Automated Reasoning of Database Queries

Shumo Chu

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Dan Suciu, Chair

Zachary Tatlock

Emina Torlak

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science

University of Washington

Abstract

Automated Reasoning of Database Queries

Shumo Chu

Chair of the Supervisory Committee:
Professor Dan Suciu
Paul G. Allen School of Computer Science, University of Washington

From booking air tickets to analyzing astronomy datasets, database queries are pervasive in people's work and life. This thesis describes Cosette, the first tool for automated reasoning the equivalences of SQL queries.

The core of Cosette is a formal semantics of SQL based on semirings. This semantics covers major SQL features, including sophisticated ones such as grouping, aggregate, correlated subqueries, and integrity constraints. Also, this semantics is denotational and only adds a few equational axioms, as the interpretation of SQL, to semirings. Then, to check the equivalences, Cosette uses this semantics to encode a pair of input SQL queries in both an interactive theorem prover and a constraint solver. In the end, Cosette will either certify their equivalences using a sound decision procedure implemented in a theorem prover that covers the known decidable fragment of SQL, or show their inequivalence by providing a counter-example. Empirical studies show that Cosette can decide the equivalence or provide counter example for a wide range of practical SQL queries collected from database literature, real-world optimizer rules and bugs, and data management class homework assignment from UW.

TABLE OF CONTENTS

Page

LIST OF FIGURES

GLOSSARY

SEMANTICS: a mathmatical interpretation of the meaning of a program.

DATABASE: an organized collection of data, stored and accessed electronically from a computer.

DATABASE MANAGEMENT SYSTEMS: the software that interacts with users, applications, and the database itself to create and manipulate data.

DATABASE QUERY: the commands/programs used for interacting with database management systems .

QUERY OPTIMIZATION: the process of rewriting database query based on semantics equivalences in order to make the execution of database queries more efficient.

INTERATIVE THEOREM PROVER: the software that allows user to write mathematical definitions, theorems, and mechanized proofs. In addition, it can checks the validity of the mechanized proofs, which are scripted programs that proves the specified theorems.

CONSTRAINT SOLVING: the process of finding a solution to a set of constraints that impose conditions that the varibles must statisfy.

ACKNOWLEDGMENTS

I am grateful to many people in my life. Without them, this thesis would not have been possible. Let me start by thanking my fellow Ph.D. students collaborators on my thesis project: Konstantin Weitz, Chenglong Wang, and Jared Roesch. None of this work would have been possible without them. I would always miss the great times spent with them on brainstorming and hacking.

Next, I would thank all my collaborators through out my Ph.D.: Daniel Halperin, Victor Teixira de Almeida, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Jingjing Wang, Andrew Whitaker, and Shengliang Xu. I learnt a lot from each of them. And more importantly, I truly enjoy the friendship with them. I am also very honored to work with a great undergraduate student: Daniel Li, and a superb high school student Brendan Murphy. It is a rare chance to work with brilliant younger computer science students. I am glad that I had that fortune. I would also thank all the students that sparked my research ideas and gaves me invaluable feedbacks: Pavel Panchekha, James Wilcox, Stuart Pernsteiner, James Bornholt, Sorawee Proncharoenwase, and Kaiyuan Zhang.

This journey would be way less enjoyable without remarkable lab mates: Maaz Ahmad, Brandon Haynes, Srini Iyer, Shrainik Jain, Ryan Mass, Laurel Orr, Jennifer Ortiz, Guna Prasaad, Shana Hutchison, Walter Cai, Cong Yan, Maureen Daum, and Jonathan Leang. I would give special thanks to my "beer mates": Vincent Lee, Alex Mariakakis, Shrainik Jain, and Ignacio Cano. Without them, it is not possible for me to get through the dark days in graduate school. I would also thank the supportive friends and office mates in the department: Hanchuan Li, Haichen Shen, Sophia Wang, Shu Liang, Danyang Zhuo, Yuchen Jin, Dianqi Li, Hao Peng, Qiao Zhang, Max Forbes, Lianhui Qin, Amanda Baughan and Siddharth Iyer.

During my Ph.D., I received great mentorship from faculty members in both database group and PLSE group. I would thank all these professors for shaping my research taste and motivating

my research pursuit: Magda Balazinska, Alvin Cheung, Bill Howe, Zachary Tatlock and Emina Torlak.

Last but not the least, I would like to thank my Ph.D. advisor, Dan Suciu, for being a supportive mentor as well as a role model for research and life. I appreciate so many things from Dan. Among them, his selfless support to student's research pursuit and fearless inquiries on intellectual contributions would always motivate me in my future career. I would also like to thank my master advisor, James Cheng. Without him, I would not start this journey from beginning.

DEDICATION

to my parents, Derong Liu and Yuanquan Chu

Chapter 1

INTRODUCTION

1.1 Motivation

From booking air tickets to analyzing astronomy data, people interact with database systems on a daily basis using SQL and SQL like declarative languages. In this report, we discuss the motivations, theories, and practices of building automated reasoners for SQL, the de facto database query language.

First, what is reasoning? While the philosophical implication of reasoning is deep [14]. My definition of reasoning is simple: *reasoning is about asking questions*. And automated reasoning is trying to use a computer rather than a person to answer questions. For database queries, the first and most fundamental question is the "yes and no" question: whether two queries are *semantically* equvialent, i.e. given any input databases, whether two database queries return the same result.

An automated reasoner is a computer program that can answer this question automatically. An automated reasoner for SQL can open up many promising applications:

Application 1: Improving the reliability of database systems. Every database system has a query optimizer that transform database queries from one form to the other based on semantic preserving rewrites. Hopefully, the rewritten query can be executed more efficiently. However, creating such rewrite rules is error prone. Latent bugs in major database systems takes years to be found [66], and new bugs continue to arise [13, 7]. An automated reasoner can be used to verify the correctness these query rewrites that previously requires extensive human efforts to verify and debug.

Application 2: Semantics based query caching. Database queries are executed increasing more in cloud and distributed file systems rather than single computer. These execution results are usu-

ally materialized. An automated reasoner can be the basis of a semantic caching layer of these cloud native or distributed database systems to save the execution cost of queries that their semantically equivalent ones have already been executed and materialized. Compared with traditional materialized view reusing rule based techniques [69], an automated reasoner perhaps can identify a broader range of query equivalences at the cost of increased reasoning time. This is a new trade off that is worth to explore.

Application 3: Help users learn and tweak database queries. One common senario of online data management class like MOOC is evaluating student's learning result by asking students write database queries according to a natural language specification. An automated reasoner can be used for grading students' solution against the standard ones. This would save the efforts of graders and scales the online education. In addition, the real world database optimizer is not perfect. Thus, in practice, users like to tweak their database queries to make their queries run faster. An automated reasoner can let database users "hack without fear" by verifying that the tweaked database queries are still semantic preserving.

The query equivalence problem is extensively studied. As discovered by Trakhtenbrot more than half a century ago, the equivalence of first order (FO) query is undecidable [108]. Since SQL can express all FO queries, it follows that SQL equivalence is undecidable, too. The subsequent research has been focus on identifying fragments of SQL where equivalence is decidable, e.g. conjunctive query under set semantics [38], unions of conjunctive query under set semantics [98], and conjunctive query under bag semantics (so called "real conjunctive queries") [41]. This line of work mostly study the theoretical aspects of the problem. There are very few implementations, most of which restricted to applying the chase procedure to conjunctive queries [32].

To enable the aforementioned applications, an automated reasoner needs to support real world SQL queries instead just supporting set semantics conjunctive queries like [32]. This is challenging. First, real world SQL queries uses bag semantics with the explicit duplicate elimination operator `DISTINCT` that converts bags to sets. There lacks decision procedure for the decidable fragment: conjunctive queries under bag semantics. Second, real world SQL queries are beyond

conjunctive queries `SELECT-FROM-WHERE`: many more features like aggregate and grouping, correlated subqueries, and integrity constraints need to be supported.

Given these challenges, building an automated reasoner for SQL seems to be formidable. There are two insights can be leveraged. First, as noted by Dijkstra [60]: *When exhaustive testing is impossible – i.e. almost always – our trust can only be based on proof (be it mechanized or not).* Although this problem is undecidable, it doesn't mean we cannot find proofs for an instance. On the contrary, many query equivalences in real-world has been proved, mostly informally. With the advancement of interactive theorem prover that can validate the mechanized proof, one can show query equivalences by constructing mechanized proofs and let an interactive theorem prover (e.g. Coq [15] or Lean [55]) validate these proofs. The second insight is that the inequivalent database queries usually have a small model, a.k.a. a counterexample on which the evaluation results of the two queries are different. This insight is known as "small world" phenomena and has been exploited by formal methods researchers to build fast modern constraint solvers. Leveraging this fact, one can build a model checker for SQL by compiling SQL queries into constraints and let the constraint solvers find counterexamples to show that they are inequivalent. An automated reasoner can be built by combining these two insights. In fact, automated reasoners of undecidable theories have been built using a similar approach [30, 65].

Organization: In Section 1.2, we start by introducing the development of interactive theorem provers and constraint solvers from formal methods research. Next, in Section 2.2, we cover the semantics of SQL and the efforts to mechanize SQL semantics in proof systems. In Section 1.4, we revisit chase, the decision procedure for a fragment of SQL, conjunctive queries. Last, we summarize the research contributions of this thesis in Section 1.5.

1.2 Related Formal Methods Development

In this section, we introduce two related developments in formal methods research that are helpfull for building an automated reasoner for SQL: validating mechanized proofs using interactive theorem provers Section 1.2.1 and model checking using constraint solvers Section 1.2.2.

1.2.1 Validating Mechanized Proofs using Interactive Theorem Provers

An interactive theorem prover is a piece of software that allows user to write mathematical definitions, theorems, and mechanized proofs. In addition, it checks the validity of the mechanized proofs, scripted programs that proves the specified theorems, programmatically. It has been used for certificating properties of programming languages and formalizing mathematics.

The popular interactive theorem provers include Coq [15], Isabelle [91], and Lean [55]. These interactive theorem provers all fulfill the de Bruign criterion: the theorem prover is built on a small generic trusted core. Coq and Lean both are based on dependent type theory. Properties, programs and proofs are formalized in the same language called Calculus of Inductive Constructions (CIC) [47]. Thanks to Curry-Howard correspondence [48], the proof validation is actually type checking in their generic trusted core. Isabelle's meta-logic is Logic for Computable Functions [71]. We only cover Lean subsequently.

Below is an example Lean code fragment:

```
theorem test (p q : Prop) : p -> q -> (p /\ q /\ p) :=
begin
  intro Hp, intro Hq,
  apply and.intro, exact Hp, apply and.intro,
  exact Hq, exact Hp
end
```

The first line states the theorem $\forall\ p\,q, p \to q \to (p \wedge q \wedge p)$ (p and q are boolean predicates). The following lines are mechanized proofs that can be validated by Lean.

Instead of writing proofs for each instance, Lean provided metaprogramming ability [61] for implementing *t*actics, prodecures which facilitate interactive theorem proving by carrying out straightforward reasoning step and boilerplate constructions. One of the key feature of Lean's metaprogramming interface is the ability to reflect the syntax of dependent type theory. This makes implementing more "intelligent" tactic possible. Below is an example of Lean tactics, which searchs current context to find an assumption that can be unified with the current goal.

```
meta def find : expr -> list expr-> tactic
expr
| e [] := failed
| e (h :: hs) :=
do t <- infer_type h,
(unify e t >> return h) <|> find e hs

meta def assumption : tactic unit :=
do { ctx <- local_context,
t <- target,
h <- find t ctx,
exact h }
<|> fail "assumption tactic failed"
```

Note that, `infer_type` is crucial here. It infers the type of proof terms in the context (which is known or assumed) so that the tactic `assumption` (`meta def assumption: ...`) could find the right one to put in the proof.

1.2.2 Bounded Model Checking using Constraint Solvers

Given a program and a property, *bounded model checking* checks if the property is satisfied by all executions of the system with $\leq k$ steps on all inputs of size $\leq n$. The common practice of developing a bounded model checker is encoding the program semantics and the properties to be checked into SAT constraints and asking a SAT solver to find a model. For example, if we want to do bounded model checking for a program p on the property (e.g. some safe property) s. We first covert p into SAT constraints C_p. This usually involves unwinding all loops (e.g. k times) and convert the program to Static Single Assignment form [49]. In addtion, all the input variables are grounded into boolean forms as well. Next, we specify the property to checked as assertions C_s. Finally, we call a SAT solver to solve $C_p \wedge \neg C_s$. If the SAT solver find a model, which is an

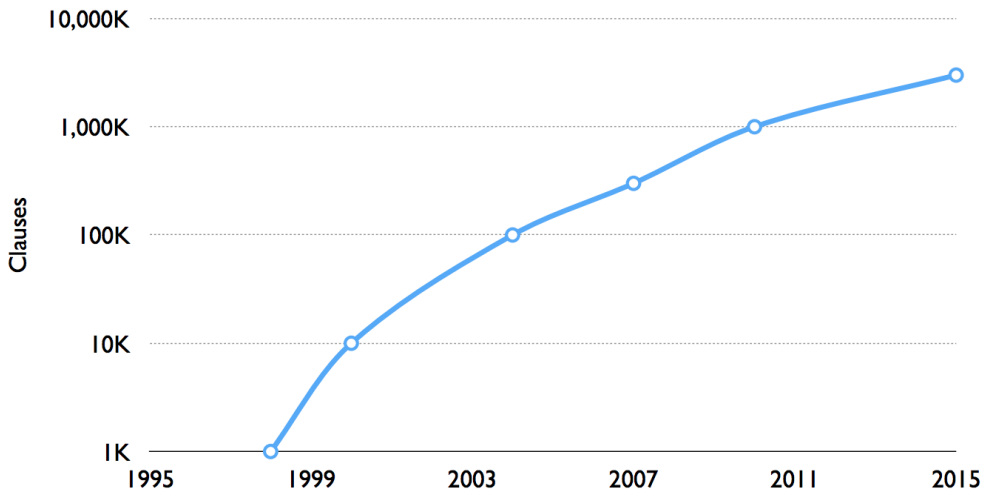

Figure 1.1: The evolution of SAT solvers (from [106])

assignment of the input variables. We then find a *counterexample* that violates the property s (it satisfies $\neg C_s$). Otherwise (the SAT solver cannot find a model), we can conclude that p will hold property s on all executions bounded by k.

What makes bounded model checking effective in practice is that the efficiency of SAT solvers increases dramatically in recent two decades (as shown in Figure 1.1). They has been successfully applied in several scenarios such as hardware verification [83] and a modular verification library on top of java collections [56].

However, reducing problems to SAT is like programming in assembly language. To make bounded verification easier, Satisfiability Modulo Theories (SMT) solvers that support richer logic (quantifier-free first-order logic) with combinations of background theories is developed [54]. Example of theories includes the theory of real numbers, the theory of integers, and theories of bit vectors. SMT solvers are usually built on top of SAT solvers. Popular SMT solvers include Z3[53] and CVC4[28].

In addition, people usually construct solver aided domain sepicific languages (SDSL) to make

ease of applying model checking to a specific domain. Users only need to write the simple programs in SDSL without the need of knowing how to convert the problem to SAT or SMT. One convenient way of constructing a SDSL is to create a shallow embedding (as a library) or a deep embedding (as a interpreter) in a solver aided language and symbolic virtual machine like Rosette [107].

1.3 Semantics of Database Queries

Although SQL is an ANSI standard [81], the semantics defined here is of litter use for building an automated reasoner. It is loosely described in English and has resulted in conflicting interpretations [51]. In this section, we discuss more "formal" semantics of SQL here. In Section 1.3.1, we introduce relational algebra, the widely used abstraction for database queries. Next in Section 1.3.2, we introduce K-relation, a more elegant formalism introduced by Green et al [74]. Finally in Section 1.3.3, we discuss efforts of developing mechanzied semantics for SQL.

1.3.1 Relational Algebra

Traditionally, relational algebra is defined in set semantics [19]. A *tuple* is an ordered n-tuple ($n \geq 0$) of constants (i.e., an element of the Cartesian product dom^n). A *relation* or relation instance over a relation schema $R[U]$ (U is a finite set of attributes) is a finite set I of tuples with sort U.

Then, relational algebra under set semantics can be defined using a family of algebra operators $\{\sigma, \pi, \times, \cup, -\}$ over relations:

Selection (σ): Selection operator applies a fitler condition over a relation:

$$\sigma_{j=a}(I) = \{t \in I | t(j) = a\}$$

For example, `SELECT * FROM Movie WHERE len = 100` can be represented as $\sigma_{len=100}(Movie)$.

Projection (π): Projection operator is used to delete and/or permute columns of a relation. The general form of the operator is $\pi_{j_1,\ldots,j_n}$, where $j_1, \ldots, j_n$ is a possibly empty sequence of positive

integers, possibly with repeats. It can be defined as:

$$\pi_{j_1,\ldots,j_n}(I) = \{\langle t(j_1), \ldots, t(j_n)\rangle | t \in I\}$$

For example, `SELECT first FROM Name` can be represented as $\pi_{first}(Name)$.

Cross-product ($\times$): Cross-product operator provides the ability to combine relations. It takes as input a pair of relations with arities n and m and returns a relation with arity $n + m$. It can be defined as:

$$I \times J = \{\langle t(1), \ldots, t(n), s(1), \ldots, s(m)\rangle | t \in I, S \in J\}$$

Union ($\cup$): Same as standard mathematics, $I_1 \cup I_2$ is the relation having this arity and containing the union of the two sets of tuples.

Difference ($-$): Similar as union, difference is the standard set difference operator that applied to relations.

SQL, the practical database query languages, is under bag semantics, where a relation is a multiset instead of the set. People usually use relational algebra under bag semantics to model the semantics of SQL. Relational albegra under bags uses the corresponding bag operators instead of set operators (i.e. bag selection, bag projection, bag cross-product, bag union, and bag difference) [67]. In addition, a duplicate elimination operator (δ) is introduced to covert a bag to a set.

Problems of relational algebra. Relational algebra is usually used as the semantics to guide the implementation of database systems, where the semantics preserving query transformations is based on rewrite rules. However, there is no meta-theory to check the correctness of these rewrite rules themselves, let alone two arbitrary database queries.

1.3.2 K-relation

A commutative semi-ring is a structure $\mathbf{K} = (K, +, \times, 0, 1)$ where both $(K, +, 0)$ and $(K, \times, 1)$ are commutative monoids, and $\times$ distributes over $+$. For a fixed set of attributes σ, denote $\texttt{Tuple}(\sigma)$

the type of tuples with attributes σ. A K-relation [74] is a function:

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

with finite support, meaning that $\{t \mid [\![R]\!]\ t \neq 0\}$ is finite.

Intuitively, $[\![R]\!]\ t$ represents the multiplicity of t in R. For example, a bag is an $\mathbb{N}$-relation, and a set is a $\mathbb{B}$-relation. All relational operators are expressed in terms of the semi-ring operations, for example:

$$[\![R\ \texttt{UNION ALL}\ S]\!] = \lambda\, t\ .\ [\![R]\!]\ t + [\![S]\!]\ t$$

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

$$[\![\texttt{SELECT * FROM}\ R\ \ x\ \texttt{WHERE}\ b]\!] = \lambda\, t\ .\ [\![R]\!]\ t \times [\![b]\!]\ t$$

$$[\![\texttt{SELECT}\ x.a_1\ \texttt{as}\ a_2\ \texttt{FROM}\ R\ \ x]\!] =$$

$$\lambda\, t_2\ .\ \sum_{t_1 \in \texttt{Tuple}\ \sigma_1} ([\![a_2]\!]t_2 = [\![a_1]\!]t_1) \times [\![R]\!]\ t_1$$

where, for any predicate b: $[\![b]\!]\ t = 1$ if the predicate holds on t, and $[\![b]\!]\ t = 0$ otherwise. One advantages of a semi-ring semantics is that formal proofs of equivalences can be reduced to applying semi-ring axioms.

1.3.3 Mechanized Semantics

Most previous approaches to mechanizing formal proofs of SQL query equivalences represent bags as lists [86, 112, 113]. Every SQL query admits a natural interpretation over lists, using a recursive definition [36]. To prove that two queries are equivalent, one uses their inductive definition on lists, and proves that the two results are equal up to element reordering and duplicate elimination (for set semantics). The challenges in this approach are coming up with the induction hypothesis, and dealing with list equivalence under permutation and duplicate elimination. Inductive proofs quickly grow in complexity, even for simple query equivalences. Consider the following queries:

```
SELECT DISTINCT x.a AS a FROM R x  -- Q2
SELECT DISTINCT x.a AS a FROM R x, R y WHERE x.a = y.a -- Q3
```

`Q3` is equivalent to `Q2`, because it performs a redundant self-join: the inductive proof of their equivalence is quite complex, and has, to the best of our knowledge, not been done formally before. A much simpler rewrite rule, the commutativity of selection, requires 65 lines of Coq proof in prior work [86], Powerful database query optimizations, such as magic sets rewrites and conjunctive query equivalences, are based on generalizations of redundant self-joins elimination like `Q2` $\equiv$ `Q3`, but significantly more complex, and inductive proofs become impractical.

Grossman et al. [76] encode a small fragment (`map, filter, fold, foldByKey`) of Spark program to SMT formulas. Their semantics restriction makes automated verfication possible.

1.4 Deciding Equivalences of Conjunctive Queries

In this section, we discuss the algorithms that can decide the equivalences of conjunctive queries.

Conjuntive Queries (CQs) are the SQL queries that only contains `SELECT-FROM-WHERE` and conjunctive predicates. Alternatively, they are relational algebra expressions with only $\sigma, \pi, \times$.

Under Set Semantics For two queries q_1, q_2 over the same schema R, q_1 is contained in q_2, denoted $q_1 \subseteq q_2$, if for each I over R, $q_1(I) \subseteq q_2(I)$. We can conclude $q_1 \equiv q_2$ if and only if $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$. The homomorphism threorem shows a characterization for containment and equivalence of conjunctive queries [19]:

Theorem 1.4.1. *Let $q = (T, u)$ and $q' = (T', u')$ be conjunctive queries over the same schema R. Then $q \subseteq q'$ if and only if there exists a homomorphism from (T', u') to (T, u).*

Here, a homomorphism from q' to q is a substituion θ such that $\theta(T') \subseteq T$ and $\theta(u') = u$.

Under Bag Semantics CQ containment and equivalence under bag semantics results are from Chaudhuri and Vardi [41]. Under bag semantics, the containment problem for conjunctive queries is Π_2^p-hard. The exact complexity of the query containment under bag semantics is still open. However, the equivalence of conjunctive queries under bag semantics is polynomially equivalent to graph isomorphism:

Theorem 1.4.2. *Let q and q' be conjunctive queries under bag semantics, $q \equiv q'$ if and only if q and q' are isomorphic.*

With constraints As a database query and additional constraints like uniqueness and referential constraints could be both expressed as first order logic sentences. A chase is equivalent to apply Modus ponens ($\{A \to B, A\} \Rightarrow B$) over a database query.

The standard chase algorithm for database queries is defined on embedded dependencies [63] in the form of:

$$\phi(\bar{u}, \bar{w}) \to \exists v\ \psi(\bar{u}, \bar{v})$$

Then the query equivalence under constraints can be solved by keep applying chase until an "Universal Plan" is reached. The problem is reduced to homomorphism again. There is no detailed chase algorithm under bag semantics in peer reviewed publication except a special case in a technical report [59].

1.5 Summary of Research Contributions

Mechanized Semantics for SQL

As discussed in Section 2.2, enabling automated reasoning on existing mechanized semantics is difficult. We need a "better" encoding of SQL to make reasoning SQL equivalence easier.

Our first contribution is a new mechanized semantics for SQL that is both simple and allows easy equivalence proofs [44]. Our semantics consists of two non-trivial generalizations of K-relations [74], which represent a relation as a mathematical function that takes as input a tuple and returns its multiplicity, with 0 meaning that the tuple does not exist in the relation. A K-relation is required to have finite support, meaning that only a finite set of tuples have multiplicity > 0. K-relations greatly simplify reasoning about SQL: under set semantics, a relation is simply a function that returns 0 or 1 (i.e., a Boolean value), while under bag semantics it returns a natural number (i.e., a tuple's multiplicity). Database operations such as join or union denote into arithmetic operations on the corresponding multiplicities: join becomes multiplication, union becomes addition. Checking if two queries are equivalent reduces to checking the equivalence of the functions they denote; for example, proving that the join operation is associative reduces to proving that multi-

plication is associative. As we will show, reasoning about functions over cardinals is much easier than writing inductive proofs on data structures such as lists.

However, K-relations, as defined by [74], are difficult to use in proof assistants, because one needs to prove for every SQL expression that its result has finite support. This is easy with pen-and-paper, but hard to encode for a proof assistant. Without a guarantee of finite support, some operations are undefined, for example projection on an attribute leads to infinite summation. Hence, our first generalization of K-relations is to drop the finite support requirement, and meanwhile allow the multiplicity of a tuple to be any cardinal number as opposed to a (finite) natural number. Then the possibly infinite sum corresponding to a projection is well defined. With this change, a relation in SQL is interpreted as a possibly infinite bag, where a tuple may have a possibly infinite multiplicity. To the best of our knowledge, ours is the first SQL semantics that interprets relations as both finite and infinite.

Our second generalization of K-relations is to replace cardinal numbers with univalent types. Homotopy Type Theory (HoTT) [105] has been introduced recently as a generalization of classical type theory by adding membership and equality proofs. A *univalent type* is a cardinal number (finite of infinite) together with the ability to prove equality. Since univalent types have been integrated into the Coq proof assistant, this was a convenient engineering choice for us, which simplified many of the formal proofs.

In summary, in our semantics a relation is a function mapping each tuple to a univalent type (representing its multiplicity), and a SQL query is a function from relations to relations. Our semantics covers all major features of SQL, including set and bag semantics, nested queries, etc. We implemented our semantics as part of a new tool called COSETTE [43, 42] for proving equivalence of SQL rewrite rules, and used it to prove many rewrite rules from the data management research literature, some of which have never been formally proven before: aggregates [40], magic sets rewriting [26], query rewriting using indexes [110], and equivalence of conjunctive queries using the homomorphism theorem [19]. In the implementation of COSETTE, all our proofs require at most a few dozens lines of Coq code. All definitions and proofs presented in this paper are

open-source and available online.[1]

Axiomatic Foundations and Automating Proofs

With the new mechanized semantics for SQL, we constructed machine-checkable proofs for a wide range of rewrite rules, many of them are first time proven. However, it is hard to imagine that a practioner who is developing a database system has to write Coq proofs whenever she want to add a new rewrite rule to the database system. Our next contribution is an axiomation of SQL semantics, based on the mechanized semantics with univalent types in Homotopy Type Theory (HoTT).

In our previous mechanized semantics [44], it resorts HoTT as the meta-theory: an equivalence proof of a pair of SQL queries relies on proving the equivalence of two univalent types. While this works in practice, we realized that SQL's equivalent proofs may not require such a complex meta-theory. In addition, lacking automation tactics in the extended Coq with HoTT library prevent us from futher automate the proof construction. 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 frame-

[1] `http://cosette.cs.washington.edu`

work 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,[2] 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 [94] 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 [55]. 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.

Determine Inequivalences of SQL Queries

In addition to proving equvialent SQL queries, a automated reasoning tool needs to report inequivalent SQL queries as well. Our third contribution is developing techniques that can determine that the input queries are inequivalent by using contraint solvers to find counterexamples, a input database on which the two input queries will evaluate to different results.

---

[2] 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.

In order to find counterexamples, we developed a SQL symbolic execution tool powered by constraint solvers. Given a logic formula containing symbolic (i.e., unknown) variables, constraint solvers are developed with various specialized heuristics [52, 90] that can efficiently find *models*, i.e., values for the symbolic variables, that make the formula true, or return unsatisfiable otherwise. For instance, given the Boolean logic formula `v1 && (v2 || v3) == (v1 && v2) || (v1 && v3)`, a constraint solver might return the model `{v1:True, v2:True, v3:False}`. Users could use this model-finding property of constraint solvers to show the *falsity* of logic formulas: if the solver can find a model (also called a counterexample in this case) for the negated formula, then the original formula must be false. This strategy has been applied in numerous real-world scenarios (e.g., [114]).

However, mapping SQL queries to solver constraints is not easy, for similar reasons with the difficulties of modeling SQL in interactive theorem provers. We didn't convert SQL queries to solver contraints directly. Instead, we use Rosette [107], a solver aided programming language and symbolic virtual machine to build a symbolic SQL executor. For a given pair of SQL queries, this symolic SQL executor will try to find counterexamples of these input queries by calling Z3 [53], the underlying SMT Solver. In order to make the solving more efficient, we make various efforts. First, we use a "compressed" relation encoding by letting the cardinalities of database tuples be symbolic variables. Second, we make the solving process incremental by using smaller symbolic relations first to avoid necessary exploration of search spaces and to get smaller models as possible.

We evaluate the effectiveness of our symbolic executor using real world bugs in database management systems, data managament class exam questions, and queries from SQL test generation framework. Our symbolic executor is able to counterexamples for all the input queries within a short period of time (< 10 Sec).

Chapter 2

*HoTT*SQL :A MECHANIZED SQL SEMANTICS WITH UNIVALENT TYPES

In this chapter, we describe a mechanized SQL semantics with univalent types. Section 4.3 overviews the idea of our mechanized SQL semantics. Section 2.2 presents the constructs of the core part and derived part of our semantics. Section 3.2.2 shows how to translate *HoTT*SQL to expressions in univalent types. Section 3.5.2 demonstrates the implementation of *HoTT*SQL and rewrite rules proven using *HoTT*SQL. Section 2.5 discusses the limitations of this approach. Section 2.6 summarizes this chapter.

2.1 Overview

SQL The basic datatype in SQL is a *relation*, which has a *schema* (a relation name R plus attribute names σ), and an *instance* (a bag of tuples). A SQL query maps one or more input relations to a (nameless) output relation. For example, if a relation with schema $R(a, b)$ has instance $\{(1, 40), (2, 40), (2, 50)\}$ then the SQL query

```
SELECT x.a AS a FROM R x  -- Q1
```

returns the bag $\{1, 2, 2\}$.

SQL freely mixes set and bag semantics, where a set is simply a bag without duplicates. One uses the `distinct` keyword to remove duplicates. For example, the query:

```
SELECT DISTINCT x.a AS a FROM R x  -- Q2
```

returns the set $\{1, 2\}$ given the definition of $R(a, b)$ above.

List Semantics Previous approaches to mechanizing formal proofs of SQL query equivalences represent bags as lists [86, 112, 113]. Every SQL query admits a natural interpretation over lists,

Rewrite Rule:

```
SELECT * FROM (R UNION ALL S) x WHERE b ≡
(SELECT * FROM R x WHERE b) UNION ALL
(SELECT * FROM S y WHERE b)
```

Denotation:

$$\Rightarrow \lambda\, t\, .\, (\llbracket R \rrbracket\, t + \llbracket S \rrbracket\, t) \times \llbracket b \rrbracket\, t \equiv$$
$$\lambda\, t\, .\, \llbracket R \rrbracket\, t \times \llbracket b \rrbracket\, t + \llbracket S \rrbracket\, t \times \llbracket b \rrbracket\, t$$

Proof: Apply distributivity of $\times$ over $+$.

Figure 2.1: Proving a rewrite rule using *HoTT*SQL. Recall that `UNION ALL` means bag-union in SQL, which in *HoTT*SQL is translated to addition of tuple multiplicities in the two input relations.

using a recursive definition [36]. To prove that two queries are equivalent, one uses their inductive definition on lists, and proves that the two results are equal up to element reordering and duplicate elimination (for set semantics). The challenges in this approach are coming up with the induction hypothesis, and dealing with list equivalence under permutation and duplicate elimination. Inductive proofs quickly grow in complexity, even for simple query equivalences. Consider the following query:

```
SELECT DISTINCT x.a AS a FROM R x, R y WHERE x.a = y.a  -- Q3
```

`Q3` is equivalent to `Q2`, because it performs a redundant self-join: the inductive proof of their equivalence is quite complex, and has, to the best of our knowledge, not been done formally before. A much simpler rewrite rule, the commutativity of selection, requires 65 lines of Coq proof in prior work [86], and only 10 lines of Coq proof in our semantics. Powerful database query optimizations, such as magic sets rewrites and conjunctive query equivalences, are based on generalizations of redundant self-joins elimination like `Q2` $\equiv$ `Q3`, but significantly more complex (see Sec. 3.5.2), and inductive proofs become impractical. This motivated us to consider a different semantics; we do

not use list semantics in this paper.

K-Relation SQL Semantics A commutative semi-ring is a structure $\mathbf{K} = (K, +, \times, 0, 1)$ where both $(K, +, 0)$ and $(K, \times, 1)$ are commutative monoids, and $\times$ distributes over $+$. For a fixed set of attributes σ, denote $\texttt{Tuple}(\sigma)$ the type of tuples with attributes σ. A K-relation [74] is a function:

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

with finite support, meaning that $\{t \mid [\![R]\!]\ t \neq 0\}$ is finite.

Intuitively, $[\![R]\!]\ t$ represents the multiplicity of t in R. For example, a bag is an $\mathbb{N}$-relation, and a set is a $\mathbb{B}$-relation. All relational operators are expressed in terms of the semi-ring operations, for example:

$$[\![R\ \texttt{UNION ALL}\ S]\!] = \lambda\, t\ .\ [\![R]\!]\ t + [\![S]\!]\ t$$

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

$$[\![\texttt{SELECT * FROM}\ R\ \ x\ \texttt{WHERE}\ b]\!] = \lambda\, t\ .\ [\![R]\!]\ t \times [\![b]\!]\ t$$

$$[\![\texttt{SELECT}\ x.a_1\ \texttt{as}\ a_2\ \texttt{FROM}\ R\ \ x]\!] =$$

$$\lambda\, t_2\ .\ \sum_{t_1 \in \texttt{Tuple}\ \sigma_1} ([\![a_2]\!]t_2 = [\![a_1]\!]t_1) \times [\![R]\!]\ t_1$$

$$[\![\texttt{SELECT DISTINCT * FROM}\ R\ \ x]\!] = \lambda\, t\ .\ \|[\![R]\!]\ t\|$$

where, for any predicate b: $[\![b]\!]\ t = 1$ if the predicate holds on t, and $[\![b]\!]\ t = 0$ otherwise. The function $\|\ \|$ is defined as $\|x\| = 0$ when $x = 0$, and $\|x\| = 1$ otherwise (see Sec. 2.2.3). The projection $[\![a]\!]\ t$ returns the attribute a of the tuple t, while equality $(x = y)$ is interpreted as 0 when $x \neq y$ and 1 otherwise.

To prove that two SQL queries are equal one has to prove that two semi-ring expressions are equal. For example, Fig. 2.1 shows how we can prove that selections distribute over unions, by reducing it to the distributivity of $\times$ over $+$, while Fig. 2.2 shows the proof of the equivalence for $\texttt{Q2} \equiv \texttt{Q3}$.

Notice that the definition of projection requires that the relation has finite support; otherwise, the summation is over an infinite set and is undefined in $\mathbb{N}$. This creates a major problem for

Rewrite Rule:

```
SELECT DISTINCT x.a AS a1 FROM R x, R y WHERE x.a = y.a
≡ SELECT DISTINCT x.a AS a1 FROM R x
```

Equational *HoTT*SQL Proof:

$$
\begin{aligned}
\Rightarrow & \lambda\ t\ .\ \left\| \sum_{t_1,t_2} (t = [\![a]\!]\ t_1) \times ([\![a]\!]\ t_1 = [\![a]\!]\ t_2) \times [\![R]\!]\ t_1 \times [\![R]\!]\ t_2 \right\| \\
\equiv & \ \lambda\ t\ .\ \left\| \sum_{t_1,t_2} (t = [\![a]\!]\ t_1) \times (t = [\![a]\!]\ t_2) \times [\![R]\!]\ t_1 \times [\![R]\!]\ t_2 \right\| \\
\equiv & \ \lambda\ t\ .\ \| (\sum_{t_1} (t = [\![a]\!]\ t_1) \times [\![R]\!]\ t_1) \times \\
& \qquad (\sum_{t_2} (t = [\![a]\!]\ t_2) \times [\![R]\!]\ t_2) \| \\
\equiv & \ \lambda\ t\ .\ \left\| \sum_{t_1} (t = [\![a]\!]\ t_1) \times [\![R]\!]\ t_1 \right\|
\end{aligned}
$$

We used the following semi-ring identities:

$$
\begin{gathered}
(a = b) \times (b = c) \equiv (a = b) \times (a = c) \\
\sum_{t_1,t_2} E_1(t_1) \times E_2(t_2) \equiv \sum_{t_1} E_1(t_1) \times \sum_{t_2} E_2(t_2) \\
\|n \times n\| \equiv \|n\|
\end{gathered}
$$

Deductive *HoTT*SQL Proof:

$$
\begin{aligned}
\Rightarrow & \ \forall\ t. \exists_{t_0} ([\![a]\!]\ t_0 = t) \wedge [\![R]\!]\ t_0 \leftrightarrow \\
& \quad \exists_{t_1,t_2} ([\![a]\!]\ t_1 = t) \wedge [\![R]\!]\ t_1 \wedge [\![R]\!]\ t_2 \wedge ([\![a]\!]\ t_1 = [\![a]\!]\ t_2)
\end{aligned}
$$

Then case split on $\leftrightarrow$. Case $\rightarrow$: instantiate both t_1 and t_2 with t_0, then apply hypotheses. Case $\leftarrow$: instantiate t_0 with t_1, then apply hypotheses.

Figure 2.2: The proof of equivalence `Q2` $\equiv$ `Q3`.

our equivalence proofs, since we need to prove, for each intermediate result of a SQL query, that it returns a relation with finite support. This adds significant complexity to the otherwise simple proofs of equivalence.

***HoTT*SQL Semantics** To handle this challenge, our semantics generalizes K-Relation in two ways: we no longer require relations to have finite support, and we allow the multiplicity of a tuple to be an arbitrary cardinality (possibly infinite). More precisely, in our semantics a relation is interpreted as a function:

$$[\![R]\!] : \texttt{Tuple}\ \sigma \to \mathcal{U}$$

where $\mathcal{U}$ is the class of homotopy types. We call such a relation a *HoTT-relation*. A homotopy type $n \in \mathcal{U}$ is an ordinary type with the ability to prove membership and equality between types.

Homotopy types form a commutative semi-ring and can well represent cardinals. The cardinal number 0 is the empty homotopy type $\mathbf{0}$, 1 is the unit type $\mathbf{1}$, multiplication is the product type $\times$, addition is the sum type $+$, infinite summation is the dependent product type Σ, and truncation is the squash type $\|\cdot\|$. Homotopy types generalize natural numbers and their semiring operations, and is now well integrated with automated proof assistants like Coq.[1] As we will show in the rest of this paper, the equivalence proofs retain the simplicity of $\mathbb{N}$-relations and can be easily mechanized, but without the need to prove finite support.

In addition, homotopy type theory unifies squash types and propositions. Using the fact that propositions are types in homotopy type theory [105, Ch 1.11], in order to prove the equivalence of two squash types, $\|p\|$ and $\|q\|$, it suffices to just prove the bi-implication of types (i.e., $p \leftrightarrow q$), which is arguably easier in Coq. For example, transforming the equivalence proof of Figure 2.2 to bi-implication would not require a series of equational rewriting using semi-ring identities any more, which is complicated because it is under the variable bindings of the dependent product type Σ. The bi-implication can be easily proved in Coq by deduction.

The queries in rewrite rule shown in Figure 2.2 fall under the well-studied category of conjunctive queries, for which equality is decidable (while equality between arbitrary SQL queries is

[1] After adding the Univalence Axiom to Coq's underlying type theory.

undecidable). Using Coq's support for automating deductive reasoning (i.e., *Ltac*), we have implemented a decision procedure to determine the equality of conjunctive queries. With that, the aforementioned rewrite rule can be proven in one line of Coq code.

Extending K-relations to infinite size frees us from needing to prove the finiteness of relations in every proof step. Furthermore, using homotopy types gives us an easy way to model cardinal numbers (to represent the size of relations), which is not supported by Coq's standard library. Even though we allow relations to be of infinite in size in *HoTT*SQL, doing so does not alter the meaning of queries as compared to standard SQL, as standard SQL does not give semantics to relations of infinite size. However, in practice, we are unaware of any scenarios where distinguishing between finite vs infinite relations matters.

*2.2 HoTT*SQL *and Its Semantics*

In this section, we present *HoTT*SQL, our formal representation of SQL which covers all major features of the SQL query language. *HoTT*SQL is in fact a superset of SQL, since it includes additional language constructs, like generic predicates that range over all Boolean predicates, or generic schemas that range over a set of schemas. As we will see, these are needed to express and prove rewrite rules (i.e., query equivalences) that hold for a set concrete instantiations of generic predicates and schemas.

2.2.1 Data Model

We first describe how schemas for relations and tuples are modeled in *HoTT*SQL. Both of these foundational concepts are from relational theory [**?**] that *HoTT*SQL builds upon.

Schema and Tuple Conceptually, a database schema is an unordered bag of attributes represented as pairs of (n, τ), where n is the attribute name, and τ is the type of the attribute. We assume that each of the SQL types can be translated into their corresponding Coq type.

In *HoTT*SQL, a user can declare a schema σ with three attributes as follows:

```
schema σ(Name:string, Age:int, Married:bool);
```

A database tuple is a collection of values that conform to a given schema. For example, the following is a tuple with the aforementioned schema:

$$\{\text{Name} : \text{“Bob”};\ \text{Age} : 52;\ \text{Married} : \texttt{true}\}$$

Attributes are accessed using record syntax. For instance t.Name returns “Bob” where t refers to the tuple above.

The goal of *HoTT*SQL is to enable users express rewrite rules over arbitrary schemas. As such, we distinguish between *concrete* schemas, in which all of their attributes are known, from *generic* schemas, which might contain unknown attributes that are denoted using ??. For example, the following rewrite rule:

```
SELECT x.a AS a FROM R x WHERE TRUE AND x.a = 10
≡ SELECT x.a AS a FROM R x WHERE x.a = 10
```

holds for any table with a schema containing at least the integer attribute a. In *HoTT*SQL, this is expressed as a generic schema that is declared as:

$$\texttt{schema}\ \sigma(\texttt{a:int, ??})$$

Relation In *HoTT*SQL a relation is modeled as a function from tuples to homotopy types called HoTT-Relations with type $\texttt{Tuple}\ \sigma \to \mathcal{U}$, as discussed in Section 4.3; we will define homotopy types shortly.

*2.2.2 HoTT*SQL*: A DSL to express SQL rewrites*

We now describe *HoTT*SQL, our frontend language for expressing rewrite rules. Figure 3.2 shows the syntax of *HoTT*SQL.

A *HoTT*SQL program consists of statements. Each statement is either a declaration (of a schema, table, predicate, function, or an aggregate) or a verify statement that contains two SQL queries whose equivalence is to be checked. Each SQL query in the verify statement is constructed using the declarations mentioned.

Schema and table declaration As described in Section 2.2.1, a schema declaration statement declares a schema with its attributes and the type of each attribute. A generic schema is declared

by appending ?? after the last known attribute. Additionally, a table declaration declares a relation of a declared schema. Notice that a table declaration represents either a base table or a SQL subquery. For example, the rule in Figure 2.3 expresses an identity of two queries where t_1, t_2 stand for either table names *or* subqueries.

Predicate declaration A predicate declaration declares a generic predicate that ranges over all possible Boolean predicates on a list of schemas (e.g., `predicate` $\beta_1(\sigma_1, \sigma_2)$`;` in Figure 2.3). Concrete predicates, such as `x.a=1`, are written explicitly in *HoTT*SQL. When a generic predicate is used in a query, users need to provide a list of tuple variables that the predicate will be applied to (e.g., $\beta_1(x, y)$ in the verify statement in Figure 2.3, where x and y are tuple variables ranging over the relations t_1 and t_2 respectively).

Function declaration A function declaration declares an uninterpreted function of expressions $f(e_1, \ldots, e_n)$ that are used to represent arithmetic operations, including constants (which are nullary functions).

Aggregate declaration An aggregate declaration constructs a generic aggregate (i.e., one that ranges over all of the six aggregates defined in SQL) and the schema of relation that it will be applied.

Verify Statement A verify statement states the two queries whose equivalence is to be decided. Each query is a standard SQL query that uses the declarations mentioned earlier. Figure 2.3 shows an example *HoTT*SQL program illustrating the rewrite rule that a generic predicate (β_2) applied to table t_2 can be pushed down into a subquery on t_2. While previous formalization in database literature like relational algebra lacks the ability to reason about generic predicates, *HoTT*SQL allows users to express rewrite rules with generic predicates in a concise and user-friendly way.

2.2.3 UniNomials

To prove the equivalence of two *HoTT*SQL SQL queries, we interpret *HoTT*SQL's `Query` expressions using UNINOMIALS, which is an algebra based on univalent types.

```
schema σ1(??); schema σ2(??);
table t1(σ1); table t2(σ2);
predicate β1(σ1, σ2);
predicate β2(σ2);
verify SELECT * FROM t1 x, t2 y WHERE β1(x, y) AND β2(y)
    ≡ SELECT *
      FROM t1 x, (SELECT * FROM t2 y WHERE β2(y)) z
      WHERE β1(x, z);
```

Figure 2.3: An example *HoTT*SQL program

Definition 2.2.1. UNINOMIAL*, the algebra of univalent Types, is* $(\mathcal{U}, \mathbf{0}, \mathbf{1}, +, \times, \cdot \to \mathbf{0}, \|\cdot\|, \sum)$*, where:*

- $(\mathcal{U}, \mathbf{0}, \mathbf{1}, +, \times)$ *forms a semi-ring, where* $\mathcal{U}$ *is the universe of univalent types,* $\mathbf{0}, \mathbf{1}$ *are the empty and singleton types, and* $+, \times$ *are binary operations.* $\mathcal{U} \times \mathcal{U} \to \mathcal{U}$*:* $n_1 + n_2$*, is the direct sum, and* $n_1 \times n_2$ *is the Cartesian product.*

- $\cdot \to \mathbf{0}, \|\cdot\|$ *are derived unary operations with type* $\mathcal{U} \to \mathcal{U}$*, where* $(\mathbf{0} \to \mathbf{0}) = \mathbf{1}$ *and* $(n \to \mathbf{0}) = \mathbf{0}$ *when* $n \neq 0$*, and* $\|n\| = (n \to \mathbf{0}) \to \mathbf{0}$*.*

- $\sum : (A \to \mathcal{U}) \to \mathcal{U}$ *is the infinitary operation:* $\sum f$ *is the direct sum of the set of types* $\{f(a) \mid a \in A\}$*.*

Importantly for us, every natural number k is represented by a finite univalent type consisting of k elements. Following standard notation [105], we say that the type n inhabits some universe $\mathcal{U}$. The base cases of n come from the denotation of HoTT-Relation and equality of two tuples (In HoTT, propositions are squash types, which are either **0** or **1** [105, Ch 1.11]). There are 5 type-theoretic operations on $\mathcal{U}$:

Cartesian product ($\times$) Cartesian product of univalent types is analogous to Cartesian product of two sets. For $A, B : \mathcal{U}$, the cardinality of $A \times B$ is the cardinality of A multiplied by the cardinality of B; when A, B are natural numbers, then $A \times B$ is their standard product. For example, the cross product of two HoTT-Relations is defined using the Cartesian product of univalent types:

$$[\![\texttt{SELECT * FROM } R_1 \ x, \ R_2 \ y]\!] \triangleq \lambda \, t. \ ([\![R_1]\!] \, t.1) \times ([\![R_2]\!] \, t.2)$$

In English: the cross product of R_1 and R_2 is a HoTT-Relation of type $\texttt{Tuple} \ (\texttt{node} \ \sigma_{R_1} \ \sigma_{R_2}) \to \mathcal{U}$, defined as follows. The number of times that a tuple t occurs in the output schema equals to the product of how many times $t.1$ occurs in R_1 multiplied by how many times $t.2$ occurs in R_2.

Disjoint union ($+$) Disjoint union of univalent types is analogous to union of two disjoint sets. For $A, B : \mathcal{U}$, the cardinality of $A + B$ is the cardinality of A plus the cardinality of B; for natural numbers, this corresponds to standard addition. For example, SQL's `UNION ALL` is defined in terms of $+$:

$$[\![R_1 \ \texttt{UNION ALL} \ R_2]\!] \triangleq \lambda \, t. \ ([\![R_1]\!] \, t) + ([\![R_2]\!] \, t)$$

In SQL, `UNION ALL` means bag union of two relations.

We also denote logical `OR` of two predicates using $+$. $A + B$ corresponds type-theoretic operation of logical OR if A and B are squash types (recall that squash types are $\mathbf{0}$ or $\mathbf{1}$ [105, Ch 1.11]).

Squash ($\|n\|$) Squash is a type-theoretic operation that truncates a univalent type to $\mathbf{0}$ or $\mathbf{1}$. For $A : \mathcal{U}$, $\|A\| = \mathbf{0}$ if A's cardinality is zero or $\|A\| = \mathbf{1}$ otherwise. An example of using squash types is in denoting `DISTINCT` (`DISTINCT` removes duplicated tuples, i.e., converting a bag to a set):

$$[\![\texttt{DISTINCT } R]\!] \triangleq \lambda \, t. \ \|[\![R]\!] \, t\|$$

For a tuple $t \in \texttt{DISTINCT } R$, t's cardinality equals to 1 if its cardinality in R is non-zero and equals to 0 otherwise. This is exactly $\|[\![R]\!] \, t\|$.

Negation ($n \to \mathbf{0}$) If n is a squash type, then $n \to \mathbf{0}$ is the negation of n. We have $\mathbf{0} \to \mathbf{0} = \mathbf{1}$ and $\mathbf{1} \to \mathbf{0} = \mathbf{0}$. Negation is used to denote `EXCEPT` and negating a predicate, for example:

$$[\![R_1 \ \texttt{EXCEPT} \ R_2]\!] \triangleq \lambda \, t. \ ([\![R_1]\!] \, t) \times (\|[\![R_2]\!] \, t\| \to \mathbf{0})$$

A tuple $t \in R_1$ `EXCEPT` R_2 retains its multiplicity in R_1 if its multiplicity in R_2 is not 0 (since if $[\![R_2]\!]\ t \neq \mathbf{0}$, then $\|[\![R_2]\!]\ t\| \to \mathbf{0} = \mathbf{1}$).

Summation ($\sum$) Given $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, $\sum_{x:A} B(x)$ is a dependent pair type ($\sum$) and is used to denote projection. For example:

$$[\![\texttt{SELECT}\ k\ \texttt{FROM}\ R]\!] \triangleq \lambda\ t.\ \sum_{t' : \texttt{Tuple}\ \sigma_R} \big\|[\![k]\!]\ t' = t\big\| \times [\![R]\!]\ t'$$

For a tuple t in the result of this projection query, its cardinality is the summation of the cardinalities of all tuples of schema σ_A that also has the same value on column k with t. Here $\|[\![k]\!]\ t' = t\|$ equals to $\mathbf{1}$ if t and t' have same value on k, otherwise it equals to $\mathbf{0}$. Unlike K-Relations, using univalent types allow us to support summation over an infinite domain and evaluate expressions such as the projection described above.

In general, proving rewrite rules using UNINOMIAL allows us to use powerful automatic proving techniques such as associative-commutative term rewriting in semi-ring structures (recall that $\mathcal{U}$ is a semi-ring) similar to the `ring` tactic [27] and Nelson-Oppen algorithm on congruence closure [90]. Both mitigate our proof burden.

2.2.4 Derived HoTTSQL Constructs

*HoTT*SQL supports additional SQL features including grouping, integrity constraints, and indexes. All such features are commonly utilized in query optimization. These features are supported by automatic syntactic rewrites in *HoTT*SQL.

Grouping Grouping is a widely-used relational operator that projects rows with a common value into separate groups, and applies an aggregation function (e.g., average) to each group. In SQL, this is supported via the `GROUP BY` operator that takes in the attribute names to form groups. *HoTT*SQL supports grouping by de-sugaring `GROUP BY` using a correlated subquery that returns a single attribute relation, and applying aggregation function to the resulting relation [35]. Below is an example of such a rewrite expressed using SQL, where α represents any of the standard SQL aggregates:

```
SELECT k as k, α(x.a) as a1 FROM R x GROUP BY x.k
                        ⇓
SELECT DISTINCT k AS k,
       α(SELECT x.a AS a FROM R x WHERE x.k = y.k) AS a1
FROM R y
```

We will illustrate grouping in rewrite rules in Section C.0.2.

Integrity Constraints Integrity constraints are used in database systems and facilitate various semantics-based query optimizations [50]. *HoTT*SQL supports two important integrity constraints: keys and functional dependencies, again through syntactic rewrites.

A *key constraint* requires an attribute to have unique values among all tuples in a relation. In *HoTT*SQL, declaring an attribute a as a key in relation R is rewritten to the following assertion:

```
key k(R);
   ⇓
SELECT * FROM R x = SELECT x.* FROM R x, R y WHERE x.k = y.k
```

To see why this rewrite satisfies the key constraint, note that k is a key in R if and only if R is equal to its self-join on k after converting the result into a set. Intuitively, if k is a key, then self-join of R on k will keep all the tuples of R with each tuple's multiplicity unchanged. Conversely, if some value of k occurs $n > 1$ times in R, then the second query increases the multiplicity of all those tuples by n, thus the two queries are not equivalent.

Functional Dependencies Keys are used in defining functional dependencies and indexes. A *functional dependency* constraint from attribute a to b requires that for any two tuples t_1 and t_2 in R, $(t_1.a = t_2.a) \rightarrow (t_1.b = t_2.b)$ This is equivalent to saying that a is a key in the projection of R on the attributes a, b:

```
fd a → b in R;
      ⇓
key a(DISTINCT SELECT x.a AS a, x.b AS b FROM R x);
```

Index An index on an attribute a is a data structure that speeds up the retrieval of tuples with a given value of a [67, Ch. 8].

To reason about rewrite rules that use indexes, we follow the idea that an index can be treated as a logical relation rather than physical data structure [110]. Since defining index as a relation requires a unique identifier of each tuple (analogous to a pointer to each tuple in the physical implementation of an index in database systems), we define index as a *HoTT*SQL query that projects on the a key of the relation and the index attribute. For example, if k is a key of relation R, an index i of R on attribute a can be defined as:

$$\texttt{index}\ i(a, R);$$

$$\Downarrow$$

$$\texttt{SELECT}\ x.k\ \texttt{AS}\ k,\ x.a\ \texttt{AS}\ a\ \texttt{FROM}\ R\ \texttt{x}$$

Here := means by definition (rather than proved). In Section **??**, we show example rewrite rules that utilize indexes that are commonly used in query optimizers.

2.2.5 Limitations

*HoTT*SQL does not currently support `ORDER BY`. `ORDER BY` is usually used with `LIMIT` n, e.g., output the first n tuples in a sorted relation. In addition, we currently do not support `NULL`s (i.e., 3-valued logic), and leave them as future work.

*2.3 Denoting HoTT*SQL

We translate *HoTT*SQL to UniNomial by first compiling *HoTT*SQL to an intermediate language *HoTT*IR. In this section, we describe the implementation of *HoTT*SQL's data model using *HoTT*IR, and provide a denotational semantics of *HoTT*IR to UniNomial.

*2.3.1 Implementing Data Model in HoTT*IR

Figure 2.5 shows *HoTT*IR's implementation of *HoTT*SQL's relational data model in Coq. Schemas are modeled as a collection of types organized in a binary tree, with each type corresponding to

an attribute. Tuples in *HoTT*SQL are implemented as dependent types on a schema. As shown in Figure 2.5, given a schema s, if s is the empty schema, then only the empty (i.e., `Unit`) tuple can be constructed from it. Otherwise, if s is a leaf node in the schema tree with type τ, then a tuple is simply a value of type $[\![\tau]\!]$. Finally, if s is recursively defined using two schemas s_1 and s_2, then the resulting tuple is an instance of a product type $\texttt{Tuple}(s_1) \times \texttt{Tuple}(s_2)$. An attribute in *HoTT*SQL is implemented as an uninterpreted function from a tuple of its schema to its data type.

Figure 2.6 shows three examples of *HoTT*IR's Coq implementation of schemas and tuples. A declaration of a generic schema σ`(a:int,??)` is implemented as a Schema declaration (σ`:Schema`), and an attribute declaration (`a:Attribute` σ τ) in Coq. In the second example, the output schema of the SQL query is represented as a tree. Lastly, the output schema of a SQL query that performs a Cartesian product of two tables is implemented as a tree with each table's schema as the left and right node, respectively.

*HoTT*IR implements schemas as trees. We do so simply for engineering convenience, as using trees allows us to easily support generic schemas in Coq. Consider the third example in Figure 2.6, which joins two tables with generic schema σ_1 and σ_1 into a table t with schema (`node` σ_1 σ_2). Since we know that every tuple of t has type (`Tuple` (`node` σ_1 σ_2)), by the computation rule for `Tuple`, it has type (`Tuple` $\sigma_1 \times$ `Tuple` σ_2). This computational simplification enables accesses to the components of a tuple which was generated by joins of tables with generic schemas. A straightforward alternative implementation is to model schemas as lists. Then t's schema is the list concatenation of σ_1 and σ_2, i.e., $\texttt{append}(\sigma_1, \sigma_2)$, and every tuple of t has type (`Tuple` $\texttt{append}(\sigma_1, \sigma_2)$). However, this cannot be further simplified computationally, because the definition of $append$ gets stuck on the generic schema σ_1.

*2.3.2 From HoTT*SQL *to HoTT*IR

*HoTT*SQL programs are first translated to an intermediate representation called *HoTT*IR. *HoTT*SQL and *HoTT*IR have the same syntax, except that *HoTT*IR uses the unnamed approach to represent schemas and attributes [19], similar to De Bruijn indexes [34]. Doing so decouples the equivalence proof from naming resolution, and allows schema equivalences to be determined based on struc-

tural equality. Translating from *HoTT*SQL to *HoTT*IR is straightforward except for two aspects: 1) each construct in *HoTT*IR comes with a *context* for evaluation, and contexts need to be inferred from the input *HoTT*SQL program, and 2) resolving names to convert from named to unnamed approach when accessing attributes is based on the inferred contexts.

First, contexts are represented as schemas. We infer the contexts for each query construct in *HoTT*SQL following standard evaluation of SQL, starting from the `FROM` clause of the outermost query. For instance, we start with query q_1 in the *HoTT*SQL query shown in Figure 2.7 and infer its context, Γ_1, to be that of a tree with two leaves: Γ_0 and σ_{R_1} (recall that schemas are represented as trees as discussed in Section 2.3.1). Γ_1 is then passed to the inner query q_2 to continue the inference. Each *HoTT*SQL construct is then appended with its inferred context during the translation. For predicates, which already come with the schema that it is intended to be applied to, we use `CASTPRED` to alter its context to the appropriate one depending on where it is used.

Given that, attribute naming is resolved using the inferred contexts. For instance, the inferred context for evaluating the `WHERE` clause in q_2 in Figure 2.7 is $(\texttt{node}\ (\texttt{node}\ \texttt{empty}\ \sigma_{R_1})\ \sigma_{R_2})$. Thus, as $x.\texttt{a}$ in the selection predicate refers to the schema of R_1, it is converted to the unnamed version as `left.right.a`.

Figure 2.7 shows an example of translating *HoTT*SQL query with correlated subqueries to *HoTT*IR. The full set of translation rules can be found in the appendix.

*2.3.3 Denoting HoTT*IR *to* UNINOMIAL

We now describe how each of the constructs in *HoTT*IR is denoted to UNINOMIAL.

Queries A query q is denoted to a function from q's context tuple (of type $\texttt{Tuple}\ \Gamma$) to a HoTT-Relation (of type $\texttt{Tuple}\ \sigma \to \mathcal{U}$):

$$[\![\Gamma \vdash q : \sigma]\!] : \texttt{Tuple}\ \Gamma \to \texttt{Tuple}\ \sigma \to \mathcal{U}$$

The `FROM` clause is recursively denoted to a series of cross products of HoTT-Relations. Each cross product is denoted using $\times$ as shown in Section 2.2.3. For example:

$$\begin{aligned}&[\![\Gamma \vdash \texttt{FROM}\ q_1,\ q_2 : \sigma]\!] \quad \triangleq \\ &\lambda\ g\ t.\ ([\![\Gamma \vdash q_1 : \sigma]\!]\ g\ t.1) \times ([\![\Gamma \vdash q_2 : \sigma]\!]\ g\ t.2)\end{aligned}$$

where $t.1$ and $t.2$ index into the context tuple Γ to retrieve the schemas of q_1 and q_2 respectively.

Note the manipulation of the context tuple in the denotation of `WHERE`: for each tuple t, we first evaluate t against the query before `WHERE`, using the context tuple g. After that, we evaluate the predicate b by first constructing a new context tuple as discussed (namely, by concatenating Γ and σ, the schema of q), passing it the combined tuple (g, t). The combination is needed as t has schema σ while the predicate b is evaluated under the schema $\texttt{node}\ \Gamma\ \sigma$, and the combination is easily accomplished as g, the context tuple, has schema Γ.

`UNION ALL`, `EXCEPT`, and `DISTINCT` are denoted using +, negation ($n \to \mathbf{0}$), and squash ($\|n\|$) on univalent types are denoted as shown in Section 2.2.3.

Predicates A predicate b is denoted to a function from a tuple (of type $\texttt{Tuple}\ \Gamma$) to a univalent type (of type $\mathcal{U}$):

$$[\![\Gamma \vdash b]\!] : \texttt{Tuple}\ \Gamma \to \mathcal{U}$$

More specifically, the return type $\mathcal{U}$ must be a *squash type* [105, Ch. 3.3]. A squash type can only be a type of 1 element, namely $\mathbf{1}$, and a type of 0 element, namely $\mathbf{0}$. A *HoTT*SQL program with the form $q\ \texttt{WHERE}\ b$ is denoted to the Cartesian product between a univalent type and a mere proposition.

As an example, suppose a particular tuple t has multiplicity 3 in query q, i.e., $q\ t = [\![R]\!]t = \mathbf{3}$, where $\mathbf{3}$ is a univalent type. Since predicates are denoted to propositions, applying the tuple to the predicate returns either $\mathbf{1}$ or $\mathbf{0}$, and the overall result of the query for tuple t is then either $\mathbf{3} \times \mathbf{0} = \mathbf{0}$, or $\mathbf{3} \times \mathbf{1} = \mathbf{1}$, i.e., a squash type.

Expressions and Projections A value expression e is denoted to a function from a tuple (of type $\texttt{Tuple}\ \Gamma$) to its data type, such as `int` and `bool` ($[\![\tau]\!]$):

$$[\![\Gamma \vdash e : \tau]\!] : \texttt{Tuple}\ \Gamma \to [\![\tau]\!]$$

A projection p from Γ to Γ' is denoted to a function from a tuple of type $\texttt{Tuple}\ \Gamma$ to a tuple of type $\texttt{Tuple}\ \Gamma'$.

$$[\![p : \Gamma \Rightarrow \Gamma']\!] : \texttt{Tuple}\ \Gamma \rightarrow \texttt{Tuple}\ \Gamma'$$

Projections are recursively defined. Projections can be composed using ".". The composition of two projections, "$p_1.\ p_2$", where p_1 is a projection from Γ to Γ' and p_2 is a projection from Γ' to Γ'', is denoted to a function from a tuple of type $\texttt{Tuple}\ \Gamma$ to a tuple of type $\texttt{Tuple}\ \Gamma''$ as follows:

$$\lambda\ g.\ [\![p_2 : \Gamma' \Rightarrow \Gamma'']\!]\ ([\![p_1 : \Gamma \Rightarrow \Gamma']\!]\ g)$$

We apply the denotation of p_1, which is a function of type $\texttt{Tuple}\ \Gamma \rightarrow \texttt{Tuple}\ \Gamma'$, to the argument of composed projection g, then apply the denotation of p_2 to the result of the application. A projection can be combined by two projections using ",". The combining of two projection, $p_1,\ p_2$, denotes to:

$$\lambda\ g.\ ([\![p_1 : \Gamma \Rightarrow \Gamma_0]\!]\ g,\ [\![p_2 : \Gamma \Rightarrow \Gamma_1]\!]\ g)$$

where we apply the denotation of p_1 and the denotation of p_2 to the argument of combined projection (g) separately, and combine their results using the pair constructor.

2.4 Proving Rewrite Rules using HoTTSQL

To demonstrate the effectiveness of *HoTT*SQL, we implemented it using Coq as a tool for checking the equivalence of SQL rewrite rules. The *HoTT*SQL's Coq implementation consists of five parts, 1) a parser that parses *HoTT*SQL programs and translates them to *HoTT*IR, 2) the denotational semantics of *HoTT*IR, 3) a library consisting of lemmas and tactics that can be used as building blocks for constructing proofs of arbitrary rewrite rules, 4) a fully automated decision procedure for the equivalence of rewrite rules consisting only of conjunctive queries, and 5) a number of proofs of existing rewrite rules from the database literature and real world systems as showcases. Besides the parser implemented in Haskell, all other parts are implemented in Coq.

The Coq implementation of *HoTT*SQL relies on the Homotopy Type Theory Coq library [75]. Its trusted code base contains the parser (531 lines of Haskell) and 296 lines of specification of

*HoTT*IR. Its verified part contains 405 lines of library code (including the decision procedure for conjunctive queries), and 1094 lines of code that prove well-known SQL rewrite rules.

In the following sections, we first show various rewrite rules and the lemmas we use from the *HoTT*SQL's Coq library, and then explain our automated proof search tactic.

2.4.1 Proving Rewrite Rules in COSETTE *by Examples*

We proved 23 rewrite rules from both the database literature and real world optimizers using *HoTT*SQL. Figure 2.9 shows the number of rewrite rules that we proved in each category and the average lines of code (LOC) required per proof.

The following sections show a sampling of interesting rewrite rules in these categories. Sec 2.4.1 shows how two basic rewrite rules are proved. Sec C.0.2 shows how to prove a rewrite rule involving aggregation. Sec C.0.3 shows how to prove the magic set rewrite rules, and Sec **??** shows how to state a rewrite rule involving indexes.

Basic Rewrite Rules

Basic rewrites are simple rewrite rules that are fundamental building blocks of query optimizers in relational database systems. We demonstrate how to prove the correctness of basic rewrite rules in COSETTE using two examples: selection push down and commutativity of joins.

Selection Push Down Selection push down moves a selection (filter) directly after the scan of the input table to dramatically reduce the amount of data in the execution pipeline as early as possible. It is known to be one of most powerful rules in database optimizers [67]. We formulate selection push down as the following rewrite rule in *HoTT*SQL:

```
SELECT * FROM  R x WHERE β1(x) AND β2(x) ≡
SELECT * FROM (SELECT  * FROM R x WHERE β1(x)) y
WHERE β2(y)
```

This is denoted to:

$$\lambda\ g\ t.\ [\![\beta_1]\!]\ (g,t) \times [\![\beta_2]\!]\ (g,t) \times [\![R]\!]\ g\ t \quad \equiv$$
$$\lambda\ g\ t.\ [\![\beta_2]\!]\ (g,t) \times ([\![\beta_1]\!]\ (g,t) \times [\![R]\!]\ g\ t)$$

The proof proceeds by functional extensionality,[2] along with the associativity and commutativity of $\times$.

Commutativity of Joins Commutativity of joins allows an optimizer to rearrange the order of joins to obtain the join order with the best performance. This is one of the most fundamental rewrite rules that almost every SQL optimizer uses. We formulate the commutativity of joins in *HoTT*SQL as follows:

```
SELECT * FROM R x, S y ≡
SELECT y.*, x.* FROM S x, R y
```

Note that the select clause flips the tuples from S and R, such that the order of the tuples matches the original query. A proof of this rewrite rule is provided in the appendix.

Aggregation and Group By Rewrite Rules

Aggregation and Group By are widely used in analytic queries [40]. The standard data analytic benchmark TPC-H [109] has 16 queries with group by and 21 queries with aggregation out of a total of 22 queries. Following is an example rewrite rule for aggregate queries. The query on the left-hand side groups the relation R by the column k, sums all values in the b column for each resulting partition, and then removes all results except the partition whose column k is equal to the constant l. This can be rewritten to the faster query that first removes all tuples from R whose column $k \neq l$, and then computes the sum. A proof of this rewrite rule is provided in the appendix.

```
SELECT *
FROM (SELECT k, α(b) FROM R x GROUP BY x.k) y
WHERE x.k = l   ≡
SELECT k, α(b) FROM R x WHERE  x.k = l GROUP BY x.k
```

[2]Function extensionality is implied by the Univalence Axiom.

Magic Set Rewrite Rules

Magic set rewrites are well-known rewrite rules that were originally used in processing recursive queries in deductive databases [26, 97]. It was then used for rewriting complex decision support queries and has been implemented in commercial systems such as IBM's DB2 database [102, 89]. As described in [102], all magic set rewrites can be composed from three basic rewrite rules on semijoins: introduction of θ-semijoin, pushing θ-semijoin through join, and pushing θ-semijoin through aggregation.

We firstly define θ-semijoin as a syntactic rewrite in *HoTT*SQL:

```
A SEMIJOIN B ON θ   ≜
SELECT * FROM A x
WHERE EXISTS (SELECT * FROM B y WHERE θ(x, y))
```

Introduction of θ-semijoin This rules shows how to introduce semijoin from join and selection. Using *HoTT*SQL, the rewrite can be expressed as follows:

```
SELECT * FROM R2 x, R1 y WHERE θ(x, y)  ≡
SELECT *
FROM (R2 x SEMIJOIN R1 y ON θ(x, y)) z1, R1 z2
WHERE θ(z1, z2)
```

which is denoted to:

$$\begin{aligned}
&\lambda\ g\ t.\ [\![\theta]\!]\ (g, t) \times [\![R_2]\!]\ g\ t.1 \times [\![R_1]\!]\ g\ t.2 \qquad \equiv \\
&\lambda\ g\ t.\ [\![\theta]\!]\ (g, t) \times [\![R_2]\!]\ g\ t.1 \times [\![R_1]\!]\ g\ t.2 \times \\
&\qquad \left\| \textstyle\sum_{t_1} [\![\theta]\!]\ (g, (t.1, t_1)) \times [\![R_1]\!]\ g\ t_1 \right\|
\end{aligned}$$

The proof uses Lemma C.0.3 provided by the COSETTE library.

Lemma 2.4.1. *$\forall P, T : \mathcal{U}$, where P is either **0** or **1**, we have:*

$$(T \to P) \Rightarrow ((T \times P) = T)$$

Proof. Intuitively, this can be proven by cases on T. If T is inhabited, then P holds by assumption, and $T \times \mathbf{1} = T$. If $T = 0$, then $\mathbf{0} \times P = \mathbf{0}$. □

Using this lemma, it remains to be shown that $[\![\theta]\!]\ (g, t)$ and $[\![R_2]\!]\ g\ t.1$ and:

$$[\![R_1]\!]\ g\ t.2 \Rightarrow \left\| \sum_{t_1} [\![\theta]\!]\ (g, (t.1, t_1)) \times [\![R_1]\!]\ g\ t_1 \right\|$$

We show this by instantiating t_1 with $t.2$, and then by hypotheses.

Pushing θ-semijoin through join The second rule in magic set rewrites is the rule for pushing θ-semijoin through join. Using *HoTT*SQL, the rewrite can be expressed as follows:

```
(SELECT * FROM R1 x, R2 y WHERE θ1(x, y))
SEMIJOIN R3 ON θ2    ≡
(SELECT *
 FROM R1 x,
      (R2 SEMIJOIN (SELECT * FROM R1, R3) ON θ1 AND θ2) y
 WHERE θ1(x, y)) S
SEMIJOIN R3 ON θ2
```

The rule is denoted to:

$$\begin{aligned}
&\lambda\ g\ t.\ \left\|\textstyle\sum_{t_1} [\![\theta_2]\!]\ (g, (t, t_1)) \times [\![R_3]\!]\ g\ t_1\right\| \times \\
&\qquad [\![\theta_1]\!]\ (g, t) \times [\![R_1]\!]\ g\ t.1 \times [\![R_2]\!]\ g\ t.2 \qquad \equiv \\
&\lambda\ g\ t.\ \left\|\textstyle\sum_{t_1} [\![\theta_2]\!]\ (g, (t, t_1)) \times [\![R_3]\!]\ g\ t_1\right\| \times \\
&\qquad [\![\theta_1]\!]\ (g, t) \times [\![R_1]\!]\ g\ t.1 \times [\![R_2]\!]\ g\ t.2 \times \\
&\qquad \| \textstyle\sum_{t_1} [\![\theta_1]\!]\ (g, (t_1.1, t.2)) \times [\![\theta_2]\!]\ (g, ((t_1.1, t.2), t_1.2)) \\
&\qquad \times [\![R_1]\!]\ g\ t_1.1 \times [\![R_3]\!]\ g\ t_1.2 \|
\end{aligned}$$

We can prove this rule by using a similar approach to the one used to prove introduction of θ-semijoin: rewriting the right hand side using Lemma C.0.3. and then instantiating t_1 with $(t.1, t_1)$ (t_1 is the witness of the Σ hypothesis).

Pushing θ-semijoin through aggregation The final rule pushes θ-semijoin through aggregation. Using *HoTT*SQL, the rewrite can be expressed as follows (a proof of this rewrite rule is provided

in the website):

```
(SELECT x.c1 AS c1, α(a) FROM R1 x GROUP BY x.c1)
SEMIJOIN R2 ON θ ≡
SELECT x.c1, α(a)
FROM (R1 SEMIJOIN R2 ON θ) x GROUP BY x.c1
```

Index Rewrite Rules As introduced in Section 2.2.4, we define an index as a *HoTT*SQL query that projects on the indexed attribute and the primary key of a relation. Assuming k is the primary key of relation R, and i is an index on column a (`index` $i(a,\ R)$), we prove the following common rewrite rule that converts a full table scan to a lookup on an index and a join:

```
SELECT * FROM R x WHERE x.a = l   ≡
SELECT * FROM i x, R y
WHERE x.a = l AND x.k = y.k
```

We omit the proof here for brevity.

2.4.2 Automated Decision Procedure for Conjunctive Queries

The equivalence of two SQL queries is in general undecidable [108]. The most well-known decidable subclass are conjunctive queries, which are of the form `DISTINCT SELECT` p `FROM` q `WHERE` b, where p is a sequence of arbitrarily many projections, q is the cross product of arbitrarily many input relations, and b is a conjunct consisting of arbitrarily many equality predicates between attribute projections.

We implement a decision procedure to automatically prove the equivalence of conjunctive queries in *HoTT*SQL. After denoting the *HoTT*SQL query to UniNomial, the decision procedure automates the steps similar to the proof in Section C.0.2. First, after applying functional extensionality, both sides become squash types due to the `DISTINCT` clause. The procedure then applies the fundamental lemma about squash types $\forall AB, (A \leftrightarrow B) \Rightarrow (A = B)$. In both cases of the resulting bi-implication, the procedure tries all possible instantiations of the Σ, which is due to the `SELECT` clause. This search for the correct instantiation is implemented using Ltac's built-in

backtracking support. The procedure then rewrites all equalities and tries to discharge the proof by direct application of hypotheses.

The following is an example of two equivalent conjunctive SQL queries that COSETTE can solve using its decision procedure:

```
SELECT DISTINCT x.c1 AS c1 FROM R1 x, R2 y
WHERE x.c2 = y.c3    ≡
SELECT DISTINCT x.c1 AS c1 FROM R1 x, R1 y, R2 z
WHERE x.c1 = y.c1 AND x.c2 = z.c3
```

which is denoted as:

$$
\begin{aligned}
&\lambda\ g\ t.\ \|\textstyle\sum_{t_1} [\![R_1]\!]\ g t_1.1 \times [\![R_2]\!]\ g\ t_1.2 \times \\
&\qquad ([\![c_2]\!]\ t_1.1 = [\![c_3]\!]\ t_1.2) \times \\
&\qquad ([\![c_1]\!]\ t_1.1 = t) \times \| \qquad \equiv \\
&\lambda\ g\ t.\ \|\textstyle\sum_{t_1} [\![R_1]\!]\ g\ t_1.1.1 \times [\![R_1]\!]\ g\ t_1.1.2 \times [\![R_2]\!]\ g\ t_1.2 \times \\
&\qquad ([\![c_1]\!]\ t_1.1.1 = [\![c_1]\!]\ t_1.1.2) \times ([\![c_2]\!]\ t_1.1.1 = [\![c_3]\!]\ t_1.2) \times \\
&\qquad ([\![c_1]\!]\ t_1.1.1 = t)\|
\end{aligned}
$$

The decision procedure turns this goal into a bi-implication, which it proves by cases. For the $\rightarrow$ case, the decision procedure destructs the available Σ witness into tuple t_x from R_1 and t_y from R_2, and tries all instantiations of t_1 using these tuples. The instantiation $t_1 = ((t_x, t_x), t_y)$ allows the procedure to complete the proof after rewriting all equalities. For the $\leftarrow$ case, the available tuples are t_x from R_1, t_y from R_1, and t_z from R_2. The instantiation $t_1 = (t_x, t_z)$ allows the procedure to complete the proof after rewriting all equalities.

2.5 Discussion

Limitations Our system does currently not support three SQL features: NULL's with their associated three-valued-logic, outer joins, and windows functions. However, all can be expressed in *HoTT*SQL, at the cost of some added complexity, as we explain now.

When any argument to an expression is NULL, then the expression's output is NULL; this feature can easily be supported by modifying the external operators. When an argument of a comparison predicate is NULL, then the resulting predicate has value `unknown`, and SQL uses three valued logic to compute predicates: it defines $0 = \texttt{false}, 1/2 = \texttt{unknown}, 1 = \texttt{true}$, and the logical operators x `and` $y = \min(x, y)$, x `or` $y = \max(x, y)$, $\texttt{not}(x) = 1 - x$; a select-from-where query returns all tuples for which the where-predicate evaluates to `true` (i.e. not `false` or `unknown`). As a consequence, the law of excluded middle fails, for example the query:

```
SELECT * FROM R WHERE R.a = 5 OR R.a ≠ 5
```

is not equivalent to `SELECT * FROM` R. This, too, could be currently expressed *HoTT*SQL by encoding the predicates as external functions that implement the 3-valued logic. However, by doing so one hides from the rewrite rules the equality predicate, which plays a key role in joins. In future versions, we plan to offer native support for NULL's, to simplify the task of proving rewrite rules over relations with NULLs.

Both outer joins and windows functions are directly expressible in *HoTT*SQL. For example, a left outer join of two relations $R(a, b)$, $S(b, c)$ can be expressed by first joining R and S on b, and union-ing the result with

```
SELECT R.*, NULL FROM S EXCEPT
SELECT R.*, NULL FROM S WHERE R.b = S.b
```

A direct implementation in *HoTT*SQL would basically have to follow the same definition of left outer joins.

Finite v.s. Infinite Semantics Recall that our semantics extends the standard bag semantics of SQL in two ways: we allow a relation to have infinitely many distinct elements, and we allow each element to have an infinite multiplicity. To the best of our knowledge, our system is the first that interprets SQL over infinite relations. This has two consequences. First, our system cannot check the equivalence of two SQL expressions that return the same results on all finite relations, but differ on some infinite relations. It is well-known that there exists First Order sentences, called *infinity axioms*, that do not admit any finite model, but admit infinite models. For example [33,

pp.307] the sentence $\varphi \equiv \forall x \exists y \forall z (\neg R(x,x) \wedge R(x,y) \wedge (R(y,z) \rightarrow R(x,z)))$ is an infinity axiom. It is possible to write a SQL query that checks φ, then returns the empty set if φ is false, or returns a set consisting of a single value (say, 1) if φ is true: call this query Q_1. Call Q_2 the query `SELECT DISTINCT 1 FROM R WHERE 2=3`. Then $Q_1 = Q_2$ over all finite relations, but $Q_1 \neq Q_2$ not over infinite relations. Thus, one possible disadvantage of our semantics is that we cannot prove equivalence of queries that encode infinity axioms. However, none of the optimization rules that we found in the literature, and discussed in this paper, encode an infinity axiom. Hence we argue that, for practical purposes, extending the semantics to infinite relations is a small price to pay for the added simplicity of the equivalence proofs. Second, by generalizing SQL queries to both finite and infinite relations we make our system theoretically complete: if two queries are equivalent then, by Gödel's completeness theorem, there exists a proof of their equivalence. Finding the proof is undecidable (it is recursively enumerable, r.e.): our system does not search for the proof, instead the user has to find it, and our system will verify it. Contrast this with a system whose semantics is based on finite relations: such a system cannot have a complete proof system for SQL query equivalence. Indeed, if such a complete proof system existed, then SQL query equivalence would be r.e. (since we can enumerate all proofs and search for a proof of $Q_1 = Q_2$), and therefore equivalence would be decidable (since it is also co-r.e., because we can enumerate all finite relations, searching for an input s.t. $Q_1 \neq Q_2$). However, by Trakthenbrot's, query equivalence is undecidable. Recall that Trakthenbrot's theorem [108**?**] states that the problem *given an FO sentence φ, check if φ has a finite model* is undecidable. We can reduce this problem to query equivalence by defining Q_1 to be a query that checks checks φ and returns the empty set if φ is false, or returns some non-empty set if φ is true, and defining Q_2 to be the query that always returns the empty set (as above), then checking $Q_1 \equiv Q_2$). Thus, by extending our semantics to infinite relations we guarantee that, whenever two queries are equivalent, there exists a proof of their equivalence.

2.6 Summary

In this chapter, we defined a formal language, *HoTT*SQL, following SQL's syntax closely. *HoTT*SQL extends the semantics of SQL from finite to infinite relations, and uses univalent types from Homotopy Type Theory to represent and prove equalities of cardinal numbers (finite and infinite). We implemented *HoTT*SQL in Coq and demonstrated the power and flexibility of *HoTT*SQL by proving the correctness of several powerful optimization rules found in the database literature, with some of them not proven correct before.

$$
\begin{array}{lcl}
h \in \texttt{Program} & ::= & s_1; \ldots; s_n; \\
s \in \texttt{Statement} & ::= & \texttt{schema}\ \sigma(a_1 : \tau_1, \ldots, a_n : \tau_n, \texttt{??}) \\
& | & \texttt{table}\ t(\sigma) \\
& | & \texttt{predicate}\ \beta(\sigma_1, \ldots, \sigma_n) \\
& | & \texttt{function}\ f(e_1, \ldots, e_n) : \tau \\
& | & \texttt{aggregate}\ \alpha(\sigma) : \tau \\
& | & \texttt{verify}\ q_1 \equiv q_2 \\
a \in \texttt{Attribute} & & \\
q \in \texttt{Query} & ::= & t \\
& | & \texttt{SELECT}\ p\ q \\
& | & \texttt{FROM}\ q_1\ x_1, \ldots, q_n\ x_n \\
& | & q\ \texttt{WHERE}\ p \\
& | & q_1\ \texttt{UNION ALL}\ q_2 \\
& | & q_1\ \texttt{EXCEPT}\ q_2 \\
& | & \texttt{DISTINCT}\ q \\
x \in \texttt{TableAlias} & & \\
b \in \texttt{Predicate} & ::= & e_1 = e_2 \\
& | & \texttt{NOT}\ b \mid b_1\ \texttt{AND}\ b_2 \mid b_1\ \texttt{OR}\ b_2 \\
& | & \texttt{TRUE} \mid \texttt{FALSE} \\
& | & \beta(x_1, \ldots, x_n) \\
& | & \texttt{EXISTS}\ q \\
e \in \texttt{Expression} & ::= & x.a \mid f(e_1, \ldots, e_n) \mid \alpha(q) \\
p \in \texttt{Projection} & ::= & \texttt{*} \mid x.\texttt{*} \mid e\ \texttt{AS}\ a \mid p_1,\ p_2
\end{array}
$$

Figure 2.4: Syntax of *HoTT*SQL

$$
\begin{array}{lcl}
\tau \in \texttt{Type} & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid \ \ldots \\
[\![\texttt{int}]\!] & ::= & \mathbb{Z} \\
[\![\texttt{bool}]\!] & ::= & \mathbb{B} \\
\ldots & & \\
\sigma \in \texttt{Schema} & ::= & \texttt{empty} \\
 & \mid & \texttt{leaf}\ \tau \\
 & \mid & \texttt{node}\ \sigma_1\ \sigma_2 \\
 & & \\
\texttt{Tuple empty} & ::= & \texttt{Unit} \\
\texttt{Tuple}\ (\texttt{node}\ \sigma_1\ \sigma_2) & ::= & \texttt{Tuple}\ \sigma_1 \times \texttt{Tuple}\ \sigma_2 \\
\texttt{Tuple}\ (\texttt{leaf}\ \tau) & ::= & [\![\tau]\!] \\
 & & \\
\texttt{Attribute}\ \sigma\ \tau & ::= & \texttt{Tuple}\ \sigma \to [\![\tau]\!]
\end{array}
$$

Figure 2.5: *HoTT*IR's implementation of *HoTT*SQL's Data Model

$$
\begin{array}{ll}
\texttt{schema}\ \sigma(\texttt{a:int, ??}); & \Rightarrow\ \sigma : \texttt{Schema}. \\
 & \quad \texttt{a: Attribute}\ \sigma\ \tau. \\
 & \\
\text{schema of} & \Rightarrow\ \texttt{node}\ (\texttt{leaf int}) \\
\texttt{SELECT "Bob" AS Name,} & \qquad (\texttt{node}\ (\texttt{leaf int}) \\
\texttt{52 AS Age, TRUE AS Married} & \qquad\quad (\texttt{leaf bool})) \\
 & \\
\text{schema of} & \Rightarrow\ \texttt{node}\ \sigma_1\ \sigma_2 \\
\texttt{SELECT * FROM t1 x, t2 y} & \\
\texttt{(table t1}\ \sigma_1\texttt{; table t2}\ \sigma_2\texttt{;)} &
\end{array}
$$

Figure 2.6: Examples of *HoTT*IR's implementation of schemas

```
SELECT * FROM R1 x WHERE                                  -- q1
  EXISTS SELECT * FROM R2 y WHERE x.a = y.a AND           -- q2
    EXISTS SELECT * FROM R3 z WHERE β(z)                  -- q3
      (...; predicate β(σR3);...)
                          ⇓
SELECT * FROM R1 WHERE                                    -- q1
  EXISTS SELECT * FROM R2 WHERE                           -- q2
        left.right.a=right.a AND
    EXISTS SELECT * FROM R3                               -- q3
          WHERE (CASTPRED Right β)
```

Query	Context schema of WHERE clause
init	Γ_0=empty
q_1	Γ_1=node Γ_0 σ_{R_1}
q_2	Γ_2=node Γ_1 σ_{R_2}
q_3	Γ_3=node Γ_2 σ_{R_3}

Figure 2.7: Translating *HoTT*SQL query with correlated subqueries to *HoTT*IR

$\boxed{[\![\Gamma \vdash q : \sigma]\!] : \texttt{Tuple}\ \Gamma \to \texttt{Tuple}\ \sigma \to \mathcal{U}}$ (* *Query* *)

$$
\begin{aligned}
&[\![\Gamma \vdash table : \sigma]\!] &&\triangleq \lambda\ g\ t.\ [\![table]\!]\ t \\
&[\![\Gamma \vdash \texttt{SELECT}\ p\ q : \sigma]\!] &&\triangleq \lambda\ g\ t.\ \textstyle\sum_{t' : \texttt{Tuple}\ \sigma'} ([\![p : \texttt{node}\ \Gamma\ \sigma' \Rightarrow \sigma]\!]\ (g, t') = t) \times [\![\Gamma \vdash q : \sigma']\!]\ g\ t' \\
&[\![\Gamma \vdash \texttt{FROM}\ q_1, q_2 : \texttt{node}\ \sigma_1\ \sigma_2]\!] &&\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q_1 : \sigma_1]\!]\ g\ t.1 \times [\![\Gamma \vdash q_2 : \sigma_2]\!]\ g\ t.2 \\
&[\![\Gamma \vdash \texttt{FROM}\ q : \sigma]\!] &&\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q : \sigma]\!]\ g\ t \\
&[\![\Gamma \vdash q\ \texttt{WHERE}\ b : \sigma]\!] &&\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q : \sigma]\!]\ g\ t \times [\![\texttt{node}\ \Gamma\ \sigma \vdash b]\!]\ (g, t) \\
&[\![\Gamma \vdash q_1\ \texttt{UNION ALL}\ q_2 : \sigma]\!] &&\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q_1 : \sigma]\!]\ g\ t + [\![\Gamma \vdash q_2 : \sigma]\!]\ g\ t \\
&[\![\Gamma \vdash q_1\ \texttt{EXCEPT}\ q_2 : \sigma]\!] &&\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q_1 : \sigma]\!]\ g\ t \times (([\![\Gamma \vdash q_2 : \sigma]\!]\ g\ t) \to \mathbf{0}) \\
&[\![\Gamma \vdash \texttt{DISTINCT}\ q : \sigma]\!] &&\triangleq \lambda\ g\ t.\ \|[\![\Gamma \vdash q : \sigma]\!]\ g\ t\|
\end{aligned}
$$

$\boxed{[\![\Gamma \vdash b]\!] : \texttt{Tuple}\ \Gamma \to \mathcal{U}}$ (* *Predicate* *)

$$
\begin{aligned}
&[\![\Gamma \vdash b_1\ \texttt{AND}\ b_2]\!] &&\triangleq \lambda\ g.\ [\![\Gamma \vdash b_1]\!]\ g \times [\![\Gamma \vdash b_2]\!]\ g \\
&[\![\Gamma \vdash \texttt{NOT}\ b]\!] &&\triangleq \lambda\ g.\ ([\![\Gamma \vdash b]\!]\ g) \to \mathbf{0} \\
&[\![\Gamma \vdash \texttt{EXISTS}\ q]\!] &&\triangleq \lambda\ g.\ \left\|\textstyle\sum_{t : \texttt{Tuple}\ \sigma} [\![\Gamma \vdash q : \sigma]\!]\ g\ t\right\| \\
&[\![\Gamma \vdash \texttt{CASTPRED}\ p\ b]\!] &&\triangleq \lambda\ g.\ [\![\Gamma' \vdash b]\!]\ ([\![p : \Gamma \Rightarrow \Gamma']\!]\ g)
\end{aligned}
$$

$\boxed{[\![\Gamma \vdash e : \tau]\!] : \texttt{Tuple}\ \Gamma \to [\![\tau]\!]}$ (* *Expression* *)

$$
\begin{aligned}
&[\![\Gamma \vdash f(e_1, \ldots) : \tau]\!] &&\triangleq \lambda\ g.\ [\![f]\!]([\![\Gamma \vdash e_1 : \tau_1]\!]\ g, \ldots) \\
&[\![\Gamma \vdash agg(q) : \tau']\!] &&\triangleq \lambda\ g.\ [\![agg]\!]\ ([\![\Gamma \vdash q : \texttt{leaf}\ \ \tau]\!]\ g)
\end{aligned}
$$

$\boxed{[\![p : \Gamma \Rightarrow \Gamma']\!] : \texttt{Tuple}\ \Gamma \to \texttt{Tuple}\ \Gamma'}$ (* *Projection* *)

$$
\begin{aligned}
&[\![* : \Gamma \Rightarrow \Gamma]\!] &&\triangleq \lambda\ g.\ g \\
&[\![\texttt{Left} : \texttt{node}\ \Gamma_0\ \Gamma_1 \Rightarrow \Gamma_0]\!] &&\triangleq \lambda\ g.\ g.1 \\
&[\![\texttt{Right} : \texttt{node}\ \Gamma_0\ \Gamma_1 \Rightarrow \Gamma_1]\!] &&\triangleq \lambda\ g.\ g.2 \\
&[\![p_1.\ p_2 : \Gamma \Rightarrow \Gamma'']\!] &&\triangleq \lambda\ g.\ [\![p_2 : \Gamma' \Rightarrow \Gamma'']\!]\ ([\![p_1 : \Gamma \Rightarrow \Gamma']\!]\ g) \\
&[\![p_1,\ p_2 : \Gamma \Rightarrow \texttt{node}\ \Gamma_0\ \Gamma_1]\!] &&\triangleq \lambda\ g.\ ([\![p_1 : \Gamma \Rightarrow \Gamma_0]\!]\ g,\ [\![p_2 : \Gamma \Rightarrow \Gamma_1]\!]\ g)
\end{aligned}
$$

Figure 2.8: Selected rules of the denotational semantics of *HoTT*SQL, more rules are shown in the appendix.

Category	No. of rules	Avg. LOC (proof)
Basic	8	11.1
Aggregation	1	50
Subquery	2	17
Magic Set	7	30.3
Index	3	64
Conjunctive Query	2	1 (automatic)
Total	23	25.2

Figure 2.9: Rewrite rules proved

Chapter 3

AXIOMATIC FOUNDATIONS AND PROOF AUTOMATION

In this chapter, 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

[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.

the presence of indexes, views, and other ICs. At a high level, our algorithm performs rewrites reminiscent to the chase/back-chase procedure [94] 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 Coq proof assistant. 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.

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 (Section 3.2).

- 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) that are easy to use in an automated theorem prover. Doing so allows us to easily utilize them in equivalence proofs (Sec. 3.3).

- 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

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

Conjunctive Queries under set and bag semantics (Section 3.4).

- We have implemented UDP using the Coq proof assistant, and have evaluated UDP by collecting 45 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, only 1 of these rules have been proven correct before, while UDP can automatically prove 39 of them (Section 4.2).

3.1 Overview

We motivate our new semantics using an example query rewrite. As shown in Figure 3.1, the rewrite changes the original query `Q1` by using an index I for look up. We follow the GMAP

framework [111], 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 [51] is expressed in English and is difficult to implement programmatically. One recent attempt is Q*cert [24], 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 [12] 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. [78] 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 [74]. 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 Figure 3.1, 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 [19] (i.e., logical formulas of the form $\forall \mathbf{x}(\varphi \Rightarrow \exists \mathbf{y}\psi)$). For example, consider the two queries in Fig. 3.1: 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. 3.1; 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.

[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 [19] are expressed as axioms, $\forall \mathbf{x}(\varphi \Rightarrow \exists \mathbf{y}\psi)$, and only

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 Figure 3.1 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. (3.15)):

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

Since k is a key of R, applying the U-semiring definition of the key constraint (Def. 3.3.1) we get:

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

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. (3.15) again to the above makes Q_2 is equivalent to Q_1. We present a detailed proof in Ex 3.3.7.

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.

3.2 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.

3.2.1 U-semirings

Under standard bag semantics, a SQL query and its input relations can be modeled as $\mathbb{N}$-relations [74], i.e., relations over the semiring $(\mathbb{N}, 0, 1, +, \times)$. However, as we saw SQL queries include con-

apply to queries under set semantics.

structs 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 3.2.1. *An* unbounded semiring*, or U-semiring, is* $(\mathcal{U}, \boldsymbol{0}, \boldsymbol{1}, +, \times, \|\cdot\|, \mathtt{not}(\cdot), (\sum_D)_{D\in\mathcal{D}})$*, where:*

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

$$\|\boldsymbol{0}\| = \boldsymbol{0} \quad , \quad \|\boldsymbol{1} + x\| = \boldsymbol{1} \tag{3.1}$$

$$\|\|x\| + y\| = \|x + y\| \tag{3.2}$$

$$\|x\| \times \|y\| = \|x \times y\| \tag{3.3}$$

$$\|x\| \times \|x\| = \|x\| \tag{3.4}$$

$$x \times \|x\| = x \tag{3.5}$$

$$x^2 = x \Rightarrow \|x\| = x \tag{3.6}$$

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

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

- $\mathcal{D}$ *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.*

[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 [74] for details.

- *Unbounded summation further satisfies the following axioms:*

$$\sum_t \big(f_1(t) + f_2(t)\big) = \sum_t f_1(t) + \sum_t f_2(t) \tag{3.7}$$

$$\sum_{t_1}\sum_{t_2} f(t_1, t_2) = \sum_{t_2}\sum_{t_1} f(t_1, t_2) \tag{3.8}$$

$$x \times \sum_t f_2(t) = \sum_t x \times f_2(t) \tag{3.9}$$

$$\left\|\sum_t f(t)\right\| = \left\|\sum_t \|f(t)\|\right\| \tag{3.10}$$

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.(3.2) implies $\|\|x\|\| = \|x\|$, which, as we will see, helps us prove that `DISTINCT` of `DISTINCT` equals `DISTINCT`. Eq.(3.4) 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.(3.6) captures even more subtle interactions between subqueries with and without `DISTINCT` as described in [93]. We will illustrate this in Section 3.4.4.

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 $\|0\| = \mathtt{not}(x) = 0$, and $\|x\| = \mathtt{not}(0) = 1$ 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 [**?**] form an U-semiring; in our prior system, COSETTE [44], we used

[4] $+$ and $\times$ are extended by $x + \infty = \infty$, $0 \times \infty = 0$, and $x \times \infty = \infty$ for $x \neq 0$. Unbounded summation

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* [62] 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 $\mathtt{not}(\cdot)$ are also kept to a minimum; for example, we omitted unnecessary conditional identities like *if* $x \neq \mathbf{0}$ *then* $\|x\| = \mathbf{1}$. 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* [70]) 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. 3.3).

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.

3.2.2 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-

is defined as $\sum_D f \stackrel{\text{def}}{=} f(a_1) + \cdots + f(a_n)$ when the support of f is finite, $\mathsf{supp}(f) \stackrel{\text{def}}{=} \{x \mid f(x) \neq 0\} = \{a_1, \ldots, a_n\}$, and $\sum_D f = \infty$ otherwise.

expressions, then checks their equivalence using the UDP algorithm described in Section 3.4. In this section, we describe the translation from SQL queries to U-expressions.

Figure 3.2 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 Figure 3.2, e.g., $t.price/100$), and denote a SQL predicate by b (Predicate in Figure 3.2, 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 [74], we do not require the support of R to be finite, as that would make it difficult to axiomatize as discussed.

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

$$[b] = \|[b]\| \tag{3.11}$$

Under the standard interpretation of SQL, $[b]$ is either **0** (false) or **1** (true), but formally we only require Eq.(3.11). The equality predicate has a special interpretation in U-semiring and is required to satisfy the following axioms, called *excluded middle* (Eq. (3.12)), *substitution of equals by equals* (Eq. (3.13)), and *uniqueness of equality* (Eq. (3.14)):

$$[e_1 = e_2] + [e_1 \neq e_2] = \mathbf{1} \tag{3.12}$$

$$f(e_1) \times [e_1 = e_2] = f(e_2) \times [e_1 = e_2] \tag{3.13}$$

$$\sum_t [t = e] = \mathbf{1} \tag{3.14}$$

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

$$\sum_t [t = e] \times f(t) = f(e) \tag{3.15}$$

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

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. 4.3); 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 3.2.2. *Every SQL expression q in Figure 3.2 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 [**?**].*

[5]Using (3.13), 3.9, and (3.14) we can show that: $\sum_t [t = e] \times f(t) = \sum_t [t = e] \times f(e) = f(e) \times \sum_t [t = e] = f(e) \times \mathbf{1} = f(e)$.

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.

3.2.3 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 3.2.3. *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:*

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

  where each tuple variable t_i ranges over $\mathtt{Tuple}(\sigma_i)$*. Each expression inside the summation is called a* factor*. Multiple summations are combined into a single sum using Eq.* (3.8)*. There is no summation in the case where $m = 0$.*

- *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* $\mathtt{not}(E_n)$*, where E_n is an expression in* SPNF*. When $E_n = \mathbf{0}$,* $\mathtt{not}(E_n)$ *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 3.2.4. *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:

$$E_1 \times (E_2 + E_3) \rightsquigarrow E_1 \times E_2 + E_1 \times E_3 \tag{3.1}$$

$$(E_1 + E_2) \times E_3 \rightsquigarrow E_1 \times E_3 + E_2 \times E_3 \tag{3.2}$$

$$E_1 \times (E_2 \times E_3) \rightsquigarrow (E_1 \times E_2) \times E_3 \tag{3.3}$$

$$\ldots \times M \times [b] \times \ldots \rightsquigarrow \ldots \times [b] \times M \times \ldots \tag{3.4}$$

$$\sum_t \big(f_1(t) + f_2(t)\big) \rightsquigarrow \sum_t f_1(t) + \sum_t f_2(t) \tag{3.5}$$

$$E \times \sum_t f(t) \rightsquigarrow \sum_t E \times f(t) \tag{3.6}$$

$$\big(\sum_t f(t)\big) \times E \rightsquigarrow \sum_t f(t) \times E \tag{3.7}$$

$$\|E_1\| \times \|E_2\| \rightsquigarrow \|E_1 \times E_2\| \tag{3.8}$$

$$\texttt{not}(E_1) \times \texttt{not}(E_2) \rightsquigarrow \texttt{not}(E_1 + E_2) \tag{3.9}$$

Each rule above corresponds to an axiom of U-semirings. Rule (3.1)-(3.4) 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 (3.1) and (3.5) apply distributivity to remove any $+$ inside each T_i. Rule (3.3) and (3.4) normalize the $\times$ expressions so they remain left associative with all boolean expressions moved to the left. The last two rules, Rule (3.8) and (3.9), consolidate a product of multiple factors into a single factor. □

Figure 3.3 shows how a U-expression is converted into SPNF by applying Rules (3.4, 3.7), and (3.6). In general, the rules described above are applied recursively until none of them is applicable.

3.3 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 Figure 3.2). In this section we introduce a new axiomatic interpretation of integrity constraints using identities in a U-semiring.

3.3.1 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 3.3.1. *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 3.3.2. *If $R.k$ satisfies the key constraint (Def. 3.3.1) over the U-semiring of natural numbers $\mathbb{N}$, then $R.k$ is a standard key.*

We briefly discuss our choice in Def. 3.3.1. The standard axiom for a key is $\forall t, t'.(t \in R \wedge t' \in R \wedge t.k = t'.k \;\Rightarrow\; t = t')$ [19]. 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:

$$\sum_t R(t) \times [t.k = e] \leq \mathbf{1} \tag{3.10}$$

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, (3.10) requires an ordered semiring, which leads to additional complexity, as argued earlier. In contrast, Def. 3.3.1 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:

$$R(t') \times \sum_t R(t) \times [t.k = e] \quad = \quad R(t') \tag{3.11}$$

This axiom is an identity and, furthermore, it can be shown that Eq.(3.10) implies Eq.(3.11) in any ordered semiring; in other words, Eq.(3.11) seems to be the right reformulation of Eq.(3.10) without requiring order. On the other hand, Eq.(3.11) already follows from our key identity (Def. 3.3.1): 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 Section 3.4.4 to prove some non-trivial SQL identities described by [93]:

Theorem 3.3.3. *If $R.k$ satisfies the key constraint (Def. 3.3.1), 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.(3.6) in Section 3.2.1, it suffices to show that the LHS squared equals itself. From Def. 3.3.1, 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:

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

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

By applying Eq. (3.9) in Section 3.2.1 twice on the left, and Eq. (3.15) in Section 3.2.2 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 3.3.3 follows from Eq. (3.6) in Section 3.2.1: $x^2 = x$ implies $\|x\| = x$. □

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

Definition 3.3.4. *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 3.3.5. *If $S.k', R.k$ satisfies the foreign key constraint (Def. 3.3.4) 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 Figure 3.2 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 [111] 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. 3.3.1) of R. The indexes are treated as views and inlined in the main query when compiling to U-expressions.

3.3.2 SQL Equivalence with Integrity Constraints

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

Definition 3.3.6. *Consider two queries, q_1 and q_2, along with definitions of schemas, base relations, and constraints (Fig. 3.2). 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 3.3.7. *We show how to formally prove that the optimization rule at the top of Figure 3.1 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.3.3, 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. (3.15) *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:*

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

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

Applying the key constraint definition (Def. 3.3.1) 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. (3.15) *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. 3.3.6.

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 3.3.6 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` $0 \neq 0$), but Q, Q' are not U-equivalent, because they are distinct over the U-semiring $\bar{\mathbb{N}}$.

[8] A different, shorter infinity axiom is given in [33, pp.307]: $\varphi \equiv \forall x \exists y \forall z (\neg R(x, x) \wedge R(x, y) \wedge (R(y, z) \rightarrow R(x, z)))$.

3.4 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 3.3.6.

UDP supports the SQL fragment in Figure 3.2. 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. 3.2.3 the structure of a normalized U-expression E and of a term T:

$$E ::= T_1 + \ldots + T_n$$
$$T_i ::= \sum_{t_1,\ldots,t_m} [b_1] \times \ldots \times [b_k] \times \left\| E' \right\| \times \texttt{not}(E'') \times M_1 \times \ldots \times M_j$$

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 Section 3.4.1). 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.

3.4.1 Canonical Form

As we have illustrated in Ex 3.3.7, 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 Section 3.3.1. We call these rewritten U-expressions the canonical form of U-expressions under integrity constraints.

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. (3.15) on E'
4: **for** $c \in C$ **do**
5: $E' \leftarrow$ *recursively* and *repeatedly* apply Def. 3.3.1 and Theorem 3.3.3 on E'
6: $E' \leftarrow$ *recursively* and *repeatedly* apply Def. 3.3.4 on E'
7: **end for**
8: **return** E' // E' is now in canonical form
9: **end procedure**

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 Section 3.2 and Section 3.3. 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. (3.15) (line 3).

3. For each key constraint, rewrite E using the identities defined in Def. 3.3.1 and Theorem 3.3.3 (line 5).

4. For each foreign key constraint, rewrite E using the identity defined in Def. 3.3.4 (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 (Section 3.4.2). 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 [94]. The difference is that our rewrites are on U-expressions rather than on relational queries. Like chase, canonize is complete for conjunctive queries (CQ) under set semantics and database integrity constraints, and will terminate if the query is weak acyclic [64].

3.4.2 Equivalence of U-expressions

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

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 Section 3.4.1. 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

Algorithm 2 UDP: U-expression Decision Procedure

1: **procedure** UDP(E_1, E_2, C) // E_1 and E_2 are in SPNF
2: $E_1 \leftarrow \texttt{canonize}(E_1, C), E_2 \leftarrow \texttt{canonize}(E_2, C)$
// 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** $\forall i \in \{1, \ldots, n\}, \texttt{TDP}(p[i], T_{2,i}, C) ==$ **true then**
8: **return true**
9: **end if**
10: **end for**
11: **return false**
12: **end procedure**

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 [90], 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.

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

- 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. 3.2.3).

Algorithm 3 `TDP`: Decision Procedure for Terms

```
1: 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 ∑_{t_1⃗}[b_{1,1}] × ... × [b_{1,k}] × ‖E_{1,1}‖ ×
   //                 not(E_{1,2}) × M_{1,1} × ... × M_{1,j}, and
   // T_2 has the form ∑_{t_2⃗}[b_{2,1}] × ... × [b_{2,m}] × ‖E_{2,1}‖ ×
   //                 not(E_{2,2}) × M_{2,1} × ... × M_{2,ℓ}
2:     for p ∈ BI_{t_2⃗,t_1⃗} do
3:         T_2' ← p(T_2)      // substitute t_2⃗ in T_2 using p(t_2⃗)
4:         if T_1 == T_2' then
5:             return true
6:         end if
7:     end for
8:     return false
9: end procedure
```

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:

Lemma 3.4.1. *The following holds in any U-semiring:*

$$\|a \times \|x\| + y\| = \|a \times x + y\| \tag{3.12}$$

Proof. By Eq. (3.2) in Section 3.2, $LHS = \|\|a \times \|x\|\| + y\|$. And by Eq. (3.3), $LHS = \|\|a\| \times \|\|x\|\| + y\|$. By Eq. (3.2) (when $y = \mathbf{0}$ in Eq. (3.2)), $\|\|x\|\| = x$; thus:

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

Apply Eq. (3.3) again:

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

And apply Eq. (3.2) from right to left:

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

□

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 [98]: 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.

Similar to the classical algorithm [98], SDP proceeds by checking whether $\|T_1 + \cdots + T_m\| \subseteq \|T'_1 + \cdots + T'_m\|$ and vice versa. This is done by checking if there exist homomorphisms on both directions, from $\|E_1\|$ to $\|E_2\|$ and from $\|E_2\|$ to $\|E_1\|$ (line 5). For example, to find the existence of a homomorphism from $\|E_2\|$ to $\|E_1\|$, SDP checks if for every term T_i, we can find a term T'_j

Algorithm 4 SDP: Decision Procedure for Squashed Expressions

procedure SDP($\|E_1\|, \|E_2\|, C$) // E, E' are in SPNF
  remove $\|\|$ inside E_1, E_2 by applying Lem. 3.4.1
  $\|E_1\| \leftarrow \|\texttt{canonize}(E_1, C)\|$
  // $\|E_1\|$ is now in the form of $\|T_1 + \ldots + T_m\|$
  $\|E_2\| \leftarrow \|\texttt{canonize}(E_2, C)\|$
  // $\|E_2\|$ is now in the form of $\|T'_1 + \ldots + T'_n\|$
  return $\forall i \exists j.$ `is-homo`(T_i, T'_j) && $\forall j \exists i$. `is-homo`(T'_j, T_i)
end procedure

procedure `is-homo`(T_1, T_2)
// returns **true** if an homomorphism is found, else return **false**
// T_1 has the form $\sum_{\vec{t_1}}[b_{1,1}] \times \ldots \times [b_{1,k}] \times \texttt{not}(E'_1) \times$
// $M_{1,1} \times \ldots \times M_{1,j}$, and
// T_2 has the form $\sum_{\vec{t_2}}[b_{2,1}] \times \ldots \times [b_{2,m}] \times \texttt{not}(E'_2) \times$
// $M_{2,1} \times \ldots \times M_{2,l}$
  for $h \in \mathcal{F}_{\vec{t_2},\vec{t_1}}$ **do**
    return true if *all* of the following satisfied:
    1. $E'_1 = h(E'_2)$
    2. $\forall i \in \{0, \ldots, l\}, h(M_{2,i}) \in \{M_{1,1}, \ldots, M_{1,j}\}$
    3. $\bigwedge_i [b_{1,i}] \Rightarrow h(\bigwedge_i [b_{2,i}])$
  end for
  return false
end procedure

such that there is a homomorphism from T'_j to T_i by calling `is-homo`. The homomorphism on the other direction is checked similarly. $\|E_1\|$ and $\|E_2\|$ are equivalent if and only if a homomorphism is found on each direction.

Precisely, a homomorphism is present if there exists a function $h \in \mathcal{F}_{\vec{t_2},\vec{t_1}}$ from the set of variables that T_2 sums over, $\vec{t_2}$, to the set of variables that T_1 sums over, $\vec{t_1}$, such that:

- If we substitute $\vec{t_2}$ with $h(\vec{t_2})$ in E'_2, the expression within `not` of T_2, the new expression (denoted as $h(E'_2)$) is equivalent E'_1, the expression within `not` of T_1.
- After the same substitution, every $h(M_{2,i})$ must be in $\{M_{1,1}, \ldots, M_{1,j}\}$, which is part of T_1.
- After the same substitution in T_2, the predicates in T_1 logically implies the predicates of T_2:

$$\bigwedge_i [b_{1,k}] \;\Rightarrow\; h(\bigwedge_i [b_{2,i}])$$

Note that, for squashed expressions representing SQL queries beyond UCQ, `SDP` is not complete, however, it is still sound and applicable. Next, we illustrate using `SDP` to check for term equivalence in U-expression with the following example.

Example 3.4.2. *To check if the these two queries are equivalent:*

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

`SDP` *first converts their corresponding U-expressions to canonical form (line 2-4):*

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

$$Q_2(t) = \left\| \sum_{t'_1} [t'_1.a = t] \times R(t'_1) \right\|$$

First, let $Q_1(t) = \|T\|$ *and* $Q_2(t) = \|T'\|$*; Let* h *be the homomorphism* $\{t'_1 \to t_1\}$*, then:*

$$h(T') = \sum_{t_1} [t_1.a = t] \times R(t_1)$$

The three conditions stated in lines 11-13 of Alg. 4 are now satisfied, hence there is a homomorphism from T' to T, and `is-homo`(T, T') *returns* ***true****.*

Similarly, let h' be $\{t_1 \to t'_1, t_2 \to t'_1\}$; then:

$$h(T) = \sum_{t'_1} [t'_1.a = t] \times R(t'_1) \times R(t'_1)$$

Likewise, the three conditions in lines 11-13 of Alg. 4 are satisfied. A homomorphism exists between T and T', `is-homo`(T', T) *returns* ***true*** *as well, and the two queries are proven to be equivalent.*

3.4.3 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 3.4.3. *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 [51].*

Proof. All transformations in algorithms 1 and 2 are based on axioms of U-semiring and proven identities. Since the standard SQL semantics [51] 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 3.4.4. *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 [98] (see also [46, Theorem 4.3, 4.4]), implying that our algorithm is complete in this case. □

Theorem 3.4.5. *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 $\|\sum_{\vec{t}}[b_1] \times \cdots \times [b_n] \times R_1(t_1) \times \cdots \times R_j(t_j)\|$, 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 [98]. □

3.4.4 An Illustration

We demonstrate how UDP works using an example rewrite evaluated under mixed set-bag semantics from the Starburst optimizer [93]:

```
-- 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 $Q_1(t)$:

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

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 $[t_1.\mathsf{np} = t.\mathsf{np})]$ and $[t_1.\mathtt{itn} = t_2.\mathtt{itemno}]$, the summation on t_1 is removed after applying Eq. (3.15) in line 3 in canonize:

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

Since ite.itemno is a key, Theorem 3.3.3 is applied (line 5 in canonize):

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

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

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

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.

3.5 Evaluation

In this section we describe our implementation and evaluation of UDP. We first describe our implementation in Section 3.5.1. Then, in Section 3.5.2, 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 Section 3.5.2 and characterize these results in Section 3.5.3. Section 3.5.4 concludes with the limitations of our current implementation.

3.5.1 Implementation

We have implemented our UDP equivalence checking algorithm, and Figure 3.4 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 Figure 3.2) 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 in Section 3.2 and Section 3.3 consists of 129 lines of Lean. UDP is implemented using 1422 lines of Lean's metaprogramming language for proof search [61]. 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.

3.5.2 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 [102]), 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 [44], including the well-known magic set rewrites [102].

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 [11, 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 Section 3.5.4. 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 Figure 3.2, 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

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

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.

Figure 3.5 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 [66]. As expected, our system failed to prove equivalence within the time limit of 30 minutes; two other bugs in the literature [7, 13] are based on the NULL semantics, which we currently do not support. As our prior work work [43] 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.

3.5.3 Characterizing Results

As shown in Figure 3.6, 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 Figure 3.6 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 [93] (we described one of them in Section 3.4.4). 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.

Figure 3.7 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 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 3.1 and Rule 3.2 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 [44]. 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 (Figure 3.5) 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. 3.3.7) 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. 3.3.7, in contrast to UDP automatically proving this rewrite rule.

3.5.4 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.

3.6 Summary

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. One significant advantage of U-semiring is to make automated proof search much easier. Based on

U-semiring, we push the boundary of automated proof search by designing a novel algorithm, UDP, for checking the equivalence of SQL queries. We show that UDP is sound in general, and complete for unions of conjunctive queries with integrity constraints under bag/set semantics. We have used it to prove the validity of 62 real-world SQL rewrites, many of which were proven for the first time.

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

Figure 3.2: SQL fragment supported by our semantics

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

$$
\begin{aligned}
[\![q]\!](t) &= \sum_{t_1,t_2} [t_2 = t] \times (\sum_{t_3} [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times \\
&\quad R(t_3)) \times R(t_2) \times [t_1.k = t_2.k] \times [t_1.a \geq 12] \\
&= \sum_{t_1,t_2} [t_2 = t] \times [t_1.k = t_2.k] \times [t_1.a \geq 12] \times \\
&\quad \Big(\sum_{t_3} [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times R(t_3)\Big) \times R(t_2) && \text{Rule (3.4)} \\
&= \sum_{t_1,t_2} [t_2 = t] \times [t_1.k = t_2.k] \times ([t_1.a \geq 12] \times \\
&\quad \sum_{t_3} [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times R(t_3) \times R(t_2) && \text{Rule (3.7)} \\
&= \sum_{t_1,t_2,t_3} [t_2 = t] \times [t_1.k = t_2.k] \times [t_1.a \geq 12] \times \\
&\quad [t_3.k = t_1.k] \times [t_3.a = t_1.a] \times R(t_3) \times R(t_2) && \text{Rule (3.6)}
\end{aligned}
$$

Figure 3.3: A SQL query q (the second query shown in Figure 3.1), its semantics $[\![q]\!]$ in U-semiring and its rewriting into sum-product normal form.

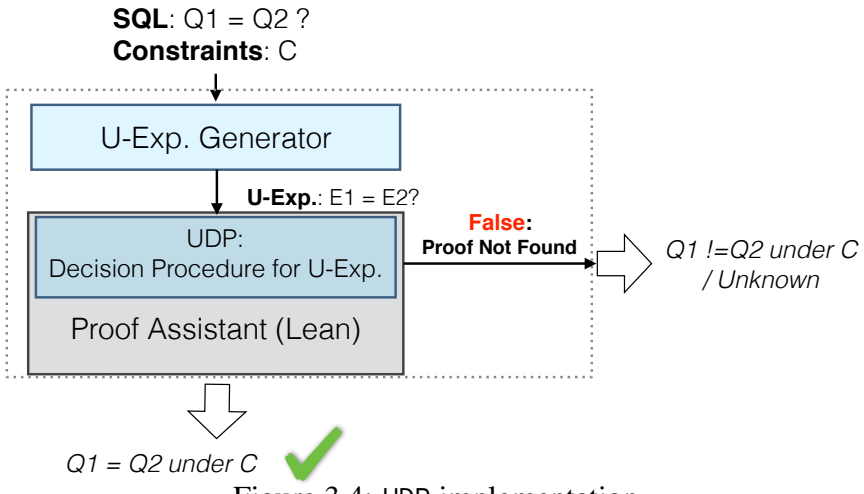

Figure 3.4: UDP implementation

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 3.5: 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 3.6: Characterization of the proved cases, where the categories are not mutually exclusive.

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 3.7: `UDP` execution time (ms)

Chapter 4

FINDING COUNTEREXAMPLES FOR INEQUIVALENT SQL QUERIES

Finding proofs for equivalent SQL queries only solved the first half of the problem. For inquivalent SQL queries, one can never find such a proof because the proof doesn't ever exist. To determine inequivalent SQL queries, we also build a symbolic executor in COSETTE which tries to find counterexamples of input SQL queries by symbolic executing of SQL queries with constraint solvers. In this chapter, Section 4.1 presents the details of finding counterexamples with constraint solver. Section 4.2 shows the empirical evaluation result of COSETTE's symbolic execution part. Section 4.3 demonstrates how to combining proof search in interactive theorem prover and symbolic execution for SQL queries, as well as the the overall architecture of COSETTE.

4.1 Finding Counterexamples With Constraint Solver

In this section we describe how COSETTE translates input queries into constraints using symbolic execution, with the generated constraints sent to a constraints solver in search of counterexamples. If found, the input queries are proven to be inequivalent, and the counterexamples are returned to the user as evidence.

To illustrate this process, we use the queries shown in Figure 4.2 as a running example. The example illustrates the famous `COUNT` bug that involves rewriting of correlated subqueries. It involves two symbolic relations, `Parts` and `Supply`, with a derived view `Temp`. Using the architecture of the constraint generator shown in Figure 4.1, we explain the solving process in detail below.

4.1.1 Data Model

COSETTE models relations using bag semantics. Following prior work on modeling relations [113, 86], in the constraints generator a tuple is encoded as a list of integers (strings are modeled as

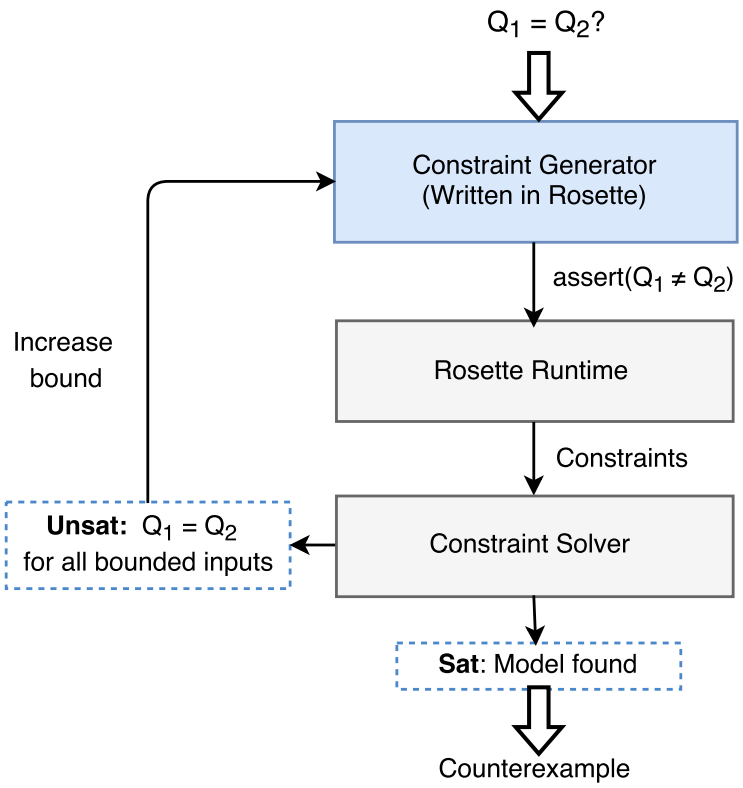

Figure 4.1: Architecture of the constraints generator its interaction with the underlying constraint solver.

```
SELECT pnum FROM Parts                          -- Q1
WHERE qoh = (SELECT COUNT(shipdate)
             FROM Supply
             WHERE Supply.pnum = Parts.pnum
             AND shipdate < 10);

WITH Temp AS                                    -- Q2
  SELECT pnum, COUNT(shipdate) AS ct
  FROM Supply
  WHERE shipdate < 10
  GROUP BY pnum
SELECT pnum FROM Parts, Temp
WHERE Parts.qoh = Temp.ct
AND Parts.pnum = Temp.pnum;
```

Figure 4.2: Constraints generation example using the `COUNT` bug.

integers, and floating point numbers are currently not supported), and a relation is defined as an unordered set of `Pairs`:

```
Tuple    := List<Integer>
Relation := List<Pair<Tuple, Integer>>
```

As shown in the above, each element of a `Relation` is a `Pair`, with first element being a tuple, and the second element represents the multiplicity of the tuple. For example, the list `[([1,2],2), ([2,5],3)]` represents a relation with 2 tuples of `[1,2]` and 3 tuples of `[2,5]`.[1]

The constraints generator compiles each symbolic relation into a fixed sized list consisting of symbolic values. For example, the symbolic relations in the running example (`Parts` and `Supply`) are compiled to:

[1] We use bold font for tuple multiplicities for ease of read.

```
 Parts = [([sv0,sv1],sv2), ([sv3,sv4],sv5)]
Supply = [([sv6,sv7],sv8)]
```

where each $\texttt{sv}_i$ represents a symbolic value whose value will be assigned by the solver. Note that the multiplicity of each tuple is a symbolic value as well. In Section 4.1.4 we describe how COSETTE uses incremental solving to dynamically increase the size of each symbolic relation. In this example, COSETTE uses 2 iterations to find the counterexample, with the first iteration setting the size of `Parts` to 1, and increasing it to 2 on the second iteration.

4.1.2 Compiling Queries to Constraint Programs

Given the data model, the symbolic execution engine in the constraints generator compiles a SQL query to a function written in Rosette that computes over symbolic relations. Rosette [107] is a language for constraints programming. When executed, the query function will generate constraints that can be sent to the constraints solver to solve. As an example, `Q2` in Figure 4.2 is compiled to the query function written in Rosette as shown below:[2]

```
def Q2():
  r = []
  for t in xprod(Parts, Temp):
    if p1(t) and p2(t):
      r.append([t[0]])
  return r
```

In the code fragment, `p1` represents the predicate `Parts.qoh = Temp`, `p2` is the predicate `Parts.pnum = Temp.pnum`, and `xprod` is the Cartesian product operator on two relations that we have implemented in Rosette. Just like its SQL counterpart, the constraint program iterates over each tuple from the Cartesian product of `Parts` and `Temp`, and appends the projection of the iterated tuple (`t[0]`, where `0` is the index of the projected `pnum` attribute) to the output relation if it satisfies the two predicates. Unlike SQL queries, however, the contents of `r` is not a set of concrete tuples, but

[2]The part that computes `Temp` is not shown.

rather a number of constraints that encodes the semantics of Q2 over the input symbolic variables, as we will describe in Section 4.1.3.

Correlated Subqueries. Correlated subqueries (as in that in Q1) are compiled in a similar manner, except that the generated query functions take in a tuple from the enclosing query as parameter. For example, the subquery in the WHERE clause in Q1 is compiled to the procedure SubQ1 below (left), where out_t represents the tuple that is passed in from the enclosing query. Q1 is compiled to procedure Q1 (right), with a call to SubQ1 on line 4.

```
def SubQ1(out_t):
  r = []
  for t in Supply:
    if (t[0] == out_t[1]
        and t[1] < 10):
      r.append([t[1]])
  return [r.size()]
```

```
def Q1():
  r = []
  for t in Parts:
    if t[1] == SubQ1(t):
      r.append([t[0]])
  return r
```

Aggregation and Grouping. The COSETTE constraint generator also supports aggregation functions such as COUNT, SUM, and AVERAGE on individual attributes. Grouping operations are rewritten to correlated subqueries and aggregates on single-columned relations following standard practice [35].

Other Features and Limitations. The COSETTE constraints generator currently supports most standard SQL features. Besides the features mentioned above, EXISTS, IN, and LEFT OUTER JOIN are also supported. Currently COSETTE does not support string operations (e.g., LIKE) since COSETTE currently does not model strings as character arrays, along with ORDER BY.

4.1.3 Generating Constraints

After compiling SQL queries to Rosette programs, they are executed by the Rosette runtime to generate constraints expressed in SMT-LIB format [29]. For example, running Q1 in Section 4.1.2 through Rosette generates the following set of constraints:

```
(assert r[0] =
  (if (sv1 = subQ1(([sv0,sv1],sv2))
   then ([sv0],sv2)
   else (if (sv4 = subQ1(([sv3,sv4],sv5))
         then ([sv3],sv5)
         else Nil))
... ...
```

Note that the contents of `r` is a set of constraints. The constraint for the first row of the table (i.e., `r[0]`), for instance, says that `r[0]` equals to a single element list containing the second element (`sv0`) from the table `Parts` with multiplicity **`sv2`** if `sv1` equals to the result of evaluating `SubQ1` on the first tuple of the `Parts` table. Otherwise, it equals to either `([sv3],sv5)` or an empty list `Nil`. Similar constraints are generated for `r[1]` as well.

All these constraints restrict the set of values that the constraint solver can assign to each of the symbolic variables, a topic that we discuss next.

4.1.4 Finding Counterexamples

After compiling queries to constraints over symbolic variables, COSETTE send the constraints to the solver by asking it to find a model to the formula `Q1()`$\neq$`Q2()`. As discussed in Section **??**, two queries are inequivalent if a model (i.e., a counterexample) is found. While many solvers are available, the current COSETTE prototype uses the solver that comes with the Rosette runtime. On the other hand, if the solver is unable to find a counterexample, then that means either a counterexample does not exist for the given size of the symbolic relations, or that the two queries are actually equivalent. For the former, COSETTE will increase the size of the symbolic relations and regenerate new constraints (as shown in Figure 4.1) until a counterexample found or a predetermined timeout is reached. Once timed out, COSETTE will forward the queries to the Uninomial generator and the proof assistant in attempt to prove their equivalence, as we will describe next.

Dataset	Equiv.?	Total No.	Solved No.	Avg. Time
Bugs	No	3	3	8.8 s
Exams	No	5	5	1.3 s
XData	No	9	9	< 1 s

Figure 4.3: Evaluation Summary.

4.2 Evaluation

We have implemented the symbolic execution part of the COSETTE solver using Rosette (version 2.2). Our prototype includes 3k lines of Rosette code. In this section we evaluate COSETTE's ability to determine query equivalence on four real-world datasets:

- **Bugs** contains 3 real-world bug reports including the `COUNT` bug [66], and two other optimizer errors in real-world DBMSs [7, 13].
- **Exams** is a set of questions from the undergraduate data management class [9] where students are asked to identify whether two queries are equivalent or not.
- **XData** contains pairs of original and mutant queries collected from XData [103]. Each mutant query is generated by mutating the original query using the tool, and COSETTE is asked to identify if the mutant preserves the original query's semantics.

Figure 4.3 shows the summary of the evaluation. COSETTE found counterexamples for all **Bugs**, all inequivalent queries in **Exams** and all mutant queries listed in **XData**. We select a few inequivalent queries and describe how COSETTE generates small counterexamples for them below.

4.2.1 Finding Counterexamples

The `COUNT` Bug. The `COUNT` bug is an incorrect optimizer rewrite rule [66] expressed using `Q1` and `Q2` as shown in Figure 4.2. COSETTE returns a counterexample containing the following two concrete tables when asked whether the two queries are equivalent:

pnum	qoh	**multiplicity**
0	0	**8**
2	2	**15**

pnum	shipdate	**multiplicity**
2	9	**2**

When executed, Q1 returns `[([0],8), ([2],15)]` while Q2 returns `[([2],15)]`. COSETTE took 2 iterations to find the counterexamples since the `Parts` table requires at least two unique tuples to demonstrate the inequivalence, and during the first iteration COSETTE only considered tables of size 1.

Oracle 12c Optimizer Bug. We next asked COSETTE to determine query equivalence of a real-world bug report on Oracle 12c [13]. In the report, the user mentioned that the result of the query below is incorrect due to an optimizer bug where it converted the first left outer join (Line 7) to a hash join. This resulted in a wrong execution plan, since tuples from `thing` that have no match in `tr` are removed by the hash join but retained by the original outer join.

```
1 --Table schemas:
2 --  thing(tid, tname), tr(rid, tid, type),
3 --  ta_status(rid, status), tb_status(rid, status)
4 SELECT t.tid, t.name, tas.status, tbs.status
5 FROM thing t LEFT JOIN tr
6   ON t.tid = tr.tid
7 LEFT JOIN ta_status tas
8   ON (tr.rid IS NOT NULL
9       AND tr.type = 1 AND tr.rid = tas.rid)
10 LEFT JOIN tb_status tbs
11   ON (tr.rid IS NOT NULL
12       AND tr.type = 2 AND tr.rid = tbs.rid)
```

Although the bug is difficult for users to locate, COSETTE identified the incorrectness of the optimization efficiently:[3] when fed with the original and optimized queries into COSETTE, the coun-

[3]Recall that COSETTE currently uses integers to model strings.

terexample below is returned:

```
thing = [([0,0],15)]      tr = []
ta_status = [([0,0],4)]   tb_status = [([0,0],3)]
```

Given the generated counter example, the original query evaluates to `[([0,0,null,null],15)]` while the incorrect optimized query evaluates to an empty table, indicating that the rewrite is incorrect.

Two Inequivalent Queries from Exams. One question from the exams ask students whether the two queries below are equivalent:

```
SELECT x.uid, x.uname,                          -- Q1
       (SELECT count(*) FROM Picture y
        WHERE x.uid = y.uid AND y.num > 1000000)
FROM Usr x
WHERE x.city = 'Denver';

SELECT x.uid, x.uname, COUNT(*)                 -- Q2
FROM Usr x, Picture y
WHERE x.uid = y.uid AND y.num > 1000000
      AND x.city = 'Denver'
GROUP BY x.uid, x.uname;
```

`Q1` and `Q2` are inequivalent as `Q2` filtered out all the cities that are not `'Denver'` first, so the count only considers `'Denver'` tuples, whereas `Q1` counts all tuples prior to filtering. COSETTE took 3 iterations to find a counterexample.

4.3 Combining SQL Prover and Symbolic Executor

We build a system,COSETTE, which combines the strengths of constraint solvers and proof assistants to determine the equivalence of SQL queries. It consists of two components: a compiler that translates the input SQL queries into logic formulas, and subsequently uses a constraint solver to

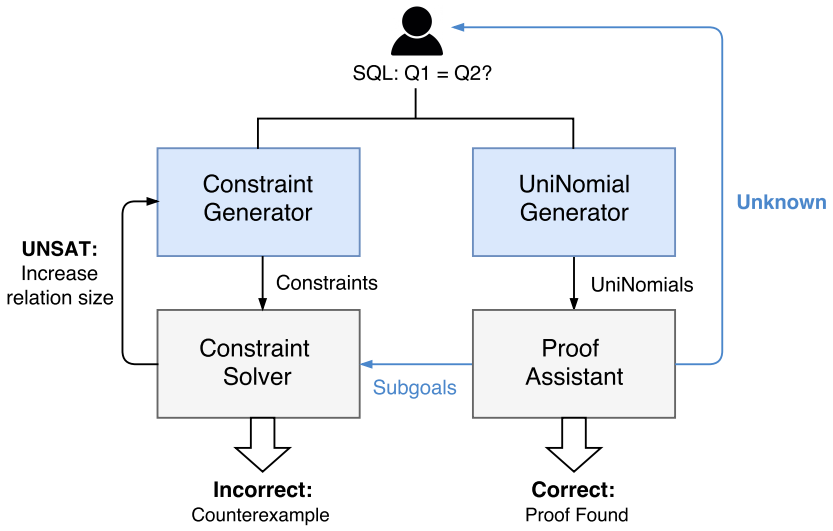

Figure 4.4: COSETTE architecture, where texts and arrows in blue indicate user interactions.

find counterexamples to show that the input queries are not equivalent; and a separate compiler translating queries into K-relations [74], and then uses a proof assistant to validate the equivalence of the two queries. The encoding of SQL to logic formulas allows us find counter examples of inequivalent SQL queries by constraint solvers. On the other hand, the encoding of SQL to K-Relations, where each relation is represented as mathematical function that takes as input a tuple and returns its multiplicity, allows proof assistants to easily search for machine checkable proofs of equivalence. As we will see, this unique combination of proving techniques enables COSETTE to efficiently determine the equivalence SQL queries.

In general, the query equivalence problem for arbitrary SQL queries is undecidable.[4] So an automated proof system for SQL will never be complete. However, our experiments show that COSETTE can already determine equivalence for a wide variety of real-world queries.

Figure 4.4 shows the architecture of COSETTE. COSETTE takes in two SQL queries and returns either "equivalent," "inequivalent," or "unknown." Besides the input queries, users can provide the actual contents of the relations that the queries are executed on. If the contents of all involved relations are provided, then determining the equivalence of the two queries reduces to executing

[4]This follows from Trakthenbrot's theorem [108].

them and checking if their result sets are the same. Otherwise, for the *symbolic relations*, i.e., relations that are queried but whose contents are not provided, COSETTE assumes that they can range over any valid schemas and values. The goal of COSETTE then is to check whether the input queries are equivalent when executed on all possible relations. To infer the schemas for the symbolic relations, COSETTE scans the input queries for the attributes that are referenced. For example, if `Emp` is a symbolic relation, and one of the queries contains the predicate `Emp.age > 21`, then COSETTE will infer that `Emp` contains at least the integer attribute `age`. COSETTE assumes that symbolic relations with different names are distinct. Furthermore, predicates can be symbolic as well, meaning that they represent any Boolean functions that take tuples as input. This is useful for checking the equivalences of query rewrite rules that are part of query optimizers, where such rules are often expressed over arbitrary predicates.

COSETTE passes the input queries to the two compilation toolchain. On the one hand, the *constraints generator* translates the input queries into constraints. This involves bounding the size of each symbolic relation and determining its schema. The compiler then uses fresh symbolic variables to represent each of the tuples in the symbolic relations, and translates the semantics of the input queries into constraints over the symbolic variables. The generated constraints are sent to a constraints solver. If the solver returns a counterexample, then the input queries are proven to be inequivalent, and the counterexample is returned to the user.

On the other hand, if the constraint solver cannot find a counterexample, COSETTE will then forward the queries to the *uninomial generator* (which is described in detail in Chapter 3). The generator compiles the symbolic relations to K-relations, which are mathematical functions that return the multiplicity of a given tuple, and translates the queries into algebraic expressions over K-relations called UniNomials. The UniNomials are sent to the proof assistant to look for an equivalence proof.

4.4 Summary

In this chapter, we demonstrate how to find counterexamples for inequivalent SQL queries by symbolic executing them. We build a symoblic exectuor for SQL using Rosette, a solver-aided

programming language and virtual machine, Rosette. This symbolic executor could explore finite sized input relations and find a counterexample for input SQL queries. We uses incremental solving and model compression techniques to reduce the solving time. The empirical study shows that this symbolic executor is able to find counterexamples for a wide range of queries within a short period of time. In addition, we also show how to integrate this symbolic executor with the prover to build a SQL solver that could both verify equivalences and determine inequivalences by finding counterexamples.

Chapter 5

FUTURE RESEARCH DIRECTIONS

In this chapter, we discuss a few future research directions that could extend this thesis research. We expect that this new axiomatic foundation unveils new search spaces for query optimization and thus brings new query optimization techniques (Section 5.1). Also, our new formalization could narrow the gap of building a fully optimized end-to-end verfied database systems (Section 5.2). In addition, beyond answering the "yes or no" question, it would be useful to answer the "why" question on reasoning SQL queries, as well. For example, give users high level information to explain why the input SQL queries are inequivalent (Section 5.3). We conclude this chapter in Section 5.4.

5.1 Rethinking Query Optimization

One of the most appealing property of relational database systems is that they are highly efficient in term of execution time while support a high-level declarative query language. This is largely due to sophiscated query optimizers that can choose a relatively optimal query plan [39].

However, the bases of the search space of existing query optimizers is a set of simple rewrite rules in relational algebra, such as push down selection and exchange join orders [67]. It is not clear that these simple rules can be composed to express all the equivalences. As a result, the search space defined by them could be imcomplete. This means that current query optimizers may miss many subtle optimization opportunities. For example, as far as we known, no query optimizer utilizes the following equivalences that we use as an axiom (Eq.(3.4)):

$$\|x\| \times \|y\| = \|x \times y\|$$

Practically, there are 3 possible ways that our research could help improving query optimizations:

First, the axiomatic foundation that we discovered in Chapter 3 could serve as a new basis of query plan enumeration in a query optimizer. This could make an optimizer consider plans that would be otherwise never considered before.

Second, COSETTE could be used to discover new rewrite rules that have never been discovered before. So that the existing optimizers could add these new rules to their rewrite rule bases.

Third, we could actually build superoptimizers for databases, e.g. an optimzier that guaranteed to explore all equivalent spaces up to a certain size [87]. More specifically, COSETTE could be used as the verifier part of a synthesize based superoptimizer for databases. It is not hard to imagine that the running time of the optimizer will be higher due to larger search spaces. However, like STOKE [99], Markov Chian Monte Carlo (MCMC) sampling could be applied to mitigate this problem.

5.2 End-to-end Verified Database Systems

Database system is a critical layer of modern information infrastructure. The bugs or misunderstandings of the behaviors of database systems could leads to huge loss of financial asset. For example, a double spending based on a LevelDB/BerkeleyDB behavor mismatch caused the loss of 211 Bitcoin (roughly $ 168,000 in today's value) [22].

End-to-end verfied database systems could provide the utimate correctness guarantee. However, this is not easy. The existing attempts include Q*cert [25, 24], COSETTE [44, 43, 42], and Ynot [86]. They all suffer some significant limitations. Q*cert is a verified by construction query optimizer. COSETTE is a automated reasoner for SQL queries. Using them, one can verify the correctness of the query optimization. But the underlying execution engine, as well as the interface between the optimizer and the execution engine is not verified. Ynot provide end-to-end verfication guarantee. However, it lacks many important features like indices and sophiscated query optimization.

To close the gap, we believe that layered abstractions are needed. In fact, one reason that Ynot could not support sophiscated query optimization is that the proof engineering cost is too high due to its low abstraction level. One possible approach is to develop lower level operational semantics from COSETTE's current denotational semantics, and prove their equivalences. For this lower level

operational semantics, we could develop verified by construction execution engines.

5.3 Counterfactual Reasoning of Database Queries

Currently, COSETTE can only answer the "yes or no" question on database queries. We imagine that an advanced reasoner should be able to answer "why" question as well. More specifically, the reasoner should give users high level explanations when the two input SQL queries are inequivalent.

One possible way of providing such high level explanations is *counterfactual reasoning*. More specifically, if two SQL queries Q_1 and Q_2 are not equivalent ($Q_1 \equiv Q_2 \rightarrow$ `false`), a possible explanation is a precondition P, such that Q_1 and Q_2 are equivalent if P hold: $P \rightarrow Q_1 \equiv Q_2$. Program synthesis techniques [20] can be applied in order to efficiently finding preconditions. One inherent trade-off here is the expressiveness of the precondition language and the efficiency of precondition synthesis. We propose to start from these set of preconditions that most real world relational database systems support:

- **Key Constraint.** A key cosntraint is to specify that the values of one or more attributes are unique in a table. E.g. attribute a is a key to table t, means that there is no duplicate in `select` a `from` t. A key can be composed by multiple attributes as well.

- **Referential Constraint.** A referential constraint specifies a primary key and foreign key pairs. For example, if attribute a_1 of table t_1 is a foreign key to attribute a_2 of table t_2. This means a_2 must be a key to table t_2, and every value of a_1 in t_1 must be a value of a_2 in t_2.

- **Table Assertion.** A table assertion is a predicate on a single table, for example, a `>` `10` `AND` b `<` `5` on a table with attributes a and b.

5.4 Summary

In this chapter, we outlook the future research directions from this thesis. We think the axiomatic foundation and the reasoning tool developed in this thesis could lead to better query optimizaitons,

facilitate the development of end-to-end verified database systems, and open the door of counterfactual reasoning of database queries.

BIBLIOGRAPHY

[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] CMU 15-811: Verifying Complex Systems. https://www.cs.cmu.edu/~15811/.

[9] CSE 344 Introduction to Data Management, Fall 2013, University of Washington. https://courses.cs.washington.edu/courses/cse344/13au/.

[10] Homotopy Type Theory Blog. https://homotopytypetheory.org/2016/09/26/hottsql-proving-query-rewrites-with-univalent-sql-semantics/.

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

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

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

[14] Reason. https://en.wikipedia.org/wiki/Reason.

[15] The Coq Proof Assistant. https://coq.inria.fr/.

[16] The Gallina specification language. https://coq.inria.fr/refman/gallina.html.

[17] the morning paper, Oct. 5, 2017. ttps://blog.acolyer.org/2017/10/05/hottsql-proving-query-rewrites-with-univalent-sql-semantics.

[18] The tactic language. https://coq.inria.fr/refman/ltac.html.

[19] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[20] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8. IEEE, 2013.

[21] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". 2002.

[22] Gavin Andresen. BIP-0050: March 2013 Chain Fork Post-Mortem.

[23] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD Conference*, pages 1383–1394. ACM, 2015.

[24] Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme 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.

[25] Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. Q*cert: A platform for implementing and verifying query compilers. In *SIGMOD Conference*, pages 1703–1706. ACM, 2017.

[26] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15, 1986.

[27] Bruno Barras, Benjamin Grégoire, Assia Mahboubi, and Laurent Théry. Coq reference manual chapter 25: The ring and field tactic families. `https://coq.inria.fr/refman/Reference-Manual028.html`.

[28] Clark Barrett, Haniel Barbosa, Martin Brain, Duligur Ibeling, Tim King, Paul Meng, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, and Cesare Tinelli. CVC4 at the SMT competition 2018. *CoRR*, abs/1806.08775, 2018.

[29] Clark Barrett et al. SMT-LIB standard v2.5. `http://smtlib.cs.uiowa.edu`.

[30] Andreas Bauer, Markus Pister, and Michael Tautschnig. Tool-support for the analysis of hybrid systems and models. In *DATE*, pages 924–929. EDA Consortium, San Jose, CA, USA, 2007.

[31] Michael Benedikt. How can reasoners simplify database querying (and why haven't they done it yet)? In *PODS*, pages 1–15. ACM, 2018.

[32] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. Benchmarking the chase. In *SIGMOD Conference*, pages 37–52, 2017.

[33] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

[34] De Bruijn and Nicolaas Govert. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[35] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[36] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[37] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system R. *Commun. ACM*, 24(10):632–646, 1981.

[38] Ashok K. Chandra and Philip 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.

[39] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43. ACM Press, 1998.

[40] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[41] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* conjunctive queries. In *PODS*, pages 59–70. ACM Press, 1993.

[42] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. Demonstration of the cosette automated SQL prover. In *SIGMOD Conference*, pages 1591–1594. ACM, 2017.

[43] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR*. www.cidrdb.org, 2017.

[44] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017.

[45] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among conjunctive aggregate queries. *J. ACM*, 54(2):5, 2007.

[46] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *PODS*, pages 155–166. ACM Press, 1999.

[47] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[48] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[49] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[50] Shaul Dar, Michael J. Franklin, Björn Thór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341. Morgan Kaufmann, 1996.

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

[52] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[53] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[54] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[55] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *CADE*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[56] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In *ISSTA*, pages 109–120. ACM, 2006.

[57] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In *PODS*, pages 149–158. ACM, 2008.

[58] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, pages 459–470. Morgan Kaufmann, 1999.

[59] Alin Deutsch, Lucian Popa, and Val Tannen. Chase & backchase: A method for query optimization with materialized views and integrity constraints. 01 2001.

[60] E. W. Dijkstra. Under the spell of Leibniz's Dream. In *EWD1298*.

[61] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017.

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

[63] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 207–224. Springer, 2003.

[64] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 207–224. Springer, 2003.

[65] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1(3-4):209–236, 2007.

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

[67] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[68] K. Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications, 1992.

[69] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342. ACM, 2001.

[70] Michel Gondran and Michel Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms (Operations Research/Computer Science Interfaces Series)*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[71] Mike Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–186. The MIT Press, 2000.

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

[73] G. Gratzer. *Universal Algebra*. Springer-Verlag, 1980.

[74] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[75] Jason Gross, Mike Shulman, Andrej Bauer, Peter LeFanu Lumsdaine, Assia Mahboubi, and Bas Spitters. The HoTT libary in Coq. `https://github.com/HoTT/HoTT`.

[76] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. Verifying equivalence of spark programs. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 282–300. Springer, 2017.

[77] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. Verifying equivalence of spark programs. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 282–300. Springer, 2017.

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

[79] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the Myria big data management service. In *SIGMOD Conference*, pages 881–884. ACM, 2014.

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

[81] ISO/IEC. Iso/iec 9075-1:2011. `https://www.iso.org/obp/ui/#iso:std:iso-iec:9075:-1:ed-4:v1:en`.

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

[83] Robert B. Jones, Jens U. Skakkebæk, and David L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *FMCAD*, volume 1522 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1998.

[84] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, pages 95–104. ACM Press, 1995.

[85] Yuliang Li, Alin Deutsch, and Victor Vianu. VERIFAS: A practical verifier for artifact systems. *PVLDB*, 11(3):283–296, 2017.

[86] J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *POPL*, pages 237–248, 2010.

[87] Henry Massalin. Superoptimizer - A look at the smallest program. In *ASPLOS*, pages 122–126. ACM Press, 1987.

[88] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[89] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *SIGMOD Conference*, pages 247–258, 1990.

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

[91] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 673–676. Springer, 1992.

[92] Giuseppe Peano. *Arithmetices Principia Novo Methodo Exposita*. Bocca, 1889.

[93] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD Conference*, pages 39–48. ACM Press, 1992.

[94] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A chase too far? In *SIGMOD Conference*, pages 273–284. ACM, 2000.

[95] Lucian Popa and Val 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.

[96] M. Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*. publisher not identified, 1931.

[97] Jean Rohmer, R. Lescoeur, and Jean-Marc Kerisit. The alexander method - A technique for the processing of recursive axioms in deductive databases. *New Generation Comput.*, 4(3):273–285, 1986.

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

[99] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, pages 305–316. ACM, 2013.

[100] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. Optimizing big-data queries using program synthesis. In *SOSP*, pages 631–646. ACM, 2017.

[101] Ziv Scully and Adam Chlipala. A program optimization for automatic database result caching. In *POPL*, pages 271–284. ACM, 2017.

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

[103] Shetal Shah et al. Generating test data for killing SQL mutants: A constraint-based approach. In *ICDE*, pages 1175–1186, 2011.

[104] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *VLDB*, pages 318–329. Morgan Kaufmann, 1996.

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

[106] Emina Torlak. Practical applications of sat.

[107] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, page 54. ACM, 2014.

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

[109] Transaction Processing Performance Council (TPC). Tpc benchmark h revision 2.17.1. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf.

[110] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. In *VLDB*, pages 367–378. Morgan Kaufmann, 1994.

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

[112] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic query exploration. In *ICFEM*, pages 49–68, 2009.

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

[114] Willem Visser et al. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[115] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. The Myria big data management and analytics system and cloud services. In *CIDR*. www.cidrdb.org, 2017.

[116] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *PACMPL*, 2(POPL):56:1–56:29, 2018.

Appendix A

TRANSLATING *HoTT*SQL TO *HoTT*IR

In this appendix we provide details on translating *HoTT*SQL to *HoTT*IR.

Figure A.1 shows the syntax of *HoTT*IR. It is an unnamed version of *HoTT*SQL.

Figure A.2 and Figure A.3 shows the translation rules and the scheme inference rules from *HoTT*SQL to *HoTT*IR. A transformation rule ($\texttt{Tr}_c(\ldots)$) will translate a *HoTT*SQL AST to a *HoTT*IR AST given a context schema c. A schema inference rule will infer the schema of a query or a projection.

In Figure A.3, the function `topath` will convert a table alias to a path expression given a context c. The function `table_schema` will look up *HoTT*SQL's declarations and find the schema of table t. The function `alias_schema` will look up the schema of a table alias given a context c.

$$
\begin{array}{lcl}
q \in \texttt{Query} & ::= & t \\
& | & \texttt{SELECT}\ p\ q \\
& | & \texttt{FROM}\ q_1, \ldots, q_n \\
& | & q\ \texttt{WHERE}\ b \\
& | & q_1\ \texttt{UNION ALL}\ q_2 \\
& | & q_1\ \texttt{EXCEPT}\ q_2 \\
& | & \texttt{DISTINCT}\ q \\
t \in \texttt{Table} & & \\
b \in \texttt{Predicate} & ::= & e_1 = e_2 \\
& | & \texttt{NOT}\ b \mid b_1\ \texttt{AND}\ b_2 \mid b_1\ \texttt{OR}\ b_2 \\
& | & \texttt{TRUE} \mid \texttt{FALSE} \\
& | & \texttt{CASTPRED}\ p\ b \\
& | & \texttt{EXISTS}\ q \\
e \in \texttt{Expression} & ::= & \texttt{P2E}\ \ p \\
& | & f(e_1, \ldots, e_n) \mid agg(q) \\
& | & \texttt{CASTEXPR}\ p\ e \\
p \in \texttt{Projection} & ::= & \texttt{*} \mid \texttt{Left} \mid \texttt{Right} \mid \texttt{Empty} \\
& | & p_1.\ p_2 \\
& | & p_1,\ p_2 \\
& | & \texttt{E2P}\ e
\end{array}
$$

Figure A.1: Syntax of *HoTT*IR

$\mathsf{Tr}_{\mathsf{Ctx}}(\mathsf{AST}_{\mathit{HoTTSQL}}) \rightsquigarrow \mathsf{AST}_{\mathit{HoTTIR}}$

% Translating HoTTSQL Query

$\mathsf{Tr}_c(t)$	$\rightsquigarrow$	t
$\mathsf{Tr}_c(\texttt{SELECT}\ p\ q)$	$\rightsquigarrow$	$\texttt{SELECT}\ \mathsf{Tr}_{c \oplus \mathsf{Ct}_c(q)}(p)\ \mathsf{Tr}_c(q)$
$\mathsf{Tr}_c(\texttt{FROM}\ q_1\ x_1,\ q_2\ x_2,\ \ldots,\ q_n\ x_n)$	$\rightsquigarrow$	$\texttt{FROM}\ \mathsf{Tr}_c(q_1),\ \mathsf{Tr}_c(\texttt{FROM}\ q_2\ x_2,\ \ldots,\ q_n\ x_n)$
$\mathsf{Tr}_c(\texttt{FROM}\ q\ x)$	$\rightsquigarrow$	$\texttt{FROM}\ \mathsf{Tr}_c(q)$
$\mathsf{Tr}_c(q\ \texttt{WHERE}\ b)$	$\rightsquigarrow$	$\mathsf{Tr}_c(q)\ \texttt{WHERE}\ \mathsf{Tr}_{c \oplus \mathsf{Ct}_c(q)}(b)$
$\mathsf{Tr}_c(q_1\ \texttt{UNION ALL}\ q_2)$	$\rightsquigarrow$	$\mathsf{Tr}_c(q_1)\ \texttt{UNION ALL}\ \mathsf{Tr}_c(q_2)$
$\mathsf{Tr}_c(q_1\ \texttt{EXCEPT}\ q_2)$	$\rightsquigarrow$	$\mathsf{Tr}_c(q_1)\ \texttt{EXCEPT}\ \mathsf{Tr}_c(q_2)$
$\mathsf{Tr}_c(\texttt{DISTINCT}\ q)$	$\rightsquigarrow$	$\texttt{DISTINCT}\ \mathsf{Tr}_c(q)$

% Translating HoTTSQL Predicate

$\mathsf{Tr}_c(e_1 = e_2)$	$\rightsquigarrow$	$\mathsf{Tr}_c(e_1) = \mathsf{Tr}_c(e_2)$
$\mathsf{Tr}_c(\texttt{NOT}\ b)$	$\rightsquigarrow$	$\texttt{NOT}\ \mathsf{Tr}_c(b)$
$\mathsf{Tr}_c(b_1\ \texttt{AND}\ b_2)$	$\rightsquigarrow$	$\mathsf{Tr}_c(b_1)\ \texttt{AND}\ \mathsf{Tr}_c(b_2)$
$\mathsf{Tr}_c(\texttt{TRUE})$	$\rightsquigarrow$	$\texttt{TRUE}$
$\mathsf{Tr}_c(\texttt{FALSE})$	$\rightsquigarrow$	$\texttt{FALSE}$
$\mathsf{Tr}_c(\beta(x_1, x_2, \ldots, x_n))$	$\rightsquigarrow$	$\texttt{CASTPRED}\ (\texttt{topath}(c, x_1), \texttt{topath}(c, x_2),\ \ldots,\ \texttt{topath}(c, x_n))\ \beta$
$\mathsf{Tr}_c(\texttt{EXISTS}\ q)$	$\rightsquigarrow$	$\texttt{EXISTS}\ \mathsf{Tr}_c(q)$

% Translating HoTTSQL Expression

$\mathsf{Tr}_c(x.a)$	$\rightsquigarrow$	$\texttt{P2E}\ \texttt{topath}(c, x).a$
$\mathsf{Tr}_c(f(e_1, \ldots, e_n))$	$\rightsquigarrow$	$f(\mathsf{Tr}_c(e_1), \ldots, \mathsf{Tr}_c(e_n))$
$\mathsf{Tr}_c(\alpha(q))$	$\rightsquigarrow$	$\alpha(\mathsf{Tr}_c(q))$

% Translating HoTTSQL Projection

$\mathsf{Tr}_c(*)$	$\rightsquigarrow$	$*$
$\mathsf{Tr}_c(x.*)$	$\rightsquigarrow$	$\texttt{topath}(c, x).*$
$\mathsf{Tr}_c(p_1,\ p_2)$	$\rightsquigarrow$	$\mathsf{Tr}_c(p_1),\ \mathsf{Tr}_c(p_2)$

Figure A.2: Translating *HoTT*SQL to *HoTT*IR

$\boxed{\texttt{Ct}_{\texttt{Ctx}}(\texttt{AST}_{\textit{HoTT}\text{SQL}}) \rightsquigarrow \texttt{Ctx}}$

$\texttt{Ct}_c(t)$	$\rightsquigarrow$	$\texttt{table\_schema}(t)$
$\texttt{Ct}_c(\texttt{SELECT}\ p\ q)$	$\rightsquigarrow$	$\texttt{Ct}_{c \oplus \texttt{Ct}_c(q)}(p)$
$\texttt{Ct}_c(\texttt{FROM}\ q_1\ x_1,\ q_2\ x_2,\ \ldots,\ q_n\ x_n)$	$\rightsquigarrow$	$\texttt{Ct}_c(q_1) \bigoplus \texttt{Ct}_c(q_2, \ldots, q_n)$
$\texttt{Ct}_c(\texttt{FROM}\ q\ x)$	$\rightsquigarrow$	$\texttt{Ct}_c(q)$
$\texttt{Ct}_c(q\ \texttt{WHERE}\ b)$	$\rightsquigarrow$	$\texttt{Ct}_c(q)$
$\texttt{Ct}_c(q_1\ \texttt{UNION ALL}\ q_2)$	$\rightsquigarrow$	$\texttt{Ct}_c(q_1)$
$\texttt{Ct}_c(q_1\ \texttt{EXCEPT}\ q_2)$	$\rightsquigarrow$	$\texttt{Ct}_c(q_1)$
$\texttt{Ct}_c(\texttt{DISTINCT}\ q)$	$\rightsquigarrow$	$\texttt{Ct}_c(q)$
$\texttt{Ct}_c(\texttt{*})$	$\rightsquigarrow$	c
$\texttt{Ct}_c(x\texttt{.*})$	$\rightsquigarrow$	$\texttt{alias\_schema}(c, x)$
$\texttt{Ct}_c(e\ \texttt{AS}\ a)$	$\rightsquigarrow$	$\texttt{typeof}(c, e)$
$\texttt{Ct}_c(p_1,\ p_2)$	$\rightsquigarrow$	$\texttt{Ct}_c(p_1) \bigoplus \texttt{Ct}_c(p_2)$

Note: The Schema Inference Rule of FROM ... also add the table alias into context for future look up, we omit that for brevity.

Figure A.3: Schema Inference Rules for Translating *HoTT*SQL to *HoTT*IR

Appendix B

TRANSLATING *HoTT*IR TO UNINOMIAL

Figure B.1 and Figure B.1 show the denotational semantics for all constructs in *HoTT*IR.

$\boxed{[\![\Gamma \vdash q : \sigma]\!] : \texttt{Tuple}\ \Gamma \to \texttt{Tuple}\ \sigma \to \mathcal{U}}$ (* *Query* *)

$$
\begin{aligned}
[\![\Gamma \vdash table : \sigma]\!] &\triangleq \lambda\ g\ t.\ [\![table]\!]\ t \\
[\![\Gamma \vdash \texttt{SELECT}\ p\ q : \sigma]\!] &\triangleq \lambda\ g\ t.\ \textstyle\sum_{t' : \texttt{Tuple}\ \sigma'} ([\![p : \texttt{node}\ \Gamma\ \sigma' \Rightarrow \sigma]\!]\ (g, t') = t) \times [\![\Gamma \vdash q : \sigma']\!]\ g\ t' \\
[\![\Gamma \vdash \texttt{FROM}\ q_1, q_2 : \texttt{node}\ \sigma_1\ \sigma_2]\!] &\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q_1 : \sigma_1]\!]\ g\ t.1 \times [\![\Gamma \vdash q_2 : \sigma_2]\!]\ g\ t.2 \\
[\![\Gamma \vdash \texttt{FROM}\ q : \sigma]\!] &\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q : \sigma]\!]\ g\ t \\
[\![\Gamma \vdash q\ \texttt{WHERE}\ b : \sigma]\!] &\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q : \sigma]\!]\ g\ t \times [\![\texttt{node}\ \Gamma\ \sigma \vdash b]\!]\ (g, t) \\
[\![\Gamma \vdash q_1\ \texttt{UNION ALL}\ q_2 : \sigma]\!] &\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q_1 : \sigma]\!]\ g\ t + [\![\Gamma \vdash q_2 : \sigma]\!]\ g\ t \\
[\![\Gamma \vdash q_1\ \texttt{EXCEPT}\ q_2 : \sigma]\!] &\triangleq \lambda\ g\ t.\ [\![\Gamma \vdash q_1 : \sigma]\!]\ g\ t \times (([\![\Gamma \vdash q_2 : \sigma]\!]\ g\ t) \to \mathbf{0}) \\
[\![\Gamma \vdash \texttt{DISTINCT}\ q : \sigma]\!] &\triangleq \lambda\ g\ t.\ \|[\![\Gamma \vdash q : \sigma]\!]\ g\ t\|
\end{aligned}
$$

$\boxed{[\![\Gamma \vdash b]\!] : \texttt{Tuple}\ \Gamma \to \mathcal{U}}$ (* *Predicate* *)

$$
\begin{aligned}
[\![\Gamma \vdash e_1 = e_2]\!] &\triangleq \lambda\ g.\ ([\![\Gamma \vdash e_1 : \tau]\!]\ g = [\![\Gamma \vdash e_2 : \tau]\!]\ g) \\
[\![\Gamma \vdash b_1\ \texttt{AND}\ b_2]\!] &\triangleq \lambda\ g.\ [\![\Gamma \vdash b_1]\!]\ g \times [\![\Gamma \vdash b_2]\!]\ g \\
[\![\Gamma \vdash b_1\ \texttt{OR}\ b_2]\!] &\triangleq \lambda\ g.\ \|[\![\Gamma \vdash b_1]\!]\ g + [\![\Gamma \vdash b_2]\!]\ g\| \\
[\![\Gamma \vdash \texttt{NOT}\ b]\!] &\triangleq \lambda\ g.\ ([\![\Gamma \vdash b]\!]\ g) \to \mathbf{0} \\
[\![\Gamma \vdash \texttt{EXISTS}\ q]\!] &\triangleq \lambda\ g.\ \left\|\textstyle\sum_{t : \texttt{Tuple}\ \sigma} [\![\Gamma \vdash q : \sigma]\!]\ g\ t\right\| \\
[\![\Gamma \vdash \texttt{FALSE}]\!] &\triangleq \lambda\ g.\ \mathbf{0} \\
[\![\Gamma \vdash \texttt{TRUE}]\!] &\triangleq \lambda\ g.\ \mathbf{1} \\
[\![\Gamma \vdash \texttt{CASTPRED}\ p\ b]\!] &\triangleq \lambda\ g.\ [\![\Gamma' \vdash b]\!]\ ([\![p : \Gamma \Rightarrow \Gamma']\!]\ g)
\end{aligned}
$$

Figure B.1: Denotational Semantics of *HoTT*SQL (First Part)

$\boxed{[\![\Gamma \vdash e : \tau]\!] : \texttt{Tuple}\ \Gamma \to [\![\tau]\!]}$ (* *Expression* *)

$$
\begin{aligned}
[\![\Gamma \vdash \texttt{P2E}\ \ p : \tau]\!] &\triangleq \lambda\, g.\ [\![p : \Gamma \Rightarrow \texttt{leaf}\ \ \tau]\!]\ g \\
[\![\Gamma \vdash f(e_1, \ldots) : \tau]\!] &\triangleq \lambda\, g.\ [\![f]\!]([\![\Gamma \vdash e_1 : \tau_1]\!]\ g, \ldots) \\
[\![\Gamma \vdash agg(q) : \tau']\!] &\triangleq \lambda\, g.\ [\![agg]\!]\ ([\![\Gamma \vdash q : \texttt{leaf}\ \ \tau]\!]\ g) \\
[\![\Gamma \vdash \texttt{CASTEXPR}\ p\ e : \tau]\!] &\triangleq \lambda\, g.\ [\![\Gamma' \vdash e : \tau]\!]\ ([\![c : \Gamma \Rightarrow \Gamma']\!]\ g)
\end{aligned}
$$

$\boxed{[\![p : \Gamma \Rightarrow \Gamma']\!] : \texttt{Tuple}\ \Gamma \to \texttt{Tuple}\ \Gamma'}$ (* *Projection* *)

$$
\begin{aligned}
[\![* : \Gamma \Rightarrow \Gamma]\!] &\triangleq \lambda\, g.\ g \\
[\![\texttt{Left} : \texttt{node}\ \Gamma_0\ \Gamma_1 \Rightarrow \Gamma_0]\!] &\triangleq \lambda\, g.\ g.1 \\
[\![\texttt{Right} : \texttt{node}\ \Gamma_0\ \Gamma_1 \Rightarrow \Gamma_1]\!] &\triangleq \lambda\, g.\ g.2 \\
[\![\texttt{Empty} : \Gamma \Rightarrow \texttt{empty}]\!] &\triangleq \lambda\, g.\ \texttt{unit} \\
[\![p_1.\ p_2 : \Gamma \Rightarrow \Gamma'']\!] &\triangleq \lambda\, g.\ [\![p_2 : \Gamma' \Rightarrow \Gamma'']\!]\ ([\![p_1 : \Gamma \Rightarrow \Gamma']\!]\ g) \\
[\![p_1,\ p_2 : \Gamma \Rightarrow \texttt{node}\ \Gamma_0\ \Gamma_1]\!] &\triangleq \lambda\, g.\ ([\![p_1 : \Gamma \Rightarrow \Gamma_0]\!]\ g,\ [\![p_2 : \Gamma \Rightarrow \Gamma_1]\!]\ g) \\
[\![\texttt{E2P}\ e : \Gamma \Rightarrow \texttt{leaf}\ \ \tau]\!] &\triangleq \lambda\, g.\ [\![\Gamma \vdash e : \tau]\!]\ g
\end{aligned}
$$

Figure B.2: Denotational Semantics of *HoTT*SQL (Second Part)

Appendix C

PROOFS OF SQL REWRITE RULES

In this section, we show the additional details and proofs of the rewrite rules.

C.0.1 Commutativity of Joins

Commutativity of joins allows an optimizer to rearrange the order of joins in order to get the join order with best performance. This is one of the most fundamental rewrite rules that almost every optimizer uses. We formulate the commutativity of joins in *HoTT*SQL as follows:

$$\begin{aligned}&\texttt{SELECT * FROM } R\ x\texttt{, } S\ y \quad \equiv \\ &\texttt{SELECT } y.\texttt{*, } x.\texttt{* FROM } S\ x\texttt{, } R\ y\end{aligned}$$

Note that the select clause flips the tuples from S and R, such that the order of the tuples matches the original query. This will be denoted to:

$$\begin{aligned}&\lambda\ g\ t.\ [\![R]\!]\ g\ t.1 \times [\![S]\!]\ g\ t.2 \quad \equiv \\ &\lambda\ g\ t.\ \textstyle\sum_{t_1} [\![S]\!]\ g\ t_1.1 \times [\![R]\!]\ g\ t_1.2 \times ((t_1.2, t_1.1) = t)\end{aligned}$$

The proof uses Lemma C.0.1 provided by the COSETTE library.

Lemma C.0.1. *Let A inhabit $\mathcal{U}$, and have $P : A \to \mathcal{U}$ be a type family, then we have:*

$$\sum_{x:A\times B} P\ x = \sum_{x:B\times A} P\ (x.2, x.1)$$

Together with the fact that $t_1 = (t_1.1, t_1.2)$, the rewrite rule's right hand side becomes:

$$\lambda\ g\ t.\ \sum_{t'} [\![S]\!]\ g\ t'.2 \times [\![R]\!]\ g\ t'.1 \times (t' = t)$$

The proof then uses Lemma C.0.2 provided by the COSETTE library.

Lemma C.0.2. *Let A and B inhabit $\mathcal{U}$, and have $P : A \times B \to \mathcal{U}$, then we have:*

$$P\ x = \sum_{x'} P\ x' \times (x' = x)$$

After applying Lemma C.0.2, the right hand side becomes the following, and we can finish the proof by applying commutativity of $\times$:

$$\lambda\ g\ t.\ [\![S]\!]\ g\ t.2 \times [\![R]\!]\ g\ t.1$$

C.0.2 Aggregation and Group By Rewrite Rules

Aggregation and Group By are widely used in analytic queries [40]. The standard data analytic benchmark TPC-H [109] has 16 queries with group by and 21 queries with aggregation out of a total of 22 queries. Following is an example rewrite rule for aggregate queries. The query on the left-hand side groups the relation R by the column k, sums all values in the b column for each resulting partition, and then removes all results except the partition whose column k is equal to the constant l. This can be rewritten to the faster query that first removes all tuples from R whose column $k \neq l$, and then computes the sum.

```
(SELECT * FROM R1 x, R2 y WHERE θ1(x,y))
SEMIJOIN R3 ON θ2   ≡
(SELECT *
 FROM R1 x,
    (R2 SEMIJOIN (FROM R1, R3) ON θ1 AND θ2) y
 WHERE θ1(x,y)) SEMIJOIN R3 ON θ2
```

We use a correlated subquery and a unary aggregate function (which takes a *HoTT*SQL query as its input) to represent aggregation on group by SQL queries. After de-sugaring, the group by query becomes `SELECT DISTINCT ...`. The rule will thus be denoted to:

$$\begin{aligned}
&\lambda\ g\ t.\ (t.1 = [\![l]\!]) \times \| \textstyle\sum_{t_1} [\![R]\!]\ t_1 \times (t.1 = [\![k]\!]\ t_1) \times \\
&\qquad (t.2 = [\![\alpha]\!]\ (\lambda\ t'.\ \textstyle\sum_{t_2} ([\![k]\!]\ t_1 = [\![k]\!]\ t_2) \times \\
&\qquad\qquad [\![R]\!]\ t_2 \times ([\![b]\!]\ t_1 = t')) \| \\
&\equiv \\
&\lambda\ g\ t.\ \| \textstyle\sum_{t_1} ([\![k]\!]\ t_1 = [\![l]\!]) \times [\![R]\!]\ t_1 \times (t.1 = [\![k]\!]\ t_1) \times \\
&\qquad (t.2 = [\![\alpha]\!]\ (\lambda\ t'.\ \textstyle\sum_{t_2} ([\![k]\!]\ t_1 = [\![k]\!]\ t_2) \times ([\![k]\!]\ t_2 = [\![l]\!]) \\
&\qquad \times [\![R]\!]\ t_2 \times ([\![b]\!]\ t_1 = t')) \|
\end{aligned}$$

The proof proceeds by functional extensionality, after which both sides become squash types. The proof then uses the fundamental lemma about squash types, where for all squash types A and B, $(A \leftrightarrow B) \Rightarrow (A = B)$. It thus suffices to prove by cases the bi-implication ($\leftrightarrow$) of both sides. In both cases, instantiate t_1 with t_1 (t_1 is the witness of the Σ hypothesis). It follows that $t.1 = [\![l]\!] = [\![k]\!]\ t_1$, and thus that $[\![k]\!]\ t_2 = [\![l]\!]$ inside `SUM`.

C.0.3 Magic Set Rewrite Rules

Magic set rewrites are well known rewrite rules that were originally used in the recursive query processing in deductive databases [26, 97]. It was then used for rewriting complex decision support queries and has been implemented in commercial systems such as IBM's DB2 database [102, 89]. Below is an example of a complex magic set rewrite from [102].

Original Query:

```
CREATE VIEW DepAvgSal AS
    (SELECT E.did, AVG(E.sal) AS avgsal FROM Emp E
    GROUP BY E.did);
SELECT E.eid, E.sal
FROM Emp E, Dept D, DepAvgSal V
WHERE E.did = D.did AND E.did = V.did AND E.age < 30
    AND D.budget > 100000 AND E.sal > V.avgsal
```

Rewritten Query:

```
CREATE VIEW PartialResult AS
    (SELECT E.eid, E.sal, E.did FROM Emp E, Dept D
    WHERE E.did = D.did AND E.age < 30 AND
        D.budget > 100000);
CREATE VIEW Filter AS
    (SELECT DISTINCT P.did FROM PartialResult P);
CREATE VIEW LimitedDepAvgSal AS
    (SELECT F.did, AVG(E.sal) AS avgsal
    FROM Filter F, Emp E
    WHERE E.did = F.did
    GROUP BY F.did);
SELECT P.eid, P.sal
FROM PartialResult P, LimitedDepAvgSal V
WHERE P.did = V.did AND P.sal > V.avgsal
```

This query aims to find each young employee in a big department ($D.budget > 100000$) whose salary is higher then the average salary in her department. Magic set rewrites use the fact that only the average salary of departments that are big and have young employees need to be computed. As described in [102], all magic set rewrites can be composed from just three basic rewrite rules

on semijoins, namely introduction of θ-semijoin, pushing θ-semijoin through join, and pushing θ-semijoin through aggregation.

Following, we show how to state all three rewrite rules using COSETTE, and show how to prove two.

We firstly define θ-semijoin as a syntactic rewrite in *HoTT*SQL:

$$
\begin{aligned}
&A \texttt{ SEMIJOIN } B \texttt{ ON } \theta \quad \triangleq \\
&\texttt{SELECT * FROM } A\ x \\
&\texttt{WHERE EXISTS (SELECT * FROM } B\ y \texttt{ WHERE } \theta(x,y)\texttt{)}
\end{aligned}
$$

Introduction of θ-semijoin This rules shows how to introduce semijoin from join and selection. Using semijoin algebra notation, this rewrite can be expressed as follows:

$$R_1 \bowtie_\theta R_2 \equiv R_1 \bowtie_\theta (R_2 \ltimes_\theta R_1)$$

Using *HoTT*SQL, the rewrite can be expressed as follows:

$$
\begin{aligned}
&\texttt{SELECT * FROM } R_2\ x\texttt{, } R_1\ y \texttt{ WHERE } \theta(x,y) \quad \equiv \\
&\texttt{SELECT *} \\
&\texttt{FROM (}R_2\ x \texttt{ SEMIJOIN } R_1\ y \texttt{ ON } \theta(x,y)\texttt{) } z_1\texttt{, } R_1\ z_2 \\
&\texttt{WHERE } \theta(z_1, z_2)
\end{aligned}
$$

which is denoted to:

$$
\begin{aligned}
&\lambda\ g\ t.\ [\![\theta]\!]\ (g,t) \times [\![R_2]\!]\ g\ t.1 \times [\![R_1]\!]\ g\ t.2 \quad \equiv \\
&\lambda\ g\ t.\ [\![\theta]\!]\ (g,t) \times [\![R_2]\!]\ g\ t.1 \times [\![R_1]\!]\ g\ t.2 \times \\
&\qquad \left\| \textstyle\sum_{t_1} [\![\theta]\!]\ (g, (t.1, t_1)) \times [\![R_1]\!]\ g\ t_1 \right\|
\end{aligned}
$$

The proof uses Lemma C.0.3 provided by the COSETTE library.

Lemma C.0.3. *$\forall P, T : \mathcal{U}$, where P is either **0** or **1**, we have:*

$$(T \to P) \Rightarrow ((T \times P) = T)$$

Proof. Intuitively, this can be proven by cases on T. If T is inhabited, then P holds by assumption, and $T \times \mathbf{1} = T$. If $T = 0$, then $\mathbf{0} \times P = \mathbf{0}$. □

Using this lemma, it remains to be shown that $[\![\theta]\!]\ (g, t)$ and $[\![R_2]\!]\ g\ t.1$ and:

$$[\![R_1]\!]\ g\ t.2 \Rightarrow \left\| \sum_{t_1} [\![\theta]\!]\ (g, (t.1, t_1)) \times [\![R_1]\!]\ g\ t_1 \right\|$$

We show this by instantiating t_1 with $t.2$, and then by hypotheses.

Pushing θ-semijoin through join The second rule in magic set rewrites is the rule for pushing θ-semijoin through join, represented in semijoin algebra as:

$$(R_1 \bowtie_{\theta_1} R_2) \ltimes_{\theta_2} R_3 \equiv (R_1 \bowtie_{\theta_1} R_2') \ltimes_{\theta_2} R_3$$

where $R_2' = E_2 \ltimes_{\theta_1 \wedge \theta_2} (R_1 \bowtie R_3)$. This rule can be written in *HoTT*SQL as below:

```
(SELECT * FROM R1 x, R2 y WHERE θ1(x,y))
SEMIJOIN R3 ON θ2   ≡
(SELECT *
 FROM R1 x,
     (R2 SEMIJOIN (FROM R1, R3) ON θ1 AND θ2) y
 WHERE θ1(x,y)) SEMIJOIN R3 ON θ2
```

The rule is denoted to:

$$\begin{aligned}
&\lambda\ g\ t.\ \left\|\textstyle\sum_{t_1} [\![\theta_2]\!]\ (g, (t, t_1)) \times [\![R_3]\!]\ g\ t_1\right\| \times \\
&\qquad [\![\theta_1]\!]\ (g, t) \times [\![R_1]\!]\ g\ t.1 \times [\![R_2]\!]\ g\ t.2 \qquad \equiv \\
&\lambda\ g\ t.\ \left\|\textstyle\sum_{t_1} [\![\theta_2]\!]\ (g, (t, t_1)) \times [\![R_3]\!]\ g\ t_1\right\| \times \\
&\qquad [\![\theta_1]\!]\ (g, t) \times [\![R_1]\!]\ g\ t.1 \times [\![R_2]\!]\ g\ t.2 \times \\
&\qquad \| \textstyle\sum_{t_1} [\![\theta_1]\!]\ (g, (t_1.1, t.2)) \times [\![\theta_2]\!]\ (g, ((t_1.1, t.2), t_1.2)) \\
&\qquad \times [\![R_1]\!]\ g\ t_1.1 \times [\![R_3]\!]\ g\ t_1.2 \|
\end{aligned}$$

We can prove this rule by using a similar approach to the one used to prove introduction of θ-semijoin: rewriting the right hand side using Lemma C.0.3. and then instantiating t_1 with $(t.1, t_1)$ (t_1 is the witness of the Σ hypothesis).

Pushing θ-semijoin through aggregation The final rule is pushes θ-semijoin through aggregation:

$$ {}_{\bar{g}}\mathcal{F}_{\bar{f}}(R_1) \ltimes_{c_1=c_2} R_2 \equiv_{\bar{g}} \mathcal{F}_{\bar{f}}(R_1 \ltimes_{c_1=c_2} R_2) $$

where ${}_{\bar{g}}\mathcal{F}_{\bar{f}}$ is a grouping/aggregation operator (${}_{\bar{g}}\mathcal{F}_{\bar{f}}$ was firstly defined in [102]), and $\bar{g}$ denotes the group by attributes and $\bar{f}$ denotes the aggregation function. In this rule, one extra condition is that c_1 is from the attributes in $\bar{g}$ and c_2 is from the attributes of R_2. This rule can be written in *HoTT*IR as below:

$$
\begin{aligned}
&[\![\Gamma \vdash \texttt{(SELECT } c_1\texttt{, COUNT(}a\texttt{) FROM } R_1 \texttt{ GROUP BY } c_1\texttt{)} \\
&\qquad \texttt{SEMIJOIN } R_2 \texttt{ ON } c_1 = c_2 \ : \sigma]\!] \quad \equiv \\
&[\![\Gamma \vdash \texttt{SELECT } c_1\texttt{, COUNT(}a\texttt{)} \\
&\qquad \texttt{FROM (}R_1 \texttt{ SEMIJOIN } R_2 \texttt{ ON } c_1 = c_2\texttt{) GROUP BY } c_1 : \sigma]\!]
\end{aligned}
$$

We omit the proof here for brevity.