



# QED: A Powerful Query Equivalence Decider for SQL

Shuxian Wang

University of California, Berkeley  
wsx@berkeley.edu

Sicheng Pan

University of California, Berkeley  
pansicheng@berkeley.edu

Alvin Cheung

University of California, Berkeley  
akcheung@cs.berkeley.edu

## ABSTRACT

Checking query equivalence is of great significance in database systems. Prior work in automated query equivalence checking sets the first steps in formally modeling and reasoning about query optimization rules, but only supports a limited number of query features. In this paper, we present QED, a new framework for query equivalence checking based on bag semantics. QED uses a new formalism called *Q-expressions* that models queries using different normal forms for efficient equivalence checking, and models features such as integrity constraints and NULLs in a principled way unlike prior work. Our formalism also allows us to define a new query fragment that encompasses many real-world queries with a complete equivalence checking algorithm, assuming a complete first-order theory solver. Empirically, QED can verify 299 out of 444 query pairs extracted from the Calcite framework and 979 out of 1287 query pairs extracted from CockroachDB, which is more than 2× the number of cases proven by prior state-of-the-art solver.

### PVLDB Reference Format:

Shuxian Wang, Sicheng Pan, and Alvin Cheung. QED: A Powerful Query Equivalence Decider for SQL. PVLDB, 17(11): 3602 - 3614, 2024.

doi:10.14778/3681954.3682024

### PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/qed-solver/prover.git>.

## 1 INTRODUCTION

Query equivalence checking is widely used in databases for ensuring the semantic-equivalence of query rewrites in query execution engines [10, 11]. Although the general case is undecidable [16], much theoretical work showed the decidability of query equivalence under special fragments. For example, Cohen [9] showed the SQL equivalence problem under bag semantics is decidable for unions of conjunctive queries (UCQ) by constructing a decision procedure and showing its soundness and completeness. However, besides the restricted use of query operators, the UCQ fragment additionally requires the filter conditions be conjunctions of equalities or inequalities, and only field reference are allowed in projections. We instead define a general fragment parameterized by some first-order theory  $T$ , where the filter conditions and projection rules are allowed to use any expressions definable in  $T$ . We then construct a complete algorithm for deciding bag semantic query equivalence

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.  
doi:10.14778/3681954.3682024

within this fragment with respect to an oracle that can decide satisfiability in the theory  $T$ . When realizing the oracle using modern SMT solvers, the complete algorithm covers a rich set of data types and operations used in real-world queries, to be shown in Sec. 6.

More recently, automatic reasoning tools were built to tackle the full complexity of real-world SQL queries. Cosette [6, 8] axiomatizes queries using the theory of U-semiring and implemented the U-expression decision procedure (UDP) [7], where the bag semantics of a SQL query is modeled as a symbolic arithmetic expression representing the multiplicities of elements in the query result viewed as a bag/multiset. Equivalence checking is then performed on the formal expressions of multiplicity (so-called U-expressions) without any further knowledge of SQL, producing a machine-checkable proof of equivalence down to the axioms of U-expressions. The UDP algorithm works by checking for the equality of two U-expressions after a normalization process, which unfortunately is overly reliant on syntactic equality in many places where stronger equivalence should be used. For example, predicates embedded in queries that are equivalent but syntactically distinct such as  $[a + b = 0]$  and  $[a = -b]$  would fail to unify in UDP. Additionally, UDP does not model NULL semantics [13] and all of the related relational operators (such as outer-joins), rendering it unusable for many queries.

Meanwhile, SPES [17, 18], the current state-of-the-art solver in terms of completeness, operates by first repeatedly applying a set of selected query rewrite rules to put SQL queries into a normal form, and then using an SMT solver to help check the equivalence of the normalized queries. Formally, since SPES does not use a separate representation like the U-expressions to model the semantics of SQL query, the semantics of the relational operators are encoded implicitly in the set of query rewrites for normalization, which we must trust as axioms before accepting the new rewrites that are checked by SPES. Moreover, their normalization rules fail to capture many interactions between different SQL features, and hence compose poorly in practice. For example, SPES has two rewrite rules concerning primary key constraints, which only covers self-joins [18], but not the more common case of joining different tables on primary keys, despite table joins and primary keys both being supported features of SPES. Finally, SMT reasoning is not incorporated during normalization, making SPES still prone to pure syntactical matching to be discussed in Sec. 6.3.

To address these shortcomings, we propose a new formalism called *Q-expressions* that models the general semiring semantics similar to U-expressions, while additionally admits an efficient encoding to SMT formulas. We leverage the expressiveness of first-order logic to fully model NULL semantics and various integrity constraints, and allow unknown query operators to be captured as uninterpreted functions. For example, our new modeling of integrity constraints (Sec. 3.3) only introduce one simple rule for each kind of constraint, but they are designed to compose well with the

formalism of other SQL features during equivalence checking, as we will discuss in Sec. 6.3 when compared to prior work.

We also introduce a new equivalence checking algorithm for Q-expressions that leverages SMT solvers in both the normalization and unification steps (Sec. 5), and first-class support for reasoning with unknown query operators. SMT reasoning allows us to systematically apply normalization and term matching rules without overly relying on ad-hoc syntactical conditions, resulting in significant improvement in completeness over prior works to be discussed in Sec. 6. Moreover, our algorithm can propagate constraints and perform reasoning across boundary introduced by unknown query operators, which are otherwise complete black-boxes in prior approaches, to be illustrated with the motivating example in Sec. 2.

To summarize, using SMT solvers enables more powerful semantics reasoning and expressivity compared to the purely syntactic approaches used in UDP and some parts of SPES, while the semiring formalism of query semantics forms a small yet composable foundation for SQL reasoning, compared to the query-expression level reasoning in SPES. We describe a new approach that combines *both* formal modeling of queries and leveraging SMT solvers for the *entire pipeline* of equivalence checking, capturing and vastly improving upon prior work. In sum:

- We propose a new formalism called Q-expressions that models queries based on semiring semantics and allows efficient encoding in first-order logical formula, leveraging the reasoning power of SMT solvers. Using Q-expressions substantially increases the number of query features modeled compared to prior work, such as NULL semantics, integrity constraints, and first-class support of unknown query operators as uninterpreted symbols.
- We describe two new algorithms to decide semantic query equivalence: a complete one on a new query fragment that we define, generalizing prior theoretical results; and another general algorithm for the full set of query features modeled by Q-expressions.
- We implemented our algorithms in QED, and evaluated it using the rewrite rules from Calcite and CockroachDB. The results show that QED substantially outperforms prior state-of-the-art by proving 2× more cases in all benchmark suites.

## 2 OVERVIEW

In this section we use a concrete example to motivate our approach. Consider the table  $R(x, y)$  with two columns  $x$  and  $y$ , and the table  $S(a)$  with column  $a$  which is also a primary key of  $S$ . The following example originates from a test case for Calcite’s optimizer [2] that describes the following equivalence:

```
select x, sum(y) from R join S on x = a group by x;
select x, r from (select x, sum(y) as r from R group by x
  join S on x = a;
```

Informally, the outer aggregation is pushed down to the left side of the join if the aggregating function and the group-by clause only depends on fields from the left, and the join condition is an equality between the grouped columns from the left with some columns that have distinct values from the right. The key to proving equivalence amounts to the following two points:

- The distinctness of  $S.a$  imposed by the primary key constraint ensures the multiplicity of rows in the final results are preserved

before and after the rewrite, and the equivalence would be unprovable otherwise. However, prior work like SPES [18] fails to discover the distinctness as it models primary key constraints using an incomplete set of query preprocessing rules, in which none is applicable for this specific case.

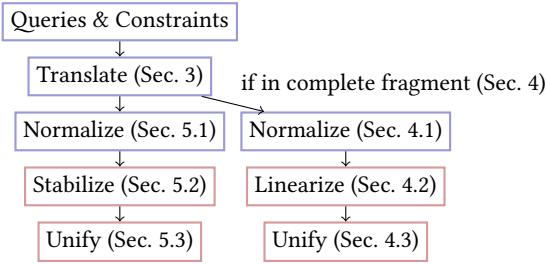
- The group-by/aggregation clauses appear in different places before and after the rewrite. When regarding the aggregation as uninterpreted operators with the tables aggregated over as subqueries applied to the aggregation, prior work [7, 18] would perform a recursive equivalence check of the subqueries to determine the equivalence of the aggregations. However, the recursive check fails to prove the equivalence of those seemingly different subqueries, since originally we have the outer aggregation applying to a filtered version of table  $R$  (filtered by the join condition of  $R.x$  must equal to some  $S.a$ ); while after the transformation, the aggregation is applied directly on  $R$  itself.

Our approach, illustrated in Fig. 1, addresses the above challenges and proves the validity of the rewrite naturally. To start with, both queries are translated into Q-expressions (Sec. 3), which models relations and queries as algebraic expressions that compute row multiplicities, as in prior work [12]. For our two queries, we get:

$$\begin{aligned} Q_1(x, r) &= \|\sum_{y,a}[x = a] \times R(x, y) \times \|S(a)\|\| \\ &\quad \times [r = \text{Sum}(\lambda y. \sum_a[x = a] \times R(x, y) \times \|S(a)\|)], \\ Q_2(x, r) &= \sum_a[x = a] \times \|\sum_y R(x, y)\| \\ &\quad \times [r = \text{Sum}(\lambda y. R(x, y))] \times \|S(a)\|. \end{aligned}$$

Queries  $Q_1$  and  $Q_2$  are modeled as functions that take in a potential value of rows (here being a tuple  $(x, r)$  as the result has two columns) and return its multiplicities in each query’s result (Sec. 3.2). The square bracket notation  $[P]$  denotes a multiplicity of 1 or 0 depending on whether the inner predicate  $P$  is true or false, and when combined like  $[P] \times m$ , models filtering by behaving like  $m$  when  $P$  is true, and trivializing to the zero multiplicity when  $P$  is false.  $\sum_x f(x)$  denotes a (potentially) unbounded summation of terms indexed over a variable  $x$ , similar to the notation for denoting a series mathematically. Here, forms like  $\sum_a[x = f(a)] \times R(a)$  models projection/mapping of multisets by some function  $f$ , by ranging all potential values in the domain  $(a)$ , and only accumulates the multiplicity if  $a$  get mapped to the end value  $x$  that we are interested in (through the predicate  $[x = f(a)]$ ). Finally, the squash notation  $\|m\|$  denotes 1 when  $m \geq 1$  and 0 otherwise, which is used to model uniqueness by effectively truncating multiplicity to be at most 1. Specifically, the primary key constraint on  $S$  makes use of squash by translating  $S$  into  $\lambda x. \|S(x)\|$  using the rule eq. (8), which is a complete formalism of primary key in that applying it results in an equivalent query equivalence problem. Moreover, such modeling is convenient for SMT solvers to reason about, as  $\|S(x)\|$  can be simply represented as an uninterpreted predicate.

Our full pipeline then picks between Alg. 3 and Alg. 6 depending on whether the input queries are in the complete fragment to be defined in Sec. 4. Since our example is not in that fragment, we showcase the algorithm presented in Sec. 5. The queries  $Q_1$  and  $Q_2$  are first rewritten into a normal form to be described in Sec. 5.1, and later in Sec. 5.2 have redundant summation scopes systematically eliminated while maintaining the normal form. Those two steps trivialize many equivalences, and in our case results in a very similar



**Figure 1: Overview of QED’s equivalence checking pipeline, starting from top to bottom. The left lane depicts the algorithm in Sec. 5 that accepts general SQL queries, and the right lane shows the complete algorithm for a specialized fragment discussed in Sec. 4. The procedures in red boxes would invoke an SMT solver as an oracle.**

pair of Q-expressions:

$$Q_1(x, r) = \|\sum_y R(x, y) \times \|S(x)\| \times [r = \text{Sum}(\lambda y. R(x, y) \times \|S(x)\|)]\|$$

$$Q_2(x, r) = \|\sum_y R(x, y) \times \|S(x)\| \times [r = \text{Sum}(\lambda y. R(x, y))]\|$$

At this point, we are ready to compare the now highly-structured pair of Q-expressions, using the unification algorithm in Sec. 5.3. Concretely in the example, we are interested in verifying

$$\forall x, r. \phi_1(x, r) \leftrightarrow \phi_2(x, r), \text{ where} \quad (1)$$

$$\phi_1(x, r) = \exists y. R(x, y) \neq 0 \wedge S(x) \neq 0 \wedge r = v_1,$$

$$\phi_2(x, r) = \exists y. R(x, y) \neq 0 \wedge r = v_2 \wedge S(x) \neq 0,$$

$$v_1 = \text{Sum}(\lambda y. R(x, y) \times \|S(x)\|), \quad v_2 = \text{Sum}(\lambda y. R(x, y)).$$

Notice that the two aggregations are substituted with two distinct variables  $v_1$  and  $v_2$ , and we may further extract their congruence information by recursively checking the equivalence between

$$\lambda y. R(x, y) \times \|S(x)\| \quad \text{and} \quad \lambda y. R(x, y).$$

However, directly performing the recursive check will fail, since the right side is missing the term  $\|S(x)\|$  (or equivalently the condition that  $x$  occurs in table  $S$  at least once). The key observation is that we can automatically read off the missing condition from the Q-expressions that we are comparing at the top-level, by asserting  $\phi_1(x, r) \vee \phi_2(x, r)$  as an *additional assumption* when performing the recursive check. To be made formal in eq. (15), this additional assumption captures constraints within the ambient environment of the subqueries, including the fact that  $x$  does occur in table  $S$ , and preserves them across recursive equivalence checks. At this point, the two subqueries are equivalent as the following is valid:

$$(\phi_1(x, r) \vee \phi_2(x, r)) \rightarrow \forall y. R(x, y) \times \|S(x)\| = R(x, y). \quad (2)$$

Therefore, we have  $v_1 = v_2$ , and with that, the top-level equivalence can be concluded as now we can check eq. (1) by validating

$$v_1 = v_2 \rightarrow (\forall x, r. \phi_1(x, r) \leftrightarrow \phi_2(x, r)). \quad (3)$$

We implement and evaluate this algorithm in Sec. 6 by running it over query optimization pairs present in real-world databases, and compare the results with similar tools.

### 3 SYNTAX AND SEMANTICS OF QUERIES

Before discussing QED’s equivalence checking algorithm, we first define the syntax of SQL handled by QED and its semantics.

$Q$	$\coloneqq$	$\text{Table}(R: S)$	<i>Table scan on R of schema S</i>
		$\text{Values}(v_1, \dots, v_n)$	<i>Raw values as table</i>
		$\text{Filter}(P, Q)$	<i>Filter by predicate P</i>
		$\text{Proj}(f, Q)$	<i>Projection by function f</i>
		$\text{Join}(Q_1, Q_2)$	<i>Cross product join</i>
		$\text{Union}(Q_1, Q_2)$	<i>Union all</i>
		$\text{Minus}(Q_1, Q_2)$	<i>Set minus</i>
		$\text{Distinct}(Q)$	<i>Row deduplication</i>
		$\text{GroupBy}(k, \alpha(f), Q)$	<i>Group by keys k</i>
		$\text{QOp}(o, v_1, \dots, v_n, Q)$	<i>Uninterpreted query operator</i>
$v, P, f$	$\coloneqq$	$x$	<i>Variable symbol</i>
		$\text{Exists}(Q)$	<i>Query emptiness</i>
		$\text{Op}(o, v_1, \dots, v_n)$	<i>Scalar operator</i>
		$\text{HOOp}(o, v_1, \dots, v_n, Q)$	<i>Higher-order operator</i>
$\alpha, o$	$\in$	Operator symbols	
$C$	$\coloneqq$	$R.k$ PrimaryKey	<i>Primary key on k</i>
		$R.k$ References $S$	<i>Foreign key on k to table S</i>
		$R$ Checks $P$	<i>Satisfies predicate P</i>

**Figure 2: Syntax for supported query expressions ( $Q$ ), scalar expressions ( $v, P, f$ ), and integrity constraints ( $C$ ).**

### 3.1 Query syntax

QED assumes that the input queries to be checked for equivalence are written using the grammar shown in Fig. 2, intended to capture the bag semantics of SQL queries that we model in Sec. 3.2. In addition to query expressions, we define scalar expressions in the syntax that is allowed to appear as condition to filter, expressions to project, or arguments to Values. Moreover, QED also supports integrity constraints on tables to be discussed in Sec. 3.3.

Inspired by relational algebra, a query  $Q$  can be constructed from different query operators, starting from table scan  $\text{Table}(R : S)$  that scans a named table with name  $R$  containing rows of type  $S$ , or  $\text{Values}(v_1, \dots, v_n)$  that explicitly constructs a table with the rows  $v_1, \dots, v_n$ .  $\text{Filter}(P, Q)$  only keeps the rows in the subquery  $Q$  that satisfies the predicate  $P$ , and  $\text{Proj}(f, Q)$  transforms every row  $r$  of  $Q$  into  $f(r)$ .  $\text{Join}(Q_1, Q_2)$  forms the cross product of two queries, and  $\text{Union}(Q_1, Q_2)$  is concatenation, i.e., having all rows in  $Q_1$  followed by those in  $Q_2$  and preserving any duplication. For  $\text{Minus}(Q_1, Q_2)$ , it keeps any row in  $Q_1$  that is not present in  $Q_2$  and removes duplicated rows; the  $\text{Distinct}$  operator also eliminates duplicated rows.<sup>1</sup> We use the squash operator  $\|\cdot\|$ , which truncates multiplicities up to 1 as later defined in eq. (4), to formally express row deduplication.

QED models group-by ( $\text{GroupBy}(k, \alpha(f), Q)$ ) explicitly. For query  $Q$  that produces rows of type  $S$ , we group by some key function  $k : S \rightarrow K$  that partitions rows in  $Q$  into groups, where each group is a bag of rows that share the same key. Each group is transformed by applying their rows with  $f : S \rightarrow V$ , forming a bag of  $V$ , and aggregated by the function  $\alpha$  to create a bag of values (now of type  $V$ ) returning a scalar representing the aggregated value.

Other remaining SQL operators can be expressed using the general uninterpreted query operator,  $\text{QOp}$ . For example, to express limiting the result of  $Q$  to  $n$  rows, one can use  $\text{QOp}(\text{Limit}, n, Q)$ . Similarly, compound scalar expressions are built with  $\text{Op}$  and  $\text{HOOp}$ . For

<sup>1</sup>We model set minus, but not the multiset/bag minus, which would be captured as an uninterpreted query operator, similarly, set intersection is expressible, but not multiset intersection. Supporting the bag variants requires some ordering/monus operator at the semiring level, which we don’t introduce in our formalism.

example, the SQL syntax  $a \text{ And } b$  can be encoded as  $\text{Op}(\text{And}, a, b)$ , and the SQL syntax  $a \text{ In } Q$  (involving a subquery  $Q$ ) can be encoded as  $\text{HOp}(\text{In}, a, Q)$ . Generally, we can give semantics to scalar expressions involving  $\text{Op}$  and  $\text{HOp}$  if it can be encoded in SMT solvers (details in Sec. 3.2), or otherwise we would treat them as uninterpreted symbols.

### 3.2 Query semantics

The semantics of a query is defined by the table it produces, and a table of schema  $S$  is regarded as a multiset of rows of type  $S$ . Following the semiring semantics [12], we interpret such multiset as a function  $S \rightarrow \bar{\mathbb{N}}$  taking in some  $s \in S$  and returning the multiplicity of  $s$  in the multiset, where  $\bar{\mathbb{N}}$  is the set of extended natural numbers ( $\mathbb{N} \cup \{\infty\}$ ). We define:

$$\begin{aligned} \infty + a &= \infty, & 0 \times \infty &= 0, & a \times \infty &= \infty \text{ if } a \neq 0, \\ \sum_s f(s) &= \begin{cases} f(s_1) + \dots + f(s_n) & \text{if } f \text{ has finite support } \{s_1, \dots, s_n\}, \\ \infty & \text{otherwise} \end{cases} \\ [P] &= \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}, & \|a\| &= [a \neq 0] & \neg a &= [a = 0], \end{aligned} \quad (4)$$

with  $+$  and  $\times$  being the usual operation when acting on finite numbers, and the support of  $f$  is defined to be  $\{x \mid f(x) \neq 0\}$ . This makes  $\bar{\mathbb{N}}$  a model of U-semiring as introduced by [7], and additionally has an efficient encoding in SMT solvers as shown later in Sec. 5.3. We refer to expressions of type  $\bar{\mathbb{N}}$  as Q-expressions. Formally, the semantics of a query  $Q$  denoted as  $\llbracket Q \rrbracket$ , is given inductively as:

$$\begin{aligned} \llbracket \text{Table}(R: S) \rrbracket &= \lambda x. R(x) \\ \llbracket \text{Values}(v_1, \dots, v_n) \rrbracket &= \lambda x. [x = v_1] + \dots + [x = v_n] \\ \llbracket \text{Filter}(P, Q) \rrbracket &= \lambda x. [\llbracket P \rrbracket(x) = \text{True}] \times \llbracket Q \rrbracket(x) \\ \llbracket \text{Proj}(f, Q) \rrbracket &= \lambda x. \sum_s [x = \llbracket f \rrbracket(s)] \times \llbracket Q \rrbracket(s) \\ \llbracket \text{Join}(Q_1, Q_2) \rrbracket &= \lambda x_1, x_2. \llbracket Q_1 \rrbracket(x_1) \times \llbracket Q_2 \rrbracket(x_2) \\ \llbracket \text{Union}(Q_1, Q_2) \rrbracket &= \lambda x. \llbracket Q_1 \rrbracket(x) + \llbracket Q_2 \rrbracket(x) \\ \llbracket \text{GroupBy}(k, \alpha(f), Q) \rrbracket &= \lambda x, y. \sum_s [x = \llbracket k \rrbracket(s)] \times \llbracket Q \rrbracket(s) \\ &\quad \times [y = \text{HOp}(\alpha, \lambda y'. \sum_s \llbracket Q \rrbracket(s)) \\ &\quad \times [x = \llbracket k \rrbracket(s) \wedge y' = \llbracket f \rrbracket(s)]]] \\ \llbracket \text{Distinct}(Q) \rrbracket &= \lambda x. \llbracket Q \rrbracket(x) \\ \llbracket \text{Minus}(Q_1, Q_2) \rrbracket &= \lambda x. \llbracket Q_1 \rrbracket(x) \times \neg \llbracket Q_2 \rrbracket(x) \\ \llbracket \text{QOp}(o, v_1, \dots, v_n, Q) \rrbracket &= \lambda x. \text{QOp}(o, v_1, \dots, v_n, \llbracket Q \rrbracket)(x) \end{aligned} \quad (5)$$

Moreover, the semantics of scalar expressions is defined as:

$$\begin{aligned} \llbracket \text{Exists}(Q) \rrbracket &= \|\sum_s \llbracket Q \rrbracket(s)\| \\ \llbracket \text{Op}(o, v_1, \dots, v_n) \rrbracket &= \text{Op}(o, \llbracket v_1 \rrbracket, \dots, \llbracket v_n \rrbracket) \\ \llbracket \text{HOp}(o, v_1, \dots, v_n, Q) \rrbracket &= \text{HOp}(o, \llbracket v_1 \rrbracket, \dots, \llbracket v_n \rrbracket, \llbracket Q \rrbracket). \end{aligned}$$

To keep track of context information, we sometimes annotate expression  $Q$  as  $\Gamma \mid \Delta \mid \Phi \vdash Q$ , where  $\Gamma$  and  $\Delta$  are respectively the list of table symbols and other uninterpreted symbols that  $Q$  may refer to, and  $\Phi$  is a predicate over  $\Gamma$  and  $\Delta$  that would be satisfied. For a pair of queries of the same schema under the same context, we define the query equivalence problem  $\Gamma \mid \Delta \mid \Phi \vdash Q_1 \stackrel{?}{=} Q_2$  to be the problem of deciding equality of the Q-expressions

$$\Gamma \mid \Delta, s \mid \Phi \vdash \llbracket Q_1 \rrbracket(s) = \llbracket Q_2 \rrbracket(s).$$

That is, for all instantiations of the tables in  $\Gamma$  and variables in  $\Delta$  such that  $\Phi$  is satisfied, and any possible row  $s$ , the multiplicity of  $s$  in  $Q_1$  is the same as that of  $s$  in  $Q_2$ . For brevity, in the following we omit context annotations when they can be inferred.

### 3.3 Integrity constraints

SQL allows user to define three types of integrity constraints within `CREATE TABLE` clauses, namely the primary key (uniqueness), the foreign key, and user-defined check constraints. QED models such constraints by rewriting the Q-expressions as well as their surrounding context, such that checking the equality of those Q-expressions are equivalent before and after applying the rewrites.

*Primary key constraints.* If table  $R$  has schema  $K \times S$ , where there is a column  $k$  of type  $K$  being the primary key of  $R$ , then there exists a function  $f : K \rightarrow S$  representing the functional dependency of the non-key columns on the key columns, over the subset of  $K$  that is contained in  $R$ . Conversely, any subset of  $K$  combined with a function  $f : K \rightarrow S$  induces a table  $R$  with primary key on  $K$ . In fact, for any logical predicate  $P$  over a table, we have that

$$\begin{aligned} &\forall R \text{ with primary key } k. P(R) \\ &\iff \forall R', f_R. P(\lambda k, s. \|R'(k)\| \times [s = f_R(k)]). \end{aligned} \quad (6)$$

This means that for table  $R$  with a primary key  $k$ , we can rewrite

$$R \vdash R \rightsquigarrow R' \mid f_R \vdash \lambda k, s. \|R'(k)\| \times [s = f_R(k)]. \quad (7)$$

Note that we change from having the table symbol  $R$  (of schema  $K \times S$ ) to another table symbol  $R'$  (of schema  $K$ ) in the context, and substitute  $R$  with an equivalent relation containing  $R'$  in the body. When  $k$  is the only column in  $R$  we can regard the non-key columns being of the unit type and trivializing the functional dependency  $f_R$ . Hence, the above rule simplifies to:

$$R \vdash R \rightsquigarrow R \vdash \lambda k. \|R(k)\|. \quad (8)$$

Compared to modeling primary keys through rewrite rules on the query syntax as in SPES [18], which fails to capture the full semantics, our approach is complete as shown in eq. (6). Moreover, eq. (7) only introduces new uninterpreted symbols that is easily encoded and reasoned by SMT solvers, in comparison to maintaining an extra group of special query rewrite rules that may or may not be captured by the normal forms as presented in SPES.

*Foreign key constraints.* Consider  $R$  is a table of schema  $K \times A$  with a primary key on  $K$ , and  $S$  being a table of schema  $B \times K$  with a foreign key of type  $K$  referencing to keys in  $R$ . This means that for any  $(b, k)$  in  $S$ , there exist exactly one  $a \in A$  such that  $(k, a)$  occurs exactly once in  $R$ , which is captured by the following equality:

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

This rule is employed eagerly in UDP [7], but if  $R$  additionally has a foreign key constraint referring back to  $S$ , such process would not terminate due to the cyclical reference.

We resolve the non-termination of rewrites by encoding the foreign key constraint as a single global logical formula, allowing SMT solvers to lazily unfold the consequences as needed. Concretely, the above intuition can be expressed as:

$$S \vdash S \rightsquigarrow S, R' \mid \forall b, k. \|S(b, k)\| \rightarrow \|R'(k)\| \vdash S. \quad (9)$$

As a derived rule, when  $S$  has both the primary key constraint and foreign key constraint, we can apply both eqs. (7) and (9) to obtain

$$S \vdash S \rightsquigarrow S', R' \mid \forall b. \|S'(b)\| \rightarrow \|R'(f_S(b))\| \vdash S.$$

*Check constraints.* Suppose on a table  $R$  of schema  $S$ , we have a check constraint  $C$  expressed as a logical predicate using the syntax for scalar expressions in Fig. 2 over  $S$ . The check constraint ensures all rows in  $R$  satisfy  $C$ , hence gives the rewrite rule:

$$S \vdash S \rightsquigarrow S \mid \forall s. \|R(s)\| \rightarrow C(s) \vdash S.$$

### 3.4 NULL semantics

Since NULL values are allowed to occur under any typing context in SQL, we equip every type  $T$  with a distinguished value  $\text{Null}_T$ . For elementary types such as Integer and Boolean, we do this systematically by lifting them to the option type  $\text{Option}(T)$  in the semantics, using the additional element as the distinguished NULL value. Such translation is supported by modern SMT solvers as the option types and can be defined as custom algebraic data types. In general, for an  $n$ -ary primitive operation  $f$  that is well-defined on non-NUL values, the behavior of  $\text{Op}(f, a_1, \dots, a_n)$  is extended to return NULL exactly when some  $a_i$  is NULL. Hence, we can systematically lift many primitive operations such as  $+$  and  $\times$  that occurs in the surface syntax into the corresponding operations that also operates on nullable values.

The above rule, however, has a few exceptions [14]. For the SQL Boolean operations `And` and `Or`, we follow three-value logic:

$$\begin{aligned} a \text{ And } b &:= \text{ite}(a = \text{Null}, \quad a \text{ Or } b := \text{ite}(a = \text{Null}, \\ &\quad \text{ite}(b = \text{True}, \text{Null}, b), \quad \text{ite}(b = \text{False}, \text{Null}, b), \\ &\quad \text{ite}(a = \text{True}, b, a)), \quad \text{ite}(a = \text{False}, b, a)), \end{aligned}$$

where `ite` denotes the if-then-else construct. Another special case is the `Some` operator, appearing in SQL as  $a \langle \text{cmp} \rangle \text{Some } Q$ , which checks if there is some row  $r$  in the subquery  $Q$  that compares with  $a$  using the comparison operator  $a \langle \text{cmp} \rangle r$ . The usual  $a \text{ In } Q$  construct is just the special case when  $\langle \text{cmp} \rangle$  is the SQL equality operator. However, complications arise when  $a$  itself is NULL or when  $R$  contains NULL values, and we use the following definition:

$$\begin{aligned} a \langle \text{cmp} \rangle \text{Some } Q &:= \text{ite}(\exists x. \|Q(x)\| \wedge (x \langle \text{cmp} \rangle a) = \text{True}, \text{True}, \\ &\quad \text{ite}(\exists x. \|Q(x)\| \wedge (x \langle \text{cmp} \rangle a) = \text{Null}, \text{Null}, \text{False})). \end{aligned}$$

In aggregations, the difference in NULL value processing (the `COUNT(*)` vs. `COUNT(k)` functions) is modeled by optionally applying a non-NUL filter before applying the aggregating function.

## 4 EQUIVALENCE CHECKING ALGORITHM FOR THE COMPLETE QUERY FRAGMENT

We first describe an algorithm that decides on bag-semantic equivalence of queries for fragment we define below. Unlike prior work [9] that focuses on UCQ, we define a new fragment that additionally allows scalar expressions from some first-order theory  $T$ , instead of confining to only  $=$  and  $\wedge$  in UCQ. Although it is restricted compared to the full query syntax given in Sec. 3, we find a substantial number of real-world query pairs fall into this fragment as we will discuss in Sec. 6. Assuming the existence of an oracle  $\mathcal{O}$  that can decide the satisfiability problem of  $T$  (e.g., an ideal SMT solver), we describe a complete query equivalence checking algorithm where

the structure of the new fragment is captured by introducing the scoped and linearized normal form (SNF and LNF, see Secs. 4.1 and 4.2), we define a complete unification algorithm (Alg. 3) on LNFs below, and prove its completeness in Sec. 4.4.<sup>2</sup>

Concretely, we define our complete fragment as follows.

**DEFINITION 1.** *For some first-order theory  $T$ , we define the query fragment  $\mathcal{F}_T$  composed of Table, Values, Filter, Proj, Join, and Union operators defined in Fig. 2, where all the involved values  $v_i$ , projection expressions  $f$ , and filter conditions  $P$  are definable in the theory  $T$ . We also allow primary key constraints on the tables involved.*

As mentioned, we assume the existence of an oracle  $\mathcal{O}_T$  for  $\mathcal{F}_T$  where  $\mathcal{O}_T$  can decide any satisfiability problem in  $T$ . Equivalently, for any context  $\Gamma$  and formula  $\phi$  in  $T$  under the context  $\Gamma$ , we can check if  $\phi$  holds under all possible instantiation of the context  $\Gamma$ , denoted  $\Gamma \vdash_{\mathcal{O}_T} \phi$ , by querying the oracle  $\mathcal{O}_T$  whether  $\neg\phi$  is unsatisfiable with uninterpreted symbols from  $\Gamma$ . Our method is generic over the underlying theory  $T$ , hence the user can choose any theory  $T$  and its solver. Here we fix a theory  $T$  (decidable by some oracle  $\mathcal{O}$ ) with the following 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 in the oracle.
- (3) The oracle also supports reasoning with natural number addition  $+$ , and the if-then-else construct `ite`.

These assumptions are satisfied by modern solvers such as SMT. With this background, we describe an algorithm that completely decides the query equivalence problem in  $\mathcal{F}$  with the oracle  $\mathcal{O}$ .

### 4.1 Normalization

The queries in the fragment  $\mathcal{F}$  can be normalized in a way that we can exploit during equivalence check. Hence our first step is to rewrite any  $Q$ -expression  $\llbracket Q \rrbracket$  obtained from some query  $Q \in \mathcal{F}$  into the following normal form.

**DEFINITION 2.** *An  $Q$ -expression  $U$  under some context  $\Gamma \mid \Delta$  is in scoped normal form (SNF) when it is the sum of  $n$  normal terms*

$$\Gamma \mid \Delta \vdash U = T_1 + \dots + T_n,$$

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

$$\Gamma \mid \Delta \vdash T = \sum_{R_1^{k_1}(s_1)} \dots \sum_{R_m^{k_m}(s_m)} [P], \quad (10)$$

Here  $P$ , called the body of  $T$  and denoted  $\text{Bdy } T$ , is a predicate under the context  $\Delta, s_1, \dots, s_m$ , i.e., it does not contain any table variables. We use the notation

$$\sum_{R^0(s)} U := \sum_s \|R(s)\| \times U, \quad \sum_{R^k(s)} U := \sum_s \underbrace{R(s) \times \dots \times R(s)}_{k \text{ times}} \times U$$

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

<sup>2</sup>The completeness of Alg. 3 relies on the oracle  $\mathcal{O}$  being complete, but even when  $\mathcal{O}$  can only incompletely solve satisfiability problems, Alg. 3 would still be sound.

Additionally, we impose an ordering of the table variables in  $\Gamma$ , which gives  $\Gamma = R_1, R_2, \dots$ , and require the summations in  $T$  introduce variables applied to  $R_i$  before those applied to  $R_j$  whenever  $i < j$ .

Since SNF is highly stylized, we define some additional operations that would be helpful for equivalence checking later.

- For a normal term  $T$ , we use  $\text{Scp } T$  to denote the entire scope of  $T$ , namely the list of all variables introduced by the summations in  $T$  (e.g., in Eq. (10),  $\text{Scp } T = s_1, \dots, s_m$ ).
- Furthermore, for a table variable  $R$  in  $T$ , we let  $\text{Scp}_R T$  be the sub-list of  $\text{Scp } T$  that contains only the summation variables that are applied to  $R$  when introduced by the summations in eq. (10).
- Similarly, we define  $\text{Rel } T$  (the relational scope of  $T$ ) to be the list of table variables applied with summation variables (including their power) introduced by the summations in  $T$ , and  $\text{Rel}' T$  to be the list of variables without the power (e.g., in (10),  $\text{Rel } T = R_1^{k_1}, \dots, R_m^{k_m}$  and  $\text{Rel}' T = R_1, \dots, R_m$ ).

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 over  $s$  by  $R^k(s)$  merged with that over  $s'$  by  $R^l(s')$  to form one summation over  $s'$  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  $Q$ -expression  $\llbracket Q \rrbracket$  can always be rewritten into the normal form. We proceed by induction on a strengthened hypothesis: For  $\Gamma \mid \Delta, x \vdash \llbracket Q \rrbracket(x)$ , we can write it as the sum of normal terms where each is of the form

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

and  $P$  does not contain  $x$ . The proof is given by construction in Alg. 1. For the base cases, Table is already a normal term (line 3), and similarly with Values (line 5). Inductively, Filter pushes down its condition  $P$  into each term with conjunction (line 7), and Proj pushes down the projection function  $f$  into each term with post-composition, without introducing additional summation scopes (line 10). For the binary operator Join, every term in the first SNF is combined with every term in the other SNF by merging their summation scopes and conjoining their bodies (line 13). Finally, Unioning SNFs are simply concatenation of their terms (line 17).

## 4.2 Linearization

Deciding the equivalence of two SNF formulas can be reduced down to comparing their normal terms, but equivalence checking of normal terms requires fixing an order for their respective summation variables. For example, the terms

$$\begin{aligned} R \mid x \vdash T_1 &= \sum_{R(s_1)} \sum_{R(s_2)} [x = s_1], \\ R \mid x \vdash T_2 &= \sum_{R(s_1)} \sum_{R(s_2)} [x = s_2], \end{aligned}$$

are identical if we (rightfully) swap the summation variables  $s_1$  and  $s_2$  in  $T_2$  for direct comparison. But  $T_1$  and  $T_2$  are not the same in the sense of checking

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

---

### Algorithm 1 Normalizing query to scoped normal form

---

```

Require:  $Q \in \mathcal{F}$ 
1: function  $\text{snf}(Q)$ 
2:    $U \leftarrow 0$ 
3:   if  $Q$  is of the form  $\text{Table}(R)$  then
4:     return  $R \mid x \vdash \sum_{R(s)} [x = s]$ 
5:   else if  $Q$  is of the form  $\text{Values}(v_1, \dots, v_n)$  then
6:     return  $x \vdash [x = v_1] + \dots + [x = v_n]$ 
7:   else if  $Q$  is of the form  $\text{Filter}(P, Q')$  then
8:     for all term  $T$  of  $\text{snf}(Q')$  do
9:        $U \leftarrow U + T[\text{Bdy } T \wedge P]$ 
10:    else if  $Q$  is of the form  $\text{Proj}(f, Q')$  then
11:      for all term  $y \vdash \sum_{R^k(s)} [y = g(\bar{s}) \wedge P]$  of  $\text{snf}(Q')$  do
12:         $U \leftarrow U + x \vdash \sum_{R^k(s)} [x = f(g(\bar{s})) \wedge P]$ 
13:    else if  $Q$  is of the form  $\text{Join}(Q_1, Q_2)$  then
14:      for all term  $x \vdash \sum_S [x = f \wedge P]$  of  $\text{snf}(Q_1)$  do
15:        for all term  $x' \vdash \sum_{S'} [x' = f' \wedge P']$  of  $\text{snf}(Q_2)$  do
16:           $U \leftarrow U + x, x' \vdash \sum_{S, S'} [(x, x') = (f, f') \wedge P \wedge P']$ 
17:    else if  $Q$  is of the form  $\text{Union}(Q_1, Q_2)$  then
18:      return  $\text{snf}(Q_1) + \text{snf}(Q_2)$ 
19: return  $U$ 

```

---

This is because that variables in  $\text{Scp}_R T_1$  or  $\text{Scp}_R T_2$  can be equivalently introduced in any order, while when comparing normal terms by their body, we enforce an ordering of scope by putting both sides under the same universal quantifier.

Another problematic case comes from redundant variables:

$$\begin{aligned} R \mid x \vdash T_1 &= \sum_{R(s_1)} \sum_{R(s_2)} [(x = s_2) \wedge (s_1 = s_2)], \\ R \mid x \vdash T_2 &= \sum_{R^2(s_1)} [x = s_1], \end{aligned}$$

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. In general, our semiring semantics over  $\bar{\mathbb{N}}$  admits the following for eliminating summation:

$$\sum_x [x = a] \times f(x) = f(a) \quad (11)$$

Our solution is to fix variable ordering and eliminate redundancy in normal terms via linearization.

**DEFINITION 3.** A  $Q$ -expression  $U$  is in linearized normal form (LNF) if it is in SNF and that for all normal term  $T$  in  $U$  and distinct table variable  $R$  in  $T$ ,  $\text{Bdy } T$  implies the variables  $\text{Scp}_R T$  are pairwise-distinct and form a linearly ordered chain.

Such property effectively fixes the ordering of  $\text{Scp}_R T$  for any  $R$  in  $T$ , and since we have already fixed the ordering of distinct table variables, the entire scope  $\text{Scp } T$  now have a well-defined total order of summation variables.

The procedure to rewrite an SNF  $U$  into an equivalent and LNF is presented in Alg. 2, where  $\text{Lin}(X)$  enumerates all possible relations between the variables in set  $X$  using the total order  $<$  or equality. Concretely, if  $X = \{a, b, c\}$ , then we have

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

---

**Algorithm 2** Linearizing the normal form

---

**Require:**  $\Gamma \mid \Delta \vdash U$  in SNF

- 1: **function**  $\text{lin}(U)$
- 2:     $U' \leftarrow 0$
- 3:    **for all** term  $T$  of  $U$  **do**
- 4:       $R_1, \dots, R_n \leftarrow$  table symbols appear in  $T$
- 5:      **for all**  $(C_1, \dots, C_n) \in \text{Lin}(\text{Scp}_{R_1} T) \times \dots \times \text{Lin}(\text{Scp}_{R_n} T)$  **do**
- 6:         $T' \leftarrow T[C_1 \wedge \dots \wedge C_n \wedge \text{Bdy } T]$
- 7:        **for all**  $i \in \{1, \dots, n\}$  **do**
- 8:           $s_1, \dots, s_m \leftarrow \text{Scp}_{R_i} T'$
- 9:          **for all**  $0 < j < k \leq m$  **do**
- 10:            **if**  $\Delta, \text{Scp } T' \vdash_O \text{Bdy } T' \rightarrow s_j = s_k$  **then**
- 11:               $T' \leftarrow T'[s_j/s_k]$
- 12:          Reorder variables in  $\text{Scp } T'$  based on  $C_1, \dots, C_n$
- 13:           $U' \leftarrow U' + T'$
- 14:     **return**  $U'$

---

Every normal term is expanded on the possible cases given by  $\text{Lin}$  on the set of summation variables applied to the same table variable (line 5). For example, expanding a term with two summation variables  $\sum_{R(a)} \sum_{R(b)} [P]$  gives

$$\begin{aligned} & \sum_{R(a)} \sum_{R(b)} [(a < b) \wedge P] + \sum_{R(a)} \sum_{R(b)} [(a > b) \wedge P] \\ & + \sum_{R(a)} \sum_{R(b)} [(a = b) \wedge P]. \end{aligned}$$

After expansion, we can eliminate redundant summation variables in each term  $T$  (line 10) by substitution  $T[s'/s]$  whenever any pair of variables  $s$  and  $s'$  are equal (implied by  $\text{Bdy } T$ ). At this point in each term, all variables applied to the same table symbol should form a totally-ordered chain by  $<$ , and we can rearrange them in the strictly ascending order.

### 4.3 Unification

We finally check for equivalence of two LNF expressions using Alg. 3. To do so, we first attempt to merge terms in each LNF. For example, consider the following two LNFs

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

The two LNFs are equivalent since

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

In general, we group terms by their relational scope (as given by  $\text{Rel } T$ ) on each side (line 4 and 5), and for some group of terms with the scope  $R^k(s)$ , we then have the equivalence problem

$$\overline{\sum_{R^k(s)} [P_1] + \dots + [P_n]} \stackrel{?}{=} \overline{\sum_{R^k(s)} [P'_1] + \dots + [P'_m]},$$

which can be checked by the oracle by comparing the bodies over all possible instantiation of the variables  $\bar{s}$ , i.e.,

$$\bar{s} \vdash_O [P_1] + \dots + [P_n] \stackrel{?}{=} [P'_1] + \dots + [P'_m].$$

The  $[ \cdot ]$  and  $+$  operator can be modeled using the *ite* construct and addition on natural numbers in the oracle. Hence, we can finish the equivalence check with the unification procedure in Alg. 3.

---

**Algorithm 3** Unifying LNF pair

---

**Require:**  $\Gamma \mid \Delta \vdash U_1, U_2$  in LNF

- 1: **function**  $\text{unify}(U_1, U_2)$
- 2:    **for all** distinct relational scope  $S$  appeared in  $U_1$  or  $U_2$  **do**
- 3:       $\bar{s} \leftarrow$  summation variables in  $S$
- 4:       $\sum_S [P_1], \dots, \sum_S [P_n] \leftarrow$  terms  $T$  of  $U_1$  where  $\text{Rel } T = S$
- 5:       $\sum_S [P'_1], \dots, \sum_S [P'_m] \leftarrow$  terms  $T'$  of  $U_2$  where  $\text{Rel } T' = S$
- 6:      **if**  $\Delta, \bar{s} \nvdash_O [P_1] + \dots + [P_n] = [P'_1] + \dots + [P'_m]$  **then**
- 7:        **return** False
- 8:      **return** True

---

### 4.4 Completeness of decision procedure

**THEOREM.** *For a pair of queries under the same context  $\Gamma \mid \Delta \mid \Phi \vdash Q_1, Q_2$ , our equivalence deciding algorithm described above is complete. In other words:*

$$\llbracket Q_1 \rrbracket = \llbracket Q_2 \rrbracket \rightarrow \text{unify}(\text{linear}(\text{norm}(Q_1)), \text{linear}(\text{norm}(Q_2))).$$

**PROOF.** We prove by the contrapositive. Let  $s \vdash U_1$  and  $s \vdash U_2$  be the LNFs of  $s \vdash \llbracket Q_1 \rrbracket(s)$  and  $s \vdash \llbracket Q_2 \rrbracket(s)$  respectively, and we wish to show  $U_1 \neq U_2$ . During unification, we group the terms of  $U_1$  and  $U_2$  by their relational scope, and perform group-wise comparison. Hence, for any relational scope  $S$ , there is a pair of groups of terms

$$G_S = \{T \in U_1 \mid \text{Rel } T = S\}, \quad G'_S = \{T' \in U_2 \mid \text{Rel } T' = S\},$$

that is checked for equality between ( $\oplus$  denoting n-ary plus  $+$ )

$$B_S = \bigoplus_{T \in G_S} [\text{Bdy } T], \quad B'_S = \bigoplus_{T' \in G'_S} [\text{Bdy } T']$$

as described in Line 6 of Alg. 3. By the assumption, some pairs of groups failed such equality check, and among which we pick one with a relational scope  $S^*$  that has the minimal number of variables. For all the other groups that are equal, we can safely ignore them going forward as they can be canceled out equally on both sides. Additionally, we use  $S^{*\prime}$  be the same scope as  $S^*$  but with the power on table variables ignored (i.e.,  $S^{*\prime} = \text{Rel}' T$  for some  $T \in G_{S^*}$ ), and  $S_R^*$  for the variables in  $S^*$  that are applied to some table symbol  $R$  (i.e.,  $S_R^* = \text{Rel}_R T$  for some  $T \in G_{S^*}$ ).

We now construct an instance of  $\Delta$  and a family of instances of  $\Gamma$  based on  $S^*$  and later show at least one such instantiation leads to the desired conclusion  $U_1 \neq U_2$ . First, since the pair of groups from  $S^*$  failed the equality check during unification, there must exist some instantiation  $\delta$  of  $\Delta$  and  $\sigma$  of the summation variables in  $S^*$  under which the bodies of the group are different

$$B_{S^*}[\delta, \sigma] \neq B'_{S^*}[\delta, \sigma]. \quad (12)$$

The instantiation  $\sigma$  can be regarded as a sequence of values, where the  $i$ -th value is the instantiation of the  $i$ -th summation variable in  $S^*$ . Moreover, we use  $\sigma_R$  to denote the subsequence of  $\sigma$  which form the instantiation of the variables in  $S^*$  that are applied to  $R$ .

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

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

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  $\sigma_R$  to be pairwise distinct, which guarantees  $\gamma_R$  to be a well-formed function.

Under the constructed context instantiations  $\delta$  and  $\gamma$ , we may evaluate the LNF of both sides,  $U_1$  and  $U_2$ . Many terms in both sides will evaluate to zero, and we claim that the remaining terms that are non-zero will have a relational scope similar to  $S$ , in that for any  $T \neq 0$ , we have  $\text{Rel}' T = S^{**}$ . Otherwise for some term  $T \neq 0$  with  $\text{Rel}' T \neq S^{**}$ , there must exist some table variable  $R$  such that  $\text{Scp}_R T$  is longer than  $S_R^*$ , since we have chosen  $S^*$  to have the minimal number of variables. Since  $T$  is in LNF, there must be some  $s \in \text{Scp}_R T$  where  $\text{Bdy } T$  implies  $s$  is always distinct from the values in  $\sigma_R$ . But since  $s$  is applied to  $R$  in the term, and the instantiation  $\gamma_R$  vanishes at all points beyond those of  $\sigma_R$ , the term  $T'$  must also vanish under the evaluation.

Finally, we inspect the remaining non-zero terms, which all belong to groups with a scope similar to  $S^*$  as shown above. For any such group  $G_S$  in  $U_1$  (or similarly in  $U_2$ ), the variable and table variable introduced by the  $i$ -th summation  $R_i^{k_i}(s_i) \in S$  evaluates to  $\gamma_{R_i}^{k_i}(\sigma_i) = n_i^{k_i}$ . And as we have a total order of variables (ensured by LNF), the entire group evaluates to a monomial over the  $n_i$ 's

$$G_S[\delta, \sigma, \gamma] = B_S[\delta, \sigma] \times n_1^{k_1} \times \cdots \times n_m^{k_m}.$$

Therefore, the remaining terms in  $U_1$  (and similarly in  $U_2$ ) evaluates to a polynomial  $P_1$  (and similarly  $P_2$ ) over the variables  $n_1, \dots, n_m$ , where each term in  $P_1$  comes from a remaining group  $G_S$  in  $U_1$  with the coefficient being its body  $B_S[\delta, \sigma]$  and the powers of variables given by the scope of the group,  $S$ . Two polynomials over variables of natural numbers are equal if and only if all monomials of a certain power have the same coefficient on both sides. However, since the monomial corresponds to  $S^*$  have different coefficients as assumed in eq. (12), in the family of instantiation  $\gamma$  over  $N$ , there must be some instantiation for which  $P_1 \neq P_2$ , and hence  $U_1 \neq U_2$ .  $\square$

## 5 A GENERAL CHECKING ALGORITHM

While the procedure described above is complete (modulo a complete solver for the chosen theory  $T$ ), it can only handle a subset of queries defined in Sec. 3. We now describe an incomplete checking algorithm for our formalism that consists of two stages: first, we normalize arbitrarily formed Q-expressions into a normal form (Secs. 5.1 and 5.2), and then attempt to unify them (Sec. 5.3). The normal form here is weaker than those defined in Secs. 4.1 and 4.2, since we are working with a more general query fragment that is less structured than the fragment in Sec. 4.1. Even though it is incomplete, the wide range of input queries and good runtime characteristic in the common cases make it performs exceptionally well as we show in Sec. 6.

Similar to Sec. 4, we would assume the existence of an oracle  $O$ , but here with only the power of *incompletely* deciding the satisfiability problem of the logical theories in interest. That is, when checking a logical formula with free variables,  $O$  may give three results: satisfiable, not satisfiable, or unknown. And as presented later in Sec. 6, SMT solvers serve the role of the oracle  $O$  in QED's implementation. We use the notation

$$\Gamma \mid \Delta \mid \Phi \vdash_O P$$

to denote whether in  $O$ , checking the formula  $\Phi \wedge \neg P$  with uninterpreted symbols from  $\Gamma$  and  $\Delta$  returns the “not satisfiable” result.

---

### Algorithm 4 Converting to sum–product normal form

---

**Require:** Q-expression  $\Gamma \mid \Delta \mid \Phi \vdash U$

```

1: function spnf( $U$ )
2:    $U' \leftarrow 0$ 
3:   if  $U$  is of the form  $U_1 + U_2$  then
4:     return spnf( $U_1$ ) + spnf( $U_2$ )
5:   else if  $U$  is of the form  $U_1 \times U_2$  then
6:     for all term  $\sum_{\overline{s_1}}[L_1] \times V_1$  of spnf( $U_1$ ) do
7:       for all term  $\sum_{\overline{s_2}}[L_2] \times V_2$  of spnf( $U_2$ ) do
8:          $U' \leftarrow U' + \sum_{\overline{s_1}, \overline{s_2}}[L_1 \wedge L_2] \times V_1 \times V_2$ 
9:   else if  $U$  is of the form  $\sum_{\overline{s}} V$  then
10:    for all term  $T$  of spnf( $V$ ) do
11:       $U' \leftarrow U' + \sum_{\overline{s}} T$ 
12:   else if  $U$  is of the form  $\|V\|$  then return [ $\text{spnf}(V) \neq 0$ ]
13:   else if  $U$  is of the form  $\neg V$  then return [ $\text{spnf}(V) = 0$ ]
14:   else return  $U$ 
15: return  $U'$ 

```

---

### 5.1 Normalization

As  $\times$  and  $\sum$  distributes over  $+$ , and  $\sum$  commutes with  $\times$ , all Q-expressions can be normalized as follows, allowing the equalities generated by those rules become trivially provable.

DEFINITION 4. An Q-expression  $U$  under some context  $\Gamma \mid \Delta \mid \Phi$  is in sum–product normal form (SPNF) when it is the sum of some  $n$  sum–product normal terms

$$\Gamma \mid \Delta \mid \Phi \vdash U = T_1 + \cdots + T_n,$$

where a sum–product normal term  $T$  is some  $m$  nested summations

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

Here  $L$  is any first-order logical formula, and  $V$  is the product of some  $k$  applications of some table variable  $R$  (or uninterpreted query operator  $\text{QOp}(o, v_1, \dots, v_n, Q)$ ) with some expression, namely

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

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

Compared to SNF introduced in Sec. 4.1, SPNF allows any Q-expressions to be normalized into it. Moreover, SPNF is less structured in that table variables are now free to occur anywhere in the body ( $[L] \times V$ ) of terms  $\sum_{\overline{s}}[L] \times V$ , whereas in SNF, table variables are only allowed to be applied with a fresh summation variable. Alg. 4 describes a one-pass normalization procedure by recursion on the structure of Q-expressions. For example, the Q-expression:

$$R(x) \times (\sum_y (S(y) + R(y)) + \sum_z \|S(z)\| \times [P(z)])$$

would be normalized into

$$\begin{aligned} & \sum_y R(x) \times S(y) + \sum_y R(x) \times R(y) \\ & + \sum_z [S(z) \neq 0] \times R(x) + \sum_z [P(z)] \times R(x). \end{aligned}$$

One may notice that the form  $[U \neq 0]$  generated when we encounter a squash operator  $\| \cdot \|$  (and similarly for the negation  $\neg$ ), would make a Q-expression  $U$  present in a logical formula. As presented later, we would include such logical formula in the satisfiability problem for the oracle  $O$ , seemingly requiring  $O$  to already have the power of deciding equality in the theory of Q-expressions,

---

**Algorithm 5** SPNF Stabilization

---

**Require:** Q-expression in SPNF  $\Gamma \mid \Delta \mid \Phi \vdash U$

```

1: function stable( $U$ )
2:    $U' \leftarrow 0$ 
3:   for all term  $\sum_{\bar{s}}[L] \times V$  of  $U$  do
4:      $\bar{s}' \leftarrow \bar{s}$ ,  $L' \leftarrow L$ ,  $V' \leftarrow V$ 
5:     for all variable  $s_i \in \bar{s}$  do
6:        $\bar{s}'' \leftarrow \bar{s}' \setminus s_i$            Remove  $s_i$  from the variable list  $\bar{s}'$ 
7:       Eliminate  $s_i$  if a dependency over others is found.  $\leftarrow$ 
8:       if  $\exists (\Delta \vdash f). \Gamma \mid \Delta, \bar{s}'' \mid \Phi \vdash_O L' \rightarrow f(\bar{s}'') = s_i$  then
9:          $\bar{s}' \leftarrow \bar{s}''$ ,  $L' \leftarrow L'[f(\bar{s}'')/s_i]$ ,  $V' \leftarrow V'[f(\bar{s}'')/s_i]$ 
10:         $U' \leftarrow U' + \sum_{\bar{s}'}[L'] \times V'$ 
11:   return  $U'$ 

```

---

and defeating the point of our own equivalence checking algorithm! Instead, using the following rewrites,

$$\begin{aligned} U_1 + U_2 \neq 0 &\rightsquigarrow U_1 \neq 0 \vee U_2 \neq 0, & \|U\| \neq 0 &\rightsquigarrow U \neq 0, \\ U_1 \times U_2 \neq 0 &\rightsquigarrow U_1 \neq 0 \wedge U_2 \neq 0, & [P] \neq 0 &\rightsquigarrow P, \\ \sum_s U \neq 0 &\rightsquigarrow \exists s. U \neq 0, \end{aligned}$$

the form  $U \neq 0$  (or similarly  $U = 0$ ) can be translated into a Q-expression-free first-order logical formula. In this way, any remaining Q-expressions would only appear in the form of fully applied table symbols or uninterpreted query operators, e.g.,

$$R(a) \neq 0 \quad \text{or} \quad \text{QOp}(v_1, \dots, v_n, Q)(a) \neq 0,$$

which is further encoded in the manner described in Sec. 5.3.

## 5.2 Stabilization

After converting to SPNF, we need an additional procedure before we can check for equivalence. Consider the rule of composing two consecutive projections into one, which results in the following:

$$R \mid x \vdash \sum_{y,z} [x = f(y) \wedge y = g(z)] \times R(z) \stackrel{?}{=} \sum_z [x = f(g(z))] \times R(z). \quad (13)$$

While the two expressions differ in their summation scopes, the summation variable  $y$  on the left can be expressed in terms of other variables as  $g(z)$ , making the summation over  $y$  unnecessary by applying eq. (11). In general, for a normal term like  $\sum_{\bar{x}}[L] \times V$ , we are interested in any potential equalities (between each summation variable  $x_i$  and some independent expression)  $P$  that are *implied* by  $L$ , since  $[L] = [P \wedge L] = [P] \times [L]$  whenever  $L \rightarrow P$ . Unlike prior work [7], this allows us to discover variable dependencies beyond those present in the syntactic level, resulting in more unaligned summations being reduced to have aligned scope. We call this process the stabilization of SPNF as shown in Alg. 5.

Every variable  $s_i$  of a term is checked for redundancy by searching for some functional dependency  $f$  over the set of variables  $\bar{s}'$  that are not yet eliminated (line 8). We have devised two ways to find such  $f$ . First is to use synthesis tools such as Syntax-Guided Synthesis (SyGuS) [1] to directly synthesize  $f$ . Another way is to first obtain the congruence classes of all the expressions involved in the term by calling SMT solvers, where each class contains a subset of the terms that are equal to each other, and based on which we can analyze the congruence classes to find expressions that are equal to  $s_i$  and get a suitable  $f$ . In our example (eq. (13)), for the left-hand side we ask the SMT solver for the congruence classes of  $x, y, z, f(y)$ , and  $g(z)$  to get the groups  $\{x, f(y)\}, \{y, g(z)\}, \{z\}$ ,

which can be used to discover the dependency  $y = g(z)$  and eliminate the variable  $y$ . For the right-hand side, the congruence classes are  $\{x, f(g(z))\}$  and  $\{z\}$ , and since the summation variable  $z$  is the only member of its class, no elimination is performed.

The second strategy can theoretically miss cases if the solution is absent as an expression already in the term. For example, given the condition  $[x + 3 = 10]$ , querying the SMT solver for the congruence information of  $x, 3, x+3$ , and 10 will return that  $x$  is the only member of its congruence class, and thus  $x$  will not be eliminated when it is a summation variable, despite having a true dependency  $x = 7$  that may be discovered by a SyGuS solver. In practice, however, we find both approaches equally powerful in our evaluation and choose the latter for implementation ease.

## 5.3 Unification

Once we normalize and stabilize the Q-expressions to SPNF, we implement equivalence checking using Alg. 6. The core of the unification algorithm is the equality checking between two normal terms and recursively handling the higher-order query and scalar expressions as defined in the termEq function in Alg. 6. Before that, we need to reduce the problem of comparing two SPNFs  $U_1$  and  $U_2$  down to comparing normal terms. We first eliminate terms that are empty by checking if their body contains a trivially false predicate (line 2 and 4). The commutativity of  $+$  is then accommodated by performing pairwise comparisons for the terms in  $U_1$  and  $U_2$ , and cancelling out equated terms (line 7) to ensure a bijection between terms of  $U_1$  and  $U_2$ . Now we are ready to consider the equality between two normal terms  $\sum_{\bar{s}_1}[L_1] \times V_1$  and  $\sum_{\bar{s}_2}[L_2] \times V_2$  where  $\bar{s}_1$  and  $\bar{s}_2$  are of the same length.

**5.3.1 Summation scope permutation.** For two summations sharing the exact same summation variables, their equality can be checked by comparing their bodies, i.e.,

$$(\forall \bar{s}. f(\bar{s}) = g(\bar{s})) \rightarrow \sum_{\bar{s}} f(\bar{s}) = \sum_{\bar{s}} g(\bar{s}). \quad (14)$$

But here we have two distinct sets of summation variables  $\bar{s}_1$  and  $\bar{s}_2$  when comparing the normal terms, and commuting the summation variables of a summation does not change its semantics. Hence, before we align the scope of the normal terms and check for the universal quantification as in (14), we need to enumerate all possible permutations of  $\bar{s}_2$  over  $\bar{s}_1$ . For a certain permutation  $\sigma$ , we can apply that to the right-hand term and reduce the term equality down to equality of term bodies:

$$\Gamma \mid \Delta, \bar{s}_1 \mid \Phi \vdash_O [L_1] \times V_1 = [L_2[\sigma]] \times V_2[\sigma]. \quad (15)$$

There can be a lot of permutations if there are many summation variables. In practice, we set an upper limit of 24 permutation attempts in the implementation, and empirically only about 2% of cases in our evaluation dataset in Sec. 6 would go past the first trivial permutation to find a match.

Linearization described in Sec. 4.2 is similar to scope permutation, but the more structured SNF in the complete fragment allows us to explore fewer permutations. On the other hand, delaying permutation down to term matching as we do here avoids the up-front cost of eager linear expansion in Alg. 2, and the high bias towards the trivial permutation in real-world query pairs make this approach performs much faster in practice.

5.3.2 *Recursive equivalence check.* We denote the goal of comparing the body of the normal terms as

$$\Gamma \mid \Delta \mid \Phi \vdash [L_1] \times V_1 = [L_2] \times V_2 \quad (16)$$

However, as we may have higher-order operators (introduced by QOp and HOOp) that contains *functions* returning  $\mathbb{N}$  as their operands, we cannot directly translate such construct into a first-order logic formula. Instead, we first substitute every higher-order query expression such as  $\text{QOp}(o, v_1, \dots, v_n, Q)$  with a fresh table variable  $R$ , and every such scalar expression  $\text{HOOp}(o, v_1, \dots, v_n, Q)$  with a fresh variable  $x$ . A priori, two syntactically different expressions such as  $\text{QOp}(o, v_1, \dots, v_n, \lambda x. W)$  and  $\text{QOp}(o', v'_1, \dots, v'_n, \lambda x. W')$  would be assigned two different fresh variables  $R$  and  $R'$ , but we recover any congruence information like  $\forall x. R(x) = R'(x)$  if we can determine that  $o$  and  $o'$  are the same operator, the scalar arguments  $v_i$  and  $v'_i$  match, and recursively, the  $Q$ -expressions  $W$  and  $W'$  are equal.

To obtain the congruence information about the freshly introduced variables, the recursive equivalence check of  $W$  and  $W'$  can be soundly taken under the additional assumption  $L'_1 \vee L'_2$  in line 22 of Alg. 6. Intuitively, we would otherwise have  $\neg L'_1 \wedge \neg L'_2$ , in which both  $[L'_1]$  and  $[L'_2]$  will be trivially zero, and so the two terms ( $[L'_1] \times V'_1$  and  $[L'_2] \times V'_2$  in line 23) would be trivially equal no matter what the congruence information of the fresh variables would have been. This allows us to automatically pass down certain conditions when checking for equivalence of the sub-queries embedded inside higher-order operators, which can be essential as illustrated earlier in the motivating example of Sec. 2. Concretely in eq. (1), we first introduce fresh variables for the aggregation functions:

$$v_1 = \text{Sum}(\lambda y. R(x, y) \times [S(x) \neq 0]), \quad v_2 = \text{Sum}(\lambda y. R(x, y)),$$

turning the two terms into  $T'_1 = [L'_1]$  and  $T'_2 = [L'_2]$  with

$$L'_1 = \exists y. R(x, y) \neq 0 \wedge S(x) \neq 0 \wedge r = v_1,$$

$$L'_2 = \exists y. R(x, y) \neq 0 \wedge r = v_2 \wedge S(x) \neq 0.$$

The equivalence between  $T'_1$  and  $T'_2$  reduces to the problem of deciding  $v_1 \stackrel{?}{=} v_2$ , which is checked by recursively comparing  $R(x, y) \times [S(x) \neq 0]$  and  $R(x, y)$ . Although they seemingly differ by the part  $[S(x) \neq 0]$ , our technique allows us to make the assumption  $L'_1 \vee L'_2$  (which contains the missing piece  $S(x) \neq 0$ ) while doing the recursive comparison, under which we can conclude  $v_1 = v_2$ , and thus prove the equivalence. Formally, to verify eq. (16) it is enough to check the following two clauses:

$$\Gamma' \mid \Delta' \mid \Phi \wedge (L'_1 \vee L'_2) \vdash C, \quad \Gamma' \mid \Delta' \mid \Phi \wedge C \vdash [L'_1] \times V'_1 = [L'_2] \times V'_2 \quad (17)$$

Here  $L'_1$  is  $L_1$  with all higher-order operators substituted to fresh variables, and so are  $L'_2$ ,  $V'_1$ , and  $V'_2$ . The context  $\Gamma'$  denotes the context extended from  $\Gamma$  with the newly introduced table variables, and similarly  $\Delta'$  from  $\Delta$  for the scalar variables. The proposition  $C$  denotes the congruence information about the fresh variables obtained through recursive equivalence checking.

5.3.3 *Encoding term equivalence.* Based on our semantics, for any table  $R$  of schema  $S$  in the context  $\Gamma'$ , we should be ranging over all of  $S \rightarrow \mathbb{N}$  when checking for the equivalence in eq. (17). However, to better reflect real-world usage, we require that any such table  $R$  contain any value  $s : S$  only finitely many times, i.e., can be

---

#### Algorithm 6 Unifying SPNFs

---

**Require:**  $\Gamma \mid \Delta \mid \Phi \vdash U_1, U_2$  are SPNFs under the same context.

```

1: function unify( $U_1, U_2$ )
2:   for all term  $T$  of the form  $\sum_{\bar{s}[L] \times V}$  of  $U_1$  do
3:     if  $\Gamma \mid \Delta, \bar{s} \mid \Phi \vdash_O \neg(L \wedge \|V\|)$  then Remove  $T$  from  $U_1$ 
4:   for all term  $T$  of the form  $\sum_{\bar{s}[L] \times V}$  of  $U_2$  do
5:     if  $\Gamma \mid \Delta, \bar{s} \mid \Phi \vdash_O \neg(L \wedge \|V\|)$  then Remove  $T$  from  $U_2$ 
6:   if  $U_1$  and  $U_2$  have different length then return Unknown
7:   for all term  $T_1$  of  $U_1$  do
8:      $M \leftarrow$  False
9:     for all term  $T_2$  of  $U_2$  do
10:      if termEq( $\Gamma \mid \Delta \mid \Phi \vdash T_1 \stackrel{?}{=} T_2$ ) then
11:        Remove  $T_2$  from  $U_2$ 
12:         $M \leftarrow$  True
13:      break
14:    if  $\neg M$  then return Unknown
15:  return True
16: function termEq( $\Gamma \mid \Delta \mid \Phi \vdash \sum_{\bar{s}_1[L_1] \times V_1} = \sum_{\bar{s}_2[L_2] \times V_2}$ )
17:   if  $\bar{s}_1$  and  $\bar{s}_2$  have different length then return False
18:   for all bijective substitution  $\sigma$  of  $\bar{s}_2$  with  $\bar{s}_1$  do
19:      $L_2, V_2 \leftarrow L_2[\sigma], V_2[\sigma]$  Apply substitution  $\sigma$  to  $L_2, V_2$ 
20:      $L'_1, V'_1, L'_2, V'_2 \leftarrow L_1, V_1, L_2, V_2$  with higher-order operators substituted with fresh variables.
21:      $\Gamma', \Delta' \leftarrow \Gamma, \Delta$  extended with the fresh variables.
22:      $C \leftarrow$  congruence relation of the fresh variables through recursive
23:     unify under  $\Gamma' \mid \Delta', \bar{s}_1 \mid \Phi \wedge (L'_1 \vee L'_2)$ 
24:     if  $\Gamma' \mid \Delta', \bar{s}_1 \mid \Phi \wedge C \vdash_O [L'_1] \times V'_1 = [L'_2] \times V'_2$  then
25:       return True
26:   return False
27: function equiv( $Q_1, Q_2$ )
28:   return unify(stable(spnf([[ $Q_1$ ]])), stable(spnf([[ $Q_2$ ]])))

```

---

restricted to a function of type  $S \rightarrow \mathbb{N}$ . With this assumption, we can eliminate all sources of infinity when comparing the bodies in (17), since  $[L'_1]$  and  $[L'_2]$  are always finite, and both  $U'_1$  and  $U'_2$  are products of fully applied tables  $R(s)$ , which we now assume to be always finite. This allows us to regard each body as an expression of type  $\mathbb{N}$ , and the  $\times$ ,  $[ \cdot ]$ ,  $\| \cdot \|$ , and  $\neg$  operators can be restricted to only operate on finite multiplicities. Any oracle  $O$  supporting the Peano arithmetic can be used to reason about the equality in (17) since we can now directly encode the summation bodies.

Finally, the equivalence checking procedure `equiv` (line 26) is the composition of SPNF normalization, stabilization, and unification.

## 6 EVALUATION

We implemented the algorithm presented in Secs. 4 and 5 in 2,520 lines of Rust, with `cvc5` and `z3` as the backend SMT solver. `QED` sends the SMT formulas to `cvc5` and `z3` in parallel and waits for whichever returns first. `QED` dispatches multiple SMT formulas as presented in Algs. 3 and 5, and has a user-configurable timeout for every request to the solvers with a 60 seconds default in all the following experiments.

### 6.1 Test cases

To compare with other state-of-the-art solvers, we first run `QED` on a benchmark suite initially compiled to evaluate the UDP implementation [7], and later also run by `EQUITAS` [17] and `SPES` [18]. This benchmark suite consists of 232 query pairs that are extracted from

a previous version of the Calcite framework [2]. 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 using the same table schema information.

Among the 232 collected query pairs, 23 pairs are identical and hence trivial to prove. To obtain query pairs of higher quality, we prepare another suite extracted from a newer version of Calcite (v1.32.0) and filter out any trivial pairs. This new benchmark suite contains 444 query pairs, and we rerun the previous state-of-the-art solver SPES on this new suite for comparison. The new suite is not run on older tools like EQUITAS since we believe SPES is more powerful than the older tools.

To gain more insights on the performance of QED in comparison with SPES, we further extracted 1,287 non-trivial pairs of queries from CockroachDB (v23.1.3) [15]. Each of these query pairs also comes from a test case for the rewrite engine in CockroachDB. As a proxy for query complexity, the queries in the new Calcite test suite contain 5.6 query operators on average (as defined in Sec. 3.1), with a standard deviation of 2.9. For those from CockroachDB, the average is 5.3 and the standard deviation is 2.3.

As shown in Table 1, QED can successfully prove substantially more cases compared to SPES, in both the old and new benchmark suites. As shown, 33% of the provable Calcite cases lies within the complete fragment, while 67% does so for CockroachDB. This shows the benefit of having both decision procedures in QED to increase coverage. Compared to UDP, the only other implementation that is also based on a semiring formalism, QED can prove 3.32× more cases and prove each case 67% faster on average.

We further analyzed the detailed breakdown of QED runtime in Table 2. For the 299 query pairs where QED successfully verified in the new Calcite test suite, on average QED spends 0.83s with SMT solvers to eliminate redundant summation variables (Alg. 5) and 0.03s with SMT solvers to check term equivalence (Alg. 6), while in total QED spends 0.88s on average for each of these query pairs, as shown in Table 1. Hence the majority of runtime is consumed by SMT solvers, and the overhead of QED is minimal. The same observation holds for the CockroachDB test suite, where 0.31s are spent to eliminate redundant variables and 0.03s are spent to check term equivalence within the 0.35s average total runtime for QED.

We also note that all test cases provable by SPES are also provable by QED, and we further breakdown the provable cases of QED by whether they are also provable by SPES. For the subset of new Calcite test suite where both QED and SPES can successfully verify, on average QED spent 1.53s and SPES spends 0.99s. For the subset of CockroachDB where both can successfully verify, on average QED spent 0.16s and SPES spends 0.03s. Over the full distribution in Fig. 3, QED does perform slower than SPES on the commonly provable cases, although the slowdown is within one order of magnitude.

## 6.2 Failure breakdown

We analyzed the cases where QED fails to prove equivalence and tabulated them in Table 3.

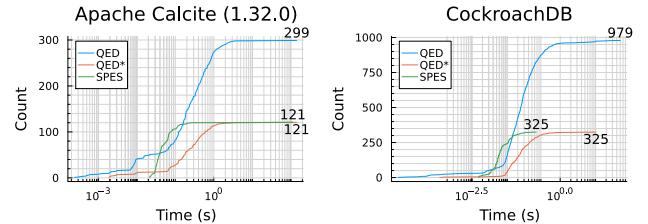
- *Custom operators.* We only modeled common operators for SQL in QED, and some test cases use operators that are not modeled. For example, QED fails to prove the equivalence of some query pairs

**Table 1:** Comparison of automated SQL query checkers. The data for UDP and EQUITAS are respectively taken from [7] and [17]. Numbers in parenthesis are standard deviations. For QED, we give both the number of provable cases within Sec. 4 (QED') and the number of provable cases within the full semantics from Sec. 3 (QED).

Test Suite		Tool				
Name	Stat	UDP	EQUITAS	SPES	QED'	QED
Calcite - Old	Provable (#)	34	67	95	49	147
	Avg. Time (s)	4.16	0.19	0.04 (0.02)	0.18 (0.23)	1.39 (12.0)
Calcite - New	Provable (#)	—	—	121	98	299
	Avg. Time (s)	—	—	0.99 (10.3)	0.24 (0.32)	0.88 (8.21)
CockroachDB	Provable (#)	—	—	325	656	979
	Avg. Time (s)	—	—	0.02 (0.02)	0.29 (1.63)	0.35 (2.26)

**Table 2:** Breakdown of QED average runtime. Numbers in parenthesis are standard deviations. We list the average time QED spent on SMT solvers (STBL when used in Alg. 5 and UNI when used in Alg. 6), the average total runtime for QED on test cases where SPES fails (New), and that on test cases where SPES can also prove (Shared).

Test Suite	QED				SPES
	STBL	UNI	New	Shared	
Calcite - Old	1.34	0.03	0.74	2.3	0.04
Avg. Time (s)	(12.0)	(0.05)	(0.68)	(16.5)	(0.02)
Calcite - New	0.83	0.03	0.71	1.53	0.99
Avg. Time (s)	(8.2)	(0.05)	(0.7)	(12.9)	(10.3)
CockroachDB	0.31	0.03	0.14	0.16	0.03
Avg. Time (s)	(2.25)	(0.03)	(0.16)	(0.67)	(0.02)



**Figure 3:** The cumulative distribution of runtime for provable cases by SPES and QED. For a given time  $t$ , the corresponding case count is the number of cases proved under  $t$  by each tool. QED\* denotes the runtime of QED, but restricted to only the cases that are also provable by SPES.

**Table 3:** Breakdown of cases where QED fails to verify in Calcite and CockroachDB.

Category	Calcite	CockroachDB
Aggregates	96	27
Custom operators	25	152
List semantic	18	103
Typing	6	26

involving the SEARCH operator in Calcite, which determines if a value is in certain ranges, but we only treat it uninterpreted.

- *List semantics.* We use bag semantics to model queries, hence QED currently does not support queries that make use of the ordering semantics of `Limit`, `Offset`, or `Order By`.
- *Aggregates.* Our normal form aims to handle the unbounded summations used by common SQL operators, but this design has limited support for aggregations. We only treat the aggregations as higher order operators without further assumptions, but this does not capture the exact semantics of aggregations, and QED fails to prove some cases as a result.

- *Typing.* Type casting is common in database systems, but SMT solvers do not currently reason about type casting. We can only interpret explicit type casting (e.g., int to float) as uninterpreted functions, while the implicit type casting will lead to error in QED due to type mismatch.

### 6.3 Illustrative cases

Besides the motivating example we give in Sec. 2, we select two more examples from the test cases that are illustrative to compare the respective approach of QED and SPES.

**6.3.1 Join Push Transitive Predicates.** One Calcite case that SPES fails but is provable using QED involves the Join Push Transitive Predicates rule [4]. This rule identifies the constant constraint across joins and propagate the constraint to the inputs of the join. Q2 below is the result of applying this rule on Q1. They are equivalent if we have two tables  $R(a)$  and  $S(b)$ :

Q1: `select * from (select * from R where R.a = 42) as X  
 left join S on X.a = S.b;`

Q2: `select * from (select * from R where R.a = 42)  
 left join (select * from S where S.b = 42) on TRUE;`

The equivalence of the two queries are easy to verify for QED once they are stabilized using Alg. 5, but SPES fails to prove it as one of its normalization rules unsoundly rewrites the left join in Q2 into a cross join when it sees the join condition is TRUE, which then results in two semantically different normal forms.

**6.3.2 Project Join Join Remove.** Another case that SPES fails but QED can prove is from the Calcite Project’s Join Join Remove rule [5]. This rule removes a join on unique key if possible. Q2 below is the result of applying this rule on Q1. They are equivalent if we have two tables  $R(a)$  and  $S(k)$ , where  $k$  in table  $S(k)$  is unique:

Q1: `select R.a from R  
 inner join S as S1 on R.a = S1.k  
 inner join S as S2 on R.a = S2.k;`

Q2: `select R.a from R inner join S on R.a = S.k;`

Following our formalism for primary keys (eq. (8)), QED restricts the multiplicity of  $S(k)$  to at most one by using  $\|S(k)\|$ , and it reduces the two queries above to the following expressions:

$$Q_1(a) = \sum_k R(a) \times \|S(k)\| \times [a = k],$$

$$Q_2(a) = \sum_{k_1, k_2} R(a) \times \|S(k_1)\| \times \|S(k_2)\| \times [a = k_1] \times [a = k_2].$$

Then, during stabilization (in Sec. 5.2), we can systematically determine (using Alg. 5) the congruence classes of all the variables involved, and replace the unnecessary bounded summation variables  $k$ ,  $k_1$ , and  $k_2$  with  $a$ :

$$Q_1(a) = R(a) \times \|S(a)\| \times [a = a],$$

$$Q_2(a) = R(a) \times \|S(a)\| \times \|S(a)\| \times [a = a] \times [a = a].$$

Finally QED decides that these expressions are equivalent, since the SMT solver suggests that for all  $R$ ,  $S$ , and  $a$ , we have:

$$\begin{aligned} R(a) \times [S(a) \neq 0 \wedge a = a] \\ = R(a) \times [S(a) \neq 0 \wedge S(a) \neq 0 \wedge a = a \wedge a = a]. \end{aligned}$$

For SPES, although it has a normalization rule that removes redundant self-joins on primary keys, it fails in this case since this

is not directly a self-join in the naive join order, i.e.,  $(R \bowtie S) \bowtie S$  instead of  $R \bowtie (S \bowtie S)$ , and it cannot infer  $S_1.k = S_2.k$  from the given join conditions  $R.a = S_1.k$  and  $R.a = S_2.k$ . In contrast, we only provide the semantics of primary keys and joins separately in Sec. 3, yet QED is able to reason with cases when both features are involved. This is mainly due to QED lowering SQL semantics down to the algebraic properties of Q-expressions and first-order logical formula, allowing our formalism of different SQL features to compose much better and be systematically reasoned about.

## 7 RELATED WORK

**Formalization of SQL semantics.** Our modeling of SQL semantics is closely related to the theory of U-semiring, which is used in prior work [7], and the previously state-of-the-art tool SPES [18]. We extend this formalism with better handling of integrity constraints and Null values in QED (Sec. 3). Cosette [8] formalized K-relations in the Coq proof assistant with Homotopy Type Theory. Later work implemented additional SQL features in Coq to extend this approach, such as the support for NULL values [3, 14]. In comparison, our approach has better coverage of SQL feature, and the Q-expression formalism is designed to enable efficient encoding in modern SMT solvers.

**SQL equivalence checking.** Although general query equivalence is undecidable [16], substantial efforts have been made to develop equivalence checkers on fragments of SQL where the problem is decidable. Cosette [8] utilizes tactics in the Coq proof assistant to automatically generate proofs of query equivalence. UDP [7] normalizes the queries on the U-semiring level and then performs syntactical equivalence check. SPES [18] normalizes the queries on the relational level and then utilizes the SMT solver to verify the equivalence. QED uses the Q-expression formalization and SMT solvers for equivalence checking. In comparison, QED has the most sophisticated equivalence pipeline demonstrates the highest fidelity in supporting real-world query pairs.

**Query optimizers.** Query optimization engines, such as the ones we obtain the evaluation test suite from [2, 15], makes use of query equivalence as their foundation. Although verifying the correctness of query transformation is not the goal, query optimizers apply query rewrites heavily. And if one regard their code base as trusted, a simple verifier can make use of those query optimizers for the normalization process.

## 8 CONCLUSION

We presented QED, a new solver for checking SQL equivalence. QED applies a novel algorithm that normalizes SQL queries in stages, and we devised a fragment of SQL where equivalence can fully be determined by QED along with a general checking algorithm. We tested QED on real-world queries and show that QED can verify 2× more cases than the previous state-of-the-art tool.

## ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This work is supported in part by NSF grants IIS-1955488, IIS-2027575, CNS-1764039, CCF-2119184, DOE award DE-SC0016260, DE-SC0021982, ARO award W911NF2110339, and ONR award N00014-21-1-2724.

## REFERENCES

[1] [n. d.]. *Syntax-Guided Synthesis*. <https://sygus-org.github.io/>

[2] 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*. ACM, Houston TX USA, 221–230. <https://doi.org/10.1145/3183713.3190662>

[3] Véronique Benzaken and Evelyn Contejean. 2019. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 249–261.

[4] Apache Calcite. 2022. *Join Push Transitive Predicates Rule*. <https://github.com/apache/calcite/blob/413ed693a9087402cc1a6eefeca7a29445d337/core/src/main/java/org/apache/calcite/rel/rules/JoinPushTransitivePredicatesRule.java>

[5] Apache Calcite. 2022. *Project Join Join Remove Rule*. <https://github.com/apache/calcite/blob/413ed693a9087402cc1a6eefeca7a29445d337/core/src/main/java/org/apache/calcite/rel/rules/ProjectJoinJoinRemoveRule.java>

[6] 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*. ACM, Chicago Illinois USA, 1591–1594. <https://doi.org/10.1145/3035918.3058728>

[7] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proceedings of the VLDB Endowment* 11, 11 (July 2018), 1482–1495. <https://doi.org/10.14778/3236187.3236200>

[8] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTSQL: proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (Sept. 2017), 510–524. <https://doi.org/10.1145/3140587.3062348>

[9] Sara Cohen. 2009. Equivalence of queries that are sensitive to multiplicities. *The VLDB Journal* 18, 3 (June 2009), 765–785. <https://doi.org/10.1007/s00778-008-0122-1>

[10] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data(base) Engineering Bulletin* 18 (1995), 19–29. <https://api.semanticscholar.org/CorpusID:260706023>

[11] Goetz Graefe and William J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. *Proceedings of IEEE 9th International Conference on Data Engineering* (1993), 209–218. <https://api.semanticscholar.org/CorpusID:5793758>

[12] Todd J. Green and Val Tannen. [n. d.]. The Semiring Framework for Database Provenance. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago Illinois USA, 2017-05-09). ACM, 93–99. <https://doi.org/10.1145/3034786.3056125>

[13] Paolo Guagliardo and Leonid Libkin. [n. d.]. A formal semantics of SQL queries, its validation, and applications. 11, 1 ([n. d.]), 27–39. <https://doi.org/10.14778/3151113.3151116>

[14] Wilmer Ricciotti and James Cheney. 2022. A Formalization of SQL with Nulls. *Journal of Automated Reasoning* 66, 4 (2022), 989–1030.

[15] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieber, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruth Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. [n. d.]. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland OR USA, 2020-06-11). ACM, 1493–1509. <https://doi.org/10.1145/3318464.3386134>

[16] B. A Trahtenbrot. 1950. The impossibility of an algorithm for the decision problem for finite domains.

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

[18] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, Kuala Lumpur, Malaysia, 2735–2748. <https://doi.org/10.1109/ICDE53745.2022.000250>