Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries

Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, Dan Suciu

Paul G. Allen School of Computer Science and Engineering
University of Washington
{chushumo, jroesch, akcheung, suciu}@cs.washington.edu, bsmurphy@uw.edu

ABSTRACT

Deciding the equivalence of SQL queries is a fundamental problem in data management. As prior work has mainly focused on studying the theoretical limitations of the problem, very few implementations for checking such equivalences exist. In this paper, we present a new formalism and implementation for reasoning about the equivalences of SQL queries. Our formalism, U-semiring, extends SQL's semiring semantics with unbounded summation and duplicate elimination. U-semiring is defined using only very few axioms and can thus be easily implemented using proof assistants such as Lean for automated query reasoning. Yet, they are sufficient enough to enable us reason about sophisticated SQL queries that are evaluated over bags and sets, along with various integrity constraints. To evaluate the effectiveness of U-semiring, we have used it to formally verify 68 equivalent queries and rewrite rules from both classical data management research papers and real-world SQL engines, where many of them have never been proven correct before.

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

1. INTRODUCTION

All modern relational database management systems (DBMSs) contain a query optimizer that chooses the best means to execute an input query. At the heart of an optimizer is a rule-based rewrite system that uses rewrite rules to transform the input SQL query into another (hopefully more efficient) query to execute. A key challenge in query optimization is how to ensure that the rewritten query is indeed semantically equivalent to the input, i.e., that the original and rewritten queries return the same results when executed on all possible input database instances. History suggests that the lack of tools to establish such equivalences has caused long-standing, latent bugs in major database systems [32], and new bugs continue to arise [7, 10]. All such errors lead to incorrect results returned that can cause dire consequences.

Proceedings of the VLDB Endowment, Vol. 11, No. 11

DOI: https://doi.org/10.14778/3236187.3236200

In the past, query optimizers were developed by only small number of commercial database vendors with dedicated teams to handle customer reported bugs. Such teams effectively acted as "manual solvers" for query equivalences. The explosive growth of new data analytic systems in recent decades (e.g., Spark SQL [12], Hive [4], Dremel [42], Myria [39, 55], among many others) has unfortunately exacerbated this problem significantly, as many such new systems are developed by teams that simply lack the dedicated resources to check every single rewrite implemented in their system. For example, the Calcite open source query processing engine [1] is mainly developed by open-source contributors. While it contains more than 232 rewrite rules in its optimization engine, none of which has been formally validated. The same holds true for similar engines.

On the other hand, deciding whether two arbitrary SQL queries are semantically equivalent is a well-studied problem in the data management research community that has shown to be undecidable in general [11]. Most subsequent research has been directed at identifying fragments of SQL where equivalence is decidable, under set semantics [17, 47] or bag semantics [24]. This line of work, focused on theoretical aspects of the problem [51, 47, 41], led to very few implementations, most of which were restricted to applying the chase procedure to conjunctive queries [14].

An alternative approach to query equivalence was recently proposed by the COSETTE system [23] based on a new SQL semantics. It interprets SQL relations as K-relations [35]. A K-relation is semiring that maps each tuple t to a value $R(t)$ that denotes the multiplicity of the tuple in the relation. Normally, the multiplicity is an integer, thus the semiring K is the semiring of natural numbers $\mathbb{N}$, but in order to support some of the SQL constructs (such as projections, with or without `DISTINCT`), COSETTE resorts to using a much more complex semiring, namely that of all univalent types in Homotopy Type Theory (HoTT) [50]. Equivalence of two SQL queries is then proven in COSETTE using the Coq proof assistant extended with the HoTT library [36]. While the system can handle many features of SQL, its reliance on HoTT makes it difficult to extend as HoTT is both a Coq library under development and an active research area itself. As a result, COSETTE has a number of shortcomings. For example, COSETTE does not support foreign keys as they are difficult to model using the HoTT library in Coq.

In this paper, we propose a new algebraic structure, called the *unbounded semiring*, or *U-semiring*. We define the U-semiring by extending the standard commutative semiring with a few simple constructs and axioms. This new algebraic structure serves as a foundation for our new formalism that models the semantics of SQL. To prove the semantic equivalence of two SQL queries, we first convert them into U-semiring expressions, i.e., *U-expressions*. Deciding the equivalence of queries then becomes determining the

equivalence of two U-expressions which, as we will show, is much easier compared to classical approaches.

Our core contribution is identifying the minimal set of axioms for U-semirings that are sufficient to prove sophisticated SQL query equivalences. This is important as the number of axioms determines the size of the trusted code base in any proof system. As we will see, the few axioms we designed for U-semirings are surprisingly simple as they are just identities between two U-expressions. Yet, they are sufficient to prove various advanced query optimization rules that arise in real-world optimizers. Furthermore, we show how integrity constraints (ICs) such as keys and foreign keys can be expressed as U-expressions identities as well, and this leads to a single framework that can model many different features of SQL and the relational data model. This allows us to devise different algorithms for deciding the equivalences of various types of SQL queries including those that involve views, or leverage ICs to rewrite the input SQL query as part of the proof.

To that end, we have developed a new algorithm, UDP, to automatically decide the equivalence of two arbitrary SQL queries that are evaluated under mixed set/bag semantics,[1] and also in the presence of indexes, views, and other ICs. At a high level, our algorithm performs rewrites reminiscent to the chase/back-chase procedure [45] but uses U-expressions rather than first order logic sentences. It performs a number of tests for isomorphisms or homomorphisms to capture the mixed set/bag semantics of SQL. Our algorithm is sound in general and is complete in two restricted cases: when the two queries are Unions of Conjunctive Queries (UCQ) under set semantics, or UCQ under bag semantics. We implement UDP on top of the Lean proof assistant [26]. The implementation includes the modeling of relations and queries as U-expressions, axiomatic representations of ICs as simple U-expression identities, and the algorithm for checking the equivalence of U-expressions.

We evaluate UDP using various optimization rules from classical data management research literature and as implemented in the Apache Calcite framework [1]. These rules consist of sophisticated SQL queries with a wide range of features such as subqueries, grouping and aggregate, DISTINCT, and integrity constraints.In fact, only 1 of them have been proven before. UDP can formally and automatically prove most of them (39 of 45). The running time of UDP on each of these 39 rules is within 30 seconds. Our system only proves *equivalence*. In prior work [22] we described the complementary task that uses a model checker to find counterexamples to identify buggy rewrites [32, 7, 10]; of course, the model checker cannot prove equivalence, which is our current focus.

In summary, our paper makes the following contributions:

- We describe a new algebraic structure, the U-semiring, which extends the standard semiring with the necessary operators to model the semantics of a wide range of SQL queries (Sec. 3).
- We propose a new formalism for expressing different kinds of integrity constraints over a U-semiring. We implement such constraints using a number of axioms (in the form of simple identities) such that they can be easily implemented using a proof assistant. Doing so allows us to easily utilize them in equivalence proofs (Sec. 4).
- We describe a new algorithm for deciding the equivalences of different types of SQL queries. The algorithm operates entirely on our representation of SQL queries as U-expressions. We show that this algorithm is sound for general SQL queries and is complete for Unions of Conjunctive Queries under set and bag semantics (Sec. 5).
- We have implemented UDP using the Lean proof assistant, and have evaluated UDP by collecting 69 real-world rewrite rules as benchmarks, both from prior research work done in the database research community, and from Apache Calcite [1], an open-source relational query optimizer. To our knowledge, the majority of these rules have been never proven correct before, while UDP can automatically prove 62 of 68 correct ones and fail as intended on the 1 buggy one (Sec. 6).

[1]Mixed set/bag semantics is the bag semantics that allows explicit DISTINCT on arbitrary subqueries, bag semantics and set semantics are special cases of mixed set/bag semantics.

Equivalent SQL Queries

```
SELECT * FROM R t WHERE t.a >= 12   -- Q1

SELECT t2.* FROM I t1, R t2         -- Q2
WHERE t1.k = t2.k AND t1.a >= 12
```

where k is a key of R, and I is an index on R defined as:

```
I := SELECT t3.k AS k, t3.a AS a FROM R t3
```

Corresponding Equivalence in Semirings

$$Q_1(t) = \lambda\, t.\ [\![R]\!](t) \times [t.a \geq 12]$$

$$Q_2(t) = \lambda\, t.\ \sum_{t_1,t_2,t_3} [t_2 = t] \times [t_1.k = t_2.k] \times [t_1.a \geq 12] \times [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times [\![R]\!](t_3) \times [\![R]\!](t_2)$$

Figure 1: Proving that a query is equivalent to a rewrite using an index I requires proving a subtle identity in a semiring.

2. OVERVIEW

We motivate our new semantics using an example query rewrite. As shown in Fig. 1, the rewrite changes the original query Q1 by using an index I for look up. We follow the GMAP framework [52], where an index is considered as a view definition, and a query plan that uses the index I to access the relation R is represented logically by a query that selects from I then joins on the key of R.

Our goal is to devise a semantics for SQL along with various integrity constraints that can be easily implemented as a tool for checking query equivalences. Unfortunately, the SQL standard [25] is expressed in English and is difficult to implement programmatically. One recent attempt is Q*cert [13], which models SQL using NRA (Nested Relational Algebra). NRA is implemented using the Coq proof assistant, with relations modeled using lists. To prove that Q_1 and Q_2 are equivalent in Q*cert, users need to prove that the output from the two are equal up to element reordering and duplicate elimination (for set semantics). As Q*cert models relations as lists, this amounts to writing an inductive proof on the size of R, i.e., if R is empty, then Q_1 outputs the same relation as Q_2; if R is of size n and the two outputs are equivalent, then the two outputs are also equivalent if R is of size $n + 1$. Writing such proofs can be tedious. As an illustration, Q*cert requires 45 lines of Coq to prove that selection can be distributed over union [9] while, as we will see, using U-semiring only requires 1 line of Lean: distribute multiplication over addition, which is one of the semiring axioms.

Another formalism proposed by Guagliardo et al. [38] models relations as bags. While it models NULL semantics, it (like Q*cert) does not model integrity constraints, which are widely used in almost all database systems and are involved in the rewrites in many real world optimizers. There is no known algorithm based on their formalism to automatically check the equivalence of SQL queries.

We instead base our semantics on K-relations, which was first proposed by Green et al [35]. Under this semantics, a relation is

modeled as a function that maps tuples to a commutative semiring, $\mathbf{K} = (K, 0, 1, +, \times)$. In other words, a K-relation, R, is a function:

$$[\![R]\!] : \texttt{Tuple}(\sigma) \to K$$

with finite support. Here, $\texttt{Tuple}(\sigma)$ denotes the (possibly infinite) set of tuples of type σ, and $[\![R]\!](t)$ represents the multiplicity of t in relation R. For example, a relation under SQL's standard bag semantics is an $\mathbb{N}$-relation (where $\mathbb{N}$ is the semiring of natural numbers), and a relation under set semantics is a $\mathbb{B}$-relation (where $\mathbb{B}$ is the semiring of Booleans). All relational operators and SQL queries can be expressed in terms of semiring operations; for example:

$$[\![\texttt{SELECT * FROM } R\ x,\ S\ y]\!] = \lambda\,(t_1, t_2)\ .\ [\![R]\!](t_1) \times [\![S]\!](t_2)$$

$$[\![\texttt{SELECT * FROM } R \texttt{ WHERE } a > 10]\!] = \lambda\,t\ .\ [t.a > 10] \times [\![R]\!](t)$$

$$[\![\texttt{SELECT } a \texttt{ FROM } R]\!] = \lambda\,t.\ \sum_{t' : \texttt{Tuple}(\sigma)} [t'.a = t] \times [\![R]\!](t')$$

For any predicate b, we denote $[b]$ the element of the semiring defined by $[b] = 1$ if the predicate holds, and $[b] = 0$ otherwise.

K-relations can be used to prove many simple query rewrites by reducing query equivalences to semiring equivalences, which are much easier since one can use algebraic reasoning rather than proofs by induction. However, for sophisticated rewrites like the one shown in Fig. 1, K-relations are not sufficient to prove query equivalences, let alone automate the proof search. One problem is that K-relation does not model integrity constraints (keys, foreign keys, etc.), which are usually expressed as generalized dependencies [11] (i.e., logical formulas of the form $\forall \mathbf{x}(\varphi \Rightarrow \exists \mathbf{y} \psi)$). For example, consider the two queries in Fig. 1: Q_1 is a selection on $R.a \geq 12$, while Q_2 rewrites the given query using an index I. These two queries are indeed semantically equivalent in that scanning a table using a given attribute (a in the example) is the same as performing an index scan on the same attribute. However, consider their corresponding expressions over K-relations, shown at the bottom of Fig. 1; it is unclear how to express formally the fact that $R.k$ is a key, yet alone prove automatically that these two expressions are equal.

To extend K-relations for *automatically* proving SQL equivalences under database integrity constraints, we develop a novel algebraic structure, U-semiring, as the semiring in K-relations. A U-semiring extends a semiring with three new operators, $\sum$, $\|\cdot\|$, $\texttt{not}(\cdot)$, and a minimal set of axioms, each of which is a simple identity[2] that is easy to implement using a proof assistant. We also discover a set of U-semiring identities that models database integrity constraints such as keys and foreign keys. With these additions, SQL equivalences can be reasoned by checking the equivalences of their corresponding U-expressions using only these axioms expressed in U-semiring identities.

Using our semantics, the rewrite shown in Fig. 1 can be proved by rewriting the Q_2 into the Q_1 using U-semiring axioms in three steps. First, the sum over t_1 can be removed by applying the axiom of the interpretation of equality over summation (Eq. (15)):

$$\lambda\,t.\ \sum_{t_2, t_3} [t_2 = t] \times [t_3.k = t_2.k] \times [t_3.a \geq 12] \times R(t_3) \times R(t_2)$$

Since k is a key of R, applying the U-semiring definition of the key constraint (Def. 4.1) we get:

$$[t_3.k = t_2.k] \times [\![R]\!](t_3) \times [\![R]\!](t_2) = [t_3 = t_2] \times [\![R]\!](t_3)$$

[2]An *axiom* is a logical sentence, such as $\forall x R(x) \Rightarrow S(x)$. An *identity*, or an *equational law*, is an equality, such as $x + y = y + x$. The implication problem for identities is much simpler than for arbitrary axioms. Traditional generalized dependencies [11] are expressed as axioms, $\forall \mathbf{x}(\varphi \Rightarrow \exists \mathbf{y} \psi)$, and only apply to queries under set semantics.

$h \in \texttt{Program}$	::=	$s_1; \ldots; s_n;$
$s \in \texttt{Statement}$	::=	verify $q_1 \equiv q_2$
	|	schema $\sigma(a_1 : \tau_1, \ldots, a_n : \tau_n)$
	|	table $r(\sigma)$
	|	key $r(a_1, \ldots, a_n)$;
	|	foreign key $r_1(a'_1, \ldots, a'_n)$ references $r_2(a_1, \ldots, a_n)$;
	|	view v q;
	|	index i on $\sigma(a_1, \ldots, a_n)$;
$a \in \texttt{Attribute}$	::=	string
$q \in \texttt{Query}$	::=	r
	|	SELECT p q
	|	FROM q_1 $x_1, \ldots, q_n$ x_n
	|	q WHERE p
	|	q_1 UNION ALL q_2
	|	DISTINCT q
$x \in \texttt{TableAlias}$	::=	string
$b \in \texttt{Predicate}$	::=	$e_1 = e_2$
	|	NOT b | b_1 AND b_2 | b_1 OR b_2
	|	TRUE | FALSE
	|	EXISTS q
$e \in \texttt{Expression}$	::=	$x.a$ | $f(e_1, \ldots, e_n)$ | $\texttt{agg}(q)$
$p \in \texttt{Projection}$	::=	* | x.* | e AS a | $p_1,\ p_2$
$f \in \texttt{UDF}, \texttt{agg} \in \texttt{UDA}$	::=	string

Figure 2: SQL fragment supported by our semantics

Thus, the Q_2 can be rewritten to:

$$\lambda\,t.\ \sum_{t_2, t_3} [t_2 = t] \times [t_3.a \geq 12] \times [\![R]\!](t_2) \times [t_2 = t_3]$$

Applying Eq. (15) again to the above makes Q_2 is equivalent to Q_1. We present a detailed proof in Ex 4.7.

We next explain these axioms in detail and show how we use them to develop an algorithm to formally and automatically check the equivalences of general SQL queries.

3. AXIOMATIC FOUNDATIONS

In this section we introduce a new algebraic structure, U-semiring, and show that it can be used to give semantics to SQL queries.

3.1 U-semirings

Under standard bag semantics, a SQL query and its input relations can be modeled as $\mathbb{N}$-relations [35], i.e., relations over the semiring $(\mathbb{N}, 0, 1, +, \times)$. However, as we saw SQL queries include constructs that are not directly expressible in a semiring, such as projection (requiring an unbounded summation), DISTINCT, and non-monotone operators (e.g., NOT EXISTS). We now define a new algebraic structure that can be used for such purposes.

DEFINITION 3.1. *An* unbounded semiring*, or U-semiring, is* $(\mathcal{U}, \mathbf{0}, \mathbf{1}, +, \times, \|\cdot\|, \texttt{not}(\cdot), (\sum_D)_{D \in \mathcal{D}})$*, where:*

- $(\mathcal{U}, \mathbf{0}, \mathbf{1}, +, \times)$ *forms a commutative semiring.*[3]
- $\|\cdot\|$ *is a unary operation called* squash *that satisfies:*

$$\|\mathbf{0}\| = \mathbf{0} \quad , \quad \|\mathbf{1} + x\| = \mathbf{1} \tag{1}$$

$$\|\|x\| + y\| = \|x + y\| \tag{2}$$

$$\|x\| \times \|y\| = \|x \times y\| \tag{3}$$

$$\|x\| \times \|x\| = \|x\| \tag{4}$$

$$x \times \|x\| = x \tag{5}$$

$$x^2 = x \ \Rightarrow\ \|x\| = x \tag{6}$$

[3]Recall that the semiring axioms are: associativity and commutativity of $+$ and $\times$, identity of $\mathbf{0}$ for $+$, identity of $\mathbf{1}$ for $\times$, distributivity of $\times$ over $+$, and $\mathbf{0} \times x = \mathbf{0}$; see [35] for details.

- $\texttt{not}(\cdot)$ *is a unary operation that satisfies the following:*

$$\begin{aligned} \texttt{not}(\boldsymbol{0}) &= \boldsymbol{1} \\ \texttt{not}(x \times y) &= \|\texttt{not}(x) + \texttt{not}(y)\| \\ \texttt{not}(x + y) &= \texttt{not}(x) \times \texttt{not}(y) \\ \texttt{not}(\|x\|) &= \|\texttt{not}(x)\| = \texttt{not}(x) \end{aligned}$$

- *$\mathcal{D}$ is a set of sets; each $D \in \mathcal{D}$ is called a* summation domain. *For each D, the operation $\sum_D : (D \to \mathcal{U}) \to \mathcal{U}$ is called an* unbounded summation*: its input is a function $f : D \to \mathcal{U}$, and its output is a value in $\mathcal{U}$. We will write the summation as $\sum_{t \in D} f(t)$, or just $\sum_t f(t)$, where f is an expression and t is a free variable.*
- *Unbounded summation further satisfies the following axioms:*

$$\sum_t \big(f_1(t) + f_2(t)\big) = \sum_t f_1(t) + \sum_t f_2(t) \tag{7}$$

$$\sum_{t_1} \sum_{t_2} f(t_1, t_2) = \sum_{t_2} \sum_{t_1} f(t_1, t_2) \tag{8}$$

$$x \times \sum_t f_2(t) = \sum_t x \times f_2(t) \tag{9}$$

$$\left\| \sum_t f(t) \right\| = \left\| \sum_t \|f(t)\| \right\| \tag{10}$$

Thus, a U-semiring extends the semiring with unbounded summation, the squash operator (to model the SQL `DISTINCT` operator), and negation (to model `NOT EXISTS`). We chose a set of axioms that captures the semantics of SQL queries. For example, Eq.(2) implies $\|\|x\|\| = \|x\|$, which, as we will see, helps us prove that `DISTINCT` of `DISTINCT` equals `DISTINCT`. Eq.(4) captures equality of queries under set semantics, such as `SELECT DISTINCT R.a FROM R` and `SELECT DISTINCT x.a FROM R x, R y WHERE x.a = y.a`. Eq.(6) captures even more subtle interactions between subqueries with and without `DISTINCT` as described in [44]. We will illustrate this in Sec. 5.4.

If a summation domain D is finite, say $D = \{a_1, \ldots, a_n\}$, we could define $\Sigma_D f$ directly as $f(a_1) + \cdots + f(a_n)$. However, as we saw, the meaning of a projection requires us to sum over an unspecified set, namely, all tuples of a fixed schema. Even though all SQL summation domains are finite, *proving* this automatically is difficult and adds considerable complexity even for the simplest query equivalences (for instance, we need to prove that all operators preserve domain finiteness). Instead, we retain unbounded summation as a primitive in a U-semiring.

We now illustrate four simple examples of U-semirings. (1) If all summation domains are finite, then the set of natural numbers, $\mathbb{N}$, forms a U-semiring, where the unbounded summation is the standard sum, the squash and negation operators are $\|0\| = \texttt{not}(x) = 0$, and $\|x\| = \texttt{not}(0) = 1$ for all $x \neq 0$. (2) Its *closure*, $\bar{\mathbb{N}} \stackrel{\text{def}}{=} \mathbb{N} \cup \{\infty\}$, forms a U-semiring over arbitrary summation domains.[4] (3) The univalent types [50] form an U-semiring; in our prior system, COSETTE [23], we used univalent types to prove SQL query equivalence. (4) Finally, the cardinal numbers (which form a subset of univalent types) also form a U-semiring.

To appreciate the design of U-semirings, it may be helpful to see what we *excluded*. Recall that fewer axioms translate into a simpler proof system; hence, the need for frugality. A *complete-semiring* [30] also extends a semiring with unbounded summation. However, it requires a stronger set of axioms: summation must be defined over *all* subsets of some index set, and includes additional axioms, such as $\sum_{\{a,b\}} f = f(a) + f(b)$, among others. In a U-semiring, summation is defined on only a small and fixed set of summation domains that do not include $\{a, b\}$. This removes the need for axioms involving complex index sets. Similarly, the axioms for $\|\cdot\|$ and $\texttt{not}(\cdot)$ are also kept to a minimum; for example, we omitted unnecessary conditional identities like *if $x \neq \boldsymbol{0}$ then $\|x\| = \boldsymbol{1}$*. One example of a U-semiring where this conditional axiom fails is the set of diagonal 2×2 matrices with elements in $\bar{\mathbb{N}}$, where $\mathbf{0} \stackrel{\text{def}}{=} \text{diag}(0,0)$, $\mathbf{1} \stackrel{\text{def}}{=} \text{diag}(1,1)$, and all operations are performed on the diagonal using their meaning in $\bar{\mathbb{N}}$. In this semiring, $\|x\| \in \{\text{diag}(0,0), \text{diag}(0,1), \text{diag}(1,0), \text{diag}(1,1)\}$.

Another important decision was to exclude order relations, $x \leq y$, which we could have used to define key constraints (by stating that every key value occurs $\leq \mathbf{1}$ times). An *ordered semiring* (see also a *dioid* [33]) is a semiring equipped with an order relation $\leq$ such that $\mathbf{0} \leq a$ for all a, and $x \leq y$ implies $x + z \leq y + z$. We did not require a U-semiring to be ordered, instead we define key constraints using only the existing axioms (Sec. 4).

With all these simplifications, one wonders if it is possible to prove *any* non-trivial SQL equivalences, let alone those in the presence of complex integrity constraints. We answer this in the affirmative, as we shall next explain.

3.2 U-semiring SQL Semantics

Every SQL query is translated into a U-expression, which denotes the K-relation of the query's answer; to check the equivalences of two SQL queries, the system first translates them to U-expressions, then checks their equivalence using the `UDP` algorithm described in Sec. 5. In this section, we describe the translation from SQL queries to U-expressions.

Fig. 2 shows the SQL fragment currently supported by our implementation. We require the explicit declaration of table schemas, keys, foreign keys, views, and indexes, and support a rich fragment of SQL that includes subqueries and `DISTINCT`. We also support `GROUP BY` by de-sugaring them into subqueries as follows:

```
SELECT x.k as k, agg(x.a) as a1 FROM R x
GROUP BY x.k
                is rewritten to ⇓
SELECT y.k as k,
       agg(SELECT x.a as a
           FROM R x WHERE x.k=y.k) as a1
FROM R y
```

where R can be any SQL expression. Our implementation of `UDP` currently supports aggregates by treating them as uninterpreted functions.

Each SQL query Q is translated into a *U-expression* as follows. We denote a U-expressions by a capital letter E, denote a SQL expression by a lower case letter e (`Expression` in Fig. 2, e.g., $t.price/100$), and denote a SQL predicate by b (`Predicate` in Fig. 2, e.g. $t.price/100 > s.discount$).

- For each schema σ in the program, there is a summation domain $\texttt{Tuple}(\sigma)$, which includes all tuples with schema σ.
- For each relation name R, there is a predefined function $[\![R]\!] : \texttt{Tuple}(\sigma) \to \mathcal{U}$. Intuitively, $[\![R]\!](t)$ returns the multiplicity of $t \in R$, or $\mathbf{0}$ if $t \notin R$. If the context is clear, we will drop $[\![\cdot]\!]$ and simply write $R(t)$. Unlike prior work [35], we do not require the support of R to be finite, as that would make it difficult to axiomatize as discussed.

[4] $+$ and $\times$ are extended by $x + \infty = \infty$, $0 \times \infty = 0$, and $x \times \infty = \infty$ for $x \neq 0$. Unbounded summation is defined as $\sum_D f \stackrel{\text{def}}{=} f(a_1) + \cdots + f(a_n)$ when the support of f is finite, $\texttt{supp}(f) \stackrel{\text{def}}{=} \{x \mid f(x) \neq 0\} = \{a_1, \ldots, a_n\}$, and $\sum_D f = \infty$ otherwise.

- For each SQL predicate expression b, there is a U-expression $[b] \in \mathcal{U}$ that satisfies:

$$[b] = \|[b]\| \tag{11}$$

Under the standard interpretation of SQL, $[b]$ is either $\mathbf{0}$ (false) or $\mathbf{1}$ (true), but formally we only require Eq.(11). The equality predicate has a special interpretation in U-semiring and is required to satisfy the following axioms, called *excluded middle* (Eq. (12)), *substitution of equals by equals* (Eq. (13)), and *uniqueness of equality* (Eq. (14)):

$$[e_1 = e_2] + [e_1 \neq e_2] = \mathbf{1} \tag{12}$$

$$f(e_1) \times [e_1 = e_2] = f(e_2) \times [e_1 = e_2] \tag{13}$$

$$\sum_t [t = e] = \mathbf{1} \tag{14}$$

These axioms are sufficient to capture the semantics of equality ($=$). For example, we can prove:[5]

$$\sum_t [t = e] \times f(t) = f(e) \tag{15}$$

- For each uninterpreted aggregate operator agg and U-expression $E : \texttt{Tuple}(\sigma) \to \mathcal{U}$, $e ::= \texttt{agg}(E)$ is a valid value expression. It can be used in a predicate, e.g., $[\texttt{agg}(E) = t.a]$.

Every SQL query q is translated, inductively, to a U-expression denoted $[\![q]\!]$. For example, a table R is translated into its predefined semantics $[\![R]\!]$; a `SELECT-FROM-WHERE` is translated by generalizing K-relation SQL semantics (shown in Sec. 2); predicates are also translated inductively: `NOT`, `AND`, `OR` become $\texttt{not}(\cdot), \times, +$, while $[\![\texttt{EXISTS } q]\!] = \|[\![q]\!]\|$ and $[\![\texttt{NOT EXISTS } q]\!] = \texttt{not}([\![q]\!])$. Notice that the language includes nested queries both in the `FROM` and the `WHERE` clauses; their translation is based on standard unnesting. We omit the details and state only the main property:

DEFINITION 3.2. *Every SQL expression q in Fig. 2 is translated into a U-expression $[\![q]\!] : \texttt{Tuple}(\sigma) \to \mathcal{U}$. The translation is defined inductively on the structure of q; see the full paper for details [20].*

We write q instead of $[\![q]\!]$ when context permits. Strictly speaking q is a function, $q = \lambda t.E$, but we use the more friendly notation $q(t) = E(t)$, where E is an expression with a free variable t.

3.3 Sum-Product Normal Form

To facilitate the automated equivalence proof, our algorithm, `UDP`, first converts every U-expression into the *Sum-Product Normal Form (*SPNF*)*. Importantly, this conversion is done by repeated applications of the U-semiring axioms; thus, our system can *prove* that any expression translated into SPNF is semantically equivalent to the original input U-expression.

DEFINITION 3.3. *A U-expression expression E is in* SPNF *if it has the following form:*

- $E ::= T_1 + \ldots + T_n$
- *Furthermore, each term T_i has the form:*

$$\sum_{t_1,\ldots,t_m} [b_1] \times \ldots \times [b_k] \times \|E_s\| \times \texttt{not}(E_n) \times M_1 \times \ldots \times M_j$$

where each tuple variable t_i ranges over $\texttt{Tuple}(\sigma_i)$. Each expression inside the summation is called a factor. *Multiple summations are combined into a single sum using Eq.* (8)*. There is no summation in the case where $m = 0$.*

[5] Using (13), 9, and (14) we can show that: $\sum_t [t = e] \times f(t) = \sum_t [t = e] \times f(e) = f(e) \times \sum_t [t = e] = f(e) \times \mathbf{1} = f(e)$.

SQL Query q

```
SELECT t2.*
FROM (SELECT t1.k as k, t1.a as a
      FROM R t3) t1, R t2
WHERE t1.k = t2.k AND t1.a ≥ 12
```

Its corresponding U-expression in SPNF

$$\begin{aligned}
[\![q]\!](t) &= \sum_{t_1,t_2} [t_2 = t] \times \Big(\sum_{t_3} [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times \\
&\quad R(t_3)\Big) \times R(t_2) \times [t_1.k = t_2.k] \times [t_1.a \geq 12] \\
&= \sum_{t_1,t_2} [t_2 = t] \times [t_1.k = t_2.k] \times [t_1.a \geq 12] \times \\
&\quad \Big(\sum_{t_3} [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times R(t_3)\Big) \times R(t_2) && \text{Rule (4)} \\
&= \sum_{t_1,t_2} [t_2 = t] \times [t_1.k = t_2.k] \times ([t_1.a \geq 12] \times \\
&\quad \sum_{t_3} [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times R(t_3) \times R(t_2) && \text{Rule (7)} \\
&= \sum_{t_1,t_2,t_3} [t_2 = t] \times [t_1.k = t_2.k] \times [t_1.a \geq 12] \times \\
&\quad [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times R(t_3) \times R(t_2) && \text{Rule (6)}
\end{aligned}$$

Figure 3: A SQL query q (the second query shown in Fig. 1), its semantics $[\![q]\!]$ in U-semiring and its rewriting into sum-product normal form.

- *Each factor $[b_i]$ is a predicate.*
- *There is exactly one factor $\|E_s\|$, where E_s is an expression in* SPNF*. When $E_s = \mathbf{1}$, $\|E_s\|$ can be omitted.*
- *There is exactly one factor $\texttt{not}(E_n)$, where E_n is an expression in* SPNF*. When $E_n = \mathbf{0}$, $\texttt{not}(E_n)$ can be omitted.*
- *Each factor M_i is an expression of the form $R(t)$, for some relation name R, and some tuple variable t.*

THEOREM 3.4. *For any U-expression E, there exists an* SPNF *expression E' such that $E = E'$ in any U-semiring.*

PROOF. We briefly sketch the proof idea here. Formally, any U-expression E can be rewritten into E' in SPNF using the following rewrite system:

$$E_1 \times (E_2 + E_3) \rightsquigarrow E_1 \times E_2 + E_1 \times E_3 \tag{1}$$

$$(E_1 + E_2) \times E_3 \rightsquigarrow E_1 \times E_3 + E_2 \times E_3 \tag{2}$$

$$E_1 \times (E_2 \times E_3) \rightsquigarrow (E_1 \times E_2) \times E_3 \tag{3}$$

$$\ldots \times M \times [b] \times \ldots \rightsquigarrow \ldots \times [b] \times M \times \ldots \tag{4}$$

$$\sum_t \big(f_1(t) + f_2(t)\big) \rightsquigarrow \sum_t f_1(t) + \sum_t f_2(t) \tag{5}$$

$$E \times \sum_t f(t) \rightsquigarrow \sum_t E \times f(t) \tag{6}$$

$$\Big(\sum_t f(t)\Big) \times E \rightsquigarrow \sum_t f(t) \times E \tag{7}$$

$$\|E_1\| \times \|E_2\| \rightsquigarrow \|E_1 \times E_2\| \tag{8}$$

$$\texttt{not}(E_1) \times \texttt{not}(E_2) \rightsquigarrow \texttt{not}(E_1 + E_2) \tag{9}$$

Each rule above corresponds to an axiom of U-semirings. Rule (1)-(4) are axioms of commutative semirings, while the rest are axioms of U-semirings. All rewrite rules in SPNF are unidirectional and guaranteed to make progress. For example Rule (1) and (5) apply distributivity to remove any $+$ inside each T_i. Rule (3) and (4) normalize the $\times$ expressions so they remain left associative with all boolean expressions moved to the left. The last two rules, Rule

(8) and (9), consolidate a product of multiple factors into a single factor. □

Figure 3 shows how a U-expression is converted into SPNF by applying Rules (4, 7), and (6). In general, the rules described above are applied recursively until none of them is applicable.

4. INTEGRITY CONSTRAINTS

Our system checks the equivalence of two SQL queries in the presence of a set of integrity constraints: keys, foreign keys, views, and indexes (see Fig. 2). In this section we introduce a new axiomatic interpretation of integrity constraints using identities in a U-semiring.

4.1 Axiomatic Interpretation of Constraints

Key Constraints. To the best of our knowledge, key constraints have not to date been defined for semiring semantics. Therefore, we give the following definition.

DEFINITION 4.1. *Let R be a relation with schema σ, and let k be an attribute (or a set of attributes). The* KEY *constraint is the following identity, for all $t, t' \in \texttt{Tuple}(\sigma)$:*

$$[t.k = t'.k] \times R(t) \times R(t') \;=\; [t = t'] \times R(t)$$

For each key constraint in the SQL specification the system generates one such identity, adds it as an axiom, and uses it later to prove equivalences of SQL expressions. We show two simple properties implied by the key constraint. First, setting $t = t'$, we have $(R(t))^2 = R(t)$ for every tuple t. Second, for two tuples $t \neq t'$ such that $t.k = t'.k$, we have $R(t) \times R(t') = \mathbf{0}$. Therefore, if the U-semiring is the semiring of natural numbers, $\mathbb{N}$, then $R.k$ is a standard key: the first property implies that $R(t) \in \{0, 1\}$, and the second implies that $R(t) = 0$ or $R(t') = 0$. Hence, only one tuple with a given key may occur in R, with its multiplicity being 1:

THEOREM 4.2. *If $R.k$ satisfies the key constraint (Def. 4.1) over the U-semiring of natural numbers $\mathbb{N}$, then $R.k$ is a standard key.*

We briefly discuss our choice in Def. 4.1. The standard axiom for a key is $\forall t, t'.(t \in R \wedge t' \in R \wedge t.k = t'.k \;\Rightarrow\; t = t')$ [11]. However, it applies only to set semantics and uses a first-order sentence instead of an identity in a semiring. An alternative attempt at defining a key is:

$$\sum_t R(t) \times [t.k = e] \leq \mathbf{1} \tag{10}$$

This says that the sum of all multiplicities of tuples that have their keys equal to a constant e must be $\leq \mathbf{1}$. However, (10) requires an ordered semiring, which leads to additional complexity, as argued earlier. In contrast, Def. 4.1 does not require order and is a simple identity, i.e., it has the form $E_1 = E_2$. A better attempt is to state:

$$R(t') \times \sum_t R(t) \times [t.k = e] \;=\; R(t') \tag{11}$$

This axiom is an identity and, furthermore, it can be shown that Eq.(10) implies Eq.(11) in any ordered semiring; in other words, Eq.(11) seems to be the right reformulation of Eq.(10) without requiring order. On the other hand, Eq.(11) already follows from our key identity (Def. 4.1): simply sum both sides over t and observe that the RHS is equal to $R(t')$. We prove here a consequence of the key constraint, which we use in Sec. 5.4 to prove some non-trivial SQL identities described by [44]:

THEOREM 4.3. *If $R.k$ satisfies the key constraint (Def. 4.1), then the following U-expression is preserved under the squash operator $\|\cdot\|$, for any U-expression E, expression e, predicate b:*

$$\sum_t [b] \times \|E\| \times [t.k = e] \times R(t) = \left\| \sum_t [b] \times \|E\| \times [t.k = e] \times R(t) \right\|$$

PROOF. By Eq.(6) in Sec. 3.1, it suffices to show that the LHS squared equals itself. From Def. 4.1, we derive:[6]

$$[t.k = e] \times [t'.k = e] \times [b]^2 \times \|E\|^2 \times R(t) \times R(t') = [t = t'] \times [t.k = e] \times [b] \times \|E\| \times R(t)$$

Next, we sum over t and t' on both sides:

$$\sum_{t,t'} [t.k = e] \times [t'.k = e] \times [b]^2 \times \|E\|^2 \times R(t) \times R(t') = \sum_{t,t'} [t = t'] \times [t.k = e] \times [b] \times \|E\| \times R(t)$$

By applying Eq. (9) in Sec. 3.1 twice on the left, and Eq. (15) in Sec. 3.2 on the right:

$$\Big(\sum_t [b] \times \|E\| \times [t.k = e] \times R(t)\Big) \times \Big(\sum_{t'} [b] \times \|E\| \times [t'.k = e] \times R(t')\Big)$$
$$= \sum_t [b] \times \|E\| \times [t.k = e] \times R(t)$$

Hence Theorem 4.3 follows from Eq. (6) in Sec. 3.1: $x^2 = x$ implies $\|x\| = x$. □

Foreign Key Constraints. We now define foreign keys in a U-semiring:

DEFINITION 4.4. *Let S, R be two relations, and k', k be two attributes (or two sets of attributes), in S and R, respectively. The* FOREIGN KEY *constraint from $S.k'$ to $R.k$ is the following:*

$$S(t') = S(t') \times \sum_t R(t) \times [t.k = t'.k']$$

If $S(t') \neq 0$ and the U-semiring is the standard semiring $\mathbb{N}$, then this definition implies $\sum_t R(t) \times [t.k = t'.k'] = 1$, i.e., a tuple in R has the same value k, the tuple is unique, and its multiplicity is 1. There is no constraint on the multiplicity of $S(t')$. Thus:

THEOREM 4.5. *If $S.k', R.k$ satisfies the foreign key constraint (Def. 4.4) over the U-semiring of natural numbers $\mathbb{N}$, then $R.k$ is a standard key in R, and $S.k'$ is a standard foreign key to $R.k$.*

Views and Indexes. We convert a view definition in Fig. 2 into an assertion $v(t) = q(t)$; every occurrence of v in a query is inlined according to its definition. We follow the GMAP approach [52] and consider an index to be a view definition that consists of the projection on the key and the attribute to be indexed. For example, an index I on the attribute $R.a$ is expressed as:

$$I := \texttt{SELECT } x.a,\ x.k \texttt{ FROM } R\ x$$

where k is the KEY (Def. 4.1) of R. The indexes are treated as views and inlined in the main query when compiling to U-expressions.

[6] We also used $[b]^2 = [b]$ and $\|E\|^2 = \|E\|$.

4.2 SQL Equivalence with Integrity Constraints

We can now formally define the equivalence of two SQL expressions under U-semiring semantics.

DEFINITION 4.6. *Consider two queries, q_1 and q_2, along with definitions of schemas, base relations, and constraints (Fig. 2). q_1 and q_2 are U-equivalent if, for any U-semiring and interpretation of the base relations, when all key and foreign key constraints are satisfied, then the following identity holds: $[\![q_1]\!] = [\![q_2]\!]$.*[7]

We will refer to U-equivalence simply as *equivalence* when the context is clear. We illustrate this concept with an example.

EXAMPLE 4.7. *We show how to formally prove that the optimization rule at the top of Fig. 1 is correct by showing that Q_1 is equivalent to its rewrite, Q_2 using an index lookup. Recall that R has key k, and I is an index on $R.a$.*

The U-expression of Q_1 is:

$$Q_1(t) = R(t) \times [t.a \geq 12]$$

After inlining the definition of I, Q_2 is the query shown in Fig.3, and its U-expression in SPNF *is:*

$$Q_2(t) = \sum_{t_1,t_2,t_3} [t_2 = t] \times [t_1.k = t_2.k] \times [t_1.a \geq 12] \times [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times R(t_3) \times R(t_2)$$

Since t_1 is a tuple returned by I, its schema consists of the two attributes a and k; hence, $[t_3.k = t_1.k]$ and $[t_3.a = t_1.a]$ imply $[t_1 = (t_3.k, t_3.a)]$. We use Eq. (15) *to remove the summation over t_1, the two equalities $[t_3.k = t_1.k]$ and $[t_3.a = t_1.a]$, and substitute $t_1.k$ with $t_3.k$ and $t_1.a$ with $t_3.a$ to get:*

$$Q_2(t) = \sum_{t_2,t_3} [t_2 = t] \times [t_3.k = t_2.k] \times [t_3.a \geq 12] \times R(t_3) \times R(t_2)$$

Applying the key constraint definition (Def. 4.1) on $Q_2(t)$, we get:

$$Q_2(t) = \sum_{t_2,t_3} [t_2 = t] \times [t_3.a \geq 12] \times R(t_2) \times [t_2 = t_3]$$

Now, we use Eq. (15) *to remove the summation over t_3 as $t_2 = t_3$:*

$$Q_2(t) = \sum_{t_2} [t_2 = t] \times [t_2.a \geq 12] \times R(t_2)$$

And similarly remove the summation over t_2 since $t_2 = t$:

$$Q_2(t) = [t.a \geq 12] \times R(t)$$

This shows that Q_1 and Q_2 are equivalent by Def. 4.6.

The example above reveals a strategy for proving queries: first, express both queries in SPNF. Then, repeatedly reduce the expression using the available constraints until the they become isomorphic and hence are equivalent. We formally present our equivalence deciding algorithm in the next section.

Discussion. Definition 4.6 is sound because two U-equivalent queries are also equivalent under the standard interpretation in the semiring of natural numbers. However, the definition is not complete: there exists SQL queries that are equivalent under the standard semantics but not U-equivalent. One such example consists of queries that are equivalent over all finite relations but not equivalent over infinite relations. For example, there exists sentences φ in First Order Logic, called *infinity axioms*, that are always false when R is finite, but can be satisfied by an infinite R; for example, φ may say "R is non-empty; for every x occurring in R there exists a unique y such that $R(x, y)$ (i.e. R is a function); the mapping $x \mapsto y$ is injective; and it is not surjective."[8] Such a sentence φ can be converted in SQL query Q that returns $\emptyset$ when φ is false, and returns 1 when φ is true. Then, on any finite database Q is equivalent to the empty-set query Q' (e.g, written in SQL as `select distinct 1 from R where 0 ≠ 0`), but Q, Q' are not U-equivalent, because they are distinct over the U-semiring $\mathbb{N}$.

[7] We assume all views are inlined into q_1 and q_2 as discussed above.

5. DECISION PROCEDURE FOR SQL

We now present UDP (U-expression Decision Procedure) for checking the equivalence of two U-expressions with constraints, where *equivalence* is the U-equivalence given by Definition 4.6. As we are unaware of any theorem provers that can automatically reason about the equivalences of U-expressions, we implement our decision procedure using proof assistants, which ensures that our procedure implementation is *provably correct* given the U-expression axioms that we have defined.

UDP supports the SQL fragment in Fig. 2. The input SQL queries are evaluated under mixed set and bag semantics, which is the semantics that most real-world database systems use. We show that UDP is sound and also complete when the input are Union of Conjunctive Queries (UCQ) and evaluated under set semantics only, or under bag semantics only. To the best of our knowledge, this is the first algorithm implemented that can check the equivalence of UCQ with integrity constraints and evaluated under set or bag semantics.

At a high level, UDP proceeds recursively on the structure of a U-expression as shown in Alg. 2. Recall from Def. 3.3 the structure of a normalized U-expression E and of a term T:

$$E ::= T_1 + \ldots + T_n$$

$$T_i ::= \sum_{t_1,\ldots,t_m} [b_1] \times \ldots \times [b_k] \times \|E'\| \times \texttt{not}(E'') \times M_1 \times \ldots \times M_j$$

UDP takes two U-expressions (E_1 and E_2), and a set of integrity constraints (C) in the form of U-expression identities. It first calls `canonize` (line 2) to transform each U-expression into a canonical representation (to be discussed Sec. 5.1). Then, UDP proceeds recursively on the structure of the two canonical U-expressions by calling TDP (line 7) shown in Alg. 3 to check the equivalence of each term. Similarly, procedure TDP calls SDP (line 4) shown in Alg. 4 to check the equivalence of two squashed U-expressions (U-expressions in the form of $\|E\|$). We discuss the details below.

5.1 Canonical Form

As we have illustrated in Ex 4.7, to prove the equivalence of SQL queries under integrity constraints, we rewrite the input query U-expressions using the axiomatic interpretations of these constraints as shown in Sec. 4.1. We call these rewritten U-expressions the canonical form of U-expressions under integrity constraints.

Algorithm 1 shows the detail of canonization. To convert a U-expression E from SPNF to canonical form, we rewrite E by the axioms and definitions discussed in Sec. 3 and Sec. 4. The algorithm contains the following four rewrites:

1. Apply the transitivity of the equality predicate (line 2): $[e_1 = e_2] \times [e_2 = e_3] = [e_1 = e_2] \times [e_2 = e_3] \times [e_1 = e_3]$.
2. Remove unnecessary summations using Eq. (15) (line 3).
3. For each key constraint, rewrite E using the identities defined in Def. 4.1 and Theorem 4.3 (line 5).

[8] A different, shorter infinity axiom is given in [15, pp.307]: $\varphi \equiv \forall x \exists y \forall z (\neg R(x, x) \wedge R(x, y) \wedge (R(y, z) \rightarrow R(x, z)))$.

Algorithm 1 Canonization

1: **procedure** canonize(E, C)
   // E is an U-expression in SPNF; C is a set of constraints
2: $E' \leftarrow$ TC(E) // Transitive closure of equalities
3: $E' \leftarrow$ *recursively* and *repeatedly* apply Eq. (15) on E'
4: **for** $c \in C$ **do**
5: $E' \leftarrow$ *recursively* and *repeatedly* apply Def. 4.1 and Theorem 4.3 on E'
6: $E' \leftarrow$ *recursively* and *repeatedly* apply Def. 4.4 on E'
7: **end for**
8: **return** E' // E' is now in canonical form
9: **end procedure**

4. For each foreign key constraint, rewrite E using the identity defined in Def. 4.4 (line 6).

When using an identity to rewrite E, we find a matching subexpression on E with the identity's LHS, and then replace it with the RHS. More importantly, these rewrites are performed *recursively* on the structure of E, and *repeatedly* until the following termination conditions: 1) The first 3 rewrites (line 2, 3, and 5) terminate until no further rewrite can be applied. 2) When applied on squashed expressions, e.g., $\|E'\|$, the last rewrite (line 6) is repeatedly applied until the rewritten squashed expression is equivalent to the one before applying the rewrite. The equivalence of the squashed expressions (before and after applying the rewrite) is checked using procedure SDP that will be presented in Algorithm 4 (Sec. 5.2). 3) When applied on other expressions, the last rewrite (line 6) is repeatedly applied until no new base relation is introduced.

Does canonize terminate? The rewrites using key and foreign key definitions resemble the chase procedure [45]. The difference is that our rewrites are on U-expressions rather than on relational queries. It is known that the chase procedure may not terminate [31], and so may our rewrite steps; for example, a cycle in the key/foreign-key graph may lead to non-termination. While this is possible in theory, we did not encounter any case that does not terminate, as our evaluation in Sec. 6 shows.

5.2 Equivalence of U-expressions

We now present the algorithm for checking if two U-expressions are equivalent after converting them into canonical form.

Algorithm 2 UDP: U-expression Decision Procedure

1: **procedure** UDP(E_1, E_2, C) // E_1 and E_2 are in SPNF
2: $E_1 \leftarrow$ canonize(E_1, C), $E_2 \leftarrow$ canonize(E_2, C)
   // E_1 is in the form of $T_{1,1} + \ldots + T_{1,n}$, and
   // E_2 is in the form of $T_{2,1} + \ldots + T_{2,m}$
3: **if** $m \neq n$ **then**
4: **return false**
5: **end if**
6: **for** $p \in \mathcal{P}([T_{1,1}, \ldots, T_{1,n}])$ **do**
7: **if** $\forall i \in \{1, \ldots, n\}$, TDP($p[i], T_{2,i}, C$) == **true then**
8: **return true**
9: **end if**
10: **end for**
11: **return false**
12: **end procedure**

Alg. 2 shows the detail of UDP for checking the equivalence of two U-expressions. On line 2, UDP first converts the two input U-expressions (E_1 and E_2) to canonical forms under integrity constraints as described in Sec. 5.1. Recall that a U-expression is a sum of terms $E = T_1 + \ldots + T_n$. To check for U-equivalence of E_1 and E_2, UDP searches for an isomorphism between each of their constituent terms T_i in line 3-10. It returns false if the number of terms of E_1 and E_2 are not equal (line 4). Otherwise, it searches for a permutation p of E_1's terms (here $\mathcal{P}$ represents the set of all possible permutations of its arguments) such that each pair of terms in p and E_2 are equivalent. This is determined by calling TDP in line 8, which we discuss next.

Equivalence of Terms

The TDP procedure shown in Alg. 3 checks the equivalence between terms. Recall that a term is an unbounded summation of the form $T = \sum_{t_1,\ldots,t_m}(\ldots)$. To check for the equivalence of the following input terms:

$$T_1 = \sum_{\vec{t_1}} [b_{1,1}] \times \ldots \times [b_{1,k}] \times \|E_{1,1}\| \times \texttt{not}(E_{1,2}) \times M_{1,1} \times \ldots \times M_{1,j}$$

$$T_2 = \sum_{\vec{t_2}} [b_{2,1}] \times \ldots \times [b_{2,m}] \times \|E_{2,1}\| \times \texttt{not}(E_{2,2}) \times M_{2,1} \times \ldots \times M_{2,\ell}$$

TDP searches for an isomorphism between T_1 and T_2. Let $h \in \mathcal{BI}_{\vec{t_2},\vec{t_1}}$ be a bijection from the set of variables that T_2 sums over ($\vec{t_2}$) to the set of variables that T_1 sums over ($\vec{t_1}$) (line 2).[9] TDP find an isomorphism if, after substituting $\vec{t_1}$ with $p(\vec{t_2})$ in T_2 (we call this new expression T_2') (line 3), the equivalence of T_1 and T_2' is checked as following (line 4):

- The predicate parts of two terms are equivalent: $[b_{1,1}] \times \ldots \times [b_{1,k}] = [b_{2,1}] \times \ldots \times [b_{2,m}]$. This requires checking that two Boolean expressions are equivalent. We check the equivalences of boolean expressions using the congruence procedure [43], which first computes the equivalent classes of variables and function applications and then checks for equivalence of the expressions using the equivalent classes. For example, $[a = b] \times [c = d] \times [b = e] \times [f(a) = g(d)]$ is equivalent to $[a = b] \times [a = e] \times [c = d] \times [f(e) = g(c)]$, since there are the following equivalent classes:
$$\{a, b, e\},\ \{c, d\},\ \{f(a), f(e)\}, \{g(c), g(d)\}$$
A predicate in T_1 (e.g., $[b_{1,i}]$) may contains aggregate functions (for example, $[\texttt{avg}(\cdots) = t.a]$). These aggregate functions are treated as uninterpreted functions like f and g in the example above.
- $\|E_{1,1}\|$ is U-equivalent to $\|E_{2,1}\|$. We explain the procedure for checking equivalences of squashed U-expressions, SDP, in the next section.
- The negated expressions $\texttt{not}(E_{1,2})$ and $\texttt{not}(E_{2,2})$ are U-equivalent. This is checked by calling UDP recursively in line 4.
- The two terms have the same number of M subterms, i.e., $j = \ell$, and the terms $M_{1,1}, \ldots, M_{1,j}$ are identical to the terms $M_{2,1}, \ldots, M_{2,\ell}$. Recall that each M_{i_1,i_2} has the form $R(t)$ for some relation name R and some tuple variable t (Def. 3.3).

Equivalences of Squashed Expressions

Finally, Alg. 4 shows SDP, the procedure for checking the equivalence of two squashed U-expressions, $\|E_1\|$ and $\|E_2\|$. Recall that squash expressions $\|\cdot\|$ are used to model the semantics of (sub-)queries with the DISTINCT operator.

The first step of SDP is to remove any nested squash subexpressions (line 2). We do so by applying the following lemma:

[9] For ease of presentation, we use the vectorized notation $\vec{t}$ as a shorthand for $t_1, t_2, \ldots$

Algorithm 3 TDP: Decision Procedure for Terms

procedure TDP(T_1, T_2, C)
// T_1, T_2 is in SPNF, C is a set of constraints. In particular,
// T_1 has the form $\sum_{\vec{t_1}}[b_{1,1}] \times \ldots \times [b_{1,k}] \times \|E_{1,1}\| \times$
// $\texttt{not}(E_{1,2}) \times M_{1,1} \times \ldots \times M_{1,j}$, and
// T_2 has the form $\sum_{\vec{t_2}}[b_{2,1}] \times \ldots \times [b_{2,m}] \times \|E_{2,1}\| \times$
// $\texttt{not}(E_{2,2}) \times M_{2,1} \times \ldots \times M_{2,\ell}$
  for $p \in \mathcal{BI}_{\vec{t_2},\vec{t_1}}$ **do**
    $T_2' \leftarrow p(T_2)$ // substitute $\vec{t_2}$ in T_2 using $p(\vec{t_2})$
    if $T_1 == T_2'$ **then**
      return true
    end if
  end for
  return false
end procedure

LEMMA 5.1. *The following holds in any U-semiring:*

$$\|a \times \|x\| + y\| = \|a \times x + y\| \tag{12}$$

PROOF. By Eq. (2) in Sec. 3, $LHS = \|\|a \times \|x\|\| + y\|$. And by Eq. (3), $LHS = \|\|a\| \times \|\|x\|\| + y\|$. By Eq. (2) (when $y = \mathbf{0}$ in Eq. (2)), $\|\|x\|\| = x$; thus:

$$LHS = \|\|a\| \times \|x\| + y\|$$

Apply Eq. (3) again:

$$LHS = \|\|a \times x\| + y\|$$

And apply Eq. (2) from right to left:

$$LHS = \|a \times x + y\|$$

□

Algorithm 4 SDP: Decision Procedure for Squashed Expressions

1: **procedure** SDP($\|E_1\|, \|E_2\|, C$) // E, E' are in SPNF
2: remove $\|\|$ inside E_1, E_2 by applying Lem. 5.1
3: $\|E_1\| \leftarrow \|\texttt{canonize}(E_1, C)\|$
  // $\|E_1\|$ is now in the form of $\|T_1 + \ldots + T_m\|$
4: $\|E_2\| \leftarrow \|\texttt{canonize}(E_2, C)\|$
  // $\|E_2\|$ is now in the form of $\|T_1' + \ldots + T_n'\|$
5: **return** $\forall i\, \exists\, j.\ \texttt{minimize}(\|T_i\|) == \texttt{minimize}(\|T_j'\|)$
    && $\forall j\, \exists\, i.\ \texttt{minimize}(\|T_j'\|) == \texttt{minimize}(\|T_i\|)$
6: **end procedure**

SDP then converts the expressions $\|E_1\|$ and $\|E_2\|$ to canonical forms under constraints by calling canonize on (lines 3-4). After that, SDP checks the equivalence of two expressions $\|T_1 + \cdots + T_m\|$ and $\|T_1' + \cdots + T_m'\|$.

If T_i and T_i' (for $i = 1, \ldots, m$) are conjunctive queries, then $\|T_1 + \cdots + T_m\|$ and $\|T_1' + \cdots + T_m'\|$ represent two queries in the class of unions of conjunctive queries under set semantics. A classical algorithm exists for checking the equivalence of such queries [47]: given $Q = q_1 \vee \cdots \vee q_m$ and $Q' = q_1' \vee \cdots \vee q_n'$, where each q is a conjunctive query, equivalence is established by checking whether $Q \subseteq Q'$ and $Q' \subseteq Q$. The former is checked by showing that for every i, there exists a j such that $q_i \subseteq q_j$ which, in turn, requires checking for a homomorphism from q_j to q_i. $Q' \subseteq Q$ is checked similarly.

However, it is challenging to express the homomorphism checking algorithm in [47] purely using U-semiring axioms since U-semiring axioms are all identities (equations). There is no notion of inclusion or ordering in U-semiring. To check the equivalences of two canonized squashed expressions, $\|E_1\|$ and $\|E_2\|$, we instead minimize each term, T_i, inside $\|E_1\|$ and $\|E_2\|$ using only the U-semiring axioms (this is implemented in the minimize procedure) and then check the syntactic equivalences of the minimized terms. This procedure is equivalent to minimizing conjunctive queries [11]. Thus, our equivalence checking procedure is sound and complete for squashed expressions derived from UCQs. For squashed expressions that represent SQL queries beyond UCQs, SDP is not complete, however, our procedure is still sound. Next, we explain how to minimize a term in a squashed expression using only U-semiring axioms. minimize follows exactly the same steps as shown in the example, except needs to be performed on all possible pairs of summation variables.

EXAMPLE 5.2. *We illustrate* SDP *on these two queries:*

```
SELECT DISTINCT x.a FROM R x, R y    -- Q1
SELECT DISTINCT R.a FROM R           -- Q2
```

It first converts Q_1 to a U-expressions canonical form (line 2-4):

$$Q_1(t) = \left\| \sum_{t_1,t_2} [t_1.a = t] \times R(t_1) \times R(t_2) \right\|$$

$$\overset{Eq.(12)}{=} \left\| \sum_{t_1,t_2} ([t_1 = t_2] + [t_1 \neq t_2]) \times [t_1.a = t] \times R(t_1) \times R(t_2) \right\|$$

Next, it uses the rewrite rules in Sec. 3.3 to convert it to SPNF*:*

$$Q_1(t) = \| \sum_{t_1,t_2} [t_1 = t_2] \times [t_1.a = t] \times R(t_1) \times R(t_2) + \sum_{t_1,t_2} [t_1 \neq t_2] \times [t_1.a = t] \times R(t_1) \times R(t_2) \|$$

Using Eq. (15)*, it removes the summation over t_2 as $t_1 = t_2$:*

$$Q_1(t) = \| \sum_{t_1} [t_1.a = t] \times R(t_1) \times R(t_1) + \sum_{t_1,t_2} [t_1 \neq t_2] \times [t_1.a = t] \times R(t_1) \times R(t_2) \|$$

By Eq. (10) *and Eq.* (4) *($\|R(t_1) \times R(t_1)\| = \|R(t_1)\|$), we have:*

$$Q_1(t) = \| \sum_{t_1} [t_1.a = t] \times R(t_1) + \sum_{t_1,t_2} [t_1 \neq t_2] \times [t_1.a = t] \times R(t_1) \times R(t_2) \|$$

Now, we can factorize the expression inside squash using Eq. (7)*:*

$$Q_1(t) = \| \sum_{t_1} [t_1.a = t] \times R(t_1)(1 + \sum_{t_2} [t_1 \neq t_2] \times R(t_2) \|$$

Finally, we simplify $Q_1(t)$ using Eq. (1) *and Eq.* (3)*:*

$$Q_1(t) = \| \sum_{t_1} [t_1.a = t] \times R(t_1) \times 1 \| = \| \sum_{t_1} [t_1.a = t] \times R(t_1) \| = Q_2(t)$$

thus proving that $Q_1(t)$ and $Q_2(t)$ are equivalent.

5.3 Soundness and Completeness

We now show that UDP, our procedure for checking the equivalence of two U-expressions, is sound. Furthermore, it is complete if two U-expressions are unions of conjunctive queries evaluated under set or bag semantics.

THEOREM 5.3. *Algorithm 2 is sound. For any pair of SQL queries, if algorithm 2 returns* ***true****, then the pair is equivalent under the standard SQL semantics [25].*

PROOF. All transformations in algorithms 1 and 2 are based on axioms of U-semiring and proven identities. Since the standard SQL semantics [25] is based on the semiring of natural numbers $\mathbb{N}$, it follows that the equivalence also holds under the standard semantics. □

When there is no integrity constraint and Q is a CQ evaluated under bag semantics, i.e., Q has the form `SELECT p FROM R1 t1, ... , Rn tn WHERE b`, and `b` is a conjunction of equality predicates, then $[\![Q]\!]$ has a unique canonical form, namely

$$\texttt{canonize}([\![Q]\!], \emptyset) = \sum_{t_1,\ldots,t_n} [b] \times R_1(t_1) \times \ldots \times R_n(t_n)$$

In addition, if Q is a CQ evaluated under set semantics, i.e., Q has the form `SELECT DISTINCT p FROM R1 t1, ..., Rn tn WHERE b`, then Q has a similar unique canonical form (the same form as bag semantics CQ but within $\|\cdot\|$).

This allows us to prove the following two theorems on the completeness of UDP.

THEOREM 5.4. *Algorithm 2 is complete for checking the equivalence of Unions of Conjunctive Queries (UCQ) evaluated under bag semantics.*

PROOF. Two UCQ queries under bag semantics are equivalent if and only if they are isomorphic [47] (see also [24, Theorem 4.3, 4.4]), implying that our algorithm is complete in this case. □

THEOREM 5.5. *Algorithm 2 is complete for checking the equivalence of Unions of Conjunctive Queries (UCQ) evaluated under set semantics.*

PROOF. The U-expression of a conjunctive query evaluated under set semantics is $\left\|\sum_{\vec{t}}[b_1] \times \cdots \times [b_n] \times R_1(t_1) \times \cdots \times R_j(t_j)\right\|$, where all predicates b_i are equalities $[t.a_1 = t'.a_2]$. In this case, `UDP` simply checks for the existence of a homomorphism for the input queries, which has been shown to be complete [47]. □

5.4 An Illustration

We demonstrate how `UDP` works using an example rewrite evaluated under mixed set-bag semantics from the Starburst optimizer [44]:

```
-- Q1
SELECT ip.np, itm.type, itm.itemno
FROM (SELECT DISTINCT itp.itemno as itn,
                      itp.np as np
      FROM price price
      WHERE price.np > 1000) ip, itm itm
WHERE ip.itn = itm.itemno;

-- Q2
SELECT DISTINCT price.np, itm.type, itm.itemno
FROM price price, itm itm
WHERE price.np > 1000 AND
      itp.itemno = itm.itemno;
```

Here `itemno` is a key of `itm`.

Below is the U-expression representing $Q_1(t)$:

$$\begin{aligned} Q_1(t) \quad = \quad & \textstyle\sum_{t_1,t_2} [t_1.\texttt{np} = t.\texttt{np}] \times [t_2.\texttt{type} = t.\texttt{type}] \times \\ & [t_2.\texttt{itemno} = t.\texttt{itemno}] \times [t_1.\texttt{itn} = t_2.\texttt{itemno}] \times \\ & \| \textstyle\sum_{t'} [t'.\texttt{itemno} = t_1.\texttt{itn}] \times [t'.\texttt{np} = t_1.np] \times \\ & [t'.\texttt{np} > 1000] \times \texttt{price}(t') \| \times \texttt{itm}(t_2) \end{aligned}$$

Next, $Q_1(t)$ is canonized by `canonize` (called by `UDP` in line 2). As tuple t_1 is generated by the subquery in `Q1`, it has two attributes: `np` and `itn`. Because $[t_1.\texttt{np} = t.\texttt{np})]$ and $[t_1.\texttt{itn} = t_2.\texttt{itemno}]$, the summation on t_1 is removed after applying Eq. (15) in line 3 in `canonize`:

$$\begin{aligned} Q_1(t) \quad = \quad & \textstyle\sum_{t_2} [t_2.\texttt{type} = t.\texttt{type}] \times [t_2.\texttt{itemno} = t.\texttt{itemno}] \times \\ & \| \textstyle\sum_{t'} [t'.\texttt{itemno} = t.\texttt{itemno}] \times [t'.\texttt{np} = t.np] \times \\ & [t'.\texttt{np} > 1000] \times \texttt{price}(t') \| \times \texttt{itm}(t_2) \end{aligned}$$

Since `ite.itemno` is a key, Theorem 4.3 is applied (line 5 in `canonize`):

$$\begin{aligned} Q_1(t) \quad = \quad & \| \textstyle\sum_{t_2,t'} [t_2.\texttt{type} = t.\texttt{type}] \times [t_2.\texttt{itemno} = t.\texttt{itemno}] \times \\ & [t'.\texttt{itemno} = t.\texttt{itemno}] \times [t'.\texttt{np} = t.np] \times \\ & [t'.\texttt{np} > 1000] \times \texttt{price}(t') \times \texttt{itm}(t_2) \| \end{aligned}$$

Similarly, $Q_2(t)$ is canonized to:

$$\begin{aligned} Q_2(t) \quad = \quad & \| \textstyle\sum_{t_1,t_2} [t_2.\texttt{type} = t.\texttt{type}] \times [t_2.\texttt{itemno} = t.\texttt{itemno}] \times \\ & [t_1.\texttt{itemno} = t.\texttt{itemno}] \times [t_2.\texttt{itemno} = t_1.\texttt{itemno}] \times \\ & [t_1.\texttt{np} = t.np] \times [t_1.\texttt{np} > 1000] \times \texttt{price}(t_1) \times \texttt{itm}(t_2) \| \end{aligned}$$

At the end, since $Q_1(t)$ and $Q_2(t)$ are squashed expressions, `SDP` is called. `SDP` finds a homomorphism from $Q_2(t)$ to $Q_1(t)$, namely $\{t_1 \to t', t_2 \to t_2\}$ and a homomorphism from $Q_1(t)$ to $Q_2(t)$ defined by the function $\{t' \to t_1, t_2 \to t_2\}$. Hence Q_1 is equivalent to Q_2. To the best of our knowledge, this is the first time that this rewrite rule is formally proved to be correct. By modeling the semantics of SQL using U-expressions, our procedure can be used to automatically deduce the equivalence of complex rewrites.

6. EVALUATION

In this section we describe our implementation and evaluation of `UDP`. We first describe our implementation in Sec. 6.1. Then, in Sec. 6.2, we report on the two sets of query rewrite rules or query pairs that we used in the evaluation: one set from well-known data management research literature, and another from a popular open-source query optimization framework called Calcite [1]. We summarize the evaluation results of our query equivalence checking algorithm in Sec. 6.2 and characterize these results in Sec. 6.3. Sec. 6.4 concludes with the limitations of our current implementation.

6.1 Implementation

We have implemented our `UDP` equivalence checking algorithm, and Fig. 4 shows its architecture. As shown in the figure, Our implementation takes as input a pair of SQL queries to be checked (Q_1 and Q_2) and the precondition for these queries in the form of integrity constraints (C). It compiles the queries to U-expressions (E_1 and E_2) and checks the equivalence of the resulting U-expressions using `UDP`. We implemented `UDP` in Lean, a proof assistant, which guarantees that if `UDP` returns true then the input queries are indeed equivalent according to our U-semiring semantics.

`UDP` is implemented in two components: a U-expression generator that parses the input SQL queries (expressed using the syntax shown in Fig. 2) to ASTs (Abstract Syntax Trees), and a converter that translates the ASTs to U-expressions. The parser is written in 440 lines of Haskell, and the converter is written using 202 lines of Lean. In addition, the implementation of the axioms discussed

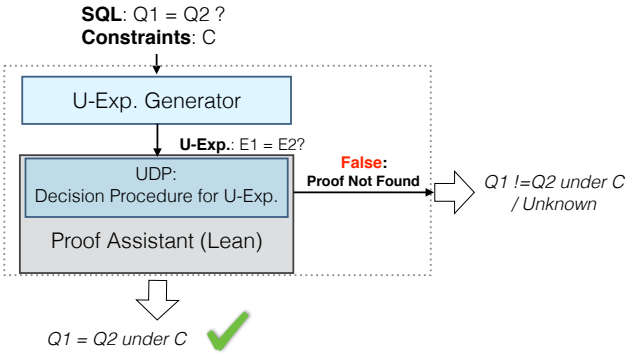

Figure 4: UDP implementation

in Sec. 3 and Sec. 4 consists of 129 lines of Lean. UDP is implemented using 1422 lines of Lean's metaprogramming language for proof search [29]. The parser, converter, and axiom code form the trusted code base of our implementation. All other parts, such as the implementation of UDP, are formally verified by implementing in Lean on top of our trusted code base.

6.2 Evaluation Summary

To evaluate UDP, we used it to prove various real-world SQL queries and rewrite rules from the following two sources[10]:

Literature. We manually examined the SIGMOD papers in the last 30 years looking for rewrite rules that contain integrity constraints. We found 6 rewrite rules from research papers published in SIGMOD (such as rules with mixed bag-set semantic queries from starburst [49]), technical reports, and blog articles. These rules are *conditional*, i.e., they all claim to be valid only under integrity constraints that are expressible in SQL. These constraints include key constraints, foreign key constraints, and the use of indexes. We also include 23 rewrite rules that are interactively proven in a proof assistant from our previous work [23], including the well-known magic set rewrites [49].

Apache Calcite. Apache Calcite [1] is an open-source query optimization framework that powers many data processing engines, such as Apache Hive [4], Apache Drill [2], and others [8, 6, 3, 5]. Calcite includes an extensive rule-based rewrite optimizer that contains 83 rules total. To ensure the correctness of these rewrite rules, Calcite comes with 232 test cases, each of which contains a SQL query, a set of input tables, and the expected results. Passing a test case means that the Calcite rewritten query returns the correct result on the specific test data. For each Calcite's test case, we use the input query as Q_1 and the rewritten query after applying Calcite's rules as Q_2. Among these 232 pairs of SQL queries, 39 pairs use SQL features that UDP currently supports, and we discuss why UDP cannot support the rest in Sec. 6.4. Note that the Calcite test cases lead us to verify pairs of *queries* rather than rules. For example, one of the test cases checks for $R \bowtie S = S \bowtie R$, where R, S are concrete tables rather than arbitrary SQL expressions. As Calcite implements its rewrite rules in Java code rather than our input language as shown in Fig. 2, we instead treat every pair of queries in the test cases as query instances by replacing the concrete table names with general SQL expressions using our syntax, while retaining any integrity constraints and ignoring the actual contents of the tables. We then send the queries to our system and ask it to prove the semantic equivalence of the two queries for all possible table contents.

Among the set of rules/equivalent query pairs used in our evaluation, 8 from the literature and 2 from Calcite require database integrity constraints as preconditions. Furthermore, 14 from the literature and 12 from Calcite were not conjunctive query rewrites.

[10]The detailed benchmark queries and rules can be found in [20].

Dataset	Total No.	No. of Supported	No. of Proved	No. of Unproved
Literature	29	29	29	0
Calcite	232	39	33	6
Bugs	3	1	0	1

Figure 5: Summary of proved and unproved cases

Dataset	Proved Total	UCQ	Cond.	Grouping, Aggregate, and Having	DISTINCT in Subquery
Literature	29	15	9	2	4
Calcite	34	21	2	11	1

Figure 6: Characterization of the proved cases, where the categories are not mutually exclusive.

Fig. 5 summarizes our evaluation result. Among the 6 conditional query rewrites from the literature, UDP can automatically prove all of them. Among Calcite's 39 test cases that use SQL features supported by UDP, 33 (i.e., 85%) of them are automatically proved. 5 unproven test cases involve integer arithmetic and string casting, which UDP currently does not support. The last unproven test case involves two very long queries, and UDP does not return a result after running for 30 minutes.

Previously Documented Bugs. We tried using our system to prove the count bug [32]. As expected, our system failed to prove equivalence within the time limit of 30 minutes; two other bugs in the literature [7, 10] are based on the NULL semantics, which we currently do not support. As our prior work work [22] already shows how to use a model checker to find counterexamples to invalidate such rules, our current system instead focuses on proving equivalent rules instead.

6.3 Characterizing Results

As shown in Fig. 6, we categorize the proved cases based on the SQL features that were used into the following:

- **UCQ**: Rewrites involving only unions of conjunctive queries, i.e., unions of SELECT-FROM-WHERE with conjunctive predicates.
- **Cond**: Rewrites that involve integrity constraints as preconditions, for instance a rewrite that is only valid in the presence of an index on a particular attribute.
- **Grouping, Aggregate, and Having**: Rewrites that use at least one of GROUP BY, aggregate functions such as SUM, and HAVING.
- DISTINCT **in Subquery**: Rewrites with DISTINCT in a subquery.

As Fig. 6 shows, UDP can formally prove the equivalence of many of the SQL rewrites described above. The running time of UDP on all these cases are within 15 seconds. Many such rewrites involve queries that are beyond UCQ, i.e., they are not part of the decidable fragment of SQL, such as the 3 rewrite rules from Starburst [44] (we described one of them in Sec. 5.4). To the best of our knowledge, none of these 3 rules (along with 35 other rules that UDP proved) were formally proven before. Proving the equivalences of these rewrite rules is non-trivial: it requires reasoning equivalence of queries evaluated under mixed of bag and set semantics, and modeling various preconditions and subqueries that use DISTINCT.

Fig. 7 shows the run time of UDP for proving the rewrites in each category. For the rewrite rules from Literature, UDP takes 6594.3 ms on average. For the ones from Calcite, UDP takes 4160.4 ms on average. As expected, UDP takes longer time on rewrites with rich

Dataset	Overall Avg.	UCQ	Cond.	Grouping, Aggregate, and Having	DISTINCT in Subquery
Literature	6594.3	3480.8	9983.9	8628.1	8223.7
Calcite	4160.4	2704.9	6429.0	6909.4	6427.7

Figure 7: UDP execution time (ms)

SQL features such as integrity constraints, grouping and aggregate, and DISTINCT in subquery.

An interesting question to ask is whether converting a U-expression to SPNF increases its size significantly in practice. Theoretically, Rule 1 and Rule 2 can increase the size of U-expression exponentially. We recorded the sizes of U-expression before and after converting them to SPNF. U-expression sizes increase by 4.1% on average in the Literature category, and increase by 0.7% on average in Calcite. Despite the exponential growth at the worst case, our evaluation shows that the growth of U-expressions after normalization is not a big concern.

Comparison to COSETTE. We also compare UDP with COSETTE [23]. UDP supports a wider range of SQL queries and provides more powerful *automated* proof search compared with COSETTE. In fact, COSETTE can only express 61 out of 69 cases that UDP proved (Fig. 5) as COSETTE does not support all types of database integrity constraints that UDP supports. For the 61 rules that COSETTE can express, only 17 of them (Ex. 4.7) was manually proven by COSETTE, and none of them can be proved automatically. As a comparison, COSETTE's manual proof script contains 320 lines of Coq to prove Ex. 4.7, in contrast to UDP automatically proving this rewrite rule.

6.4 Limitations

Unsupported SQL Features. UDP currently does not support SQL features such as CASE, UNION (under set semantics), NULL, and PARTITION BY. The queries in the rest of Calcite dataset (193 query pairs) contains at least one of these features and hence cannot be processed by our current prototype. Many of these features can be handled by syntactic rewrites. For example, UNION can be rewritten using UNION ALL and DISTINCT. Further engineering will enable us to support the majority of the remaining rewrite rules and they do not represent any fundamental obstacles to our approach.

Unproven Cases. There are a few rewrites that use only the supported features but UDP still fails to find proofs for them (6 out of 39 in the Calcite dataset). An example from Calcite is shown below:

```
SELECT *                                    -- Q1
FROM (SELECT * FROM EMP AS EMP
      WHERE EMP.DEPTNO = 10) AS t
WHERE t.DEPTNO + 5 > t.EMPNO;

SELECT *                                    -- Q2
FROM (SELECT * FROM EMP AS EMP0
      WHERE EMP0.DEPTNO = 10) AS t1
WHERE 15 > t1.EMPNO;
```

Proving the above rewrite requires modeling the semantics of integer arithmetic (which is undecidable in general), while other cases require modeling the semantics of string concatenation and conversion of strings to dates. We leave supporting such cases as future work.

7. RELATED WORK

Containment and equivalence of relational queries. Relational query containment and equivalence is a well studied topic in data management research. Equivalence of general relational queries has been proven to be undecidable [51], and subsequent research has focused on identifying decidable fragments of SQL, such as under set [17, 47] or bag semantics [24, 41, 18, 40]. As mentioned in Sec. 1, this line of work has focused on the theoretical aspects of the problem and has led to very few implementations, most of which has been restricted to applying the chase procedure to conjunctive queries for query optimization [14]. A recent work [38] proposes a new semantics to model many features of SQL (including NULL semantics), despite the lack of evaluation using real-world benchmarks and a usable implementation.

Semantic query optimization. Semantic query optimization is an important topic in query processing. While typical database engines optimizes queries using rule-based [44] or cost-based [34, 16] techniques, the line of work mentioned above has led to alternative approaches, most notably the chase and backchase (C&B) algorithm [46, 45, 27, 28], which guarantees to find a minimal semantic equivalent query for conjunctive queries under constraints. While our algorithm (Alg. 1) bears resemblance to C&B, our work fundamentally differs in that our goal is the find a formal, machine-checkable proof for the equivalence of two queries using a small number of axioms, while C&B aims to find a minimal equivalent query to the input. Second, our algorithm is sound for general SQL queries and complete for UCQ under set and bag semantics, while the original C&B are applicable only to CQ evaluated under set semantics [46, 45, 27], and the bag semantics version [28] is sparse in formal details and proofs.

Formalization of SQL semantics. The most related SQL formalizations include [23, 22, 21, 38, 13, 19]. COSETTE [23, 22, 21] formalized K-relation in the Coq proof assistant using univalent types in Homotopy Type Theory (HoTT) [50]. Compared to COSETTE, UDP has a much smaller axiomatic foundation and yet more powerful decision procedures that can find proof scripts for wider range of SQL queries. We have already discussed [38, 13] in detail in Sec. 2. Related formalizations of SQL or SQL like declarative languages in SMT solvers include Qex [54, 53], a tool for verifying the equivalences of Spark programs [37], Mediator, a tool for verifying database driven applications [56], and Blitz, a tool for synthesize big data queries [48]. Unlike UDP, Qex is used for test generation. The Spark verifier [37] can automatically verify the equivalences for a small set of Spark programs. However, it cannot be applied to SQL queries due to its syntactical restrictions. Mediator focuses on verifying transactions and programs that make updates to databases and is orthogonal to our work. Blitz [48] can only check SQL query equivalence up to bounded size inputs and is not a full verifier.

8. CONCLUSION

In this paper we presented U-semiring, a new semantics for SQL based on unbounded semirings. Using only a few axioms, U-semiring can model many SQL features including integrity constraints, which is not handled in prior work. To show the usefulness of U-semiring, we have developed a novel algorithm, UDP, for checking the equivalence of SQL queries and have used it to prove the validity of 62 real-world SQL rewrites, many of which were proven for the first time. As future work, we plan to support more SQL features and other non-relational data models such as Hive and Spark.

9. ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1614738, IIS-1651489, OAC-1739419, and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; the Intel-NSF CAPA center, and gifts from Adobe, Amazon, Google, and Huawei. We also thank the anonymous reviewers for their constructive feedback.

10. REFERENCES

[1] Apache Calcite Project. http://calcite.apache.org.

[2] Apache Drill Project. http://drill.apache.org.

[3] Apache Flink Project. https://flink.apache.org.

[4] Apache Hive Project. http://hive.apache.org.

[5] Apache Kylin Project. https://kylin.apache.org.

[6] Apache Phoenix Project. https://phoenix.apache.org.

[7] Bug 5673: Optimizer creates strange execution plan leading to wrong results. http://tinyurl.com/hwwn53r.

[8] MapD Database System. https://www.mapd.com.

[9] Q*Cert Proof of Selection Distributed over Union. https://github.com/querycert/qcert/blob/a2e924042ad44d1cb8abc352411c8ece8529d1a2/coq/NRA/Optim/NRARewrite.v#L66.

[10] Query featuring outer joins behaves differently in Oracle 12c. http://stackoverflow.com/questions/19686262.

[11] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[12] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD Conference*, pages 1383–1394. ACM, 2015.

[13] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In *SIGMOD Conference*, pages 1555–1569. ACM, 2017.

[14] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *SIGMOD Conference*, pages 37–52, 2017.

[15] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

[16] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of system R. *Commun. ACM*, 24(10):632–646, 1981.

[17] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, Proceedings of STOC, pages 77–90, 1977.

[18] S. Chaudhuri and M. Y. Vardi. Optimization of *Real* conjunctive queries. In *PODS*, pages 59–70. ACM Press, 1993.

[19] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14. ACM, 2013.

[20] S. Chu, A. Cheung, and D. Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *CoRR*, abs/1802.02229, 2018.

[21] S. Chu, D. Li, C. Wang, A. Cheung, and D. Suciu. Demonstration of the cosette automated SQL prover. In *SIGMOD Conference*, pages 1591–1594. ACM, 2017.

[22] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*. www.cidrdb.org, 2017.

[23] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017.

[24] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS*, pages 155–166. ACM Press, 1999.

[25] C. J. Date. *A Guide to the SQL Standard, Second Edition*. Addison-Wesley, 1989.

[26] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The lean theorem prover (system description). In *CADE*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[27] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, pages 459–470. Morgan Kaufmann, 1999.

[28] A. Deutsch, L. Popa, and V. Tannen. Chase & backchase: A method for query optimization with materialized views and integrity constraints. 01 2001.

[29] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017.

[30] Z. Ésik and W. Kuich. *Modern Automata Theory*.

[31] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 207–224. Springer, 2003.

[32] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD Conference*, pages 23–33, 1987.

[33] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms*. Springer, 1 edition, 2008.

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

[35] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[36] J. Gross, M. Shulman, A. Bauer, P. L. Lumsdaine, A. Mahboubi, and B. Spitters. The HoTT libary in Coq. https://github.com/HoTT/HoTT.

[37] S. Grossman, S. Cohen, S. Itzhaky, N. Rinetzky, and M. Sagiv. Verifying equivalence of spark programs. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 282–300. Springer, 2017.

[38] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1):27–39, 2017.

[39] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria big data management service. In *SIGMOD Conference*, pages 881–884. ACM, 2014.

[40] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Trans. Database Syst.*, 20(3):288–324, 1995.

[41] T. S. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for REAL conjunctive queries with inequalities. In *PODS*, pages 80–89. ACM, 2006.

[42] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[43] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[44] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD Conference*, pages 39–48. ACM Press, 1992.

[45] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far? In *SIGMOD Conference*, pages 273–284. ACM, 2000.

[46] L. Popa and V. Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 1999.

[47] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[48] M. Schlaipfer, K. Rajan, A. Lal, and M. Samak. Optimizing big-data queries using program synthesis. In *SOSP*, pages 631–646. ACM, 2017.

[49] P. Seshadri et al. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD Conference*, pages 435–446, 1996.

[50] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[51] B. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *D. Akad. Nauk USSR*, 70(1):569–572, 1950.

[52] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB J.*, 5(2):101–118, 1996.

[53] M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann. Symbolic query exploration. In *ICFEM*, pages 49–68, 2009.

[54] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 425–446. Springer, 2010.

[55] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria big data management and analytics system and cloud services. In *CIDR*. www.cidrdb.org, 2017.

[56] Y. Wang, I. Dillig, S. K. Lahiri, and W. R. Cook. Verifying equivalence of database-driven applications. *PACMPL*, 2(POPL):56:1–56:29, 2018.