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 efficiency 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 293 out of 444 query pairs extracted from the Calcite framework and 1051 out of 1400 query pairs extracted from CockroachDB, which is more than 2× the number of cases proven by prior state-of-the-art solver.

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 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.

Recently, automatic reasoning tools have been 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 by reducing down to the axioms of U-expressions. The UDP algorithm works by checking for the equality of two U-expressions after a normalization process. However, UDP is overly reliant on syntactic equality in many places where semantic 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 its subtle interactions with other relational operators, 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 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 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 compared to prior work. In sum:

- We propose a new formalism called Q-expressions that models queries based on semiring semantics and allows efficient encoding and lowering to 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 having 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:

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

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 in later stages, 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 pair of Q-expressions:

$$\begin{aligned} Q_1(x,r) &= \|\textstyle\sum_y R(x,y)\times\|S(x)\|\times[r=\mathsf{Sum}(\lambda y.\ R(x,y)\times\|S(x)\|)]\| \\ Q_2(x,r) &= \|\textstyle\sum_y R(x,y)\times\|S(x)\|\times[r=\mathsf{Sum}(\lambda y.\ R(x,y))]\| \end{aligned}$$

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

$$\begin{gathered} \forall x, r.\ \varphi_1(x,r) \leftrightarrow \varphi_2(x,r), \text{ where} \qquad (1) \\ \varphi_1(x,r) = \exists y.\ R(x,y) \neq 0 \wedge S(x) \neq 0 \wedge r = v_1, \\ \varphi_2(x,r) = \exists y.\ R(x,y) \neq 0 \wedge r = v_2 \wedge S(x) \neq 0, \\ v_1 = \mathsf{Sum}(\lambda y.\ R(x,y)\times\|S(x)\|), \qquad v_2 = \mathsf{Sum}(\lambda y.\ R(x,y)). \end{gathered}$$

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

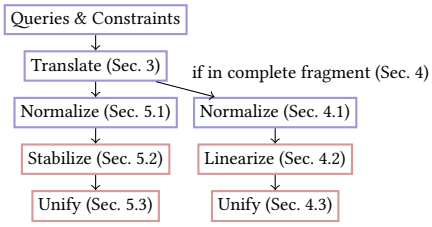

Figure 1: Overview of QED's equivalence checking pipeline, starting from top to bottom. The left lane depicts the algorithm in Sec. 5 that accepts general SQL queries, and the right lane showing the complete algorithm for a specialized fragment discussed in Sec. 4. The procedures in red boxes would invoke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

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 $\varphi_1(x,r) \vee \varphi_2(x,r)$ 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:

$$(\varphi_1(x,r) \vee \varphi_2(x,r)) \to \forall y.\ R(x,y) \times \|S(x)\| = R(x,y). \tag{2}$$

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

$$v_1 = v_2 \to (\forall x, r.\ \varphi_1(x,r) \leftrightarrow \varphi_2(x,r)). \tag{3}$$

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.

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

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

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

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

Other remaining SQL operators can be expressed using the general uninterpreted query operator, QOp. For example, to express limiting the result of Q to n rows, one can use $\mathsf{QOp}(\mathsf{Limit}, n, Q)$. Similarly, compound scalar expressions are built with Op and HOp. For example, the SQL syntax a And b can be encoded as $\mathsf{Op}(\mathsf{And}, a, b)$, and the SQL syntax a In Q (involving a subquery Q) can be encoded as $\mathsf{HOp}(\mathsf{In}, a, Q)$. Generally, we can give semantics to scalar expressions involving Op and 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 $S \to \bar{\mathbb{N}}$ 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:

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

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 $\tilde{\mathbb{N}}$ as Q-expressions. Formally, the semantics of a query Q denoted as $[\![Q]\!]$, is given inductively as:

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

Moreover, the semantics of scalar expressions is defined as:

$$\begin{aligned}
[\![\mathsf{Exists}(Q)]\!] &= \|\textstyle\sum_s [\![Q]\!](s)\|\\
[\![\mathsf{Op}(o,v_1,\dots,v_n)]\!] &= \mathsf{Op}(o,[\![v_1]\!],\dots,[\![v_n]\!])\\
[\![\mathsf{HOp}(o,v_1,\dots,v_n,Q)]\!] &= \mathsf{HOp}(o,[\![v_1]\!],\dots,[\![v_n]\!],[\![Q]\!]).
\end{aligned}$$

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

$$\Gamma \mid \Delta, s \mid \Phi \vdash [\![Q_1]\!](s) = [\![Q_2]\!](s).$$

That is, for all instantiations of the tables in Γ and variables in Δ such that Φ 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 $f\colon K \to S$ 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 $f\colon K \to S$ induces a table R with primary key on K. In fact, for any logical predicate P over a table, we have that

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

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

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

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:

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

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 must 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:

$$S \vdash S \quad \rightsquigarrow \quad S, R' \mid \forall b, k.\ \|S(b,k)\| \to \|R'(k)\| \vdash S. \tag{9}$$

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

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

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:

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

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 Null_T. For elementary types such as Integer and Boolean, we do this systematically by lifting them to the option type $\mathsf{Option}(T)$ 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-NULL values, the behavior of $\mathsf{Op}(f,a_1,\dots,a_n)$ 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. For the SQL Boolean operations And and Or, we follow three-value logic:

$$a \text{ And } b \coloneqq \text{ite}(a = \text{Null}, \quad a \text{ Or } b \coloneqq \text{ite}(a = \text{Null},$$
$$\text{ite}(b = \text{True}, \text{Null}, b), \quad \text{ite}(b = \text{False}, \text{Null}, b),$$
$$\text{ite}(a = \text{True}, b, a)), \quad \text{ite}(a = \text{False}, b, a)),$$

where ite denotes the if-then-else construct. Another special case is the Some operator, appearing in SQL as $a \langle \text{cmp} \rangle$ 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 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:

$$a \langle \text{cmp} \rangle \text{ Some } Q \coloneqq \text{ite}(\exists x.\ \|Q(x)\| \wedge (x \langle \text{cmp} \rangle a) = \text{True}, \text{True},$$
$$\text{ite}(\exists x.\ \|Q(x)\| \wedge (x \langle \text{cmp} \rangle a) = \text{Null}, \text{Null}, \text{False})).$$

In aggregations, the difference in NULL value processing (`COUNT(*)` vs `COUNT(k)`) can be modeled by optionally applying a pre-filter for NULL values 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 on 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 then prove the completeness of the procedure in Sec. 4.4.[1]

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 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. 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 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.

(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 $[\![Q]\!]$ 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*

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

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

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

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

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

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

Additionally, we impose an ordering of the table 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 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), $\text{Scp}\, T = s_1, \ldots, s_m$).
- Furthermore, for a table variable R in T, we let $\text{Scp}_R\, T$ be the sublist 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), $\text{Rel}\, T = R_1^{k_1}, \ldots, R_m^{k_m}$ and $\text{Rel}'\, T = R_1, \ldots, R_m$).

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 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 $\Delta, \mathrm{Scp}\, T \vdash P$ being a predicate, represents the new normal term based on T but with the body changed to P.

Now we show that for all $Q \in \mathscr{F}$, the Q-expression $[\![Q]\!]$ can always be rewritten into the normal form. 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

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

where 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).

Algorithm 1 Normalizing query to scoped normal form

Require: $Q \in \mathscr{F}$

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

Soundness of Alg. 1. We show by induction on the structure of any $Q \in \mathscr{F}$ that the normalization procedure of Alg. 1 preserves the semantics of the query as defined in eq. (5), i.e., under the free variable x, we have $x \vdash [\![Q]\!](x) = \mathrm{snf}(Q)$.

Case 1: if Q has the form Table(R), then:

$$[\![\mathrm{Table}(R)]\!](x) = R(x) = \sum_s [x = s] \times R(s) = \sum_{R(s)} [x = s].$$

Case 2: if Q has the form Values($v_1, \ldots, v_n$), we return the semantics given in eq. (5) as it is already in SNF.

Case 3: if Q has the form Filter(P, Q'), then

$$\begin{aligned}[\![\mathrm{Filter}(P, Q')]\!](x) &= [C] \times [\![Q']\!](x) = [C] \times T_1 + \cdots + [C] \times T_n \\ &= T_1[\mathrm{Bdy}\, T_1 \wedge C] + \cdots T_n[\mathrm{Bdy}\, T_n \wedge C],\end{aligned}$$

where $T_1, \ldots,$ and T_n are terms of snf(Q'), and $C = ([\![P]\!](x) = \mathrm{True})$.

Case 4: If Q has the form Proj(f, Q'), then we first rewrite using the fact that $\times$ and $\sum$ distributes over $+$:

$$\begin{aligned}[\![\mathrm{Proj}(f, Q')]\!](x) &= \sum_y [x = f(y)] \times [\![Q']\!](y) = [C] \times T_1 + \cdots + [C] \times T_n \\ &= \sum_y [x = f(y)] \times T_1 + \cdots + \sum_y [x = f(y)] \times T_n.\end{aligned}$$

For each of the normal term T_i of snf(Q'), it will be of the form $\sum_{\overline{R^k(s)}} [y = g(\bar{s}) \wedge P]$ by the inductive hypothesis, so we have for each of the resulting term above:

$$\begin{aligned}\sum_y [x = f(y)] \times \sum_{\overline{R^k(s)}} [y = g(\bar{s}) \wedge P] &= \sum_y \sum_{\overline{R^k(s)}} [x = f(y) \wedge y = g(\bar{s}) \wedge P] \\ &= \sum_{\overline{R^k(s)}} [x = f(g(\bar{s})) \wedge P].\end{aligned}$$

Case 5: If Q has the form Join(Q_1, Q_2), we first distributes $\times$ over $+$ to get:

$$\begin{aligned}&[\![\mathrm{Join}(Q_1, Q_2)]\!](x, x') = [\![Q_1]\!](x) \times [\![Q_2]\!](x') \\ &= T_1 \times T_1' + \cdots + T_1 \times T_m' + \cdots + T_n \times T_1' + \cdots + T_n \times T_m',\end{aligned}$$

where $T_1, \ldots, T_n$ are terms of snf(Q_1), and $T_1', \ldots, T_m'$ are terms of snf(Q_2). For each term T_i and T_j', they will be of the form $\sum_S [x = f \wedge P]$ and $\sum_{S'} [x' = f' \wedge P']$ respectively, so for every resulting term above

$$\begin{aligned}\sum_S [x = f \wedge P] \times \sum_{S'} [x' = f' \wedge P'] &= \sum_{S,S'} [x = f \wedge x' = f' \wedge P \wedge P'] \\ &= \sum_{S,S'} [(x, x') = (f, f') \wedge P \wedge P'].\end{aligned}$$

Case 6: Finally, if Q has the form Union(Q_1, Q_2), then

$$[\![\mathrm{Union}(Q_1, Q_2)]\!](x) = [\![Q_1]\!](x) + [\![Q_2]\!](x) = \mathrm{snf}(Q_1) + \mathrm{snf}(Q_2).$$

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

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

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

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

This is because that variables in $\mathrm{Scp}_R\, T_1$ or $\mathrm{Scp}_R\, T_2$ 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:

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

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

$$\sum_x [x = a] \times f(x) = f(a) \tag{11}$$

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,* $\mathrm{Bdy}\, T$ *implies the variables* $\mathrm{Scp}_R\, T$ *are pairwise-distinct and form a linearly ordered chain.*

Such property effectively fixes the ordering of $\mathrm{Scp}_R\, T$ for any R in T, and since we have already fixed the ordering of distinct table variables, the entire scope $\mathrm{Scp}\, T$ 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 $\mathsf{Lin}(X)$ 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

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

Every normal term is expanded on the possible cases given by 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

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

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 $\mathrm{Bdy}\, T$). 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.

Algorithm 2 Linearizing the normal form

Require: $\Gamma \mid \Delta \vdash U$ in SNF
1: **function** lin(U)
2: $U' \leftarrow 0$
3: **for all** term T of U **do**
4: $R_1, ..., R_n \leftarrow$ table symbols appear in T
5: **for all** $(C_1, ..., C_n) \in \mathsf{Lin}(\mathrm{Scp}_{R_1}\, T) \times \cdots \times \mathsf{Lin}(\mathrm{Scp}_{R_n}\, T)$ **do**
6: $T' \leftarrow T[C_1 \wedge \cdots \wedge C_n \wedge \mathrm{Bdy}\, T]$
7: **for all** $i \in \{1, ..., n\}$ **do**
8: $s_1, ..., s_m \leftarrow \mathrm{Scp}_{R_i}\, T'$
9: **for all** $0 < j < k \le m$ **do**
10: **if** $\Delta, \mathrm{Scp}\, T' \vdash_{\mathcal{O}} \mathrm{Bdy}\, T' \to s_j = s_k$ **then**
11: $T' \leftarrow T'[s_j/s_k]$
12: Reorder variables in $\mathrm{Scp}\, T'$ based on $C_1, ..., C_n$
13: $U' \leftarrow U' + T'$
14: **return** U'

Soundness of Alg. 2. The main loop in line 3 perform linearization by equivalently rewriting each term T into multiple terms. We first case split T over the set of linear ordering conditions (each being $C_1 \wedge \cdots \wedge C_n$) in line 6, since the conditions are pair-wise disjoint, and the disjunction of them is the trivially true condition. This is sound since in general for two propositions P_1 and P_2 that are disjoint, i.e., $\neg(P_1 \wedge P_2)$, we can split a disjunction of them with

$$[P_1 \vee P_2] = [P_1] + [P_2].$$

We then perform redundant variable elimination for each expanded sub-term T', by going through every (unordered) pair of variables (s_j, s_k) that are applied to the same table symbol and checking whether the body of T' implies the equality of them (line 10). If so, we rewrite T' into the equivalent $T'[s_j/s_k]$ (using the variable substitution notation as defined in Sec. 4.1), which is sound as the condition $s_j = s_k$ allows us to apply eq. (11). At this point, the algorithm ensures all variables in $\mathrm{Scp}_R\, T'$ are pairwise distinct, and we further reorder them based on the linear ordering condition for T' to ensure they form a chain, making T' a valid LNF term.

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

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

The two LNFs are equivalent since

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

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

$$\textstyle\sum_{\overline{R^k(s)}} [P_1] + \cdots + [P_n] \doteq \sum_{\overline{R^k(s)}} [P'_1] + \cdots + [P'_m],$$

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

$$\bar{s} \vdash_{\mathcal{O}} [P_1] + \cdots + [P_n] \doteq [P'_1] + \cdots + [P'_m].$$

The $[\bullet]$ 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 fully partitioned LNF pair

Require: $\Gamma \mid \Delta \vdash U_1, U_2$ in LNF
1: **function** 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: $\sum_S [P_1], ..., \sum_S [P_n] \leftarrow$ terms T of U_1 where $\mathrm{Rel}\, T = S$
5: $\sum_S [P'_1], ..., \sum_S [P'_m] \leftarrow$ terms T' of U_2 where $\mathrm{Rel}\, T' = S$
6: **if** $\Delta, \bar{s} \nvdash_{\mathcal{O}} [P_1] + \cdots + [P_n] = [P'_1] + \cdots + [P'_m]$ **then**
7: **return** False
8: **return** True

Soundness of Alg. 3. Alg. 3 performs the equivalence check in a group-by-group manner, where each group is formed by terms with

the same relational scope. Such reduction from the equivalence of LNFs to the equivalence of groups is sound since every term belongs to exactly one group (by their relational scope), and so we avoid over-counting and under-counting. For a specific relational scope S (and summation variables $\bar{s}$ in S), we have

$$\sum_S[P_1] + \cdots + \sum_S[P_n] = \sum_S([P_1] + \cdots + [P_n]),$$
$$\sum_S[P'_1] + \cdots + \sum_S[P'_m] = \sum_S([P'_1] + \cdots + [P'_m]),$$

respectively for terms in the group from U_1 and U_2. And then group equality would be implied by the equality of the summations' body,

$$\forall \bar{s}.\ [P_1] + \cdots + [P_n] = [P'_1] + \cdots + [P'_m],$$

as being validated by the oracle at line 6. Finally, the algorithm only returns True when all group equality checks are passed, and thus the algorithm is sound.

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:*

$$[\![Q_1]\!] = [\![Q_2]\!] \rightarrow \mathsf{unify}(\mathsf{linear}(\mathsf{norm}(Q_1), \mathsf{linear}(\mathsf{norm}(Q_2)).$$

Proof. We prove the contrapositive by first assuming the equivalence checking procedure returns false. Let $s \vdash U_1$ and $s \vdash U_2$ be the LNFs of $s \vdash [\![Q_1]\!](s)$ and $s \vdash [\![Q_2]\!](s)$ respectively, 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. That is, for any relational scope S, there is a pair of groups of terms

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

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

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

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., $S^{*\prime} = \mathrm{Rel}'\, T$ for some $T \in G_{S^*}$), and S^*_R for the variables in S^* that are applied to some table symbol R (i.e., $S^*_R = \mathrm{Rel}_R\, T$ for some $T \in G_{S^*}$).

We now construct an instance of Δ and a family of instances of Γ 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 δ of Δ and σ of the summation variables in S^* under which the bodies of the group are different

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

The instantiation σ 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 σ_R to denote the subsequence of σ which form the instantiation of the variables in S^* that are applied to R.

Now suppose there are m values in the sequence σ, and we will construct a family of instances of Γ over the set $N = \mathbb{N}^m$. Concretely, over the index $n \in N$ we construct an instantiation γ of Γ, 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 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 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 $\mathrm{Rel}'\, T = S^{*\prime}$. Otherwise for some term $T \neq 0$ with $\mathrm{Rel}'\, T \neq S^{*\prime}$, there must exist some table variable R such that $\mathrm{Scp}_R\, T$ 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 $s \in \mathrm{Scp}_R\, T$ 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.

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 $n_1, \ldots, n_m$, 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 γ over N, there must be some instantiation for which $P_1 \neq P_2$, and hence $U_1 \neq U_2$. □

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 $\mathcal{O}$, 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, $\mathcal{O}$ may give three

results: satisfiable, not satisfiable, or unknown. And as presented later in Sec. 6, SMT solvers serve the role of the oracle $\mathcal{O}$ in QED's implementation. We use the notation

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

to denote whether in $\mathcal{O}$, checking the formula $\Phi \wedge \neg P$ with uninterpreted symbols from Γ and Δ returns the "not satisfiable" result.

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

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

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 $\mathrm{QOp}(o, v_1, \ldots, v_n, Q)$) with some expression, namely

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

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.

Algorithm 4 Converting to sum–product normal form

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

```
1:  function spnf(U)
2:      U' ← 0
3:      if U is of the form U1 + U2 then
4:          return spnf(U1) + spnf(U2)
5:      else if U is of the form U1 × U2 then
6:          for all term Σ_{s̄1}[L1] × V1 of spnf(U1) do
7:              for all term Σ_{s̄2}[L2] × V2 of spnf(U2) do
8:                  U' ← U' + Σ_{s̄1,s̄2}[L1 ∧ L2] × V1 × V2
9:      else if U is of the form Σ_s̄ V then
10:         for all term T of spnf(V) do
11:             U' ← U' + Σ_s̄ T
12:     else if U is of the form ‖V‖ then return [spnf(V) ≠ 0]
13:     else if U is of the form ¬V then return [spnf(V) = 0]
14:     else return U
15:     return U'
```

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 $\sum_{\bar{s}}[L] \times V$, 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:

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

would be normalized into

$$\textstyle\sum_y R(x) \times S(y) + \sum_y R(x) \times R(y) + \sum_z [S(z) \neq 0] \times R(x) + \sum_z [P(z)] \times R(x).$$

One may notice that the form $[U \neq 0]$ generated when we encounter a squash operator $\|\bullet\|$ (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 $\mathcal{O}$, seemingly requiring $\mathcal{O}$ to already have the power of deciding equality in the theory of Q-expressions, and defeating the point of our own equivalence checking algorithm! Instead, using the following rewrites,

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

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.,

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

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

Soundness of Alg. 4. Here we show by induction that for any Q-expression U, $U = \mathsf{spnf}(U)$. Addition of Q-expressions becomes the concatenation of their SPNFs (line 3), since

$$\mathsf{spnf}(U_1 + U_2) = \mathsf{spnf}(U_1) + \mathsf{spnf}(U_2) = U_1 + U_2.$$

For multiplication, we perform a pair-wise combination of terms using the distribution of $\times$ over $+$ (line 5). Formally, we first have

$$\begin{aligned} U_1 \times U_2 &= \mathsf{spnf}(U_1) \times \mathsf{spnf}(U_2) \\ &= T_1 \times T_1' + \cdots + T_1 \times T_n' + \cdots + T_n \times T_1' + \cdots + T_n \times T_n', \end{aligned}$$

where T_1, ..., T_n are terms of $\mathsf{spnf}(U_1)$, and T_1', ..., T_n' are terms of $\mathsf{spnf}(U_2)$. For each term T_i and T_j', they will be of the form $\sum_{\bar{s}}[L] \times V$ and $\sum_{\bar{s'}}[L'] \times V'$ respectively, so for every resulting term above:

$$\sum_{\bar{s}}[L] \times V \times \sum_{\bar{s'}}[L'] \times V' = \sum_{\bar{s},\bar{s'}}[L \wedge L'] \times V \times V'.$$

And that matches line 8, hence $U_1 \times U_2 = \mathsf{spnf}(U_1 \times U_2)$.

For the case of summation, we can push down the summation operator into each normal term as $\sum$ distributes over $+$ (line 9), since

$$\sum_{\bar{s}} U = \sum_{\bar{s}}(T_1 + \cdots + T_n) = \sum_{\bar{s}} T_1 + \cdots + \sum_{\bar{s}} T_n = \mathsf{spnf}(\sum_{\bar{s}} U),$$

where T_1, ..., T_n are terms of $\mathsf{spnf}(U)$. For squash $\|\bullet\|$ and negation $\neg$, we simply rewrite them using the definition given in eq. (4) (line 12 and 13), as then they are automatically in SPNF.

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 equivalence problem of SPNFs.

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

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 $\sum_{\vec{x}}[L] \times V$, 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 $L \to P$. 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 stabilization of SPNF as shown in Alg. 5.

Algorithm 5 SPNF Stabilization

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

```
1:  function stable(U)
2:      U' ← 0
3:      for all term Σ_s̄ [L] × V of U do
4:          s̄' ← s̄, L' ← L, V' ← V
5:          for all variable s_i ∈ s̄ do
6:              s̄'' ← s̄' \ s_i                ▷ Remove s_i from the variable list s̄'
7:              ▷ Eliminate s_i if a dependency over others is found. ◁
8:              if ∃(Δ ⊢ f). Γ | Δ, s̄' | Φ ⊢_𝒪 L' → f(s̄'') = s_i then
9:                  s̄' ← s̄'', L' ← L'[f(s̄'')/s_i], V' ← V'[f(s̄'')/s_i]
10:         U' ← U' + Σ_s̄' [L'] × V'
11:     return U'
```

Every variable s_i of a term is checked for redundancy by searching for some functional dependency f over the set of variables $\vec{s}''$ 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.

Soundness of Alg. 5. Since stabilization is done on a term-by-term basis, the soundness of Alg. 5 amounts to the variable elimination rewrites done at each iteration of the loop at line 3 preserve the equality of Q-expression. In line 8, we look for a function f under the context Δ, such that $\Gamma \mid \Delta, \vec{s}' \mid \Phi \vdash_{\mathscr{O}} L' \to f(\vec{s}'') = s_i$. If such f exists, the occurrence of s_i in the term can be substituted with $f(\vec{s}'')$ and we eliminate the unnecessary summation over s_i as in line 3 of Alg. 5, which is justified by

$$\sum_{\vec{s}'}[L'] \times V' = \sum_{s_i, \vec{s}''}[s_i = f(\vec{s}'')] \times [L'] \times V' = \sum_{\vec{s}''}[L'[f(\vec{s}'')/s_i]] \times V'[f(\vec{s}'')/s_i].$$

This operation is then exhaustively performed for each summation variable s_i, but since at each step the equality is preserve, the entire rewrite generated by the loop is sound.

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 equated terms are cancelled out (line 7) to ensure that every term of U_1 is matched with another distinct term of U_2. Now we are ready to consider the equality between two normal terms $\sum_{\vec{s_1}}[L_1] \times V_1$ and $\sum_{\vec{s_2}}[L_2] \times V_2$ where $\vec{s_1}$ and $\vec{s_2}$ 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.,

$$(\forall \vec{s}.\ f(\vec{s}) = g(\vec{s})) \to \textstyle\sum_{\vec{s}} f(\vec{s}) = \sum_{\vec{s}} g(\vec{s}). \tag{14}$$

But here we have two distinct sets of summation variables $\vec{s_1}$ and $\vec{s_2}$ when comparing the normal terms, and commuting the summation variables of a summation does not change its value. Hence, before we align the scope of the normal terms and check for the universal quantification in (14), we need to first fix a bijective pairing of the variables $\vec{s_1}$ and $\vec{s_2}$ by enumerating all possible permutations of $\vec{s_2}$ over $\vec{s_1}$. For a certain permutation σ, we can apply that to the right-hand term and reduce the term equality down to equality of term bodies:

$$\Gamma \mid \Delta, \vec{s_1} \mid \Phi \vdash_{\mathscr{O}} [L_1] \times V_1 = [L_2[\sigma]] \times V_2[\sigma]. \tag{15}$$

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 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 upfront 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

$$\Gamma \mid \Delta \mid \Phi \vdash [L_1] \times V_1 = [L_2] \times V_2 \tag{16}$$

However, as we may have higher-order operators (introduced by QOp and HOp) that contains *functions* returning $\tilde{\mathbb{N}}$ as their operands, we cannot directly translate such higher-order construct into a first-order logical formula as presented in eq. (16). Instead, we first substitute every higher-order query expression $\mathsf{QOp}(o, v_1, ..., v_n, Q)$ with a fresh table variable R, and every such scalar expression $\mathsf{HOp}(o, v_1, ..., v_n, Q)$ with a fresh variable x. A priori, two syntactically different expressions such as $\mathsf{QOp}(o, v_1, ..., v_n, \lambda x.\ W)$ and $\mathsf{QOp}(o', v_1', ..., v_n', \lambda x.\ W')$ 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 additional assumptions* (namely $L_1' \vee L_2'$ in line 22 of Alg. 6). Intuitively, we justify this by looking at the other case where $\neg L_1' \wedge \neg L_2'$, 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 that case (see eq. (1)), we first introduce fresh variables for the aggregation functions:

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

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 here allows us to temporarily made 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:

$$\Gamma' \mid \Delta' \mid \Phi \wedge (L_1' \vee L_2') \vdash C, \quad \Gamma' \mid \Delta' \mid \Phi \wedge C \vdash [L_1'] \times V_1' = [L_2'] \times V_2' \tag{17}$$

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 Γ' denotes the context extended from Γ with the newly introduced table variables, and similarly Δ' from Δ 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 Γ', we should be ranging over all of $S \to \tilde{\mathbb{N}}$ 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 restricted to a function of type $S \to \mathbb{N}$. 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 $\times$, $[\bullet]$, $\|\bullet\|$, and $\neg$ operators can be restricted to only operate on finite multiplicities. Any oracle $\mathcal{O}$ supporting the Peano arithmetic can then be used to reason about the equality in (17) since we can now encode the summation bodies in a straightforward manner.

Finally, we have the equivalence checking procedure equiv (line 26) by composing SPNF normalization, stabilization, and unification.

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 Σ_{s̄:[L]×V} of U_1 do
3:          if Γ | Δ, s̄ | Φ ⊢_O ¬(L ∧ ‖V‖) then Remove T from U_1
4:      for all term T of the form Σ_{s̄:[L]×V} of U_2 do
5:          if Γ | Δ, s̄ | Φ ⊢_O ¬(L ∧ ‖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 ← False
9:          for all term T_2 of U_2 do
10:             if termEq(Γ | Δ | Φ ⊢ T_1 ≐ T_2) then
11:                 Remove T_2 from U_2
12:                 M ← True
13:                 break
14:         if ¬M then return Unknown
15:     return True
16: function termEq(Γ | Δ | Φ ⊢ Σ_{s̄_1}[L_1] × V_1 ≐ Σ_{s̄_2}[L_2] × V_2)
17:     if s̄_1 and s̄_2 have different length then return False
18:     for all bijective substitution σ of s̄_2 with s̄_1 do
19:         L_2, V_2 ← L_2[σ], V_2[σ]                ▷ Apply substitution σ to L_2, V_2
20:         L_1', V_1', L_2', V_2' ← L_1, V_1, L_2, V_2 with higher-order operators substituted with fresh variables.
21:         Γ', Δ' ← Γ, Δ extended with the fresh variables.
22:         C ← congruence relation of the fresh variables through recursive unify under Γ' | Δ', s̄_1 | Φ ∧ (L_1' ∨ L_2')
23:         if Γ' | Δ', s̄_1 | Φ ∧ C ⊢_O [L_1'] × V_1' = [L_2'] × V_2' then
24:             return True
25:     return False
26: function equiv(Q_1, Q_2)
27:     return unify(stable(spnf(⟦Q_1⟧)), stable(spnf(⟦Q_2⟧)))
```

Soundness of Alg. 6. We start by removing terms that are trivially 0 in line 2 and 4 as they do not affect the equivalence problem in any way. The early return in line 5 does not affect soundness since only Unknown can be returned. The loop in line 7 then reduces the equivalence of SPNFs down to the equivalence of normal terms, by simultaneously canceling equivalent terms at both sides and only returns True when there are no terms left.

Now we focus on checking term equivalence, which is discussed in detail in Secs. 5.3.1 to 5.3.3. As described in Sec. 5.3.1, term equivalence is reduced to (any of the multiple) equivalence of their body for terms with matching scopes, justified by the commutativity of variables introduced by $\sum$. And for term body equivalence, instead of checking eq. (16), we instead use the technique developed in Sec. 5.3.2 to check eq. (17) at line 23 to intelligently handle nested occurrence of SPNFs introduced by unknown query operators.

The soundness of this technique is justified by the following chain of implications starting from eq. (17) to eq. (16). First by

applying modus ponens, the conditions in eq. (17) imply that

$$\Gamma' \mid \Delta' \mid \Phi \vdash L_1' \vee L_2' \to [L_1'] \times V_1' = [L_2'] \times V_2'.$$

Then undoing the substitution of higher-order operators as fresh variables, we get back an equi-satisfiable condition

$$\Gamma \mid \Delta \mid \Phi \vdash L_1 \vee L_2 \to [L_1] \times V_1 = [L_2] \times V_2.$$

The above implies our goal eq. (16) since even in the case of both L_1 and L_2 being false, $[L_1] \times V_1$ and $[L_2] \times V_2$ both reduce to zero and their equality trivially holds.

The encoding of eq. (17) into an SMT formula can be soundly perform as the Q-expressions involved can be restricted to the domain of $\mathbb{N}$, and all the operations on Q-expressions correctly translated into Peano arithmetic as described in Sec. 5.3.3. Hence the soundness of term equivalence is justified, and based on the previous reduction, the soundness of unify. Finally, since the full algorithm equiv is the composition of spnf, stable, and unify, by combining all the prior soundness results we can conclude equiv is sound.

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 sharing 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) to filter out the 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. On average, the queries in the new Calcite test suite contain 5.6 query operator (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 0.87s 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 of Qed is minimal. The same observation holds for the CockroachDB test suite, where 0.28s are spent to eliminate redundant variables and 0.02s are spent to check term equivalence within the 0.31s 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 0.14s and SPES spends 0.02s. Looking over the full distribution in Fig. 3, QED does perform slower than SPES on the commonly provable cases, although the slowdown is mostly within one order of magnitude. Meanwhile, for the subset of test cases where Qed can verify but SPES cannot, Qed has similar performance as the rest of test cases. This suggests that Qed does not sacrifice its performance for the new features, such as the support for NULL in queries.

Test Suite		**Tool**				
Name	Stat	UDP	EQUITAS	SPES	Qed′	Qed
Calcite - Old	Provable (#)	34	67	95	49	147
(232 total)	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
(444 total)	Avg. Time (s)			0.99 (10.3)	0.24 (0.32)	0.87 (8.21)
CockroachDB	Provable (#)	_	_	325	705	1086
(1400 total)	Avg. Time (s)			0.02 (0.02)	0.27 (1.58)	0.31 (2.15)

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	Qed				SPES
	CONG	EQ	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	0.28	0.02	0.14	0.14	0.02
Avg. Time (s)	(2.13)	(0.03)	(0.16)	(0.63)	(0.02)

Table 2: Breakdown of Qed average runtime. Numbers in parenthesis are standard deviations. We list the average time Qed spent on SMT solvers (CONG when used in Alg. 5 and EQ when used in Alg. 6), the average total runtime for Qed on test cases where SPES fails (New), as well as the average total runtime for Qed (Shared) and SPES (SPES) on test cases where both can prove.

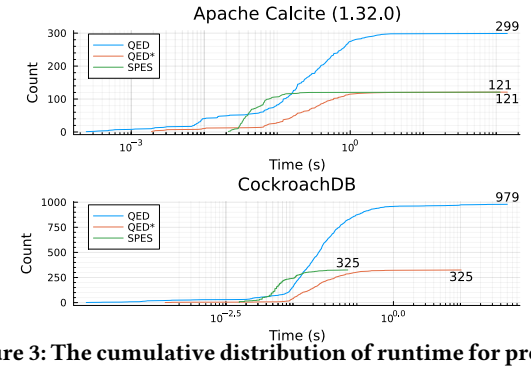

Figure 3: The cumulative distribution of runtime for provable cases by SPES and QED. For a given time t, the corresponding case count is the number of cases proved under t by each tool. QED* denotes the runtime of QED, but restricted to only the cases that are also provable by SPES.

6.2 Failure breakdown

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

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

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

- *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 involving the SEARCH operator in Calcite, which determines if a value is in certain ranges, as we do not model its semantics.
- *List semantics.* We use bag semantics to model queries instead of list semantics, hence QED currently does not support queries that use `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. This example exposes a weakness of SPES's approach, which is essentially capturing SQL semantics using a special set of rewrite rules on the query syntax level. This process can often lead to incomplete modeling, and leave many possible places to make mistakes illustrated here, as one need to consider how every query operator interacts with every other operators and codify each interaction as a rewrite rule. In contrast, after reducing query semantics down to a relatively minimal representation like the Q-expressions, we have a much smaller set of normalization rules to implement and less chance to introduce unsoundness.

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** S_1 **on** R.a = S_1.k
inner join S **as** S_2 **on** R.a = S_2.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:

$$R(a) \times [S(a) \neq 0 \wedge a = a] = R(a) \times [S(a) \neq 0 \wedge S(a) \neq 0 \wedge a = a \wedge a = a].$$

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 that $S_1.k = S_2.k$ 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.

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.

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

[4] Apache Calcite. 2022. *Join Push Transitive Predicates Rule.* https://github.com/apache/calcite/blob/413eded693a9087402cc1a6eefeca7a29445d337/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/413eded693a9087402cc1a6eefeca7a29445d337/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. HoTTSQL: 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 Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta 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/ICDE53745.2022.00250