oracles. In fact, by taking the theory of linear arithmetics, in which satisfiability is decidable, our result implies the decidability of UCQ under bag semantics without any oracles.

On the implementation side, we have compared our solver with UDP, EQUITAS, and SPES quantitatively in Section 5. We largely share the same semantics foundation with UDP, which pioneers the practise of automated equivalence proof and is the origin of the U-semiring formalism. UDP is implemented as a proof tactics using the Lean programming language, which makes it much more trustworthy compared to others. But UDP does not incorporate an SMT solver, limiting its capability in reasoning on foreign keys (potentially non-terminating) and NULL semantics (no support). Since our work is additionally SMT-based, we are able to better formalize integrity constraints and correctly model the NULL semantics.

The other solver EQUITAS and later SPES uses a different approach as they reason directly on the level of (symbolic) query expressions, and do not descent down to a lower semantic representation such as U-semiring. Essentially, a few special query rewrite rules are being selected as trusted, and queries are first normalized by repetitively applying those special rules, which are then unified with the help of SMT solvers. Such technique is effective for many simple cases, but as pointed out by Zhou et al. [8], the semantics of integrity constraints are modeled incompletely by a few ad hoc rewrite rules, which are unable to discover many real-world equivalences. We instead insists on modeling semantics on the U-semiring level, and strive to use rules that are general and composable to better capture the interaction of various aspects of SQL semantics.

7 CONCLUSION

In this paper we present a new framework for query equivalence checking, combining the U-semiring semantics with SMT solvers with better encoding of SQL features. Empirically, our implementation can verify 235 out of 415 real-world query pairs extracted from the Calcite data management framework, which doubles over the next best tool in number (104/415). To capture the power of the framework theoretically, we show that our formalism admits a large query fragment in which query equivalence is decidable relative to the satisfiability of some first-order theory, generalizing prior decidability results further. We hope that our contributions are a step towards integrating SQL semantic reasoning tools in more practical settings.

REFERENCES

[1] Edmon Begoli et al. "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". en. In: *Proceedings of the 2018 International Conference on Management of Data*. Houston TX USA: ACM, May 2018, pp. 221–230. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3190662. URL: https://dl.acm.org/doi/10.1145/3183713.3190662 (visited on 05/08/2023).

[2] Shumo Chu et al. "Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries". en. In: *Proceedings of the VLDB Endowment* 11.11 (July 2018), pp. 1482–1495. ISSN: 2150-8097. DOI: 10.14778/3236187.3236200. URL: https://dl.acm.org/doi/10.14778/3236187.3236200.

[3] Shumo Chu et al. "Demonstration of the Cosette Automated SQL Prover". en. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. Chicago Illinois USA: ACM, May 2017, pp. 1591–1594. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3058728. URL: https://dl.acm.org/doi/10.1145/3035918.3058728 (visited on 05/22/2023).

[4] Shumo Chu et al. "HoTTSQL: proving query rewrites with univalent SQL semantics". en. In: *ACM SIGPLAN Notices* 52.6 (Sept. 2017), pp. 510–524. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/3140587.3062348. URL: https://dl.acm.org/doi/10.1145/3140587.3062348 (visited on 05/22/2023).

[5] Sara Cohen. "Equivalence of queries that are sensitive to multiplicities". en. In: *The VLDB Journal* 18.3 (June 2009), pp. 765–785. ISSN: 1066-8888, 0949-877X. DOI: 10.1007/s00778-008-0122-1. URL: http://link.springer.com/10.1007/s00778-008-0122-1 (visited on 02/04/2023).

[6] Heinrich Herre et al. "The Härtig quantifier: a survey". en. In: *Journal of Symbolic Logic* 56.4 (Dec. 1991), pp. 1153–1183. ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2275466. URL: https://www.cambridge.org/core/product/identifier/S0022481200023525/type/journal_article (visited on 05/08/2023).

[7] Qi Zhou et al. "Automated verification of query equivalence using satisfiability modulo theories". en. In: *Proceedings of the VLDB Endowment* 12.11 (July 2019), pp. 1276–1288. ISSN: 2150-8097. DOI: 10.14778/3342263.3342267. URL: https://dl.acm.org/doi/10.14778/3342263.3342267 (visited on 05/08/2023).

[8] Qi Zhou et al. "SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics". In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. Kuala Lumpur, Malaysia: IEEE, May 2022, pp. 2735–2748. ISBN: 978-1-66540-883-7. DOI: 10.1109/ICDE53745.2022.00250. URL: https://ieeexplore.ieee.org/document/9835223/ (visited on 05/08/2023).