EPiC Series in Computing

Volume 100, 2024, Pages 445–463

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Verifying SQL Queries using Theories of Tables and Relations

Mudathir Mohamed[1], Andrew Reynolds[1], Cesare Tinelli[1], and Clark Barrett[2]

[1] The University of Iowa
[2] Stanford University

Abstract

We present a number of first- and second-order extensions to SMT theories specifically aimed at representing and analyzing SQL queries with join, projection, and selection operations. We support reasoning about SQL queries with either bag or set semantics for database tables. We provide the former via an extension of a theory of finite bags and the latter via an extension of the theory of finite relations. Furthermore, we add the ability to reason about tables with null values by introducing a theory of nullable sorts based on an extension of the theory of algebraic datatypes. We implemented solvers for these theories in the SMT solver cvc5 and evaluated them on a set of benchmarks derived from public sets of SQL equivalence problems.

1 Introduction

The structured query language (SQL) is the dominant declarative query language in relational databases. Two queries are equivalent in SQL if and only if they return the same table for every database instance of the same schema. Query equivalence problems are undecidable in general [1]. For *conjunctive* queries, the problem is NP-complete under set semantics [5] and $\prod_2^p$-hard under bag semantics [6]. SQL query equivalence problems have many applications in databases and software development, including query optimization and sharing sub-queries in cloud databases. There is a financial incentive to reduce the cost of these subqueries, since cloud databases charge for data storage, network usage, and computation.

Recently, these problems got some attention from researchers in formal verification who have developed software tools to prove query equivalence in some SQL fragments. To our knowledge, the state of the art of these tools is currently represented by SQLSolver [9] which supports a large subset of SQL, and SPES, an earlier tool that was used to verify queries from cloud-scale applications [26].

We present an alternative solution for the analysis of SQL queries based on a reduction to constraints in a new SMT theory of tables with bag (i.e., multiset) semantics. This work includes the definition of the theory and the development of a specialized subsolver for it within the cvc5 SMT solver. We have extended a previous theory of finite bags [16] with map and filter operators which are needed to support SQL *select* and *where* clauses, respectively. We represent table rows as tuples and define tables as bags of tuples. We also support the *product*

operator over tables. While multiset semantics captures faithfully the way tables are treated in relational database management systems, there is a lot of work in the database literature that is based on set semantics. We provide set semantics as an alternative encoding based on a theory of finite relations by Meng et al. [17], extended in this case too to accommodate SQL operations.

An initial experimental comparison of our implementation with SQLSolver and SPES places it between the two in terms of performance and supported features. While there are several opportunities for further performance improvements, our solution has two main advantages with respect to previous work: (*i*) it is not limited to SQL equivalence problems, and (*ii*) it comes fully integrated in a state-of-the-art SMT solver with a rich set of background theories. Additionally, it could be further extended to provide support for SQL queries combining set and bag-set semantics [8]. This opens up the door to other kinds of SQL query analyses (including, for instance, query containment and query emptiness problems) over a large set of types for query columns (various types of numerical values, strings, enumerations, and so on).

Specific Contributions We introduce a theory of finite tables by extending a theory of bags with support for product, filter and map operators. We also extend a theory of finite relations with map, filter, and inner join operators. We introduce a theory of nullable sorts as an extension of a theory of algebraic datatypes. These new theories enable the encoding in SMT of a large fragment of SQL under either multiset or set semantics and the automated analysis of problems such as query equivalence. We extend the cvc5 SMT solver [3] with support for quantifier-free constraints over any combination of the theories above and those already defined in cvc5. We discuss an initial experimental evaluation on query equivalence benchmarks.

Our contribution does not include support for aggregations in SQL yet but we plan to add that in future work.

1.1 Related work

A decision procedure for quantifier-free formulas in the theory of bags (QFB), or multisets, based on a reduction to quantifier-free Presburger arithmetic (QFPA) first appeared in Zarba [24]. The theory signature did not include cardinality constraints or difference operators. These are supported in a new decision procedure by Logozzo et al. [16]. The new decision procedure reduces QFB to QFPA*, which extends QFPA with formulas $\bar{u} \in \{\bar{x} \mid \phi\}$, where $\phi \in$ QFPA. Then, the QFPA* formula is translated into a QFPA formula, but with the addition of space overhead [15]. An improved decision procedure which addresses the space overhead issue by using approximations and interpolation with a set of Constrained Horn Clauses was provided in Levatich et al. [15]. Our work is closest to Zarba's [24], with additional support for map and filter operators. As in that work, we do not support the cardinality operator yet.[1] cvc5 already supports the theory of finite sets [2] and its extension to finite relations [17]. We add support for the map and filter operators to cvc5's theory solver for sets and, by extension, to the theory solver for relations, proving the decidability of the satisfiability problem in a restricted fragment of the theory of finite sets. That fragment is enough to handle the benchmarks considered in our experiments under set semantics.

Cosette is an automated tool specifically written to prove SQL query equivalence [7]. To do that, it translates the two SQL queries into algebraic expressions over an unbounded semiring, which it then normalizes to a *sum-product* normal form. Finally, it searches for an isomorphism between the two normal forms using a custom decision procedure. If an isomorphism is

[1] Indirect support for that will be provided in future work through the support for SQL aggregations.

found, the two queries are declared equivalent. To show that queries are inequivalent, Cosette translates the SQL queries into bounded lists and uses an SMT solver to find a counterexample to the equivalence. Cosette is a significant step forward in checking SQL query equivalences. However, it has several limitations. For instance, it does not support null values, intersection, difference, arithmetic operations, or string operations, which are all common in SQL queries.

EQUITAS [25] and its successor SPES [26] are used to identify shared subqueries automatically in cloud databases. Both support aggregate queries and null values. They use symbolic representations to prove query equivalence. EQUITAS follows set semantics, whereas SPES follows bag semantics for tables. EQUITAS starts by assigning symbolic tuples for the input tables in the queries, and then applies specialized algorithms to build two formulas representing output tuples for the two queries. If the two formulas are equivalent, then the two queries are classified as such. SPES also uses symbolic representations for queries. However, it reduces the query equivalence problem to the existence of an identity map between the tuples returned by the two queries [26]. In experimental evaluations [26], SPES proved more queries than EQUITAS and was 3 times as fast. Both tools have the limitation of only supporting queries with similar structure. They do not process queries with structurally different abstract syntax trees (e.g., queries with different number of joins), or queries that use basic operations such as difference or intersection. They also do not support concrete tables, built using the keyword `VALUES`.

SQLSolver, which to our knowledge represents the current state of the art, was released recently and addresses many of the limitations highlighted above [9]. It follows bag semantics and, similar to Cosette, reduces the input queries to unbounded semiring expressions. However, it then translates them into formulas in an extension of QFPA* that supports nested, parametrized, or nonlinear summation. This extension supports projection, product, and aggregate functions. SQLSolver implements algorithms similar to those in Levatich et al. [15] to translate these formulas into QFPA. SQLSolver dominates other tools both in terms of performance and expressiveness of the supported SQL fragment. However, it lacks the ability to generate counterexamples for inequivalent queries [9].

We follow a different approach from all the equivalence checkers discussed above. Our solver, incorporated into cvc5, supports difference and intersection operations, as well as evaluation on concrete tables. Thanks to the rich set of background theories provided by cvc5, it also supports arithmetic and string operations, as well as null values. Finally, our solver is not restricted to SQL query equivalence, as it supports in general any quantifier-free statements over SQL queries. As a consequence, it can also be used for other applications such as, for instance, checking for query containment or emptiness [23].

1.2 Formal Preliminaries

We define our theories and our calculi in the context of many-sorted logic with equality and polymorphic sorts and functions. We assume the reader is familiar with the following notions from that logic: signature, term, formula, free variable, interpretation, and satisfiability of a formula in an interpretation. Let Σ be a many-sorted signature. We will denote sort parameters in polymorphic sorts with α and β, and denote monomorphic sorts with τ. We will use $\approx$ as the (infix) logical symbol for equality — which has polymorphic rank $\alpha \times \alpha$ and is always interpreted as the identity relation over α. We assume all signatures Σ contain the Boolean sort Bool, always interpreted as the binary set $\{\mathit{true}, \mathit{false}\}$, and two Boolean constant symbols, $\top$ and $\bot$, for *true* and *false*. Without loss of generality, we assume $\approx$ is the only predicate symbol in Σ, as all other predicates can be modeled as functions with return sort Bool. We will

Symbol	Type	SMT-LIB syntax	Description
n	Int	n	All constants $\mathtt{n} \in \mathbb{N}$
$+$	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$	+	Integer addition
$*$	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$	*	Integer multiplication
$-$	$\mathsf{Int} \to \mathsf{Int}$	-	Unary Integer minus
$\leq$	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$	<=	Integer inequality
$\varnothing_\alpha$	$\mathsf{Bag}(\alpha)$	bag.empty	Empty bag
bag	$\alpha \times \mathsf{Int} \to \mathsf{Bag}(\alpha)$	bag	Bag constructor
m	$\alpha \times \mathsf{Bag}(\alpha) \to \mathsf{Int}$	bag.count	Multiplicity
setof	$\mathsf{Bag}(\alpha) \to \mathsf{Bag}(\alpha)$	bag.setof	Duplicate remove
$\sqcup$	$\mathsf{Bag}(\alpha) \times \mathsf{Bag}(\alpha) \to \mathsf{Bag}(\alpha)$	bag.union_max	Max union
$\uplus$	$\mathsf{Bag}(\alpha) \times \mathsf{Bag}(\alpha) \to \mathsf{Bag}(\alpha)$	bag.union_disjoint	Disjoint union
$\sqcap$	$\mathsf{Bag}(\alpha) \times \mathsf{Bag}(\alpha) \to \mathsf{Bag}(\alpha)$	bag.inter_min	Intersection
$\setminus$	$\mathsf{Bag}(\alpha) \times \mathsf{Bag}(\alpha) \to \mathsf{Bag}(\alpha)$	bag.diff_subtract	Difference subtract
$\setminus\!\!\setminus$	$\mathsf{Bag}(\alpha) \times \mathsf{Bag}(\alpha) \to \mathsf{Bag}(\alpha)$	bag.diff_remove	Difference remove
$\in$	$\alpha \times \mathsf{Bag}(\alpha) \to \mathsf{Bool}$	bag.member	Member
$\sqsubseteq$	$\mathsf{Bag}(\alpha) \times \mathsf{Bag}(\alpha) \to \mathsf{Bool}$	bag.subbag	Subbag
σ	$(\alpha \to \mathsf{Bool}) \times \mathsf{Bag}(\alpha) \to \mathsf{Bag}(\alpha)$	bag.filter	Bag filter
π	$(\alpha_1 \to \alpha_2) \times \mathsf{Bag}(\alpha_1) \to \mathsf{Bag}(\alpha_2)$	bag.map	Bag map
$\langle \ldots \rangle$	$\alpha_0 \times \cdots \times \alpha_k \to \mathsf{Tuple}(\alpha_0, \ldots, \alpha_k)$	tuple	Tuple constructor
select_i	$\mathsf{Tuple}(\alpha_0, \ldots, \alpha_k) \to \alpha_i$	(_ tuple.select i)	Tuple selector
$\mathsf{tuple.proj}_{i_1 \ldots i_n}$	$\mathsf{Tuple}(\alpha_0, \ldots, \alpha_k) \to \mathsf{Tuple}(\alpha_{i_1}, \ldots, \alpha_{i_n})$	(_ tuple.proj $i_1 \cdots i_n$)	Tuple projection
$\otimes$	$\mathsf{Table}(\boldsymbol{\alpha}) \times \mathsf{Table}(\boldsymbol{\beta}) \to \mathsf{Table}(\boldsymbol{\alpha}, \boldsymbol{\beta})$	table.product	Table cross join
$\bowtie_{i_1 j_1 \cdots i_p j_p}$	$\mathsf{Table}(\boldsymbol{\alpha}) \times \mathsf{Table}(\boldsymbol{\beta}) \to \mathsf{Table}(\boldsymbol{\alpha}, \boldsymbol{\beta})$	(_ table.join $i_1 j_1 \cdots i_p j_p$)	Table inner join
$\mathsf{table.proj}_{i_1 \ldots i_n}$	$\mathsf{Table}(\alpha_0, \ldots, \alpha_k) \to \mathsf{Table}(\alpha_{i_1}, \ldots, \alpha_{i_n})$	(_ table.proj $i_1 \cdots i_n$)	Table projection

Figure 1: Signature Σ_{Tab} for the theory of tables. Here $\mathsf{Table}(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is a shorthand for $\mathsf{Table}(\alpha_0, \ldots, \alpha_p, \beta_0, \ldots, \beta_q)$ when $\boldsymbol{\alpha} = \alpha_0, \ldots, \alpha_p$ and $\boldsymbol{\beta} = \beta_0, \ldots, \beta_q$.

write, e.g., $p(x)$ as shorthand for $p(x) \approx \top$, where $p(x)$ has sort Bool. We write $s \not\approx t$ as an abbreviation for $\neg\, s \approx t$.

A Σ-*term* is a well-sorted term, all of whose function symbols are from Σ. A Σ-*formula* is defined analogously. If φ is a Σ-formula and $\mathcal{I}$ a Σ-interpretation, we write $\mathcal{I} \models \varphi$ if $\mathcal{I}$ satisfies φ. If t is a term, we denote by $\mathcal{I}(t)$ the value of t in $\mathcal{I}$. A *theory* is a pair $\mathbb{T} = (\Sigma, \mathbf{I})$, where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in $\mathbf{I}$ only in how it interprets the variables is also in $\mathbf{I}$). $\mathbf{I}$ is also referred to as the *models* of $\mathbb{T}$. A Σ-formula φ is *satisfiable* (resp., *unsatisfiable*) in $\mathbb{T}$ if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A set Γ of Σ-formulas *entails* in $\mathbb{T}$ a Σ-formula φ, written $\Gamma \models_{\mathbb{T}} \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. We write $\Gamma \models \varphi$ to denote that Γ entails φ in the class of all Σ-interpretations. Two Σ-formulas are *equisatisfiable in* $\mathbb{T}$ if for every interpretation $\mathcal{A}$ of $\mathbb{T}$ that satisfies one, there is an interpretation of $\mathbb{T}$ that satisfies the other and differs from $\mathcal{A}$ at most in how it interprets the free variables not shared by the two formulas.

2 Theory of Tables

We define a many-sorted theory $\mathbb{T}_{\mathsf{Tab}}$ of (database) tables. Its signature Σ_{Tab} is given in Figure 1. We use α and β, possibly with subscripts, as sort parameters in polymorphic sorts. The theory includes the integer sort Int and a number of integer operators, with the same interpretation as in the theory of arithmetic. Additionally, $\mathbb{T}_{\mathsf{Tab}}$ has three classes of sorts, with a corresponding polymorphic sort constructor: function sorts, tuple sorts, and bag sorts. *Function sorts* are monomorphic instances of $\alpha_1 \times \cdots \times \alpha_k \to \alpha$ for all $k \geq 0$. *Tuple sorts* are constructed by

the varyadic constructor Tuple which takes zero or more sort arguments. With no arguments, Tuple denotes the singleton set containing the empty tuple. With $k+1$ arguments for $k \geq 0$, $\mathsf{Tuple}(\tau_0, \ldots, \tau_k)$ denotes the set of tuples of size $k+1$ with elements of sort $\tau_0, \ldots, \tau_k$, respectively. *Bag sorts* are monomorphic instances $\mathsf{Bag}(\tau)$ of $\mathsf{Bag}(\alpha)$. The sort $\mathsf{Bag}(\tau)$ denotes the set of all *finite* bags (i.e., finite multisets) of elements of sort τ. We model tables as bags of tuples. We write $\mathsf{Table}(\tau_0, \ldots, \tau_k)$ as shorthand for the sort $\mathsf{Bag}(\mathsf{Tuple}(\tau_0, \ldots, \tau_k))$. The sort Table, with no arguments, abbreviates $\mathsf{Bag}(\mathsf{Tuple})$.[2] Following databases terminology, we refer to $\tau_0, \ldots, \tau_k$ as the *columns* of $\mathsf{Table}(\tau_0, \ldots, \tau_k)$, and to the elements of a given table as its *rows*. For convenience, we index the columns of a table by natural numbers (starting with 0), instead of alphanumeric names, as in SQL.

The symbols in the first five lines in Figure 1 are from arithmetic and are interpreted as expected. The next eleven function symbols operate on bags and are defined as in Logozzo et al. [16]. Specifically, for all sorts τ, $\varnothing_\tau$ is interpreted as the empty bag of elements of sort τ. The term $\mathsf{bag}(e, n)$ denotes a singleton bag with n occurrences of the bag element e if $n \geq 1$; otherwise; it denotes $\varnothing_\tau$ where τ is the sort of e. The term $\mathsf{m}(e, s)$ denotes the *multiplicity* of element e in bag s, that is, the number of times e occurs in s. Its codomain is the set of natural numbers. For convenience, we use Int as the codomain and, during reasoning, assert $\mathsf{m}(e, s) \geq 0$ for each multiplicity term. The term $\mathsf{setof}(s)$ denotes the bag with the same elements as s but without duplicates. The predicate $e \sqsubseteq s$ holds iff element e has positive multiplicity in bag s. The predicate $s \sqsubseteq t$ holds iff bag s is contained in bag t in the sense that $\mathsf{m}(e, s) \leq \mathsf{m}(e, t)$ for all elements e. The binary operators $\sqcup, \uplus, \sqcap, \setminus, \mathbin{\backslash\!\backslash}$ are interpreted as functions that take two bags s and t and return their max union, disjoint union, subtract difference, and remove difference, respectively, making the following equalities valid in $\mathbb{T}_{\mathsf{Tab}}$:

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{m}(e, s \sqcup t) & \approx & \max(\mathsf{m}(e, s), \mathsf{m}(e, t)) & \mathsf{m}(e, s \uplus t) & \approx & \mathsf{m}(e, s) + \mathsf{m}(e, t) \\ \mathsf{m}(e, s \sqcap t) & \approx & \min(\mathsf{m}(e, s), \mathsf{m}(e, t)) & \mathsf{m}(e, s \setminus t) & \approx & \max(\mathsf{m}(e, s) - \mathsf{m}(e, t), 0) \\ \mathsf{m}(e, s \mathbin{\backslash\!\backslash} t) & \approx & \mathsf{ite}(\mathsf{m}(e, t) \geq 1, 0, \mathsf{m}(e, s)) & & & \end{array}$$

The next two symbols in Figure 1 are the filter and map functionals. These symbols require an SMT solver that supports higher-order logic, which is the case for cvc5 [4]. The term $\sigma(p, s)$ denotes the bag consisting of the elements of bag s that satisfy predicate p, with the same multiplicity they have in s. The term $\pi(f, s)$ denotes the bag consisting of the elements $f(e)$, for all e that occur in s. The multiplicity of $f(e)$ in $\sigma(f, s)$ is the sum of the multiplicities (in s) of all the elements of s that f maps to $f(e)$. Note that while $\pi(f, s)$ and s have the same cardinality, i.e., the same number of element occurrences, $\mathsf{m}(f(e), \pi(f, s))$ may be greater than $\mathsf{m}(e, s)$ for some elements e unless f is injective.

The last six symbols denote dependent families of functions over tuples and tables. The term $\langle e_1, \ldots, e_k \rangle$ is interpreted as the tuple comprised of the elements $e_1, \ldots, e_k$, in that order, with $\langle\rangle$ denoting the empty tuple. For $0 \leq i \leq k$ where $k+1$ is the size of a tuple t, $\mathsf{select}_i(t)$ is interpreted as the element at position i of t. Note that i in select_i is a numeral, not a symbolic index. This means in particular that $\mathsf{select}_3(\langle a, b \rangle)$ is an ill-sorted term. $\mathsf{tuple.proj}$ takes an unbounded number of integer arguments, followed by a tuple. $\mathsf{tuple.proj}_{i_1 \ldots i_n}$ where $n \geq 1$ and each i_j is an element of $\{0, \ldots, k\}$, applies to any tuple of size at least $k+1$ and returns the tuple obtained by collecting the values at position $i_1, \ldots, i_n$ in t. In other words, it is equivalent to $\langle \mathsf{select}_{i_1}(t), \ldots, \mathsf{select}_{i_n}(t) \rangle$. Note that $i_1, \ldots, i_n$ are not required to be distinct. When $n = 0$, the term is equivalent to the empty tuple. $\mathsf{table.proj}_{i_1 \ldots i_n}$ extends the notion of projection to tables. It is similar to $\mathsf{tuple.proj}$ except that it takes a table instead of a tuple as argument. Note that the cardinality of $\mathsf{table.proj}_{i_1 \ldots i_n}(s)$ is the same as the cardinality of s. The term

[2] And so denotes the set of all tables containing just occurrences of the empty tuple.

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{m}(e, \varnothing_\varepsilon) & \longrightarrow & 0 & s \uplus \varnothing_\varepsilon & \longrightarrow & s \\ e \sqsubset s & \longrightarrow & 1 \leq \mathsf{m}(e, s) & \varnothing_\varepsilon \uplus t & \longrightarrow & t \\ \mathsf{bag}(e, -n) & \longrightarrow & \varnothing_\varepsilon & s \sqsubseteq t & \longrightarrow & (s \setminus t) \approx \varnothing_\varepsilon \end{array}$$

Figure 2: Simplification rules for Σ_{Tab}-terms. In the last two rules, ε is the sort of e and of the elements of s, respectively; n is a numeral.

$t \otimes t'$ is interpreted as the *cross join* of tables t and t', with every tuple occurrence in t being concatenated with every tuple occurrence in t'. The operator $\bowtie_{i_1 j_1 \cdots i_n j_n}$ is indexed by n pairs of natural numbers. It takes two tables as input, each with at least n columns, and outputs the inner join of these tables on the columns specified by these index pairs. The paired columns have to be of the same sort. Notice that if $n = 0$, the join is equivalent to a product.

Simplifying Assumptions *To simplify the exposition* and the description of the calculus, from now on, we will consider only bags whose elements are not themselves bags, and only tuples whose elements are neither tuples nor bags. Note that this *non-nestedness* restriction applies to tables as well — as they are just bags of tuples. This is enough in principle to model and reason about SQL tables.[3] We stress, however, that none of these restrictions are necessary in our approach, nor required by our implementation, where we rely on cvc5's ability to reason modularly about arbitrarily nested sorts. Finally, we will not formalize in the calculus how we process constraints containing table projections (i.e., applications of table.proj). Such constraints are reduced internally to map constraints, with mapping functions generated on the fly, and added to the relevant background solver.

Definition 2.1. A (monomorphic) sort is an *element sort* if it is not an instance of $\mathsf{Bag}(\alpha)$. An *element term* is a term of an *element sort*. A *tuple/bag/table term* is a term of tuple/bag/table sort, respectively. A $\mathbb{T}_{\mathsf{Tab}}$-*atom* is an atomic Σ_{Tab}-formula of the form $t_1 \approx t_2$, $e \sqsubset s$, or $s_1 \sqsubseteq s_2$, where t_1 and t_2 are terms of the same sort, e is a term of some element sort τ and s, s_1, and s_2 are terms of sort $\mathsf{Bag}(\tau)$.

A Σ_{Tab}-formula φ is a *table constraint* if it has the form $s \approx t$ or $s \not\approx t$; it is an *arithmetic constraint* if it has the form $s \approx t$, $s \not\approx t$, or $s \leq t$, where s, t are terms of sort Int; it is an *element constraint* if it has the form $e_1 \approx e_2, e_1 \not\approx e_2, p(e), f(e_1) \approx e_2$, where e, e_1, e_2 are terms of some element sort, p is a function symbol of sort $\varepsilon \to \mathsf{Bool}$ for some element sort ε, and f is a function symbol of sort $\varepsilon_1 \to \varepsilon_2$, for some element sorts ε_1 and ε_2.

Note that table constraints include (dis)equalities between terms of any sort. This implies that (dis)equalities between terms of sort Int are both table and arithmetic constraints.

2.1 Calculus

We now describe a tableaux-style calculus with derivation rules designed to determine the satisfiability in $\mathbb{T}_{\mathsf{Tab}}$ of quantifier-free Σ_{Tab}-formulas φ. To simplify the description, we will pretend that all tables have columns of the same element sort, denoted generically by ε.

Without loss of generality, we assume that the atoms of φ are in reduced form with respect to the (terminating) rewrite system in Figure 2, which means that φ is a Boolean combination

[3] Commercial databases do allow table elements to be tuples. We could easily support this capability in the future simply by providing two kinds of tuple sorts, one for rows and one for table elements.

of only equality constraints and arithmetic constraints. Thanks to the following lemma, we will further focus on just sets of table constraints and arithmetic constraints.[4]

Lemma 2.1. *For every quantifier-free Σ_{Tab}-formula φ, there are sets $B_1, \ldots, B_n$ of table constraints, sets $A_1, \ldots, A_n$ of arithmetic constraints, and sets $E_1, \ldots, E_n$ of elements constraints such that φ is satisfiable in $\mathbb{T}_{\mathsf{Tab}}$ iff $A_i \cup B_i \cup E_i$ is satisfiable in $\mathbb{T}_{\mathsf{Tab}}$ for some $i \in [1, n]$.*

As a final simplification, we can also assume, without loss of generality, that for every term t of sort $\mathsf{Tuple}(\tau_0, \ldots, \tau_k)$ occurring in one of the sets B_i above, B_i also contains the constraint $t \approx \langle x_0, \ldots, x_k \rangle$ where $x_0, \ldots, x_k$ are variables of sort $\tau_0, \ldots, \tau_k$, respectively.

Configurations and Derivation Trees. The calculus operates on data structures we call *configurations*. These are either the distinguished configuration unsat or triples $(\mathsf{A}, \mathsf{B}, \mathsf{E})$ consisting of a set A of arithmetic constraints, a set B of table constraints, and a set E of element constraints. Our calculus is a set of derivation rules that apply to configurations.

We assume we have a (possibly multi-theory) *element solver* that can decide the satisfiability of constraints in E. This requires the computability of all predicates p and functions f used as arguments in applications of filter (σ) and map (π), respectively. We also define the set W to be an infinite set of fresh variables, which will be used in specific derivation rules.

Derivation rules take a configuration and, if applicable to it, generate one or more alternative configurations. A derivation rule *applies* to a configuration c if all the conditions in the rule's premises hold for c *and* the rule application is not redundant. An application of a rule is *redundant* if it has a conclusion where each component in the derived configuration is a subset of the corresponding component in the premise configuration.

A configuration other than unsat is *saturated with respect to a set R* of derivation rules if every possible application of a rule in R to it is redundant. It is *saturated* if it is saturated with respect to all derivation rules in the calculus. A configuration $(\mathsf{A}, \mathsf{B}, \mathsf{E})$ is *satisfiable* in $\mathbb{T}_{\mathsf{Tab}}$ if the set $\mathsf{A} \cup \mathsf{B} \cup \mathsf{E}$ is satisfiable in $\mathbb{T}_{\mathsf{Tab}}$.

A *derivation tree* is a (possibly infinite) tree where each node is a configuration whose (finitely-many) children, if any, are obtained by a non-redundant application of a rule of the calculus to the node. A derivation tree is *closed* if it is finite and all its leaves are unsat. As we show later, a closed derivation tree with root $(\mathsf{A}, \mathsf{B}, \mathsf{E})$ is a proof that $\mathsf{A} \cup \mathsf{B} \cup \mathsf{E}$ is unsatisfiable in $\mathbb{T}_{\mathsf{Tab}}$. In contrast, a derivation tree with root $(\mathsf{A}, \mathsf{B}, \mathsf{E})$ and a saturated leaf with respect to all the rules of the calculus is a witness that $\mathsf{A} \cup \mathsf{B} \cup \mathsf{E}$ is satisfiable in $\mathbb{T}_{\mathsf{Tab}}$.

The Derivation Rules. The rules of our calculus are provided in Figures 3, 4 and 5. They are expressed in *guarded assignment form* where the premise describes the conditions on the current configuration under which the rule can be applied, and the conclusion is either unsat, or otherwise describes changes to the current configuration. Rules with two conclusions, separated by the symbol ||, are non-deterministic branching rules.

In the rules, we write B, c, as an abbreviation of $\mathsf{B} \cup \{c\}$ and denote by $\mathcal{T}(\mathsf{B})$ the set of all terms and subterms occurring in B. Premises of the form $\mathsf{A} \models_{\mathsf{NIA}} c$, where c is an arithmetic constraint, can be checked by a solver for (nonlinear) integer arithmetic.[5] Premises of the form $\mathsf{E} \models_{\mathsf{EL}} \bot$ are checked by the element solver discussed earlier.

[4] Proofs of this lemma and later results can be found in a longer version of this paper [18].

[5] A linear arithmetic solver is enough for problems not containing the $\otimes$ operator. For problems with SQL joins, whose encoding to SMT does require the $\otimes$ operator, a solver for nonlinear arithmetic is needed, at the cost of losing decidability in that case.

$$\textsc{A-Conf}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} \bot}{\mathsf{unsat}} \qquad \textsc{B-Conf}\ \frac{t \not\approx t \in \mathsf{B}^*}{\mathsf{unsat}} \qquad \textsc{E-Conf}\ \frac{\mathsf{E} \models_{\mathsf{EL}} \bot}{\mathsf{unsat}}$$

$$\textsc{B-A-Prop}\ \frac{s \approx t \in \mathsf{B}^* \qquad s,t : \mathsf{Int}}{\mathsf{A} := \mathsf{A}, s \approx t} \qquad \textsc{B-E-Prop}\ \frac{e_1 \approx e_2 \in \mathsf{B}^* \qquad e_1, e_2 \text{ are elem. terms}}{\mathsf{E} := \mathsf{E}, e_1 \approx e_2}$$

$$\textsc{E-Ident}\ \frac{e_1, e_2 \in \mathcal{T}(\mathsf{B}^*) \qquad e_1, e_2 \text{ are element terms of the same sort}}{\mathsf{B} := \mathsf{B}, e_1 \approx e_2 \quad || \quad \mathsf{B} := \mathsf{B}, e_1 \not\approx e_2}$$

$$\textsc{A-Prop}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} s \approx t \qquad s,t \in \mathsf{A} \qquad s \text{ or } t \text{ is a multiplicity term}}{\mathsf{B} := \mathsf{B}, s \approx t}$$

$$\textsc{Diseq}\ \frac{s \not\approx t \in \mathsf{B}^* \qquad w \text{ is a fresh variable}}{\mathsf{B} := \mathsf{B}, \mathsf{m}(w,s) \not\approx \mathsf{m}(w,t) \quad \mathsf{A} := \mathsf{A}, \mathsf{m}(w,s) \not\approx \mathsf{m}(w,t)} \qquad \textsc{Nonneg}\ \frac{\mathsf{m}(e,s) \in \mathcal{T}(\mathsf{B}^*)}{\mathsf{A} := \mathsf{A}, 0 \leq \mathsf{m}(e,s)}$$

$$\textsc{Cons1}\ \frac{s \approx \mathsf{bag}(e,n) \in \mathsf{B}^* \qquad n \leq 0 \notin \mathsf{A} \qquad 1 \leq n \notin \mathsf{A}}{\begin{array}{c}\mathsf{A} := \mathsf{A}, n \leq 0, \mathsf{m}(e,s) \approx 0 \quad \mathsf{B} := \mathsf{B}, s \approx \varnothing_\varepsilon \\ || \quad \mathsf{A} := \mathsf{A}, 1 \leq n, \mathsf{m}(e,s) \approx n \quad \mathsf{B} := \mathsf{B}, s \not\approx \varnothing_\varepsilon\end{array}}$$

$$\textsc{Cons2}\ \frac{s \approx \mathsf{bag}(e,n) \in \mathsf{B}^* \qquad x \not\approx e \in \mathsf{B}^*}{\mathsf{A} := \mathsf{A}, \mathsf{m}(x,s) \approx 0} \qquad \textsc{Empty}\ \frac{s \approx \varnothing_\varepsilon \in \mathsf{B}^* \qquad \mathsf{m}(e,s) \in \mathcal{T}(\mathsf{B})}{\mathsf{A} := \mathsf{A}, \mathsf{m}(e,s) \approx 0}$$

$$\textsc{Disj Union}\ \frac{s \approx t \uplus u \in \mathsf{B}^* \qquad \mathsf{m}(e,v) \in \mathcal{T}(\mathsf{B}) \qquad v \in \{s,t,u\}}{\mathsf{A} := \mathsf{A}, \mathsf{m}(e,s) \approx \mathsf{m}(e,t) + \mathsf{m}(e,u)}$$

$$\textsc{Max Union}\ \frac{s \approx t \sqcup u \in \mathsf{B}^* \qquad \mathsf{m}(e,v) \in \mathcal{T}(\mathsf{B}) \qquad v \in \{s,t,u\}}{\mathsf{A} := \mathsf{A}, \mathsf{m}(e,s) \approx \max(\mathsf{m}(e,t), \mathsf{m}(e,u))}$$

$$\textsc{Inter}\ \frac{s \approx t \sqcap u \in \mathsf{B}^* \qquad \mathsf{m}(e,v) \in \mathcal{T}(\mathsf{B}) \qquad v \in \{s,t,u\}}{\mathsf{A} := \mathsf{A}, \mathsf{m}(e,s) \approx \min(\mathsf{m}(e,t), \mathsf{m}(e,u))}$$

$$\textsc{Diff Sub}\ \frac{s \approx t \setminus u \in \mathsf{B}^* \qquad \mathsf{m}(e,v) \in \mathcal{T}(\mathsf{B}) \qquad v \in \{s,t,u\}}{\begin{array}{c}\mathsf{A} := \mathsf{A}, \mathsf{m}(e,t) \leq \mathsf{m}(e,u), \mathsf{m}(e,s) \approx 0 \\ || \quad \mathsf{A} := \mathsf{A}, \mathsf{m}(e,t) > \mathsf{m}(e,u), \mathsf{m}(e,s) \approx \mathsf{m}(e,t) - \mathsf{m}(e,u)\end{array}}$$

$$\textsc{Diff Rem}\ \frac{s \approx t \setminus\!\setminus u \in \mathsf{B}^* \qquad \mathsf{m}(e,v) \in \mathcal{T}(\mathsf{B}) \qquad v \in \{s,t,u\}}{\mathsf{A} := \mathsf{A}, \mathsf{m}(e,u) \approx 0, \mathsf{m}(e,s) \approx \mathsf{m}(e,t) \quad || \quad \mathsf{A} := \mathsf{A}, \mathsf{m}(e,u) \not\approx 0, \mathsf{m}(e,s) \approx 0}$$

$$\textsc{Setof}\ \frac{s \approx \mathsf{setof}(t) \in \mathsf{B}^* \qquad \mathsf{m}(e,v) \in \mathcal{T}(\mathsf{B}^*) \qquad v \in \{s,t\}}{\mathsf{A} := \mathsf{A}, 1 \leq \mathsf{m}(e,t), \mathsf{m}(e,s) \approx 1 \quad || \quad \mathsf{A} := \mathsf{A}, \mathsf{m}(e,t) \leq 0, \mathsf{m}(e,s) \approx 0}$$

Figure 3: Bag rules.

$$\textsc{Prod Up}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(\langle \bar{x}_m \rangle, s) \wedge 1 \le \mathsf{m}(\langle \bar{y}_n \rangle, t) \qquad s \otimes t \in \mathcal{T}(\mathsf{B})}{\mathsf{A} := \mathsf{A}, \mathsf{m}(\langle \bar{x}_m, \bar{y}_n \rangle, s \otimes t) \approx \mathsf{m}(\langle \bar{x}_m \rangle, s) * \mathsf{m}(\langle \bar{y}_n \rangle, t)}$$

$$\textsc{Prod Down}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(\langle \bar{x}_m, \bar{y}_n \rangle, s \otimes t)}{\mathsf{A} := \mathsf{A}, \mathsf{m}(\langle \bar{x}_m, \bar{y}_n \rangle, s \otimes t) \approx \mathsf{m}(\langle \bar{x}_m \rangle, s) * \mathsf{m}(\langle \bar{y}_n \rangle, t)}$$

$$\textsc{Join Up}\ \frac{\begin{array}{c}\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(\langle \bar{x}_m \rangle, s) \wedge 1 \le \mathsf{m}(\langle \bar{y}_n \rangle, t) \\ s \bowtie_{i_1 j_1 \cdots i_p j_p} t \in \mathcal{T}(\mathsf{B}) \quad x_{i_1} \approx y_{j_1}, \ldots, x_{i_p} \approx y_{j_p} \in \mathsf{B}^*\end{array}}{\mathsf{A} := \mathsf{A}, \mathsf{m}(\langle \bar{x}_m, \bar{y}_n \rangle, s \bowtie_{i_1 j_1 \cdots i_p j_p} t) \approx \mathsf{m}(\langle \bar{x}_m \rangle, s) * \mathsf{m}(\langle \bar{y}_n \rangle, t)}$$

$$\textsc{Join Down}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(\langle \bar{x}_m, \bar{y}_n \rangle, s \bowtie_{i_1 j_1 \cdots i_p j_p} t)}{\begin{array}{c}\mathsf{A} := \mathsf{A}, \mathsf{m}(\langle \bar{x}_m, \bar{y}_n \rangle, s \bowtie_{i_1 j_1 \cdots i_p j_p} t) \approx \mathsf{m}(\langle \bar{x}_m \rangle, s) * \mathsf{m}(\langle \bar{y}_n \rangle, t) \\ \mathsf{B} := \mathsf{B}, x_{i_1} \approx y_{j_1}, \ldots, x_{i_p} \approx y_{j_p}\end{array}}$$

Figure 4: Table rules. The syntax $\bar{x}_m$ abbreviates $x_0, \ldots, x_m$.

We define the following closure for B where $\models_{\text{tup}}$ denotes entailment in the theory of tuples, which treats all other symbols as uninterpreted functions.

$$\begin{aligned}\mathsf{B}^* = \ & \{s \approx t \mid s,t \in \mathcal{T}(\mathsf{B}), \mathsf{B} \models_{\text{tup}} s \approx t\} \cup \{m(e,s) \approx m(e,t) \mid \mathsf{B} \models_{\text{tup}} s \approx t,\ m(e,s) \in \mathcal{T}(\mathsf{B})\} \\ & \cup \{m(e_1,s) \approx m(e_2,s) \mid \mathsf{B} \models_{\text{tup}} e_1 \approx e_2,\ m(e_1,s) \in \mathcal{T}(\mathsf{B})\} \\ & \cup \{s \not\approx t \mid s,t \in \mathcal{T}(\mathsf{B}), \mathsf{B} \models_{\text{tup}} s \approx s' \wedge t \approx t' \text{ for some } s' \not\approx t' \in \mathsf{B}\} \end{aligned} \tag{1}$$

The set B^* is computable by extending standard congruence closure procedures with rules for adding equalities of the form $\mathsf{select}_i(\langle x_0, \ldots, x_i, \ldots, x_k \rangle) \approx x_i$ and rules for deducing consequences of equalities of the form $\langle s_1, \ldots, s_n \rangle \approx \langle t_1, \ldots, t_n \rangle$.

Among the derivation rules in Figure 3, rules A-Conf, E-Conf are applied when conflicts are found by the arithmetic solver or the element solver. Likewise, rule B-Conf is applied when the congruence closure procedure finds a conflict between an equality and a disequality constraint. Rules B-A-Prop, B-E-Prop, and A-Prop communicate equalities between the three solvers. Rule Diseq handles disequality between two bags s, t by stating that some element, represented by a fresh variable w, occurs with different multiplicities in s and t. Rule Nonneg ensures that all multiplicities are nonnegative. Rule Empty enforces zero multiplicity for elements to a bag that is provably equal to the empty bag.

Rules Cons1 and Cons2 reason about singleton bags, denoted by terms of the form $\mathsf{bag}(e, n)$. The first one splits on whether n is positive or not to determine whether $\mathsf{bag}(e, n)$ is empty, and if not, it also determines the multiplicity of element e to be n. The second one ensures that no elements different from e are in $\mathsf{bag}(e, n)$. Rules Disj Union, Max Union, Inter, Diff Sub, Diff Rem, and Setof correspond directly to the semantics of their operators. For example, the Disj Union rule applies to any multiplicity term related to bags $(t, u, t \uplus u)$ or their equivalence classes if $t \uplus u \in \mathcal{T}(B)$.

The rules in Figure 4 are specific to table operations. Prod Up and Prod Down are upward and downward rules for the $\otimes$ operator. They are the ones which introduce nonlinear arithmetic constraints on multiplicities. Join Up and Join Down are similar to the product rules. However, they consider the equality constraints between joining columns, to account for the semantics of inner joins.

$$\textsc{Filter Up}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(e,t) \qquad \mathsf{m}(e,t) \in \mathcal{T}(\mathsf{B}^*) \qquad s \approx \sigma(p,t) \in \mathsf{B}^*}{\mathsf{E} := \mathsf{E}, p(e) \quad \mathsf{A} := \mathsf{A}, \mathsf{m}(e,s) \approx \mathsf{m}(e,t) \;\;||\;\; \mathsf{E} := \mathsf{E}, \neg p(e) \quad \mathsf{A} := \mathsf{A}, \mathsf{m}(e,s) \approx 0}$$

$$\textsc{Filter Down}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(e,s) \qquad \mathsf{m}(e,s) \in \mathcal{T}(\mathsf{B}^*) \qquad s \approx \sigma(p,t) \in \mathsf{B}^*}{\mathsf{E} := \mathsf{E}, p(e) \qquad \mathsf{A} := \mathsf{A}, \mathsf{m}(e,s) \approx \mathsf{m}(e,t)}$$

$$\textsc{Map Up}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(e,t) \qquad \mathsf{m}(e,t) \in \mathcal{T}(\mathsf{B}^*) \qquad s \approx \pi(f,t) \in \mathsf{B}^* \qquad e \notin W}{\mathsf{A} := \mathsf{A}, \mathsf{m}(e,t) \le \mathsf{m}(f(e),s)}$$

$$\textsc{Inj Map Down}\ \frac{\mathsf{m}(e,s) \in \mathcal{T}(\mathsf{B}^*) \qquad s \approx \pi(f,t) \in \mathsf{B}^* \qquad f \text{ is injective}}{\mathsf{E} := \mathsf{E}, f(w) \approx e \qquad \mathsf{A} := \mathsf{A}, \mathsf{m}(e,s) \approx \mathsf{m}(w,t)}$$

$$\textsc{NotInj Up}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(e,t) \qquad s \approx \pi(f,t) \in \mathsf{B}^*}{\mathsf{B} := \mathsf{B}, i \approx \mathit{ind}(e,t) \qquad \mathsf{A} := \mathsf{A}, 1 \le i \le \mathit{delem}(t)}$$

$$\textsc{NotInj Down}\ \frac{\mathsf{A} \models_{\mathsf{NIA}} 1 \le \mathsf{m}(e,s) \quad s \approx \pi(f,t) \in \mathsf{B}^*}{\mathsf{A} := \mathsf{A}, \mathit{sum}(e,t,\mathit{delem}(t)) \approx \mathsf{m}(e,s), \mathit{sum}(e,t,0) \approx 0}$$

Figure 5: Bag filter and map rules. w, i are fresh variables.

The rules in Figure 5 reason about the filter (σ) and map (π) operators. Filter Up splits on whether an element e in s satisfies (the predicate denoted by) p or not in order to determine its multiplicity in bag $\sigma(p,s)$. Filter Down concludes that every element with positive multiplicity in $\sigma(p,s)$ necessarily satisfies p and has the same multiplicity in s. Map Up applies the function symbol f to every element e that is provably in bag t. Note that it cannot determine the exact multiplicity of $f(e)$ in bag $\pi(f,t)$ since multiple elements can be mapped to the same one by f if (the function denoted by) f is not injective. Therefore, the rule just asserts that $\mathsf{m}(f(e),\pi(f,t))$ is at least $\mathsf{m}(e,t)$. To prevent derivation cycles with Inj Map Down, rule Map Up applies only if e is not a variable introduced by the downward rule.

The downward direction for map terms is more complex, and expensive, if f is not injective. Therefore, before solving, we check the injectivity of each function symbol f occurring in map terms. This is done via a subsolver instance that checks the satisfiability of the formula $f(x) \approx f(y) \wedge x \not\approx y$ for fresh variables x, y. If the subsolver returns unsat, which means that f is injective, we apply rule Inj Map Down which introduces a fresh variable w for each term $\mathsf{m}(e,\pi(f,t))$ such that $e = f(w)$ and w is in t with the same multiplicity. In contrast, if the subsolver returns sat or unknown, we treat f as non-injective and rely on a number of features of cvc5 to construct and process a set of quantified constraints which, informally speaking and mixing syntax and semantics here for simplicity, formalize the following relationship between the multiplicity of an element e in a bag $\pi(f,t)$ and that of the elements of t that f maps to e:

$$\mathsf{m}(e,\pi(f,t)) = \sum\{\, \mathsf{m}(x,t) \mid x \in t \wedge f(x) = e \,\} \tag{2}$$

To encode this constraint, we introduce three uninterpreted symbols, $\mathit{delem} : \mathsf{Bag}(\alpha) \to \mathsf{Int}$, $\mathit{ind} : \alpha \times \mathsf{Bag}(\alpha) \to \mathsf{Int}$, and $\mathit{sum} : \alpha \times \mathsf{Bag}(\alpha) \times \mathsf{Int} \to \mathsf{Int}$. The value $\mathit{delem}(t)$ represents the number of distinct elements in (the bag denoted by) t; $\mathit{ind}(e,t)$ represents a unique index in the range $[1, \mathit{delem}(t)]$ for element e in bag t; for an element e in $\pi(f,t)$, $\mathit{sum}(e,t,i)$ accumulates the

multiplicities of the elements of t with index in $[1, i]$ that f maps to e. Rule NotInj Up ensures that every element in t is assigned an index i in $[1, \mathit{delem}(t)]$, whereas NotInj Up constrains the multiplicity $\mathsf{m}(e, s)$ to be $\mathit{sum}(e, t, \mathit{delem}(t))$ when e is in s. We do not describe the encoding of (2) here due to space limitations. However, it is an axiom with bounded quantification over the interval $[1, \mathit{delem}(t)]$ that is processed by cvc5's model-based quantifier instantiation module [19].

Example 2.1. Suppose we have the constraints: $\{\, x \not\sqsubseteq s, y \sqsubset \pi(f, s), y \approx x + 1 \,\}$ where x, y are integers and f is defined in the arithmetic solver to be the integer successor function ($f(x) = x+1$). After applying simplification rules in Figure 2, we end up with $c_0 = (\mathsf{A}_0, \mathsf{B}_0, \mathsf{E}_0)$, where $\mathsf{A}_0 = \{\, \neg(1 \leq \mathsf{m}(x, s)), 1 \leq \mathsf{m}(y, \pi(f, s)), y \approx x + 1 \,\}$ and $\mathsf{E}_0 = \{\, y \approx x + 1 \,\}$. Applying rule Nonneg, we get $c_1 = (\mathsf{A}_1, \mathsf{B}_1, \mathsf{E}_1)$, where $\mathsf{A}_1 = \mathsf{A}_0 \cup \{\, 0 \leq \mathsf{m}(x, s), 0 \leq \mathsf{m}(y, \pi(f, s)) \,\}$ and $\mathsf{E}_1 = \mathsf{E}_0$. Since f is injective, we can apply rule Inj Map Down to get $c_2 = (\mathsf{A}_2, \mathsf{B}_2, \mathsf{E}_2)$, where $\mathsf{A}_2 = \mathsf{A}_1 \cup \{\, \mathsf{m}(w, s) \approx \mathsf{m}(y, \pi(f, s)) \,\}$ and $\mathsf{E}_2 = \mathsf{E}_1 \cup \{\, y \approx w + 1 \,\}$. Next, we apply the propagation rule E-Ident followed by B-A-Prop to get $\mathsf{A}_3 = \mathsf{A}_2$, $\mathsf{A}_4 = \mathsf{A}_3 \cup \{\, y \approx w + 1 \,\}$. Now, A_4 is unsatisfiable because it entails $x \approx w$, $\neg(1 \leq \mathsf{m}(x, s))$, and $1 \leq \mathsf{m}(w, s)$. Hence the rule A-Conf applies, and we get unsat. If we remove the constraint $x \not\sqsubseteq s$, then the problem is satisfiable and we can construct a model $\mathcal{I}$ where $\mathcal{I}(x)$ is any natural n, $\mathcal{I}(y) = n + 1$, and $\mathcal{I}(s)$ is a singleton bag containing n with multiplicity 1.

2.2 Calculus Correctness

Logozzo et al. [16] proved the decidability of the theory of bags with linear constraints over bag cardinality and the operators: $\varnothing$, bag, m, $\sqcup$, $\uplus$, $\sqcap$, $\setminus$, $\mathbin{\backslash\!\backslash}$, setof, $\sqsubset$, $\sqsubseteq$. In this work, we exclude the cardinality operator. However, we prove that adding the σ operator with computable predicates preserves decidability (See Proposition 2.5 below). In contrast, the further addition of the $\otimes$ and proj operators makes the problem undecidable. This is provable with a reduction from the undecidable equivalence problem for unions of conjunctive SQL queries [14]. Our calculus is not refutation complete in general in the presence of maps because of the possibility of nontermination, as shown in the example below.

Example 2.2. Suppose x is an integer variable, and s is an integer bag variable and consider just the constraint set $\{\mathsf{m}(x, s) \approx 1,\ s \approx \pi(f, s)\}$ where f is again the successor function. The set is unsatisfiable in the theory of finite bags. However, this is not provable in our calculus as it allows the repeated application of rules Map Up and A-Prop, which add fresh elements $x + 1, x + 2, x + 3, \ldots$ to s.

Another source of refutation incompleteness is the presence of constraints with the cross product operator $\otimes$, which causes the generation of nonlinear constraints for the arithmetic subsolver, making the entailment checks in rules such as A-Conf and B-A-Prop undecidable.

However, in the absence of π and $\otimes$, the calculus is both refutation and solution sound, as well as terminating. The soundness properties are a consequence of the fact that each derivation rule preserves models, as specified in the following lemma.

Lemma 2.2. *For all applications of a rule from the calculus, the premise configuration is satisfiable in $\mathbb{T}_{\mathsf{Tab}}$ if and only if one of the conclusion configurations is satisfiable in $\mathbb{T}_{\mathsf{Tab}}$.*

The proof of this lemma provides actually a stronger result than stated in the right-to-left implication above: for each of the conclusion configurations C', every model of $\mathbb{T}_{\mathsf{Tab}}$ that satisfies C' satisfies the premise configuration as well. This implies that any model that satisfies a saturated leaf of a derivation tree satisfies the root configuration as well.

Proposition 2.3 (Refutation Soundness)**.** *For every closed derivation tree with root node C, configuration C is $\mathbb{T}_{\mathsf{Tab}}$-unsatisfiable.*

The proposition above implies that deriving a closed tree with a root $\langle \mathsf{A}, \mathsf{B}, \mathsf{E} \rangle$ is sufficient to prove the unsatisfiability in $\mathbb{T}_{\mathsf{Tab}}$ of the constraints $\mathsf{A} \cup \mathsf{B} \cup \mathsf{E}$. The proof of the proposition is a routine proof by induction on the structure of the derivation tree.

Solution soundness has a more interesting proof since it relies on the construction of a satisfying interpretation for a saturated leaf of a derivation tree, which is a witness to the satisfiability of the root configuration. We describe next at a high level how to construct an interpretation $\mathcal{I}$ for a saturated configuration $C = (\mathsf{A}, \mathsf{B}, \mathsf{E})$.[6] More precisely, we construct a *valuation* $\mathcal{I}$ of the variables and terms in C that agrees with the semantics of the theory symbols and satisfies $\mathsf{A} \cup \mathsf{B} \cup \mathsf{E}$.

Once again, to simplify the exposition, we assume, *with* loss of generality, that any element sort ε in the problem can be interpreted as an infinite set. The actual implementation uses cvc5's theory combination mechanism to allow also sorts denoting finite sets, under mild restrictions on the theories involved [22].

Model Construction Steps

1. Sorts: The meaning of the sort constructors Bag, Tuple, and Int is fixed by the theory. The element sort ε is interpreted as an infinite set. As a concrete representation of bags, and hence of tables, consistent with the theory, we choose sets of pairs (e, n), where e is an element of the bag in question, and n is its (positive) multiplicity.
2. Σ_{Tab}: $\mathbb{T}_{\mathsf{Tab}}$ enforces the interpretation of all Σ_{Tab}-symbols. Saturation guarantees that equivalent bag/tuple terms will be interpreted by $\mathcal{I}$ as the same bag/tuple.
3. Integer variables: Saturation guarantees that there is some model of $\mathbb{T}_{\mathsf{NIA}}$ that satisfies A. We define $\mathcal{I}$ to interpret integer variables according to this model.
4. Variables of sort ε: The calculus effectively partitions them into equivalent classes (where x is in the same class as y iff $x \approx y$ is entailed by B). Each class is assigned a distinct element from $\mathcal{I}(\varepsilon)$, which is possible since it is infinite.
5. Bag variables: $\mathcal{I}$ interprets each bag variable s as the set

$$\mathcal{I}(s) = \{\, (\mathcal{I}(e), n) \mid \mathsf{m}(e, s) \in \mathcal{T}(\mathsf{B}^*),\ \mathcal{I}(\mathsf{m}(e, s)) = n > 0 \,\}.$$

   A well-foundedness argument on $\mathcal{I}$'s construction guarantees that the equation above is well defined. Note that $\mathcal{I}(s)$ is the empty bag iff there is no term $\mathsf{m}(e, s)$ in $\mathcal{T}(\mathsf{B}^*)$ that satisfies the conditions in the comprehension above.

Example 2.3. From Example 2.1, the set of constraints $\{\, y \sqsubseteq \pi(f, s), y \approx x + 1 \,\}$ is satisfiable. In this example the element sort is Int, interpreted as the integers. Similar to Example 2.1, the terms $\mathsf{m}(y, \pi(f, s)), w, \mathsf{m}(w, s)$ are generated during solving. After saturation, and for simplicity, suppose we end up with the following sets of equivalence classes for the terms involved: $\{\mathsf{m}(y, \pi(f, s)), \mathsf{m}(x, s)\}$, $\{x, w\}$, $\{y, x+1\}$, $\{s\}$, $\{\pi(f, s)\}$. The theory of arithmetic assigns consistent concrete values to the first three equivalence classes, say 1, 10, 11, respectively. In Step 5, s is interpreted as the set $\{(\mathcal{I}(x), \mathcal{I}(\mathsf{m}(x, s))),\ (\mathcal{I}(w), \mathcal{I}(\mathsf{m}(w, s)))\} = \{(10, 1)\}$, and similarly $\mathcal{I}(\pi(f, s)) = \{(\mathcal{I}(y), \mathcal{I}(\mathsf{m}(y, \pi(f, s))))\} = \{(11, 1)\}$.

[6] The full model construction and its correctness are discussed in detail in the longer version of this paper [18].

$$\textsc{Join Up}\ \frac{\langle x_1,\ldots,x_m\rangle \sqsubseteq s \in \mathsf{S}^* \qquad \langle y_1,\ldots,y_n\rangle \sqsubseteq t \in \mathsf{S}^* \qquad s \bowtie_{i_1 j_1 \cdots i_p j_p} t \in \mathcal{T}(\mathsf{S}) \qquad x_{i_1} \approx y_{j_1}, \ldots, x_{i_p} \approx y_{j_p} \in \mathsf{S}^*}{\mathsf{S} := \mathsf{S}, \langle x_1,\ldots,x_m,y_1,\ldots,y_n\rangle \sqsubseteq s \bowtie_{i_1 j_1 \cdots i_p j_p} t}$$

$$\textsc{Join Down}\ \frac{\langle x_1,\ldots,x_m,y_1,\ldots,y_n\rangle \sqsubseteq s \bowtie_{i_1 j_1 \cdots i_p j_p} t \in \mathsf{S}^*}{\mathsf{S} := \mathsf{S}, \langle x_1,\ldots,x_m\rangle \sqsubseteq s, \langle y_1,\ldots,y_n\rangle \sqsubseteq t, x_{i_1} \approx y_{j_1}, \ldots, x_{i_p} \approx y_{j_p}}$$

$$\textsc{Filter up}\ \frac{e \sqsubseteq s \in \mathsf{S}^* \qquad \sigma(p,s) \in \mathcal{T}(\mathsf{S})}{\mathsf{E} := \mathsf{E}, p(e) \quad \mathsf{S} := \mathsf{S}, e \sqsubseteq \sigma(p,s) \quad || \quad \mathsf{E} := \mathsf{E}, \neg p(e) \quad \mathsf{S} := \mathsf{S}, e \not\sqsubseteq \sigma(p,s)}$$

$$\textsc{Map up}\ \frac{e \sqsubseteq s \in \mathsf{S}^* \qquad \pi(f,s) \in \mathcal{T}(S) \qquad e \notin W}{\mathsf{S} := \mathsf{S}, f(e) \sqsubseteq \pi(f,s)}$$

$$\textsc{Map down}\ \frac{e \sqsubseteq \pi(f,s) \in \mathsf{S}^* \qquad w \in W}{\mathsf{E} := \mathsf{E}, f(w) \approx e \quad \mathsf{S} := \mathsf{S}, w \sqsubseteq s} \qquad \textsc{Filter down}\ \frac{e \sqsubseteq \sigma(p,s) \in \mathsf{S}^*}{\mathsf{E} := \mathsf{E}, p(e) \quad \mathsf{S} := \mathsf{S}, e \sqsubseteq s}$$

Figure 6: Set filter and map rules. w is a fresh variable unique per e, f, s.

Proposition 2.4 (Solution Soundness)**.** *For every derivation tree with root node C_0 and a saturated leaf C, configuration C_0 is satisfiable in $\mathbb{T}_{\mathsf{Tab}}$.*

While the calculus is not terminating in general, we can show that all derivations are finite in restricted cases.

Proposition 2.5 (Termination)**.** *Let C be a configuration containing no product, join, or map terms. All derivation trees with root C are finite.*

Under the restrictions in the proposition above, the rules of the calculus will never generate nonlinear arithmetic constraints. This means that for input problems that also have no such constraints, any derivation strategy for the calculus yields a decision procedure.

3 A Theory of Finite Relations with Filter and Map

To reason about SQL queries under set semantics, we rely on the theory of finite relations $\mathbb{T}_{\mathsf{Rel}}$, whose signature and calculus are described in Meng et al. [17]. We extend the signature with filter and map operators analogous to those in the theory of tables. We overload the symbols $\sqsubseteq, \sqsubseteq, \sqcup, \sqcap, \setminus, \otimes, \sigma, \pi, \bowtie$ for set operations membership, subset, union, intersection, difference, product, filter, map, and (inner) join,[7] respectively. We use $\mathsf{rel.proj}$ for relation projection.

A calculus to reason about constraints in $\mathbb{T}_{\mathsf{Rel}}$ can be defined over configurations of the form $\langle \mathsf{S}, \mathsf{E}\rangle$, where S is a set of $\mathbb{T}_{\mathsf{Rel}}$-constraints, and E is a set of element constraints similar to the one defined in Section 2. We again assume we have a solver that can decide the satisfiability of element constraints. The $\mathbb{T}_{\mathsf{Rel}}$-constraints are equational constraints of the form $s \approx t$ and $s \not\approx t$ and membership constraints of the form $e \sqsubseteq s$ and $e \not\sqsubseteq s$, where e, s, t are Σ_r-terms. Similar

[7] This is analogous to the inner join operator for tables but differs from the relational join operator defined in Meng et al. [17].

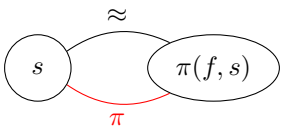

Figure 7: A cycle that contains a red edge with π label.

to bags, we also define the set W to be an infinite set of fresh variables specific for map rules. The closure S^* for the S component of a configuration is defined as:

$$\begin{aligned}\mathsf{S}^* = &\{s \approx t \mid s,t \in \mathcal{T}(\mathsf{S}), \mathsf{S} \models_{\text{tup}} s \approx t\}\\ &\cup \{s \not\approx t \mid s,t \in \mathcal{T}(\mathsf{S}), \mathsf{S} \models_{\text{tup}} s \approx s' \wedge t \approx t' \text{ for some } s' \not\approx t' \in \mathsf{S}\}\\ &\cup \{e \sqsubseteq s \mid e,s \in \mathcal{T}(\mathsf{S}), \mathsf{S} \models_{\text{tup}} e \approx e' \wedge s \approx s' \text{ for some } e' \sqsubseteq s' \in \mathsf{S}\}\end{aligned}$$

For space constraints, we refer the reader to the extended version of this paper [18] for a full description of the extended signature and calculus. We provide in Figure 6 only our additional rules: those for filter and map, which resemble the corresponding rules given in Figure 5 for bags, and those for inner joins.

Calculus correctness. Bansal et al. [2] provide a sound, complete, and terminating calculus for a theory of finite sets with cardinality. The calculus is extended to relations but without cardinality by Meng et al. [17]. That extension is refutation- and solution-sound but not terminating in general. While it is proven terminating over a fragment of the theory of relations, that fragment excludes the relational product operator, which we are interested in here.

We have proved that the calculus by Meng et al. extended with the filter rules in Figure 6 is terminating for constraints built with the operators $\{\approx, \sqcup, \sqcap, \setminus, \otimes, \sigma\}$ [18]. Termination is lost with the addition of map rules, also in Figure 6. Example 2.2 works in this case too as a witness.

However, termination can be recovered when the initial configuration satisfies a certain *acyclicity* condition. To express this condition, we associate with each configuration $C = \langle \mathsf{S}, \mathsf{E} \rangle$ an undirected multi-graph G whose vertices are the relation terms of sort Set occurring in S, and whose edges are labeled with an operator in $\{\approx, \sqcup, \sqcap, \setminus, \otimes, \bowtie, \sigma, \pi\}$. Two vertices have an edge in G if and only if they are in the same equivalence class or a membership rule can be applied (either upward or downward) to C such that one of the vertices occurs as a child of the other vertex in the premises or conclusions of the rule. Furthermore, each edge between vertices $\pi(f, s)$ and s is colored red while the remaining edges are colored black.

Proposition 3.1. *Let $C_0 = \langle \mathsf{S}_0, \mathsf{E}_0 \rangle$ be an initial configuration and let G_0 be its associated multi-graph. The calculus is terminating for C_0 if there is no cycle in G_0 with red edges, and all cycles are located in a subgraph without map terms.*

Figure 7 shows the graph for Example 2.2 which has a cycle with a red edge. Figure 8 shows a graph with no red cycles. For the purposes of this paper, the acyclicity condition is not a serious restriction because typical SQL queries translate to initial configurations whose associated graph does not have cycles with red edges.[8]

[8]Examples of SQL queries that do have cycles with red edges are queries with recursive common table expressions.

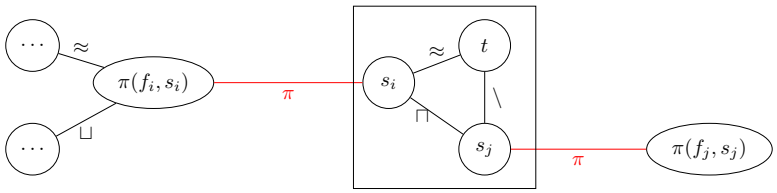

Figure 8: A graph with no cycles that contain red edges.

Symbol	Arity	SMTLIB	Description
null	$\mathsf{Nullable}(\alpha)$	`nullable.null`	Null constructor
some	$\alpha \to \mathsf{Nullable}(\alpha)$	`nullable.some`	Some constructor
val	$\mathsf{Nullable}(\alpha) \to \alpha$	`nullable.val`	Value selector
isNull	$\mathsf{Nullable}(\alpha) \to \mathsf{Bool}$	`nullable.is_null`	Is null tester
isSome	$\mathsf{Nullable}(\alpha) \to \mathsf{Bool}$	`nullable.is_some`	Is some tester
lift	$(\alpha_1 \times \cdots \times \alpha_k \to \alpha) \times$ $\mathsf{Nullable}(\alpha_1) \times \cdots \times \mathsf{Nullable}(\alpha_k) \to \mathsf{Nullable}(\alpha)$	`nullable.lift`	Lifting operators

Figure 9: Signature for the theory of nullable sorts. lift is a variadic function symbol.

4 Supporting Nullable Sorts

SQL and relational databases allow tables with *null values*. To support them in SMT, one could rely in principle on SMT solvers that accept user-defined algebraic datatypes [20], as nullable types are a form of option types. That is, however, not enough since, to encode SQL operations on nullable types, one also needs to lift all operators over a given sort to its nullable version. Since this is extremely tedious and error-prone when done at the user level, we extended the datatypes solver in cvc5 by adding built-in parametric Nullable sorts, along with selectors, testers, and a family of (second-order) lifting operators. The signature of the corresponding theory of *nullable sorts* is provided in Figure 9. The meaning of constructors null and some, the selector val, and the testers isNull, isSome is standard; that of lift is:

$$\mathsf{lift}(f, x_1, \ldots, x_k) = \begin{cases} \mathsf{null} & \text{if } x_i = \mathsf{null} \text{ for some } i \in [1, k] \\ \mathsf{some}(f(\mathsf{val}(x_1), \ldots, \mathsf{val}(x_k))) & \text{otherwise} \end{cases}$$

which is analogous to the semantics of eager evaluation in programming languages. This semantics covers most SQL operations, except for `OR` and `AND`, as SQL adopts a three-valued logic for nullable Booleans [11], interpreting, for instance, `NULL OR TRUE` as `TRUE` and `NULL AND FALSE` as `FALSE`. We can support this short-circuiting semantics through an encoding from SQL to SMT based on the *if-then-else* operator.

5 Evaluation of Benchmarks on Sets and Bags

We evaluated our cvc5 implementation using a subset of the benchmarks [12] derived from optimization rewrites of Apache calcite, an open source database management framework [10]. The benchmark set provides a database schema for all the benchmarks. Each benchmark contains two queries over that schema which are intended to be equivalent under bag semantics, in the sense that they should result in the same table for any instance of the schema. The total number of available benchmarks is 232. We modified many queries that had syntax errors or could not actually be parsed by calcite. We then excluded benchmarks with queries containing

constructs we currently do not support, such as `ORDER BY` clauses[9] or aggregate functions, or benchmarks converted to queries with aggregate functions by the calcite parser. That left us with 88 usable benchmarks, that is, about 38% of all benchmarks. For the purposes of the evaluation we developed a prototype translator from those benchmarks into SMT problems over the theory of tables and of relations presented earlier, following SQL's bag semantics and set semantics, respectively. The translator uses different encodings for bag semantics and set semantics. Each SMT problem is unsatisfiable iff the SQL queries in the corresponding benchmark are equivalent under the corresponding semantics.

We ran cvc5 on the translated benchmarks on a computer with 128GB RAM and with a 12th Gen Intel(R) Core(TM) i9-12950HX processor. We compared our results with those of SQLSolver and SPES analyzers — which can both process the calcite benchmarks directly. The results are shown in Figure 10a. The first three lines show the results for the bags semantics encoding (b) while the fourth line shows the results for the set semantics encoding (s). The column headers $\not\equiv$, $\equiv$, and uk stand for inequivalent, equivalent, and `unknown` respectively. Unknown for SPES means it returns "not proven," whereas for cvc5 it means that it timed out after 10 seconds. SQLSolver solves all benchmarks efficiently within few seconds. However, it incorrectly claims that the two queries in one benchmark, `testPullNull`, are equivalent, despite the fact that they return tables which differ by the order of their columns. The SQLSolver developers acknowledged this issue as an error after we reported it to them, and they fixed it in a later version. SPES too misclassified benchmark `testPullNull` as well as another one (`testAddRedundantSemiJoinRule`) where the two queries are equivalent only under set semantics. The SPES authors acknowledged this as an error in their code. The tool is supposed to answer `unknown` for the second benchmark, as the two queries have different structures, a case that SPES does not support. cvc5 gives the correct answer for these two benchmarks. In each case, it provides counterexamples in the form of a small database over which the two SQL queries differ, something that we were able to verify independently using the PostgreSQL database server [13]. Under set semantics, cvc5 found 83 out of 88 benchmarks to contain equivalent queries, found 1 to contain inequivalent queries (`testPullNull`), and timed out on 4 benchmarks. Under bag semantics, cvc5 proved fewer benchmarks than both SPES and SQLSolver. However, it found the benchmark whose queries are inequivalent under bag semantics but equivalent under set semantics.

Since the calcite benchmark set is heavily skewed towards equivalent queries, we mutated each of the 88 benchmarks to make the mutated queries inequivalent. The mutation was performed manually but *blindly*, to avoid any bias towards the tools being compared. For the same reason, we excluded the two original benchmarks that SPES misclassifies. The results of running the three tools on the mutated benchmarks are shown in Figure 10b. SQLSolver was able to solve all of them correctly. As expected, SPES returned `unknown` on all of them since it cannot verify that two queries are inequivalent.

Note that cvc5's performance improves with the mutated benchmarks because many of them use non-injective map functions, and it is easier to find countermodels for such benchmarks than it is to prove equivalence. Consistent with that, we observe that cvc5 times out for more benchmarks under set semantics than for bag semantics. We conjecture that this is because more queries are equivalent under set semantics than they are under bag semantics.

Looking at cvc5's overall results, more benchmarks were proven equivalent by cvc5 under set semantics than bag semantics. How to improve performance under bag semantics requires

[9]Both SPES and SQLSolver provide partial support for `ORDER BY`. They classify two input queries with `ORDER BY` clauses as equivalent if they can prove their respective subqueries without the `ORDER BY` clause equivalent. Otherwise, they return `unknown`.

		≢	≡	uk	Total
SQLSolver	(b)	1	87		88
SPES	(b)		54	34	88
cvc5	(b)	2	42	44	88
cvc5	(s)	1	83	4	88

(a)

		≢	≡	uk	Total
SQLSolver	(b)	86			86
SPES	(b)			86	86
cvc5	(b)	81		5	86
cvc5	(s)	67	9	10	86

(b)

Figure 10: Left table is the original benchmarks, right table is the mutated one. SPES can only answer equivalent or unknown.

further investigation. However, we find it interesting and useful for the overall development of SQL analyzers that cvc5 was able to expose a couple of bugs in the compared tools.

6 Conclusion and Future Work

We showed how to reason about SQL queries automatically using a number of theories in SMT solvers. We introduced a theory of finite tables to support SQL's bag semantics. We extended a theory of relations to support SQL's set semantics. We handled null values by extending a theory of algebraic datatypes with nullable sorts and a generalized lifting operator. We also showed how to translate SQL queries without aggregation into these theories. Our comparative evaluation has shown that our implementation is not yet fully competitive performance-wise with that of specialized SQL analyzers, particularly on equivalent queries with non-injective mapping functions. We plan to address this in future work.

We are experimenting with adding support for *fold* functionals in order to encode and handle SQL queries with aggregation operators. Another direction for future work is adding filter and map operators to a theory of sequences [21] to reason about SQL queries with *order-by* clauses, which current tools support only in part.

Acknowledgements This work was supported in part by a gift from Amazon Web Services. We are grateful to Qi Zhou and Joy Arulraj, the authors of SPES, for their help, detailed answers to our questions, and for sharing the source code and benchmarks of SPES, which we used in this paper. We are also grateful to Haoran Ding, the first author of SQLSolver, for his prompt answers to our questions and his clarifications on some aspects of the tool. Finally, we thank Abdalrhman Mohamed and the anonymous reviewers for their feedback and suggestions for improving the paper.

References

[1] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edn. (1995), https://dl.acm.org/doi/10.5555/551350

[2] Bansal, K., Barrett, C.W., Reynolds, A., Tinelli, C.: Reasoning with finite sets and cardinality constraints in SMT. Log. Methods Comput. Sci. **14**(4) (2018), https://doi.org/10.23638/LMCS-14(4:12)2018

[3] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y.,

Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: TACAS (1). Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022)

[4] Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11716, pp. 35–54. Springer (2019), https://doi.org/10.1007/978-3-030-29436-6_3

[5] Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In: Proceedings of the Ninth Annual ACM Symposium on Theory of Computing. pp. 77–90. STOC '77, Association for Computing Machinery, New York, NY, USA (1977), https://doi.org/10.1145/800105.803397

[6] Chaudhuri, S., Vardi, M.Y.: Optimization of real conjunctive queries. In: Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. pp. 59–70. PODS '93, Association for Computing Machinery, New York, NY, USA (1993), https://doi.org/10.1145/153850.153856

[7] Chu, S., Wang, C., Weitz, K., Cheung, A.: Cosette: An automated prover for SQL. In: CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings (2017), http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[8] Cohen, S.: Equivalence of queries combining set and bag-set semantics. In: Vansummeren, S. (ed.) Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA. pp. 70–79. ACM (2006). https://doi.org/10.1145/1142351.1142362

[9] Ding, H., Wang, Z., Yang, Y., Zhang, D., Xu, Z., Chen, H., Piskac, R., Li, J.: Proving query equivalence using linear integer arithmetic. Proc. ACM Manag. Data **1**(4) (dec 2023), https://doi.org/10.1145/3626768

[10] Foundation, A.S.: Apache calcite (2014), https://calcite.apache.org/docs/, accessed on Feb 25, 2024

[11] Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall Press, USA, 2 edn. (2008)

[12] Group, G.T.D.: Calcite benchmarks (2020), https://github.com/georgia-tech-db/spes/blob/main/testData/calcite_tests.json, accessed on Feb 25, 2024

[13] Group, T.P.G.D.: Postgresql (2024), https://www.postgresql.org/, accessed on Feb 25, 2024

[14] Ioannidis, Y.E., Ramakrishnan, R.: Containment of conjunctive queries: beyond relations as sets. ACM Trans. Database Syst. **20**(3), 288–324 (sep 1995), https://doi.org/10.1145/211414.211419

[15] Levatich, M., Bjørner, N., Piskac, R., Shoham, S.: Solving LIA* using approximations. In: Beyer, D., Zufferey, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 360–378. Springer International Publishing, Cham (2020)

[16] Logozzo, F., Peled, D.A., Zuck, L.D.: Decision procedures for multisets with cardinality constraints. In: VMCAI, Lecture Notes in Computer Science, vol. 4905, pp. 218–232. Springer Berlin / Heidelberg, Germany (2008)

[17] Meng, B., Reynolds, A., Tinelli, C., Barrett, C.W.: Relational constraint solving in SMT. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 148–165. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_10, https://doi.org/10.1007/978-3-319-63046-5_10

[18] Mohamed, M., Reynolds, A., Tinelli, C., Barrett, C.: Verifying SQL queries using theories of tables and relations. CoRR **abs/2405.03057** (2024), https://arxiv.org/abs/2405.03057

[19] Reynolds, A., Tinelli, C., Goel, A., Krstic, S., Deters, M., Barrett, C.W.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June

9-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7898, pp. 377–391. Springer (2013), https://doi.org/10.1007/978-3-642-38574-2_26

[20] Reynolds, A., Viswanathan, A., Barbosa, H., Tinelli, C., Barrett, C.W.: Datatypes with shared selectors. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10900, pp. 591–608. Springer (2018), https://doi.org/10.1007/978-3-319-94205-6_39

[21] Sheng, Y., Nötzli, A., Reynolds, A., Zohar, Y., Dill, D., Grieskamp, W., Park, J., Qadeer, S., Barrett, C., Tinelli, C.: Reasoning about vectors using an smt theory of sequences. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning. pp. 125–143. Springer International Publishing, Cham (2022)

[22] Sheng, Y., Zohar, Y., Ringeissen, C., Lange, J., Fontaine, P., Barrett, C.W.: Politeness for the theory of algebraic datatypes. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 238–255. Springer (2020), https://doi.org/10.1007/978-3-030-51074-9_14

[23] Veanes, M., Grigorenko, P., de Halleux, P., Tillmann, N.: Symbolic query exploration. In: Formal Methods and Software Engineering, pp. 49–68. Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

[24] Zarba, C.G.: Combining multisets with integers. In: Voronkov, A. (ed.) Automated Deduction—CADE-18. pp. 363–376. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)

[25] Zhou, Q., Arulraj, J., Navathe, S., Harris, W., Xu, D.: Automated verification of query equivalence using satisfiability modulo theories. Proceedings of the VLDB Endowment **12**(11), 1276–1288 (2019)

[26] Zhou, Q., Arulraj, J., Navathe, S.B., Harris, W., Wu, J.: A symbolic approach to proving query equivalence under bag semantics. CoRR **abs/2004.00481** (2020), https://arxiv.org/abs/2004.00481