HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics

Shumo Chu, Konstantin Weitz, Alvin Cheung, Dan Suciu
University of Washington, USA
{chushumo, weitzkon, akcheung, suciu}@cs.washington.edu
http://cosette.cs.washington.edu

Abstract

Every database system contains a query optimizer that performs query rewrites. Unfortunately, developing query optimizers remains a highly challenging task. Part of the challenges comes from the intricacies and rich features of query languages, which makes reasoning about rewrite rules difficult. In this paper, we propose a machine-checkable denotational semantics for SQL, the de facto language for interacting with relational databases, for rigorously validating rewrite rules. Unlike previously proposed semantics that are either non-mechanized or only cover a small amount of SQL language features, our semantics covers all major features of SQL, including bags, correlated subqueries, aggregation, and indexes. Our mechanized semantics, called *HoTT*SQL, is based on K-Relations and homotopy type theory, where we denote relations as mathematical functions from tuples to univalent types. We have implemented *HoTT*SQL in Coq, which takes only fewer than 300 lines of code, and have proved a wide range of SQL rewrite rules, including those from database research literature (e.g., magic set rewrites) and real-world query optimizers (e.g., subquery elimination), where several of them have never been previously proven correct. In addition, while query equivalence is generally undecidable, we have implemented an automated decision procedure using *HoTT*SQL for conjunctive queries: a well-studied decidable fragment of SQL that encompasses many real-world queries.

CCS Concepts **• Information systems → Database query processing; Structured Query Language; • Theory of computation → Program verification**

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062348

Keywords SQL, Formal Semantics, Homotopy Types, Equivalence

1. Introduction

From purchasing plane tickets to browsing social networking websites, we interact with database systems on a daily basis. Every database system consists of a query optimizer that takes in an input query and determines the best program, also called a query plan, to execute in order to retrieve the desired data. Query optimizers typically consist of two components: a query plan enumerator that generates query plans that are semantically equivalent to the input query, and a plan selector that chooses the optimal plan from the enumerated ones to execute based on a cost model.

The key idea behind plan enumeration is to apply *rewrite rules* that transform a given query plan into another one that, hopefully, has a lower cost than the input. While numerous plan rewrite rules have been proposed and implemented, unfortunately designing such rules remains a highly challenging task. For one, rewrite rules need to be *semantically preserving*, i.e., if a rule transforms query plan Q into Q', then the results (i.e., the relation) returned from executing Q must be the same as those returned from Q', and this has to hold for *all* possible input database schemas and instances. Obviously, establishing such a proof for any non-trivial query rewrite rule is not an easy task.

Coupled with that, the rich language constructs and subtle semantics of SQL, the de facto programming language used to interact with relational database systems, only makes the task even more difficult. As a result, while various rewrite rules have been proposed and studied extensively in the data management research community [39, 42, 43, 50], to the best of our knowledge only some simple ones have been formally proven to be semantic preserving. This has unfortunately led to dire consequences as incorrect query results have been returned from widely-used database systems due to unsound rewrite rules, and such bugs can often go undetected for extended periods of time [22, 49, 51].

In this paper, we describe a system to formally verify the equivalence of two SQL expressions. We demonstrate its utility by proving correct a large number of query rewrite rules that have been described in the literature and are currently used in popular database systems. We also show that, given counter examples, common mistakes made in query optimization fail to pass our formal verification, as they should. Our system shares similar high-level goals of building formally verified systems using theorem provers and proof assistants as recent work [12, 37, 38].

The biggest challenge in a formal verification system for query equivalence is choosing the right SQL formalization. Although SQL is an ANSI standard [34], the "formal" semantics defined there is of little use for formal verification: it is loosely described in English and has resulted in conflicting interpretations [21]. In general, two quite different formalizations exists in the literature. The first comes from the formal methods community [41, 59, 60], where SQL relations are interpreted as lists, and SQL queries as functions over lists; two queries are equivalent if they return lists that are equal up to element permutation (under bag semantics) or up to element permutation and duplicate elimination (under set semantics). The problem with this semantics is that even the simplest equivalences require lengthy proofs in order to reason about order-independence or duplicate elimination, and these proofs become huge when applied to rewrites found in real-world optimizations. The second semantics comes from the database theory community and uses only set semantics [1, 6, 44]. This line of work has led to query containment techniques for conjunctive queries using tableau or homomorphisms [1, 8] and to various complexity results for the query containment and equivalence problems [8, 24, 48, 57]. The problem here is that it is restricted only to set semantics, and query equivalence under bag semantics is quite different; in fact, the first paper that noted that is entitled "Optimization of *Real* Conjunctive Queries," with an emphasis on *real* [10]. Under set semantics equivalence for both Conjunctive Queries[1] (CQ) and Unions of Conjunctive Queries (UCQ) is NP-complete [1], but under bag semantics equivalence is the same as graph isomorphism for CQ and is undecidable for UCQ [33].

This paper contributes a new SQL formalization that is both simple and allows simple query equivalence proofs. We demonstrate its effectiveness by proving the correctness of powerful query optimization rules found in the literature.

Our semantics consists of two non-trivial generalizations of K-relations [28], 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 multiplication 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 [28], 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) [52] 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; we call this language *HoTT*SQL. Our language covers all major features of SQL, including set and bag semantics, nested queries, etc. We implemented *HoTT*SQL as part of a new tool called COSETTE [17] 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 [9, 36], magic sets rewriting [2], query rewriting using indexes [55], and equivalence of conjunctive queries using the homomorphism theorem [1]. In the implementation of COSETTE, all our proofs require at most a few dozens lines of Coq code, as shown in Fig. 9. All definitions and proofs presented in this paper are open-source and available online.[2]

[1] Also called Select-Project-Join queries.

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

Rewrite Rule:

`SELECT` $*$ `FROM` (R `UNION ALL` S) x `WHERE` b $\equiv$
(`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 1. Proving a rewrite rule using *HoTTSQL*. Recall that `UNION ALL` means bag-union in SQL, which in *HoTTSQL* is translated to addition of tuple multiplicities in the two input relations.

In summary, we make the following contributions:

- We present *HoTTSQL*, a language for expressing SQL rewrite rules. *HoTTSQL* models a large fragment of SQL and can be used to express many real-world rewrite rules evaluated using bag semantics. (Sec. 3.)
- We implement *HoTTSQL* based on a generalization of K-relations to infinite relations and univalent types. The goal of this semantics is to enable easy proofs of query equivalence. (Sec. 4.)
- We show how SQL expressions can be translated into algebraic expressions over univalent types, called *uninomials* (Sec. 4.)
- In COSETTE, we prove a wide variety of well-known and widely-used SQL rewrite rules, many of which have never been formally proven before; each proof requires at most a few dozens lines of Coq code. (Sec. 5.)

We are aware of two prior projects for formally verifying SQL query equivalence. Malecha et al. [41] represent relations as lists and prove the correctness of 5 simple rewrite rules (removing redundant predicates that always evaluate to true or false, union with empty resulting in the same relation, pushing selection through projection, conjunction of two predicate equals applying them separately), requiring more than 700 lines of Coq code; in contrast, the same rules take fewer than 40 lines in our framework. Benzaken et al. [4] use only set semantics, consider only conjunctive queries, and prove formally several textbook equivalences including the equivalence of two conjunctive queries using the homomorphism theorem. As we explained above, this approach cannot handle equivalence under bag semantics; and it also cannot represent nested SQL queries. In contrast, *HoTTSQL* allows both set and bag semantics, and includes nested queries along with other features of SQL.

2. 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

`Q1:SELECT` $x.a$ `as` a `FROM` R x

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:

`Q2:SELECT DISTINCT` $x.a$ `as` a `FROM` R x

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 [41, 59, 60]. Every SQL query admits a natural interpretation over lists, using a recursive definition [7]. 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:

`Q3:SELECT DISTINCT` $x.a$ `AS a FROM` $R\; x, R\; y$ `WHERE` $x.a = y.a$

`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 [41], 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. 5), 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 `Tuple`(σ) the type of tuples with attributes σ. A K-relation [28] is a function:

$$\llbracket R \rrbracket : \texttt{Tuple}\ \sigma \to K$$

with finite support, meaning that $\{t \mid \llbracket R \rrbracket\; t \neq 0\}$ is finite.

Intuitively, $\llbracket R \rrbracket\; t$ 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:

$$\begin{aligned}
&[\![R \texttt{ UNION ALL } S]\!] = \lambda\, t\, .\, [\![R]\!]\ t + [\![S]\!]\ t\\
&[\![\texttt{SELECT * FROM } R\ x,\ S\ y]\!] = \lambda\, (t_1, t_2)\, .\, [\![R]\!]\ t_1 \times [\![S]\!]\ t_2\\
&[\![\texttt{SELECT * FROM } R\ x \texttt{ WHERE } b]\!] = \lambda\, t\, .\, [\![R]\!]\ t \times [\![b]\!]\ t\\
&[\![\texttt{SELECT } x.a_1 \texttt{ as } a_2 \texttt{ FROM } R\ x]\!] =\\
&\qquad \lambda\, t_2\, . \sum_{t_1 \in \texttt{Tuple } \sigma_1} ([\![a_2]\!] t_2 = [\![a_1]\!] t_1) \times [\![R]\!]\ t_1\\
&[\![\texttt{SELECT DISTINCT * FROM } R\ x]\!] = \lambda\, t\, .\, \|[\![R]\!]\ t\|
\end{aligned}$$

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. 3.3). 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. 1 shows how we can prove that selections distribute over unions, by reducing it to the distributivity of $\times$ over $+$, while Fig. 2 shows the proof of the equivalence for Q2$\equiv$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 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.

***HoTTSQL* 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:

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

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 **0**, 1 is the unit type **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.[3] As we will show in the rest of this paper, the equivalence proofs retain

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

Rewrite Rule:

$\texttt{SELECT DISTINCT } x.a \texttt{ AS } a_1 \texttt{ FROM } R\ x, R\ y \texttt{ WHERE } x.a = y.a$

$\equiv \texttt{SELECT DISTINCT } x.a \texttt{ AS } a_1 \texttt{ FROM } R\ x$

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:

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

Deductive *HoTT*SQL Proof:

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

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. The proof of equivalence Q2$\equiv$Q3.

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 [52, 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 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 fall under the well-studied category of conjunctive queries, for which equality is decidable (while equality between arbitrary SQL queries is 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.

3. *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.

3.1 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 [18] 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:

```
schema σ(a:int, ??)
```

Relation In *HoTT*SQL a relation is modeled as a function from tuples to homotopy types called HoTT-Relations with type `Tuple` $\sigma \to \mathcal{U}$, as discussed in Sec. 2; we will define homotopy types shortly.

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

We now describe *HoTT*SQL, our frontend language for expressing rewrite rules. Figure 4 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 Sec. 3.1, 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 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 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 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 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

```
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 3. An example *HoTT*SQL program

```
h ∈ Program      ::=  s₁; . . . ; sₙ;
s ∈ Statement    ::=  schema σ(a₁ : τ₁, . . . , aₙ : τₙ, ??)
                  |   table t(σ)
                  |   predicate β(σ₁, . . . , σₙ)
                  |   function f(e₁, . . . , eₙ) : τ
                  |   aggregate α(σ) : τ
                  |   verify q₁ ≡ q₂
a ∈ Attribute
q ∈ Query        ::=  t
                  |   SELECT p q
                  |   FROM q₁ x₁, . . . , qₙ xₙ
                  |   q WHERE p
                  |   q₁ UNION ALL q₂
                  |   q₁ EXCEPT q₂
                  |   DISTINCT q
x ∈ TableAlias
b ∈ Predicate    ::=  e₁ = e₂
                  |   NOT b | b₁ AND b₂ | b₁ OR b₂
                  |   TRUE | FALSE
                  |   β(x₁, . . . , xₙ)
                  |   EXISTS q
e ∈ Expression   ::=  x.a | f(e₁, . . . , eₙ) | α(q)
p ∈ Projection   ::=  * | x.* | e AS a | p₁, p₂
```

Figure 4. Syntax of *HoTT*SQL

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.

3.3 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.

Definition 3.1. 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 [52], 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** [52, 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 `Tuple` (node σ_{R_1} σ_{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 **0** or **1** [52, Ch 1.11]).

Squash $(\|n\|)$ Squash is a type-theoretic operation that truncates a univalent type to **0** or **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 $t \in$ `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:

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

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 [3] and Nelson-Oppen algorithm on congruence closure [45]. Both mitigate our proof burden.

3.4 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 [6]. 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
                rewrites to ⇓
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 Sec. 5.1.2.

Integrity Constraints Integrity constraints are used in database systems and facilitate various semantics-based query optimizations [20]. *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);
                rewrites to ⇓
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, $(t_1.a = t_2.a) \to (t_1.b = t_2.b)$ 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;
                rewrites to ⇓
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 [23, 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 [55]. 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:

```
index i(a, R);
rewrites to ⇓
i := SELECT x.k AS k, x.a AS a FROM R x
```

Here := means by definition (rather than proved). In Section 5.1.4, we show example rewrite rules that utilize indexes that are commonly used in query optimizers.

3.5 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.

4. Translating *HoTT*SQL to UNINOMIAL

We translate *HoTT*SQL to UNINOMIAL by first compiling *HoTT*SQL to an intermediate language *HoTT*IR. In

$\tau \in$ `Type`	$::=$	`int` | `bool` | `string` | ...
$[\![$`int`$]\!]$	$::=$	$\mathbb{Z}$
$[\![$`bool`$]\!]$	$::=$	$\mathbb{B}$
...		
$\sigma \in$ `Schema`	$::=$	`empty`
	|	`leaf` τ
	|	`node` σ_1 σ_2
`Tuple empty`	$::=$	`Unit`
`Tuple` (`node` σ_1 σ_2)	$::=$	`Tuple` $\sigma_1 \times$ `Tuple` σ_2
`Tuple` (`leaf` τ)	$::=$	$[\![\tau]\!]$
`Attribute` σ τ	$::=$	`Tuple` $\sigma \to [\![\tau]\!]$

Figure 5. *HoTT*IR's implementation of *HoTT*SQL's Data Model

`schema` σ`(a:int, ??);`	$\Rightarrow$	σ `: Schema.` `a: Attribute` σ τ.

schema of
`SELECT "Bob" AS Name,`
`52 AS Age, TRUE AS Married`
	$\Rightarrow$	`node (leaf int)`
		`(node (leaf int)`
		`(leaf bool))`

schema of
`SELECT * FROM t1 x, t2 y`
`(table t1` σ_1`; table t2` σ_2`;)`
	$\Rightarrow$	`node` σ_1 σ_2

Figure 6. Examples of *HoTT*IR's implementation of schemas

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.

4.1 Implementing Data Model in *HoTT*IR

Figure 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 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 `Tuple`$(s_1) \times$ `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 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 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., $\mathsf{append}(\sigma_1, \sigma_2)$, and every tuple of t has type (`Tuple` $\mathsf{append}(\sigma_1, \sigma_2)$). However, this cannot be further simplified computationally, because the definition of *append* gets stuck on the generic schema σ_1.

4.2 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 [1], similar to De Bruijn indexes [5]. Doing so decouples the equivalence proof from naming resolution, and allows schema equivalences to be determined based on structural 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 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 Sec. 4.1). 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 7 is (`node (node empty` σ_{R_1}`)` σ_{R_2} . Thus, as x.`a` in the selection predicate refers to the schema of R_1, it is converted to the unnamed version as `left.right.a`.

```
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 7. Translating *HoTT*SQL query with correlated subqueries to *HoTT*IR

Figure 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 of the full version of this paper [16].

4.3 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 Tuple Γ) to a HoTT-Relation (of type 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 3.3. For example:

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

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 node Γ σ, 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 3.3.

Predicates A predicate b is denoted to a function from a tuple (of type Tuple Γ) 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* [52, Ch. 3.3]. A squash type can only be a type of 1 element, namely **1**, and a type of 0 element, namely **0**. A *HoTT*SQL program with the form q 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 Tuple Γ) 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 Tuple Γ to a tuple of type Tuple Γ'.

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

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 Tuple Γ to a tuple of type Tuple Γ'' 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 Tuple $\Gamma \to$ Tuple Γ', 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.

5. COSETTE: A Tool for Checking Rewrite Rules

To demonstrate the effectiveness of *HoTT*SQL, we implemented it using Coq as part of COSETTE, a tool for checking the equivalence of SQL rewrite rules. The *HoTT*SQL component of COSETTE consists of five parts, 1) a parser that parses *HoTT*SQL programs and translates them to

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

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

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

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

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

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

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

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

Figure 8. Selected rules of the denotational semantics of *HoTT*SQL, more rules are shown in the appendix of the full version [16].

*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 part of COSETTE relies on the Homotopy Type Theory Coq library [29]. 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 COSETTE's Coq library, and then explain our automated decision procedure.

5.1 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 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 5.1.1 shows how two basic rewrite rules are proved. Sec 5.1.2 shows how to prove a rewrite rule involving aggregation. Sec 5.1.3 shows how to prove the magic set rewrite rules, and Sec 5.1.4 shows how to state a rewrite rule involving indexes.

5.1.1 Basic Rewrite Rules

Basic rewrites are simple rewrite rules that are fundamental building blocks of query optimizers in relational database

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 9. Rewrite rules proved

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 [23]. 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:

$$\begin{aligned}&\lambda\ g\ t.\ [\![\beta_1]\!]\ (g,t) \times [\![\beta_2]\!]\ (g,t) \times [\![R]\!]\ g\ t \quad \equiv\\ &\lambda\ g\ t.\ [\![\beta_2]\!]\ (g,t) \times ([\![\beta_1]\!]\ (g,t) \times [\![R]\!]\ g\ t)\end{aligned}$$

The proof proceeds by functional extensionality,[4] 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 full version of the paper available on project website.

5.1.2 Aggregation and Group By Rewrite Rules

Aggregation and Group By are widely used in analytic queries [9]. The standard data analytic benchmark TPC-H [54] 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 full version.

```
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
```

5.1.3 Magic Set Rewrite Rules

Magic set rewrites are well-known rewrite rules that were originally used in processing recursive queries in deductive databases [2, 47]. It was then used for rewriting complex decision support queries and has been implemented in commercial systems such as IBM's DB2 database [42, 50]. As described in [50], 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:

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

The proof uses Lemma 5.1 provided by the COSETTE library.

Lemma 5.1. *$\forall P, T : \mathcal{U}$, 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.

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

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 $R_1$ $x$, $R_2$ $y$ WHERE $\theta_1(x,y)$)
SEMIJOIN $R_3$ ON $\theta_2$  ≡
(SELECT *
 FROM $R_1$ $x$,
      ($R_2$ SEMIJOIN (FROM $R_1$, $R_3$) ON $\theta_1$ AND $\theta_2$) $y$
 WHERE $\theta_1(x,y)$) SEMIJOIN $R_3$ ON $\theta_2$
```

The rule is denoted to:

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

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 5.1. 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 full version):

```
(SELECT $x.c_1$ AS $c_1$, $\alpha(a)$ FROM $R_1$ $x$ GROUP BY $x.c_1$)
SEMIJOIN $R_2$ ON $\theta$  ≡
SELECT $x.c_1$, $\alpha(a)$
FROM ($R_1$ SEMIJOIN $R_2$ ON $\theta$) $x$ GROUP BY $x.c_1$
```

5.1.4 Index Rewrite Rules

As introduced in Section 3.4, 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.

5.2 Automated Decision Procedure for Conjunctive Queries

The equivalence of two SQL queries is in general undecidable [53]. 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 5.1.2. 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:

```
DISTINCT SELECT $x.c_1$ AS $c_1$ FROM $R_1$ $x$, $R_2$ $y$
WHERE $x.c_2$ = $y.c_3$  ≡
DISTINCT SELECT $x.c_1$ AS $c_1$ FROM $R_1$ $x$, $R_1$ $y$, $R_2$ $z$
WHERE $x.c_1$ = $y.c_1$ AND $x.c_2$ = $z.c_3$
```

which is denoted as:

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

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.

6. Related Work

6.1 Query Rewriting

Query rewriting based on equivalence rules is an essential part of modern query optimizers. Rewrite rules are either fired by a forward chaining rule engine [31, 46], or are used as basis of search space [25–27].

In the implementation of Cosette, we formally prove a series of rewrite rules from the database literature. Those rules include basic algebraic rewrites such as selection push down [56], rewrite rules using indexes [23], and unnesting aggregates with joins [43]. Using Cosette, we proved one of the most complicated rewrite rules that are also widely used in practice: magic set rewrites [2, 42, 50]. Magic set rewrites involve many SQL features such as correlated subqueries, aggregation, and grouping. To our best knowledge, its correctness has not been formally proven before.

COSETTE automates proving rewrite rules on the decidable fragments of SQL. According to Codd's theorem [19], relational algebra and relational calculus (formulas of first-order logic on database instances) are equivalent in expressive power. Thus, the equivalence between two SQL queries is in general undecidable [53]. Extensive research has been done to study the complexity of containment and equivalence of fragments of SQL queries under bag semantics and set semantics [8, 11, 24, 33, 35, 48, 58].

6.2 SQL Semantics

SQL is the de-facto language for relational database systems. Although the SQL language is an ANSI/ISO standard [34], it is loosely described in English and leads to conflicting interpretations [21]. Previous related formalizations of various fragments of SQL include relational algebra [1], comprehension syntax [6], and recursive and non-recursive Datalog [11]. These formalisms are not suited for rigorous reasoning about the correctness of real world rewrite rules since they mostly focus exclusively on set semantics. In addition, to express rewrite rules in these formalism, non-trivial transformation from SQL are required.

There are a number of prior approaches in formalizing SQL in proof systems [4, 13, 14, 40, 41, 59, 60]. In Qex [59, 60], SQL semantics are encoded in the Z3 SMT solver for test generation. In Ynot RDBMS [41], an end to end verified prototype database system is implemented in Coq. In Datacert [4], a relational data model and relational algebra are implemented in Coq. Different from Datacert, our semantics uses bag semantics and denotes all major features of SQL, many of which, such as grouping, aggregation, and correlated subqueries are not supported in Datacert. The COKO-KOLA system [13, 14] can express query rewrite rules and prove them using the Larch proof system [30]. COKO-KOLA relies on a trusted collection of domain specific axioms for proving rewrite rules. The consistency of these axioms is not proven, and incorrect rewrite rules may thus be provable. Modern theorem provers like Coq avoids this problem by satisfying the De Bruijn criterion: the theorem prover is built on a small generic trusted core. Besides, the COKO-KOLA papers are sparse on details, and the front-end that handles non-trivial transformations from SQL to their combinator-based backend language is not available online. We thus were not able to perform an in-depth comparison between COKO-KOLA and *HoTT*SQL. Compared with [4, 13, 14, 40, 41], *HoTT*SQL covers all important SQL feature such as bags, aggregation, group by, indexes, and correlated subqueries. As a result, we are able to express a wide range of rewrite rules. Unlike Ynot [41], we did not build an end to end formally verified database system. And similar to prior work [40], we also use Coq's tactics to automate the proving of conjunctive queries.

In *HoTT*SQL, SQL features such as aggregation after grouping and indexes are supported through syntactic rewrites. Rewriting aggregation after grouping using correlated subqueries is based on [6]. Finally, using logical relationals to represent indexes was first proposed by Tastalos et al [55].

6.3 Related Formal Semantics in Proof Systems

A number of formal semantics in different application domains have been developed using proof systems for software verification. The CompCert compiler [38] specifies the semantics of a subset of C in Coq and provides machine checkable proofs of the correctness of compilation. HALO denotes Haskell to first-order logic for static verification of contracts [61]. Bagpipe [62] developed formal semantics for the Border Gateway Protocol (BGP) to check the correctness of BGP configurations. SEL4 [37] formally specifies the functional correctness of an OS kernel in Isabelle/HOL and developed a verified OS kernel. FSCQ [12] builds a crash safe file system using an encoding of crash Hoare logic in Coq. With the formal semantics implemented using proof systems, a number of verified systems have been developed, such as Verdi [63], Bedrock [15], and Ironclad [32].

7. Conclusion

In this paper, we described COSETTE, a tool for checking the equivalence of SQL rewrite rules. To support COSETTE, 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). Using COSETTE, we have demonstrated the power and flexibility of COSETTE by proving the correctness of several powerful optimization rules found in the database literature, with some of them not proven correct before.

8. Acknowledgments

We thank our shepherd Steve Zdancewic and the anonymous reviewers for their helpful feedback on improving the paper. This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1651489, III-1614738, AITF-1535565, and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; and gifts from Adobe, Amazon, and Google.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL http://webdam.inria.fr/Alice/.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15, 1986.

[3] B. Barras, B. Grégoire, A. Mahboubi, and L. Théry. Coq reference manual chapter 25: The ring and field tactic families. `https://coq.inria.fr/refman/Reference-Manual028.html`.

[4] V. Benzaken, E. Contejean, and S. Dumbrava. A coq formalization of the relational data model. In *ESOP*, pages 189–208, 2014.

[5] D. Bruijn and N. 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.

[6] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[7] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[8] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90. ACM, 1977.

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

[10] S. Chaudhuri and M. Y. Vardi. Optimization of *Real* conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*, pages 59–70, 1993. doi: 10.1145/153850.153856. URL `http://doi.acm.org/10.1145/153850.153856`.

[11] S. Chaudhuri and M. Y. Vardi. On the complexity of equivalence between recursive and nonrecursive datalog programs. In *PODS*, pages 107–116. ACM Press, 1994.

[12] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *SOSP*, pages 18–37, 2015.

[13] M. Cherniack and S. B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *SIGMOD Conference*, pages 401–412. ACM Press, 1996.

[14] M. Cherniack and S. B. Zdonik. Changing the rules: Transformations for rule-based optimizers. In *SIGMOD Conference*, pages 61–72. ACM Press, 1998.

[15] A. Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *ICFP*, pages 391–402. ACM, 2013.

[16] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. *CoRR*, abs/1607.04822, 2016. URL `http://arxiv.org/abs/1607.04822`.

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

[18] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[19] E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[20] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341. Morgan Kaufmann, 1996.

[21] C. J. Date. *A Guide to the SQL Standard, Second Edition*. Addison-Wesley, 1989. ISBN 978-0-201-50209-1.

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

[23] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009. ISBN 978-0-13-187325-4.

[24] G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Parallel-correctness and containment for conjunctive queries with union and negation. In *ICDT*, volume 48 of *LIPIcs*, pages 9:1–9:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

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

[26] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *SIGMOD Conference*, pages 160–172. ACM Press, 1987.

[27] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218. IEEE Computer Society, 1993.

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

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

[30] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993. ISBN 0-387-94006-5.

[31] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *SIGMOD Conference*, pages 377–388. ACM Press, 1989.

[32] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, pages 165–181. USENIX Association, 2014.

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

[34] ISO/IEC. Iso/iec 9075-1:2011. `https://www.iso.org/obp/ui/#iso:std:iso-iec:9075:-1:ed-4:v1:en`. Online; accessed 9-May-2016.

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

[36] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28, 2016.

[37] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.

[38] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[39] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *VLDB*, pages 96–107. Morgan Kaufmann, 1994.

[40] G. Malecha and R. Wisnesky. Using dependent types and tactics to enable semantic optimization of language-integrated queries. In *DBPL*, pages 49–58. ACM, 2015.

[41] J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *POPL*, pages 237–248, 2010.

[42] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *SIGMOD Conference*, pages 247–258, 1990.

[43] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *VLDB*, pages 91–102. Morgan Kaufmann, 1992.

[44] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3):513–534, 1991.

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

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

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

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

[49] D. Schmitt. Bug #5673: Optimizer creates strange execution plan leading to wrong results. `https://www.postgresql.org/message-id/201009231503.o8NF3Blt059661@wwwmaster.postgresql.org`. Online; accessed 1-July-2016.

[50] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD Conference*, pages 435–446, 1996.

[51] M. Sulik. Bug #70038: Wrong select count distinct with a field included in two-column unique key. `http://bugs.mysql.com/bug.php?id=70038`. Online; accessed 1-July-2016.

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

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

[54] 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`.

[55] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. In *VLDB*, pages 367–378. Morgan Kaufmann, 1994.

[56] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[57] J. D. Ullman. Information integration using logical views. In *ICDT*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 1997.

[58] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *PODS*, pages 331–345, 1992.

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

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

[61] D. Vytiniotis, S. L. P. Jones, K. Claessen, and D. Rosén. HALO: haskell to logic through denotational semantics. In *POPL*, pages 431–442, 2013.

[62] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Bagpipe: Verified BGP configuration checking. Technical Report UW-CSE-16-01-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Jan. 2016.

[63] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368. ACM, 2015.