Elucidating Type Conversions in SQL Engines

Wenjia Ye[1,2], Matías Toro[3], Claudio Gutierrez[3], Bruno C. d. S. Oliveira[2], and Éric Tanter[3]

[1] National University of Singapore
yewenjia@connect.hku.hk
[2] The University of Hong Kong
bruno@cs.hku.hk
[3] Computer Science Department, University of Chile & IMFD
{mtoro,cgutierr,etanter}@dcc.uchile.cl

Abstract. Practical SQL engines differ in subtle ways in their handling of typing constraints and implicit type casts. These issues, usually not considered in formal accounts of SQL, directly affect the portability of queries between engines. To understand this problem, we present a formal typing semantics for SQL, named TRAF, that explicitly captures both static and dynamic type behavior. The system TRAF is expressed in terms of abstract operators that provide the necessary leeway to precisely model different SQL engines (PostgreSQL, MS SQL Server, MySQL, SQLite, and Oracle).
We show that this formalism provides formal guarantees regarding the handling of types. We provide practical conditions on engines to prove type safety and soundness of queries. In this regard, TRAF can serve as precise documentation of typing in existing engines and potentially guide their evolution, as well as provide a formal basis to study type-aware query optimizations, and design provably-correct query translators. Additionally, we test the adequacy of the formalism, implementing TRAF in Python for these five engines, and tested them with thousands of randomly-generated queries.

Keywords: SQL, Typing Semantics, Databases

1 Introduction

Query translation between different SQL engines is a common practice arising in different scenarios, like database migration (to reduce costs, maintenance, changes in software, etc.) [6, 20, 22] and prototyping (code in lightweight databases like SQLlite and then port to a more robust database). Today there are many tools addressing this task [1, 18].

Translation between SQL engines could bring many surprises. One that particularly captures attention is the semantic discrepancy between SQL engines related to typing behavior, the problem we study in this paper. This problem is mainly due to differences in datatypes, type checking, when to perform the

checks, and explicit and implicit type casts that may or may not be performed by the engines. This poses substantial challenges for developers and database administrators. Existing migration tools, whether paid or open source, often fall short of addressing these differences, tending to prioritize syntax over behavioral disparities.[4]

For illustration purposes, consider a table `R`. In the query `SELECT 'a' + '2b' FROM R`, the engine PSQL reports a static error and Oracle a runtime error (in both engines, addition is not defined for strings); MSSQL interprets the operation as string concatenation, yielding `'a2b'`; and MySQL and SQLite yield `2`. On the other hand, the query `SELECT 1 FROM R WHERE '1' < 2` shows that MySQL and SQLite do not always exhibit identical behavior: the first yields `1`, while the other an empty result. In Table 1 we present further (minimal) examples of this wide difference in behavior, which are explained in detail in Section 2.

As these examples show, database engines have different treatment of types, following different design models, like some being statically typed while others embrace dynamic typing, and different approaches to overloading basic arithmetic and comparison operations.

Understanding and addressing these anomalies is not a simple task. A first issue is that SQL standards do not cover many issues related to typing or leave the interpretation rather open.[5] In addition, many of the design decisions of the engines are hidden under optimization mechanisms that either are not public or complex to find in the code. Furthermore, the problem has not been addressed by the research literature. Although there is solid work on the formalization of the semantics of SQL [19,31], they assume that all comparisons and operations apply to the right types. Typing in SQL have been explored [3, 13, 24, 26, 32]. Nonetheless, the problem of type constraints and casts potentially raising errors at runtime, which is addressed in this paper, has not been dealt with before.

In this paper we address the problem of discrepant type related behavior by proposing a general formal framework, called TRAF, that models both common behavior and the intricate behavioral discrepancies across SQL database engines. TRAF was designed to explicitly capture the semantics of types, both static and dynamic, of a core fragment of relational algebra. The selected minimal core (already presented in other works like [8,19,27]) is designed to include the minimal features and operators that generate the indicated anomalies, as well as being flexible enough to model different SQL engines. It comprises booleans, numbers, selections, cross-products, nested queries within `FROM` clauses, and set operations. We also add support for arithmetic operators and type casts. Although features

[4] Some examples: `https://en.wikibooks.org/wiki/Converting_MySQL_to_PostgreSQL` `http://www.sqlines.com/online` `https://www.rebasedata.com/convert-bmysql-bto-bpostgres-bonline`

[5] A good example is the following: To cast an exact number to an exact numeric type, e.g. from a real to int, the specification says: "If there is a representation of SV [source value] in the data type TV [target value] that does not lose any leading significant digits after rounding or truncating if necessary, then TV is that representation. The choice of whether to round or truncate is implementation-defined." ANSI SQL 1992, Sec. 6.10, Case 3)a)i))

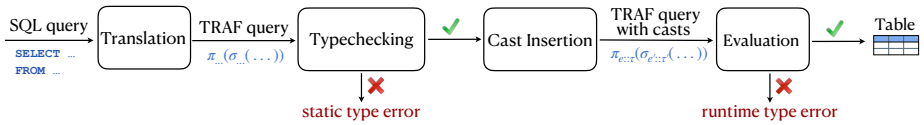

Fig. 1: Overview of TRAF. Some abstract operators in the typechecking, cast insertion, and evaluation phases must be instantiated to implement a particular engine. We provide sample instantiations for PSQL, MSSQL, Oracle, MySQL and SQLite.

such as nulls, aggregation, or `EXISTS`, are not supported within this core, it serves to illustrate the primary distinctions, leaving the extension to these additional features as potential future work.

We study two kinds of different behavior among engines: (1) different results due to type conversions, and (2) different behavior related to type errors. Regarding the latter, we identify two kinds of type errors: *static errors*, which happen during compilation/before running the query; and *runtime errors* that occur during the execution of the query. We say that two engines differ in behavior if, for a given query, the results are different (including different kinds of errors).

TRAF involves four sequential phases as illustrated in Figure 1: translation, typechecking, cast insertion, and evaluation. In the **translation** step (which is standard and can be found in the extended version of the paper), an SQL query is (1) analyzed to rule out syntactically invalid queries such as `SELECT FROM FROM` and (2) transformed to a TRAF query, that is, essentially a typed relational algebra query. Next, in the **typechecking** phase, a typechecker validates the query before evaluation (Section 3.3).[6] This typechecker is responsible for identifying mismatches between types and the number of columns of subqueries in set operations, and to ensure proper access to column names in scope. More importantly, it validates the appropriate usage of both implicit and explicit casts, rejecting operations that may result in casts known to always fail during evaluation. If the typechecker rejects the query, the process terminates and reports a static type error to the user. However, if the query is deemed well-typed, it proceeds to the third phase.

The **cast insertion** phase (Section 3.4) transforms a TRAF query by making all implicit type casts explicit. The purpose of this phase is to circumvent the complexities that would arise from handling implicit casts in the evaluation phase, thereby simplifying the dynamic semantics of TRAF. The **evaluation** phase (Section 3.5) interprets the transformed TRAF query. For this purpose, we present a monadic evaluator expressed using denotational semantics, cleanly encompassing the management of explicit type conversions and runtime errors. Should a type error arise during evaluation, the evaluation process is halted, and a runtime type error is reported. In the absence of runtime type errors, the outcome of evaluating a query is a table.

[6] TRAF does not assume the absence of type mismatches; users can write queries such as `SELECT R.A from R WHERE 'Bob' = 1`.

TRAF is designed to be flexible enough to model different real-world engines (PSQL, MSSQL, Oracle, MySQL and SQLite) in order to facilitate understanding of different behaviors and thus enable informed decisions when translating queries. Specifically, typechecking, cast insertion, and evaluation are parameterized in terms of abstract operators that need to be instantiated (Section 4).

The formal model is coherent and comprehensive, making it possible to precisely formulate and prove properties satisfied by all or some engines (Section 5). First, we prove a type safety result that ensures that well typed queries either reduce to a table or raise a (controlled) type error. In other words, evaluation of well-typed queries does not get stuck. Second, we enhance this result proving that the resulting table can be typed to the same type of the query. Third, the instantiation of the model to PSQL, MySQL and SQLite satisfies a theorem stating that if the programmer does not use explicit casts in well-typed queries, then the queries will evaluate without errors. Fourth, regarding cast insertion, independently of the engine (the proofs are parameterized by light constraints on abstract operators), translated queries preserve types, and the translation is unique.

Finally, to validate the adequacy of our formal framework, and following other approaches that also validate semantics by testing them against real-world implementations [19, 29, 30], we developed multiple database interpreters, and tested them with thousands of randomly-generated queries, comparing their results with those obtained from actual database engines. Additionally, we shed light on the impact and challenges of query optimizations on the evaluation process.

In summary, our contributions includes the identificacion of practical semantic discrepancies due to typing between SQL engines; the description of TRAF, a formal framework to reason about types both statically and dynamically, which can model different SQL engines; the metatheory of TRAF, indicating precise constraints on abstract operators required for properties to hold; and an empirical validation of our formal model, by building several interpreters tested against real engines. Additional material, including proofs and full definitions, can be found in the extended version. Also, we made available a prototype implementation as supplementary material.

The rest of the paper is organized as follows. Section 2 explains the discrepancies illustrated in Table 1. Section 3 presents the TRAF formal framework. Section 4 describes the instantiation of TRAF to practical SQL engines. Section 5 presents and proves formal properties of the model and its instantiations. Section 6 summarizes the experimental validation. In Section 7 we discuss related work, and Section 8 presents brief conclusions.

2 Typing semantic discrepancies

In this section we explain the source of discrepancies of Table 1, which we use as a starting point and develop (and justify) TRAF. For clarity, we categorize

	Query	PSQL	MSSQL	Oracle	MySQL	SQLite
E1	`SELECT 1.1 + 1 FROM R`	2.1	2.1	2.1	2.1	2.1
E2	`SELECT '1' + 1 FROM R`	2	2	2	2	2
E3	`SELECT '1.1' + 1 FROM R`	✗	↛	2.1	2.1	2.1
E4	`SELECT '1.1' + 1.1 FROM R`	2.2	2.2	2.2	2.2	2.2
E5	`SELECT '1' + '1' FROM R`	✗	'11'	2	2	2
E6	`SELECT 'a' + '2b' FROM R`	✗	'a2b'	↛	2	2
E7	`SELECT 1+A FROM R WHERE B=20`	✗	2	2	2	2
E8	`SELECT 1+A FROM R WHERE B=10`	✗	↛	↛	1	1
E9	`SELECT 1 + A FROM (SELECT '2' AS A) B`	✗	3	3	3	3
E10	`SELECT 1 FROM R WHERE '1' < 2`	1	1	1	1	∅
E11	`SELECT 1 FROM R WHERE '1.1' < 2`	✗	↛	1	1	∅
E12	`SELECT '1.1' FROM R INTERSECT SELECT 1.1 FROM R`	1.1	1.1	✗	1.1	∅
E13	`SELECT '1.1' FROM R INTERSECT SELECT 1 FROM R`	✗	↛	✗	∅	∅

Table 1: Examples of discrepant behaviors of different database engines, considering the table `R(A,B)` $= \{(\texttt{'Bob'}, 10), (\texttt{'1'}, 20), (\texttt{'1.1'}, 30)\}$. The queries are purposely chosen to exhibit minimal cases of typing issues. For simplicity we show only one element of the result. We use ✗ to denote a static error (before execution), and ↛ to denote a runtime error.

the examples of discrepancies into three groups: arithmetic, boolean, and set operations.

2.1 Arithmetic operations

For the first set of examples we focus on the expression being selected rather than the tables or the conditions.

E1 and E2. To begin, we present two examples that behave uniformly across the engines. For simplicity, we say that the result is 2.1, to denote a bag of uniform elements $\{2.1, 2.1, 2.1\}$.

E3. This is the first example that illustrates a difference in behavior. PSQL and MSSQL throw a type error, whereas the other engines return 2.1. The reason for the error is that in PSQL and MSSQL we can implicitly cast a string to an integer if the string is an integer, but not if the string is a real number. To fix this problem in PSQL, we must explicitly *cast* the string to a real number: `SELECT CAST('1.1' as FLOAT) + 1 FROM R`.

E4. If we take example 3, and change `1` for a real number, such as `1.1`, then the result is now 2.2 for every engine. The difference with respect to the previous example is that the plus operation is defined for both integers and real numbers, and now `'1.1'` can be cast directly to a real number.

E5. PSQL reports an static error due to the lack of an addition operator between two strings. MSSQL returns `'11'` as addition is overloaded for string. The other engines return 2 as addition is only defined between numbers, thus implicit casting `'1'` to `1`.

E6. Both PSQL and Oracle report an error, MSSQL uses string concatenation yielding `'a2b'`, and MySQL and SQLite yield `2`. The reason for the latter is due

Operation	SQLite	Other engines
`0 < 1`	1	t
`'0' < 1`	0	t
`'1' < 0`	0	f
`'0'+0 < 1`	1	t
`'0' < CAST(1 AS INT)`	1	t
`'0' < 1 + 0`	0	t

Table 2: Behavior of the comparison operator in SQLite

to the way these engines cast strings to numbers: they search for a number in the prefix of the string (if nothing is found then 0 is returned).

E7. PSQL rejects this query as column `A` is of type String, and the addition between numbers and strings is not defined. This behavior is more conservative than E2, as now it cannot determine statically if the given string can be cast to number or not. Other engines defer the check to runtime and return `2`. To fix this query in PSQL we can explicitly cast column `A` to integer: `SELECT 1+CAST(A as INT) FROM R WHERE B=20`.

E8. PSQL (statically) rejects this query similarly to E7. MSSQL and Oracle now fail dynamically as they cannot convert `'Bob'` to a number. MySQL and SQLite on the other hand do not fail and return `1` as `'Bob'` is cast to `0`. Casting `A` to `INT` in PSQL would make the error dynamic, and a runtime type error would be reported instead, similarly to MySQL and SQLite.

E9. Contrary to E2, PSQL raises a static error as the nested query hides the actual String returned by the subquery. All other engines yield `3` as conversions are optimistically performed at runtime. To fix this query in PSQL an explicit cast must be inserted `SELECT 1 + CAST(A AS INT) FROM (SELECT '2' AS A) B`.

2.2 Boolean operations

The following examples illustrate difference in behavior related to comparison operators on conditionals. Note that 1 and 0 are used to represent true and false in SQLite and MySQL.

E10. Almost every engine is able to cast `'1'` to `INT`, returning 1. SQLite, on the other hand, returns a empty result as the condition is `false`. This is because SQLite does not perform implicit conversions at the boundaries of comparisons. SQLite has a type hierarchy where every string is bigger than any number.

E11. Now, if the left operand is a string representing a real, then PSQL and MSSQL return a type error. This is because the best type of the comparison operator is the one that takes two integers as argument (because 2 is an integer). As there is no direct implicit conversion between a string representing a real and an integer the query is rejected. SQLite still returns an empty result, and MySQL and Oracle return 1.

Comparison operator in SQLite. SQLite warrants special attention to illustrate unique cases related to the comparison operator, as exemplified in Table 2.

Notably, operations `'0' < 1` and `'1' < 1` yields `0`. This behavior is attributed to the fact that strings are considered larger than integers, as previously explained. However, when a cast is introduced the expressions now yields `1`. For instance, `'0' < CAST(1 as INT)` yields `1`. Since when the type of one operand is explicitly specified, an implicit conversion to that type is performed on the other operand.

2.3 Set operations

E12. PSQL, MSSQL and MySQL, yield `1.1`. This is because, during intersection, two conditions are checked: (1) the number of columns of both subqueries must match, and (2) the types of the columns must also be consistent. To achieve the second condition, an implicit conversion from string to real is inserted in the left subquery (the other direction is forbidden). However, Oracle encounters a type error since the column types do not match. On the other hand, SQLite returns an empty result as `1.1` is not the same as `'1.1'`.

E13. Now as `'1.1'` cannot be implicitly cast to an integer (the column type of the right subquery), this program is rejected by PSQL, MSSQL and Oracle. Both MySQL and SQLite return an empty result.

Having illustrated the various discrepancies between SQL engines in handling queries, it becomes evident that a more structured approach is necessary to fully understand and model these differences. To achieve this, we now turn to the formalism of TRAF.

3 The (TRAF) formal framework

In this section we present the syntax, type system, cast insertion, dynamic semantics, and translation of a typed core fragment of relational algebra, supporting projections, selection, set operations, arithmetic and boolean operations, and implicit and explicit casts. The formalism captures common behavior between engines while providing leeway to model different concrete engines, and thus is parametrized by abstract operators (detailed and instantiated in Section 3.2).

3.1 Syntax

The syntax of Typed Relational Algebra Framework (TRAF) is presented in Figure 2. The formalization is inspired by the work of Guagliardo and Libkin [19], except that here we deal with typing instead of assuming a prior (unstudied) typing phase. Types play a central role in this work, because as we illustrated, typing discrepancies are a source of important behavioral differences between engines. This section first presents types and schemas, then values and expressions, and finally queries.

Types and Schemas. There are two categories of types, *value types* τ for expressions (and values), and *relation types* T for queries and tables (relations). For simplicity, we only consider reals $\mathbb{R}$, integers $\mathbb{Z}$, booleans $\mathbb{B}$ and strings String. To avoid dealing with precision issues inherent in floating-point representations,

Types and Schemas	$\tau ::=$	$\mathbb{R} \mid \mathbb{Z} \mid \mathbb{B} \mid \mathsf{String} \mid ?$	(value types)
	$T ::=$	$N \mapsto \tau \mid T, N \mapsto \tau$	(relation types)
	$\Gamma ::=$	$\emptyset \mid \Gamma, R \mapsto T$	(schema)
Values and Expressions	$\mathtt{O_A} \in$	$\{+\}$	(arithmetic ops)
	$\mathtt{O_C} \in$	$\{<, =\}$	(comparison ops)
	$\mathtt{O_B} \in$	$\{\wedge, \vee\}$	(boolean ops)
	$w ::=$	$d \mid n \mid b \mid s$	(simple values)
	$v ::=$	$w \mid w :: ?$	(values)
	$e ::=$	$N \mid v \mid e\ \mathtt{O_A}\ e \mid e :: \tau$	(general expressions)
	$\theta ::=$	$e\ \mathtt{O_C}\ e \mid \theta\ \mathtt{O_B}\ \theta \mid \neg\theta$	(boolean expressions)
	$\beta ::=$	$e\ \mathtt{as}\ N \mid \beta, e\ \mathtt{as}\ N$	(aliased expressions)
Queries	$\mathtt{O_S} \in$	$\{\cup, \cap, \times, -\}$	(set query ops)
	$Q ::=$	$R \mid \pi_\beta(Q) \mid \sigma_\theta(Q) \mid Q\ \mathtt{O_S}\ Q \mid \varepsilon(Q)$	(queries)
Rows and Tables	$r ::=$	$\overline{v_i}$	
	$t ::=$	$\{\!\{\overline{r_i}\}\!\}$	

Fig. 2: Syntax of TRAF. n, b, s denote respectively an integer number, a boolean value and a string. N is a name. R a relation.

we use the abstract type $\mathbb{R}$ to represent decimal numbers. In addition, we use the symbol ? for the unknown type, used by PSQL to type string literals [17], and to model flexible typing in SQLite. A relation type T is an ordered list of pairs of column names and their corresponding value type. A schema Γ is a list of pairs of relation names and their relation type. We assume that both T and Γ do not contain duplicated column names and relation names, respectively (and thus behave as mappings). Intuitively, schemas represent types of databases.

Values and expressions. We represent tables as bags of *rows*, where each row is a list of *values*. A *simple value* w, which represent atomic data (integers, booleans, strings). Values v are either a simple value w, or a simple value cast to the unknown type $w :: ?$ (the latter is not used directly by programmers, and it is used exclusively in engines such as SQLite). There are three kinds of expressions: general, boolean and aliased. A general expression e, used in projections and selections, is either a column name N (e.g. Name or Age), a value v, an arithmetic operation (for simplicity we only use $+$), or an explicit cast $e :: \tau$. Boolean expressions θ as usual are comparison operations or logical combinations of them.[7] As a standard, an aliased expression is a slight extension of the classical renaming, allowing binding of names to expressions, instead of only to queries. For example, in `SELECT A.C FROM (SELECT 1+1 AS C FROM R) A`, the expression `1+1` gets a name `C` that can be used in the outer query.

Queries. A query is either a relation R, a projection $\pi_\beta(Q)$, a selection $\sigma_\theta(Q)$, or a set operation, that is, cross product $(Q \times Q)$, intersection $(Q \cap Q)$, union $(Q \cup Q)$, difference $(Q - Q)$, and the removal of duplication $(\varepsilon(Q))$.

[7] Note that we do not include boolean expressions as general expressions, because some engines do not support selecting boolean values in queries (e.g. in MSSQL and Oracle, `SELECT 1 < 2 FROM R` is a syntactically invalid query while it is valid in PSQL, MySQL and SQLite).

The only novelty is that a projection here is parametrized by a list of aliased expressions, instead of a list of names. This allows to model SQL queries like `SELECT 1+1 AS C FROM R` as $\pi_{(1+1)\text{ as }C}(R)$.

Rows and Tables. A table t is a bag of *rows* $\{\!\{r_i\}\!\}$, where a row r is an ordered list of values $\overline{v_i}$.

3.2 Abstract operators

As mentioned in Section 1, certain key operators in TRAF are left abstract, as they depend on the specific engine being used. We mark with (*) the partial operators.

Bidirectional Implicit Cast(*). Operator $\mathit{biconv}(e_1, \tau_1, e_2, \tau_2) = \tau_3$ determines the optimal implicit type cast for a set operator applied to two columns of (possibly) different types. The *biconv* operator takes as argument two expressions (e_1, e_2) and their corresponding types (τ_1, τ_2), returning a type. It either returns τ_2 or τ_1 by testing if e_1 can be implicitly cast to τ_2, or if e_2 can be implicitly cast to τ_1 respectively.

Explicit Cast(*). Operator $\mathit{cast}(v, \tau) = v'$ attempts to cast value v to a value v' of type τ. This function is primarily used to evaluate casts at runtime.

Overloading Resolution(*). Operator $\mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \mathtt{O}) = \tau_3$ determines which specific operation among a set of overloaded ones should be called based on the provided arguments during invocation. More specifically, given an operator it tries to find the best candidate type for expressions e_1 and e_2, typed as τ_1 and τ_2 respectively.

Explicit Cast Feasibility. This operator takes one expression and two types, and returns a boolean. For simplicity, it is presented as a relation $e : \tau' \rightsquigarrow \tau$, and rules out explicit casts that are known to fail at runtime. It tests if it is possible to explicitly cast expression e of type τ', to an expression of type τ.

Type of Values. Operator $\mathit{ty}(v) = \tau$ computes the type of values (constants).

Type Cleaning. The operator $\mathit{clean}(T) = T'$ performs post-processing on the relation type T, returning a new relation type T'. This is primarly used in PSQL, where literal strings are initially typed as ?, a type that is later converted to String when determining the type of a subquery.

Annotation Insertion. The operator $\mathit{insert}(e, \tau, \mathtt{O}) = e'$ returns either an explicitly cast expression e to type τ or simply e, depending on the operation O. For instance, in SQLite, comparison operations do not implicitly cast their operands, whereas addition operations do perform such casts.

Value Operation Application. Operator $\mathit{apply}(\mathtt{O}, v_1, v_2) = v_3$ performs arithmetic or comparison operation O to values v_1 and v_2, yielding value v_3.

In the next subsections, we use these operators to define the dynamic semantics, the type system and the cast insertion procedure. Examples of instantiation of these abstract operators can be found in Section 4.

$$\boxed{T \vdash e : \tau} \quad (\mathrm{T}v)\frac{}{T \vdash v : \mathit{ty}(v)} \qquad (\mathrm{T}\mathtt{O}_\mathtt{B})\frac{T \vdash \theta_1 : \mathbb{B} \quad T \vdash \theta_2 : \mathbb{B}}{T \vdash \theta_1\ \mathtt{O}_\mathtt{B}\ \theta_2 : \mathbb{B}}$$

$$(\mathrm{T}N)\frac{(N \mapsto \tau) \in T}{T \vdash N : \tau} \qquad (\mathrm{T}\neg)\frac{T \vdash \theta : \mathbb{B}}{T \vdash \neg\theta : \mathbb{B}} \qquad (\mathrm{T}::)\frac{T \vdash e : \tau' \quad \mathit{e : \tau' \twoheadrightarrow \tau}}{T \vdash (e :: \tau) : \tau}$$

$$(\mathrm{T}\mathtt{O})\frac{\begin{array}{c}T \vdash e_1 : \tau_1 \quad T \vdash e_2 : \tau_2 \quad \mathtt{O} \in \mathtt{O}_\mathtt{A} \cup \mathtt{O}_\mathtt{C} \\ \mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \mathtt{O}) = \tau_3 \times \tau_4 \to \tau_5\end{array}}{T \vdash e_1\ \mathtt{O}\ e_2 : \tau_5}$$

$$\boxed{T \vdash \beta : T'} \qquad (\mathrm{T}\beta)\frac{\forall i. T \vdash e_i : \tau_i \quad \mathit{unique}(\overline{N_i})}{T \vdash \overline{e_i\ \mathtt{as}\ N_i} : \overline{N_i \mapsto \tau_i}}$$

$$\boxed{\Gamma \vdash Q : T}$$

$$(\mathrm{T}\pi)\frac{\Gamma \vdash Q : T \quad \mathit{clean}(T) \vdash \beta : T'}{\Gamma \vdash \pi_\beta(Q) : T'} \qquad (\mathrm{T}\sigma)\frac{\Gamma \vdash Q : T \quad T \vdash \theta : \mathbb{B}}{\Gamma \vdash \sigma_\theta(Q) : T}$$

$$(\mathrm{T}\times)\frac{\ell(\Gamma, Q_1) \cap \ell(\Gamma, Q_2) = \emptyset \quad \Gamma \vdash Q_1 : T_1 \quad \Gamma \vdash Q_2 : T_2}{\Gamma \vdash Q_1 \times Q_2 : T_1, T_2} \qquad (\mathrm{T}R)\frac{\Gamma(R) = T}{\Gamma \vdash R : T}$$

$$(\mathrm{T}\mathtt{O}_\mathtt{S})\frac{\begin{array}{c}\Gamma \vdash \pi_{\beta_1}(Q_1) : T_1 \qquad \Gamma \vdash \pi_{\beta_2}(Q_2) : T_2 \\ \mathit{biconv}^*(\beta_1, T_1, \beta_2, T_2) = T \quad \mathtt{O}_\mathtt{S} \in \{\cup, \cap, -\}\end{array}}{\Gamma \vdash \pi_{\beta_1}(Q_1)\ \mathtt{O}_\mathtt{S}\ \pi_{\beta_2}(Q_2) : T} \qquad (\mathrm{T}\varepsilon)\frac{\Gamma \vdash Q : T}{\Gamma \vdash \varepsilon(Q) : T}$$

Fig. 3: Type System of TRAF. Abstract operators are highlighted in gray.

3.3 Type System

The type system of TRAF is presented in Figure 3, and in the following, we briefly explain the rationale behind each rule. Boxes in gray indicate abstract operators, and we can provide specific implementations for modeling an engine (Section 3.2).

Expressions. Rule (Tv) assigns types to values based on the specific database engine (the *ty* operator in the grey box). For instance, in PSQL, integers are typed as $\mathbb{Z}$ and literal strings as ?, but in SQLite, both are typed as ?. Rule (TN) assigns the type τ to the column name N if N is mapped to τ in the relation type T.

Rule (T::) assigns the type τ to an ascription $e :: \tau$ if the following conditions are met: (1) the expression e must be well-typed for some τ'; (2) the explicit cast of e to τ (engine-dependent, thus grey box) is checked to rule out explicit casts that are known to always fail at runtime. For example, the attempt to cast `'hi'` to $\mathbb{Z}$ (`'hi'` $:: \mathbb{Z}$) is rejected in PSQL, while casting `'1'` to $\mathbb{Z}$ is accepted in both PSQL and SQLite.

Rules (T$\mathtt{O}$), (T$\mathtt{O}_\mathtt{B}$) and (T$\neg$) type operations. (T$\mathtt{O}_\mathtt{B}$) and (T$\neg$) type boolean operations as usual. The interesting case is rule (T$\mathtt{O}$) that types arithmetic and

string operations. Based on the types of e_1 and e_2 and operation `O`, the (T`O`) rule searches for the best candidate type signature for the given operation, taking into consideration that an operation might be overloaded with multiple types. It involves the operations *resolve* and *ty* that are engine-dependent. If a single candidate function type is identified, the expression is typed; otherwise, it is considered ill-typed. Note that types τ_3 and τ_4 do not need to coincide with τ_1 and τ_2 as arguments can be cast to different types. For instance, in the case of the query `SELECT 1+1 FROM P` both PSQL and SQLite choose numeric addition ($\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$). However, when dealing with strings (e.g. `SELECT 'a' + 'b' FROM P`) PSQL rejects the query because it cannot choose a best candidate operator, but SQLite instantiates the query with numeric addition, implicitly casting both string arguments to integers.

Rule (Tβ) is used to type an aliased expression β. We use the notation $\overline{A_i}$ to denote a list $A_1, \ldots, A_n$. Under T, every expression e_i yields a type τ_i. Subsequently, the type of the aliased expression as a whole is a relation type, where each name N_i is mapped to the type τ_i of their corresponding subexpression e_i. Additionally, we use metafunction *unique*(.) to ensure that names are unique.

Queries. Rule (TR) assigns a relation type to relation name R according to schema environment Γ. Rule (Tπ) types $\pi_\beta(Q)$ based on the type of Q and that of the aliased expression β, which is typed under $clean(T)$ to remove unknown occurrences. For instance, in PSQL, `SELECT '1' + 1 FROM P` runs successfully, but `SELECT C + 1 FROM (SELECT '1' AS C) B` is rejected statically: the first query `'1'` has type unknown ?, which can be cast to integer, whereas in the second query, the unknown type ? is transformed to String disallowing the implicit cast to integer.

Rule (Tσ) first typechecks the subquery, resulting in a relation type T. Then, the condition must be successfully typed as boolean under the context of T (considering that the condition may reference columns from the subquery). Finally, the selection operation is assigned the same type as the subquery T. Rule (T$\times$) types cross products using the concatenation of the relation types of both subqueries, ensuring that the sets of names in each subquery are disjoint. The list of column names of a query Q is extracted using function $\ell(\Gamma, Q)$, and defined as:

$$
\begin{aligned}
\ell(\Gamma, R) &= \overline{N_i} \quad \text{where } \Gamma(R) = \overline{N_i \mapsto \tau_i} \\
\ell(\Gamma, Q_1 \times Q_2) &= \ell(\Gamma, Q_1), \ell(\Gamma, Q_2) \\
\ell(\Gamma, Q_1 \ \mathtt{O_S}\ Q_2) &= \ell(\Gamma, Q_1) \quad \text{where } \mathtt{O_S} \in \{\cup, \cap, -\};\ (*) \\
\ell(\Gamma, \sigma_\theta(Q)) &= \ell(\Gamma, Q) \\
\ell(\Gamma, \pi_\beta(Q)) &= \ell(\Gamma, \beta) \\
\ell(\Gamma, \beta, e \ \mathtt{as}\ N) &= \ell(\Gamma, \beta), \ell(\Gamma, e \ \mathtt{as}\ N) \\
\ell(\Gamma, e \ \mathtt{as}\ N) &= N
\end{aligned}
$$

Rule (*) follows standard engine usage of using the left schema of a set expression as output schema.

Rule ($\texttt{TO}_\texttt{S}$) deals with set operations (union, intersection, and difference) which require special attention. Usually, it is assumed that the names, number of columns, and types of the columns in the subqueries match. In practice, the column names do not necessarily match, but number of columns and types must align. To achieve this, many engines perform implicit casts between the columns of the subqueries to align their types. In particular, string literals are analyzed to check the plausibility of casts. For instance in PSQL, `(SELECT '1.1' AS A FROM P) INTERSECT (SELECT 1.1 AS A FROM P)` runs successfully, resulting in a non-empty result, whereas `(SELECT CAST(Age AS TEXT) AS A FROM P) INTERSECT (SELECT Age FROM P)` is rejected before execution. To deal with these special cases, and without loss of generality, we require that both subqueries within a set operation must be projections in order to verify column casts. Therefore, the rule first typechecks both projections. Second, it examines whether the lists of aliased expressions can be implicitly cast between each other, taking their relation types into account. Finally, if this cast is feasible, the target relation type is captured and used to typecheck the whole set operation. To check casts between lists of aliased expressions, we use the $biconv^*$ operation defined as follows:

$$biconv^*(\overline{e \texttt{ as } N}, \overline{N \mapsto \tau}, \overline{e' \texttt{ as } N'}, \overline{N' \mapsto \tau'}) = \overline{N \mapsto biconv(e, \tau, e', \tau')}$$

This operation returns a relation type, where each name N is mapped to the application of abstract operator $biconv$ over each pair of expressions and their types. We select N, the name of the left subquery, as it is a more commonly-adopted practice in various database engines. This partial operator determines the optimal implicit type cast for a set operator applied to two columns of (possibly) different types, returning one of the two types as a result. The expressions are provided to the function to rule out casts that are known to always fail during evaluation. For instance, PSQL accepts query `(SELECT '1' AS A, 1 AS B FROM R) UNION (SELECT 2 AS A, '2' AS B FROM R)`, as both `'1'` and `'2'` can be cast to integers; but rejects `(SELECT '1.1' AS A FROM R) UNION (SELECT 1 AS A FROM R)` as `'1.1'` cannot be implicitly cast to an integer (note that in PSQL, implicit casts from integers to strings are not allowed).

3.4 Cast Insertion

Recall from Figure 1 that to avoid the complexity of dealing with implicit casts during runtime, before execution, in TRAF we transform each implicit cast to an explicit cast. For instance, a PSQL query `SELECT '1' + 1 AS A FROM R` is transformed to `SELECT CAST('1' AS INT) + 1 AS A FROM R`, which in TRAF corresponds to the elaboration from $\pi_{\texttt{'1'}+1 \texttt{ as A}}(R)$ to $\pi_{(\texttt{'1'}::\mathbb{Z})+1 \texttt{ as A}}(R)$.

Figure 4 presents an excerpt of the explicit cast insertion rules; the complete rules can be found in the extended version. The rules are *type directed*, meaning that (1) we only elaborate well-typed terms, and (2) we use type information during elaboration. Like for typing, elaboration rules are defined inductively and grouped in three categories: for general and aliased expressions, and for queries.

$\boxed{T \vdash e : \tau \rightsquigarrow e'}$

$$(\mathrm{E}v)\ \frac{T \vdash v : \tau}{T \vdash v : \tau \rightsquigarrow v :: \tau}$$

$$(\mathrm{E}\mathtt{O})\ \frac{\begin{array}{c} T \vdash e_1 : \tau_1 \rightsquigarrow e_1' \qquad T \vdash e_2 : \tau_2 \rightsquigarrow e_2' \\ \mathtt{O} \in \{<, =, +\} \qquad \mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \mathtt{O}) = \tau_3 \times \tau_4 \to \tau_5 \end{array}}{\begin{array}{l} T \vdash e_1\ \mathtt{O}\ e_2 : \tau_4 \rightsquigarrow \\ \quad \mathit{insert}(\mathit{insert}(e_1', \tau_3, \mathtt{O})\ \mathtt{O}\ \mathit{insert}(e_2', \tau_4, \mathtt{O}), \tau_5, \mathtt{O}) \end{array}}$$

$\boxed{\Gamma \vdash Q : T \rightsquigarrow Q'}$

$$(\mathrm{E}\mathtt{O}_\mathtt{S})\ \frac{\begin{array}{c} \Gamma \vdash \pi_{\beta_1}(Q_1) : T_1 \rightsquigarrow \pi_{\beta_1'}(Q_1') \qquad \Gamma \vdash \pi_{\beta_2}(Q_2) : T_2 \rightsquigarrow \pi_{\beta_2'}(Q_2') \\ \mathit{biconv}^*(\beta_1, T_1, \beta_2, T_2) = T \qquad \mathtt{O}_\mathtt{S} \in \{\cup, \cap, -\} \\ e_1 = \mathit{insert}^*(\beta_1', T, \mathtt{O}_\mathtt{S}) \qquad e_2 = \mathit{insert}^*(\beta_2', T, \mathtt{O}_\mathtt{S}) \end{array}}{\Gamma \vdash \pi_{\beta_1}(Q_1)\ \mathtt{O}_\mathtt{S}\ \pi_{\beta_2}(Q_2) : T \rightsquigarrow \pi_{e_1}(Q_1')\ \mathtt{O}_\mathtt{S}\ \pi_{e_2}(Q_2')}$$

Fig. 4: TRAF Cast Insertion (excerpt). Abstract operators are highlighted in gray.

Most elaboration rules directly follow their corresponding typing rule. Judgment $T \vdash e : \tau \rightsquigarrow e'$ represents that expression e typed as τ under relation type T is elaborated to e'. Judgment $\Gamma \vdash Q : T \rightsquigarrow Q'$ is defined analogously.

Rule (Ev) inserts an explicit cast to the type of each value. This is especially relevant for engines like SQLite where some values need to be tagged as unknown. In SQLite, all constants are considered to be of type unknown, unless the constant is fetched from a table. This is important at runtime, as a `1` might behave differently than `CAST(1 AS INT)`. For example, the expression `'0'` < 1 evaluates to 0. To facilitate this special case, the expression is elaborated to `'0'` $::\ ? < 1 ::\ ?$.

Rule (E$\mathtt{O}$) introduces explicit casts based on the operation and the best candidate type. Specifically, we insert casts for both operands to match their expected corresponding domain types and also insert a cast in the result of the operation to the expected codomain type. Certain engines, like SQLite, do not insert explicit casts for comparison operations. To achieve this, we use the *insert* abstract operator.

Rule (E$\mathtt{O}_\mathtt{S}$) elaborates set operations by inserting casts to the output type of biconv^*. This is done using the insert^* operation defined as

$$\mathit{insert}^*(\overline{e\ \mathtt{as}\ N}, \overline{N \mapsto \tau}, \mathtt{O}_\mathtt{S}) = \overline{\mathit{insert}(e, \tau, \mathtt{O}_\mathtt{S})\ \mathtt{as}\ N}$$

. For instance, for PSQL, query `(SELECT '1' FROM R) INTERSECT (SELECT 1 FROM R)` is elaborated to `(SELECT CAST('1' AS INT) FROM R) INTERSECT (SELECT CAST(1 AS INT) FROM R)`.

3.5 Dynamic Semantics

Figure 5 presents the dynamic semantics of TRAF, which differs from the ones of Guagliardo and Libkin [19] (GL) as follows. First, TRAF dynamic semantics are defined over a relational algebra, whereas GL uses SQL syntax to support

$$[\![R]\!]_{\mathtt{D},\Gamma} = \mathbf{ok}\ \mathtt{D}\,(R) \tag{RR}$$

$$[\![Q_1\ \mathtt{O_S}\ Q_2]\!]_{\mathtt{D},\Gamma} = \mathsf{do}\{t_1 \leftarrow [\![Q_1]\!]_{\mathtt{D},\Gamma}; t_2 \leftarrow [\![Q_2]\!]_{\mathtt{D},\Gamma}; \mathbf{ok}\ t_1\ \mathtt{O_S}\ t_2\} \tag{R$\mathtt{O_S}$}$$

$$[\![\pi_\beta(Q)]\!]_{\mathtt{D},\Gamma} = \mathsf{do}\{t \leftarrow [\![Q]\!]_{\mathtt{D},\Gamma}; \lceil \{\!\{ \underbrace{[\![\beta]\!]^r_{\eta_{\ell(\Gamma,Q)}}, \ldots, [\![\beta]\!]^r_{\eta_{\ell(\Gamma,Q)}}}_{\mathtt{k}\ \text{times}} \mid r \in_{\mathtt{k}} t \}\!\} \rceil\} \tag{Rπ}$$

$$[\![\sigma_\theta(Q)]\!]_{\mathtt{D},\Gamma} = \mathsf{do}\{t \leftarrow [\![Q]\!]_{\mathtt{D},\Gamma}; \_ \leftarrow \lceil \{\!\{ [\![\theta]\!]^r_{\eta_{\ell(\Gamma,Q)}} \mid r \in_{\mathtt{k}} t \}\!\} \rceil; \mathbf{ok}\ \{\!\{ \underbrace{r, \ldots, r}_{\mathtt{k}\ \text{times}} \mid r \in_{\mathtt{k}} t \wedge [\![\theta]\!]^r_{\eta_{\ell(\Gamma,Q)}} \}\!\}\} \tag{Rσ}$$

$$[\![\varepsilon(Q)]\!]_{\mathtt{D},\Gamma} = \varepsilon([\![Q]\!]_{\mathtt{D},\Gamma}) \tag{Rε}$$

$$[\![e\ \mathtt{as}\ N]\!]_\eta = [\![e]\!]_\eta \tag{Ras}$$

$$[\![\beta, e\ \mathtt{as}\ N]\!]_\eta = \mathsf{do}\{r \leftarrow [\![\beta]\!]_\eta; v \leftarrow [\![e]\!]_\eta; \mathbf{ok}\ r, v\} \tag{Rβ}$$

$$[\![N]\!]_\eta = \mathbf{ok}\ \eta(N) \tag{RN}$$

$$[\![v]\!]_\eta = \mathbf{ok}\ v \tag{Rv}$$

$$[\![v :: \tau]\!]_\eta = \begin{cases} \mathbf{ok}\ v' \ \text{if}\ \ \mathit{cast}(v,\tau) = v' \\ \mathbf{error}\ \text{otherwise} \end{cases} \tag{Rv ::}$$

$$[\![e :: \tau]\!]_\eta = \mathsf{do}\{v \leftarrow [\![e]\!]_\eta; [\![v :: \tau]\!]_\eta\}\ \mathsf{where}\ e \neq v \tag{Re ::}$$

$$[\![\theta_1\ \mathtt{O_B}\ \theta_2]\!]_\eta = \mathsf{do}\{b_1 \leftarrow [\![\theta_1]\!]_\eta; b_2 \leftarrow [\![\theta_2]\!]_\eta; \mathbf{ok}\ b_1\ \mathtt{O_B}\ b_2\} \tag{R$\mathtt{O_B}$}$$

$$[\![\neg\theta]\!]_\eta = \mathsf{do}\{b \leftarrow [\![\theta_1]\!]_\eta; \mathbf{ok}\ \neg b\} \tag{R$\neg$}$$

$$[\![e_1\ \mathtt{O}\ e_2]\!]_\eta = \mathsf{do}\{v_1 \leftarrow [\![e_1]\!]_\eta; v_2 \leftarrow [\![e_2]\!]_\eta; \mathbf{ok}\ \mathit{apply}(\mathtt{O}, v_1, v_2) \quad \mathtt{O} \in \mathtt{O_A} \cup \mathtt{O_C}\} \tag{R$\mathtt{O}$}$$

Fig. 5: Dynamic Semantics of TRAF. Abstract operators are highlighted in gray.

extra features. Second, and more importantly, as some casts may be invalid, the dynamic semantics must deal with the possibility of runtime errors. For instance, in PSQL the query `SELECT CAST('s' as INT) FROM R` ($\sigma_{\mathtt{'s'}::\mathbb{Z}}(R)$) evaluates to an error.

To concisely account for the possibility of runtime errors, we present the dynamic semantics in *monadic* style in order to streamline the handling of errors [37]. The monadic presentation of computations that may fail consists in so-called "optional" values: either an actual value tagged with **ok** , or **error** to denote an error. The sequential composition is given in a do block (e.g. $\mathsf{do}\{A; B; C\}$). Errors are transparently propagated through such sequences: the evaluation returns **ok** v if all steps in a do block (e.g. A, B, and C) evaluate successfully, or **error** if one step in the sequence evaluates to **error**.

There are four categories of evaluations : $[\![Q]\!]_{\mathtt{D},\Gamma}$ to reduce queries, $[\![e]\!]_\eta$ to reduce expressions, $[\![\beta]\!]_\eta$ to reduce aliased expressions, and $[\![\theta]\!]_\eta$ to reduce boolean expressions. The evaluation of a query is parametrized by a database D, which maps relation names R to tables t. We use $r \in_{\mathtt{k}} t$ to denote that r appears k times in t. The other categories of evaluation are parametrized by an environment η, which maps column names N to values v. Intuitively, this envi-

ronment is used to extract the value associated with a column name for a given row. For instance, consider a relation of persons $\texttt{P}(\texttt{Name} \mapsto \mathsf{String}, \texttt{Age} \mapsto \mathbb{Z})$, and a table $\{(\texttt{'Bob'}, \texttt{10}), (\texttt{'Alice'}, \texttt{20})\}$. Then, for the first row, $\eta(\mathsf{Name}) = \texttt{'Bob'}$ and $\eta(\mathsf{Age}) = \texttt{10}$.

Queries. The basic case is Rule (RR), which evaluates the name of a relation yielding the table associated to that name in database $\texttt{D}$. Rule (R$\texttt{O}_\texttt{S}$) evaluates set operations by first reducing the subqueries, then combining the resulting tables. Rule (Rπ) starts by evaluating subquery Q. If the result is successful (a table t), then for each row r that appears k times in t, we try to project columns as dictated by β, and duplicate the result k times. To do this, we evaluate β under environment $\eta^r_{\ell(\Gamma,Q)}$ (similarly to [19], $\eta^{r,v}_{\overline{N_i},N} = \eta^r_{\overline{N_i}}, N \mapsto v$ and $\eta^{\cdot}_{\cdot} = \cdot$). This environment is formed by matching corresponding column names of Q with the values of r. Finally, as the evaluation of aliased expressions can also produce errors, we lift the bag of optional rows $\texttt{S}$ to an optional bag of rows using the $\lceil\cdot\rceil$ function defined as: $\lceil\texttt{S}\rceil = \textbf{error}$, when $\textbf{error} \in \texttt{S}$; and $\textbf{ok}\ \{\!\{r \mid (\textbf{ok}\ r) \in \texttt{S}\}\!\}$ otherwise.

Rule (Rσ) follows a similar approach by first reducing the subquery Q. If the result is successful (yielding table t), we proceed to test the reduction of the condition θ for each row. We employ a strategy akin to that of (Rπ), but in this case, the resulting bag of booleans is unused (binding the resulting in variable "_"). This way, in the third instruction, we can be confident that the evaluation of the conditions does not result in errors. Finally, we filter the rows from table t that satisfy the given condition. The last rule for queries (Rε) removes duplicates from subquery using the function $\varepsilon(\cdot)$.

Expressions. Rule (Ras) evaluates a single aliased expression by evaluating the subexpression e and disregarding the name N. Rule (Rβ) applies when we are evaluating multiple aliased expressions. Initially, it recursively reduces the sublist to a row r, and then the head of the list to a value v, resulting in a new row r, v.

Rule (RN) successfully evaluates a column name N to its corresponding value in η. Rule (Rv) successfully evaluates values to themselves. Rule (Rv ::) attempts to cast a value into a value of a different type using the *cast* function. For instance, in SQLite, the expression `CAST('hi' as INT)` evaluates to `1`, whereas in PSQL is not defined. If the function is defined for the given value and type, the resulting value is returned; otherwise, an error is raised. Rule (Re::) applies to subexpressions that are not already values. It first reduces the subexpression to a value, and then casts the value using rule (Rv::).

Rules (R$\texttt{O}_\texttt{B}$), (R$\neg$), and (R$\texttt{O}$) operate in a similar fashion. First, each subexpression is reduced, then the resulting values are combined using the specific operation at hand. For arithmetic and comparison operations, the exact operation is performed using the *apply* operation. For instance, in PSQL, the expression $\texttt{'0'} < 1$ evaluates to true, whereas in SQLite, it yields false.

Basic definitions

$$ty(n) = \mathbb{Z} \quad ty(d) = \mathbb{R} \quad ty(s) = ?$$
$$insert(e, \tau, \mathsf{0}) = e :: \tau \qquad clean(T) = T[\mathsf{String}/?]$$
$$ty(\mathsf{0_c}) = \{\mathbb{Z} \times \mathbb{Z} \to \mathbb{B}, \mathbb{R} \times \mathbb{R} \to \mathbb{B}, \mathsf{String} \times \mathsf{String} \to \mathbb{B}\}$$
$$ty(+) = \{\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \mathbb{R} \times \mathbb{R} \to \mathbb{R}\}$$
$$apply(\mathsf{0}, v_1, v_2) = v_1\ \mathsf{0}\ v_2$$

$$\frac{}{\tau \Rightarrow \tau} \qquad \frac{}{\mathbb{Z} \Rightarrow \mathbb{R}}$$

$$\frac{\tau \Rightarrow \tau'}{e : \tau \rightsquigarrow \tau'}$$

$$\frac{icast(v, \tau) = v'}{v : ? \rightsquigarrow \tau}$$

$\boxed{biconv(e_1, \tau_1, e_2, \tau_2) = \tau}$

$$\frac{e_1 : \tau_1 \rightsquigarrow \tau_2}{biconv(e_1, \tau_1, e_2, \tau_2) = \tau_2} \qquad \frac{e_2 : \tau_2 \rightsquigarrow \tau_1}{biconv(e_1, \tau_1, e_2, \tau_2) = \tau_1}$$

$\boxed{icast(v, \tau) = v'}$

$$icast(\text{`}n\text{'}, \mathbb{R}) = n$$
$$icast(n, \mathbb{R}) = n$$
$$icast(v, ty(v)) = v$$
$$icast(\text{`}v\text{'}, ty(v)) = v \quad v \neq s$$

$\boxed{cast(v, \tau) = v'}$

$$cast(v, \mathbb{Z}) = \lfloor v \rfloor \quad v \in \mathbb{R}$$
$$cast(v, \mathsf{String}) = str(v) \quad v \neq s$$
$$cast(v, \tau) = icast(v, \tau)$$

$$\frac{\{m\} = \arg\min_i((cost(\tau_1, \tau_{1i})) + (cost(\tau_2, \tau_{2i}))) \quad \tau_1 \Rightarrow \tau_{1m} \quad \tau_2 \Rightarrow \tau_{2m}}{bestCandidate(\tau_1, \tau_2, \overline{\tau_{1i} \times \tau_{2i} \to \tau_{3i}}) = \tau_{3m} \times \tau_{4m} \to \tau_{3m}}$$

$$cost(\tau, \tau) = 0 \quad cost(\mathbb{Z}, \mathbb{R}) = 1 \quad cost(\mathsf{String}, \mathbb{Z}) = 1 \quad cost(\mathsf{String}, \mathbb{R}) = 1 \quad cost(\_, \_) = 2$$

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \mathsf{0}) = \tau}$

$$\frac{\tau_1 \neq ? \quad \tau_2 \neq ? \quad bestCandidate(\tau_1, \tau_2, ty(\mathsf{0})) = \tau_3 \times \tau_4 \to \tau_5}{resolve(e_1, \tau_1, e_2, \tau_2, \mathsf{0}) = \tau_3 \times \tau_4 \to \tau_5} \quad \ldots$$

Fig. 6: The TRAF/PSQL Instantiation (excerpt)

4 Instantiating TRAF

In this section we illustrate how to instantiate TRAF for PSQL and SQLite. The complete rules and the instantiations for three other engines (MSSQL, Oracle, and MySQL) can be found in the extended version. In general, an instantiation is achieved by providing specific definitions of the abstract operators that are engine dependent (the grey boxes in the figures). We obtained these definitions by exploring the documentation of each engine, and by conducting black-box analyses interacting with each engine whenever the documentation was lacking in details.

4.1 TRAF/PSQL

Figure 6 describes the TRAF/PSQL instantiation. PSQL is characterized by being a strongly-typed SQL engine, meaning that it is more conservative than the rest.

In many cases, if it cannot check the feasibility of casts, it rejects the query before its execution.

For the type of values, reals are typed as $\mathbb{R}$, integers to $\mathbb{Z}$, and string literals to ?. The operator for removing unknown types replaces ? occurrences with String, while leaving other types unchanged.[8] The operator for inserting annotations adds explicit casts regardless of the operation's type. The candidate types of a given operator is represented as sets of binary function types. Lastly, the semantics of operations between values are passed to the real implementation without any modifications.

Bidirectional implicit cast. Operator *biconv* is defined using the auxiliary *implicit type cast* relation $e : \tau \rightsquigarrow \tau'$. An expression e of type τ can be cast to τ' if either type τ can be implicitly cast to τ' ($\tau \Rightarrow \tau'$), or if e is a value v of type unknown ? (i.e. a string) and that value can be implicitly cast to some value v' under type τ' ($icast(v, \tau') = v'$).

Implicit type cast $\tau \Rightarrow \tau'$ is only defined between identical types $\tau \Rightarrow \tau$, or from integers to reals $\mathbb{Z} \Rightarrow \mathbb{R}$. For instance, `(SELECT 1 FROM R) INTERSECT (SELECT 1.0 FROM R)` is accepted and evaluates to $\{1.0\}$, since $\mathbb{Z}$ (the type of `1`) can be implicitly cast to a $\mathbb{R}$ (the type of `1.0`). Implicit value cast $icast(v, \tau) = v'$ is defined for extracting numbers from strings, but not viceversa.

Explicit cast. In addition to what an implicit value cast can do, explicit (value) casts $cast(v, \tau) = v'$ support casts from real numbers to integers by removing the decimals, and from non-string values to strings by enclosing them in quotes. For instance, `SELECT CAST('1.1' AS DOUBLE) FROM R` is accepted and evaluates to 1.1, but `SELECT CAST('1.1' AS INT) FROM R` is rejected by the type-checker.

Overload resolution. The definition of the *resolve* operator is divided in four cases. The general case arises when the types of the two expressions are not known. Function *bestCandidate* is used to determine the best candidate. We model this function by initially calculating the sum of the type differences between the corresponding types of the expression and the domain of each candidate. The type difference $\tau - \tau'$ quantifies the "cost" of changing type τ to τ': it yields a value of 0 if both types are identical, 1 when transitioning from an integer to a real or from a string to a number, and 2 in all other cases. If there are ties, and more than one type is selected, then the function is not defined. Furthermore, both types must satisfy the implicit cast relation with their corresponding type from the domain of the chosen best candidate. For instance, for query `SELECT 1 + 1.1 FROM R`, the `+` operation has two possible candidates: $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, and $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. The best candidate in this case is $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, as both operands can be implicitly cast to reals. The remaining cases are analogous. When only one type is unknown, the best candidate function is applied using the other known type in both positions. Additionally, the expression of unknown type is implicitly cast to the chosen type to rule out potential errors. Finally, if both types are unknown, they are assumed to be strings when looking for the best candidate. For example, in `SELECT '1' + 1.1 FROM R`, the best

[8] Notation $C[A/B]$ denotes type C where occurrences of B have been replaced by A.

Basic definitions

$$ty(v) = ? \quad dty(w :: ?) = ? \quad dty(r) = \mathbb{R} \quad dty(s) = \mathsf{String}$$
$$insert(e, \tau, \mathtt{0_c}) = e \quad insert\ (e, \tau, +) = e :: \tau$$
$$ty(\mathtt{0_c}) = \{? \times ? \to ?, \mathbb{R} \times \mathbb{R} \to ?, \mathsf{String} \times \mathsf{String} \to ?\}$$
$$ty(+) = \{\mathbb{R} \times \mathbb{R} \to ?\}$$
$$apply(+, v_1, v_2) = v_1 + v_2$$
$$apply(\mathtt{0_c}, v_1, v_2) = compare(|v_1'|, |v_2'|) :: ?$$
$$\text{where } resolve(v_1, dty(v_1), v_2, dty(v_2), ty(\mathtt{0_c})) = \tau_1 \times \tau_2 \to \tau_3,$$
$$icast(v_1, \tau_1) = v_1', icast(v_2, \tau_2) = v_2',$$
$$|w :: ?| = w \quad |w| = w$$

$\boxed{icast(v, \tau) = v'}$

$$\ldots$$
$$icast(s, \tau) = icast(number(s), \tau) \quad \tau \in \{\mathbb{Z}, \mathbb{R}\}$$
$$icast(s, \tau) = s \quad number(s) \text{ is not defined.}$$

$\boxed{cast(v, \tau) = v'}$

$$\ldots$$
$$cast(s, \tau) = cast(number(nprefix(s)), \tau) \quad \tau \in \{\mathbb{Z}, \mathbb{R}\}$$
$$cast(s, \tau) = 0 \quad number(nprefix(s)) \text{ is not defined}, \tau \in \{\mathbb{Z}, \mathbb{R}\}$$

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \mathtt{0}) = \tau}$

$$\frac{\{m\} = \arg\min_i((cost(\tau_1, \tau_{1i})) + (cost(\tau_2, \tau_{2i})))}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_{1i} \times \tau_{2i} \to \tau_{3i}}) = \tau_{3m} \times \tau_{4m} \to \tau_{3m}}$$
$$cost(\tau, \tau) = 0 \quad cost(\mathbb{Z}, \mathbb{R}) = 0 \quad cost(\mathsf{String}, \mathbb{R}) = 1 \quad cost(?, \tau) = 1 \quad cost(\_, \_) = 2$$

Fig. 7: The TRAF/SQLite Instantiation (excerpt)

candidate type is $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, as the implicit cast from `'1'` to real is possible. On the contrary, query `SELECT '1' + '1' FROM R` is rejected by the typechecker because there is more than one candidate available (int and real versions).

4.2 TRAF/SQLite

SQLite is one of the most flexible SQL engines: type enforcement is not mandatory, and types on columns are optional. However, SQLite attempts its best effort to perform casts without ever raising a type error at runtime. To capture this kind of flexibility, we model the type of values as ? and define all abstract operators as total functions. Figure 7 describes the TRAF/SQLite instantiation.

Statically, in SQLite the type of every value is unknown. We introduce a dynamic type operator $dty(v)$ to obtain precise type information during the evaluation of comparisons. The type of a value, initially unknown, may be refined at runtime to a more precise type. The operator for removing unknown types *clean* acts as the identity function, since unknown values have special meaning. For instance $1 :: ? < \texttt{'0'}$ is true, but $1 < \texttt{'0'}$ is false. The operator for

inserting annotations only inserts explicit casts for arithmetic operations, while for comparison operations, it behaves as the identity function.

The dynamic semantics for comparison is more involved. First, the best candidate type is searched, using the dynamic type information of the operands. Then, once the best candidate is found, both operands are implicitly cast to the corresponding types. Finally, the cast values, stripped of (potentially) casts to unknown, are compared using the $compare(w_1, w_2)$ function defined as 1 if $(dty(w_1) = dty(w_2) \wedge w_1 < w_2) \vee (dty(w_1) < dty(w_2))$; 0 otherwise. If the dynamic types of the operands are equal, then a regular comparison operation is performed. If the types are different then the types are compared using an arbitrary hierarchy such that $\mathbb{Z} = \mathbb{R} < \mathsf{String}$.

To illustrate, consider examples (1) `SELECT '0' < 1 FROM R`, and (2) `SELECT '0' < CAST(1 AS INT) FROM R`. The first example evaluates to `0`. Since the (Ev) elaboration rule inserts an explicit cast on every value, both values are cast to ?. Consequently, the chosen candidate for `<` is $? \times ? \to ?$, and the implicit cast leaves them untouched. Finally, as the dynamic type of both operands, with annotations removed, is String and $\mathbb{R}$ respectively, the comparison function yields `0` as result. The second example evaluates to `1`. In the process of elaboration, the left expression is cast to the unknown type, while the right expression is cast to an integer type. During the actual evaluation, the most suitable candidate is determined to be $\mathbb{R} \times \mathbb{R} \to ?$, which implies an implicit cast of both values into numerical values `0` and `1`, respectively. As both casted values share the same dynamic type, a standard comparison is carried out, resulting in `1`.

Bidirectional implicit cast. For the case of SQLite, the operator *biconv* always yields the unknown type for any pair of types and expressions. Here the relation is always defined. Consequently, for implicit casts from strings to numbers, when the string is not a valid number, the result is the same string.

Explicit cast feasibility. Operator $e : \tau \rightsquigarrow \tau'$ allows casting any expression of type τ to any type τ'.

Explicit cast. This operator is defined almost identically to implicit casts, except when the expression is a string. In this case, the cast is performed by extracting the largest numeric prefix from the string and then casting it to the required number type. If there is no numeric prefix, then the cast yields 0. For instance, `SELECT CAST('12.3hi' AS INT), CAST('hi') FROM R` evaluates to $(12, 0)$.

Overload resolution. There is only one rule for overloading resolution *resolve*. It yields the first best candidate found, using the *type difference* operator. Type difference yields 0 when the types are the same or when converting from integer to real, and 1 when converting either from an unknown type to any other type or from string to real.

4.3 SQLite ↔ PSQL Translation Examples

Given the design of TRAF/PSQL and TRAF/SQLite, we now illustrate some examples of SQL query translations between their corresponding database engines.

We show the effect of understanding the type semantics of each engine to justify translations that might seem counterintuitive.

From SQLite *to* PSQL Consider example E3, `SELECT '1.1' + 1 FROM R`. This example runs successfully in SQLite and yields $\{\!\{2.1, 2.1, 2.1\}\!\}$. This is expected due to the candidates $ty(+) = \{\mathbb{R} \times \mathbb{R} \to ?\}$, $resolve(\texttt{'1.1'}, ?, \texttt{1}, ?, ty(+)) = \mathbb{R} \times \mathbb{R} \to ?$, and

$$\begin{aligned} cast(\texttt{'1.1'}, \mathbb{R}) &= cast(number(nprefix(\texttt{'1.1'})), \mathbb{R}) \\ &= cast(number(\texttt{'1.1'}), \mathbb{R}) \\ &= cast(\texttt{1.1}, \mathbb{R}) = 1.1 \end{aligned}$$

However, this query does not typecheck in PSQL. Addition is only defined for numeric values ($ty(+) = \{\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \mathbb{R} \times \mathbb{R} \to \mathbb{R}\}$), and $resolve(\texttt{'1.1'}, ?, \texttt{1}, \mathbb{Z}, ty(+))$ requires $icast(\texttt{'1.1'}, \mathbb{Z})$ to be defined (which it is not). What is defined is the implicit cast to a real $icast(\texttt{'1.1'}, \mathbb{R})$. We can force such cast with the following translation that preserves the same behavior: `SELECT CAST('1.1' AS DECIMAL(1)) + 1 FROM R`.

Other examples such as E10, `SELECT 1 FROM R WHERE '1' < 2`, are more challenging. In SQLite, this query yields an empty result because strings are considered larger than numbers. A straightforward translation to PSQL that maintains this behavior is `SELECT 1 FROM R WHERE False`. However, a more general approach involves following the *compare* function, performing type testing using `pg_typeof` and then applying dynamic casts accordingly. For instance, the comparison `a < b` could be translated to:

```
(pg_typeof(a) = pg_typeof(b) AND
  CAST(a AS pg_typeof(a)) < CAST(b AS pg_typeof(b))) OR
  (pg_typeof(a) = 'number' AND pg_typeof(b) = 'text')
```

However, dynamic casts such as `CAST(a AS pg_typeof(a))` are not supported natively by PSQL.

From PSQL *to* SQLite Consider once again example E10, `SELECT 1 FROM R WHERE '1' < 2`, but now in the opposite direction. In PSQL, this query returns $\{\!\{1, 1, 1\}\!\}$ (the best candidate type is $\mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$). Translating this query to SQLite requires mimicking the comparison behavior of PSQL. According to the *compare* function, both arguments need to have the same dynamic type. This can be achieved either by casting the left operand: `SELECT 1 WHERE CAST('1' AS INT) < 2`, or surprisingly, by casting the right operand: `SELECT 1 WHERE '1' < CAST(2 as INT)`. By casting the right operand, *resolve* chooses $\mathbb{R} \times \mathbb{R} \to ?$ as best candidate, thus implicitly casting both operands to a number.

Future work may involve an automatic translation mechanism, which takes into account their type semantic and operational differences. Such mechanism would significantly reduce the manual effort required to adapt queries and help ensure consistency.

R1: Every operator must be deterministic.
R2: $biconv(e_1, \tau_1, e_2, \tau_2)$ yields either τ_1 or τ_2.
R3: $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3})$ must be contained in $\overline{\tau_3}$.
R4: If $cast(v, \tau) = v'$, then the (cleaned) type of v' must be τ.
R5: If $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6$, $[\![e_1]\!]_\eta \neq \textbf{error}$ and $[\![e_2]\!]_\eta \neq \textbf{error}$ then $[\![e_1 :: \tau_4]\!]_\eta \neq \textbf{error}$ and $[\![e_2 :: \tau_5]\!]_\eta \neq \textbf{error}$.
R6: If $biconv(e_1, \tau_1, e_2, \tau_2) = \tau$, $[\![e_1]\!]_\eta \neq \textbf{error}$ and $[\![e_2]\!]_\eta \neq \textbf{error}$ then $[\![e_1 :: \tau]\!]_\eta \neq \textbf{error}$ and $[\![e_2 :: \tau]\!]_\eta \neq \textbf{error}$.
R7: If $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6$ then $e_1 : \tau_1 \rightsquigarrow \tau_4$ and $e_2 : \tau_2 \rightsquigarrow \tau_5$.
R8: If $biconv(e_1, \tau_1, e_2, \tau_2) = \tau$ then $e_1 : \tau_1 \rightsquigarrow \tau$ and $e_2 : \tau_2 \rightsquigarrow \tau$.
R9: Either $clean(T) = T[\mathsf{String}/?]$ or $clean(T) = T$.
R10: If $apply(\mathtt{O}, v_1, v_2)$ then given values of the right types, primitive functions, such as boolean or arithmetic operations, will never fail.

Fig. 8: Requirements that abstract operator must meet to satisfy the properties.

5 Properties

Based on the core relational algebra of TRAF, we can establish metatheoretical results for each formalized engine, consisting of lemmas and theorems regarding queries and their evaluation. For simplicity, we will refer to each engine by the name of its corresponding TRAF formalization. Specifically, we can articulate the formal distinctions among various engines, and pinpoints the exact requirements that abstract operator (e.g. for new engines) must meet to satisfy the properties. This aids us in better comprehending the process of transforming queries from one engine to another.

To state these theorems, we need several new definitions, in particular a way to typecheck rows r, tables t and databases $\mathtt{D}$:

$$\frac{\vdash v : \tau}{\vdash v : (N \mapsto \tau)} \qquad \frac{\vdash r : T \qquad \vdash v : \tau}{\vdash r, v : T, (N \mapsto \tau)} \qquad \frac{}{\vdash \cdot : T}$$

$$\frac{\forall r \in t. \vdash r : T}{\vdash t : clean(T)} \qquad \frac{\forall R \in dom(\mathtt{D}). \vdash \mathtt{D}(R) : \Gamma(R)}{\vdash \mathtt{D} : \Gamma}$$

A row is well-typed if every value is typed to its corresponding type in T. A table is typed T if every row is typed as T. An empty row is typed to any relation type T. A database is typed Γ, if every relation in $\mathtt{D}$ is typed to its corresponding type in Γ. Note that these rules are non-deterministic, so any value can be associated to any name. The proofs require some properties about the implementation of abstract operators shown in Figure 8.

Assuming R1, R2 and R3, any instantiation of TRAF is *type safe*, meaning that well-typed queries either reduce to a table or raise a controlled type error, i.e., an error captured and raised by the language upon detecting an inconsistency. In other words, the evaluation of well-typed queries does not raise uncontrolled errors, such as getting stuck. For instance, an ill-type query, or

`SELECT Foo FROM P`, where `Foo` is not defined in `P`, gets stuck as $[\![\texttt{Foo}]\!]_\eta$ does not evaluate further.

Theorem 1 (Type Safety). *If R1, R2 and R3 hold,* $\forall T\ Q\ \texttt{D}$, *if* $\Gamma \vdash Q : T$ *and* $\vdash \texttt{D} : \Gamma$ *then* $(\exists t, [\![Q]\!]_{\texttt{D},\Gamma} = t) \vee ([\![Q]\!]_{\texttt{D},\Gamma} = \textbf{error})$.

Assuming also R4, a stronger theorem called *type soundness* states that, in addition to type safety, the resulting table indeed has the type of the query.

Theorem 2 (Type Soundness). *If R1, R2, R3 and R4 hold,* $\forall T\ Q\ \texttt{D}$, *if* $\Gamma \vdash Q : T$ *and* $\vdash \texttt{D} : \Gamma$ *then* $(\exists t, T'.[\![Q]\!]_{\texttt{D},\Gamma} = t, \vdash t : T'$ *and* $clean(T') = clean(T)) \vee ([\![Q]\!]_{\texttt{D},\Gamma} = \textbf{error})$.

The use of $clean(\cdot)$ is exclusively for PSQL, and for cases such as `SELECT CAST('hi' as String) as A from R`. This query of type $A \mapsto \mathsf{String}$ evaluates to $\{\texttt{'hi'}, ...\}$, typed as $A \mapsto ?$ (literal strings are typed ?), but $clean(\mathsf{String}) = clean(?) = \mathsf{String}$.

Type safety is satisfied by every engine we consider, but type soundness is satisfied by all except SQLite. This is because SQLite does not satisfy R4. For instance, in SQLite, `SELECT CAST(1 as INT) AS A FROM R` has type $A \mapsto \mathbb{Z}$, but its evaluation is typed $A \mapsto ?$. Also, SQLite permits storing string values in integer columns.

Moreover, PSQL, MySQL and SQLite satisfy a theorem that states that if the programmer does not use any explicit cast in a well-typed query, then the query evaluates without error. To state this theorem we use the cast-free metafunction $\texttt{CF}(Q)$, which is defined when Q does not have explicit casts of the form $e :: \tau$ (definition in the extended version).

Theorem 3 (Cast-free queries do not fail). *If R1, R5, R6, R9 and R10 hold,* $\texttt{CF}(Q)$, $\vdash \texttt{D} : \Gamma$ *and* $\Gamma \vdash Q : T \rightsquigarrow Q'$ *then* $[\![Q']\!]_{\texttt{D},\Gamma} \neq \textbf{error}$.

Neither MSSQL nor Oracle satisfy this property. Specifically, MSSQL does not satisfy R5 and R6, and Oracle does not satisfy R5. To illustrate why, let us consider table $R = \{(\texttt{'1'})\}$, schema $R \mapsto (A \mapsto \mathsf{String})$, and query `SELECT A + 1 FROM R`. This query does not typecheck in PSQL, and evaluates successfully in other engines. But with one more row to R: $\{(\texttt{'1'}), (\texttt{'hi'})\}$, the same query evaluates to a runtime error in MSSQL and Oracle.

Regarding cast insertion, the type of query translation, and the translation is unique:

Theorem 4 (Cast insertion is a type-preserving function). *If R1, R2, R4, R7, R8 and R9 hold, and* $\Gamma \vdash Q : T$ *then there exists a unique* Q' *such that* $\Gamma \vdash Q : T \rightsquigarrow Q'$ *and* $\Gamma \vdash Q' : T$.

This theorem is satisfied by the five engines we studied.

6 Experimental Validation

To validate the adequacy of the formalism and its instantiations, we adopt approaches similar to those used for the validation of formal models of Python [30], JavaScript [29], and more closely related to our work, SQL [19], by testing against real-world implementations. We develop PyTRAF, an implementation of TRAF in Python, and create one instance for each of five engines. We generate multiple random queries and verify that the results from the actual engine match those obtained from the prototype.

We generate a total of 100,000 random SQL queries for each engine, successfully confirming that our design aligns with the behavior of each individual engine. This process is challenging when dealing with engine-specific query optimizations. In particular, sometimes PyTRAF reports an error while the engine returns a table. This discrepancy occurs due to avoidance of executing certain subexpressions or subqueries that are prone to failure. For this reason, we divided the validation in two categories: a *termination-insensitive validation*, and a *termination-sensitive validation*[9].

The termination-insensitive validation approach involves verifying that, if the evaluation of a query in PyTRAF and in the engine result in tables, then these tables must be equivalent. In PyTRAF, the query generation is parameterized by the engine due to subtle discrepancies between engines For instance, MSSQL and Oracle lack a boolean type and represent booleans using integers. To avoid floating number precision mismatches, real numbers are represented as decimals in both PyTRAF and the engines. Note that comparing real numbers using a notion of closeness might be feasible, but it presents a greater challenge when these results are then cast to strings. Finally, in MySQL, we had to cast some operands of arithmetic operators to decimal to avoid precision issues. In addition, we check whether a query that succeeds in PyTRAF will also succeed in the real engine. We have observed that this is true for MySQL, MSSQL and SQLite, but not for PSQL and Oracle. The reason for this discrepancy is that some engines perform optimizations that affect the evaluation order, eagerly casting aliased subexpressions in subqueries whose condition is always false, leading to unsound results in the presence of effects such as cast errors. In other words, the optimizations performed by the engines are sound only for "pure" queries (those that do not fail). However, the impact of these optimizations on erroneous queries appears to be overlooked by engine providers.

The termination sensitive validation approach is a stronger result. It involves verifying that, if the evaluation of a query in PyTRAF yields a table, then evaluation of the query in the engine results in an equivalent table. Furthermore, if a query in PyTRAF reports an error, then the query in the engine also reports an

[9] The names "termination sensitive" (TS) and "termination insensitive" (TI) are borrowed from hyper-properties such as noninterference (NI). In NI, TI-NI means that NI holds only when both executions terminate successfully, while TS-NI means TI-NI plus equitermination. Therefore, TI validation means that the engine and model coincide whenever they both don't fail, while TS validation means that if one fails, the other must fail as well.

error. It is important to note that sometimes distinguishing between errors resulting from type checking or evaluation solely by inspecting the engine's output might not be feasible. Consequently, if PyTRAF reports a type error (either statically or dynamically), we verify that the real engine throws any kind of errors. To achieve this stronger validation, we had to perform several simplifications (explained in the extended version) on the generation of queries because some query optimizations prevent certain sub-expressions from being evaluated.

7 Related Work

Traditionally SQL has been implemented with some sort of either static typing or syntactic checking, though the issue of type errors and type disciplines has received little attention. Nonetheless, there are two lines of works that relate to this work. In the Databases literature, the consideration of corner cases such as NULLs and dynamically generated queries involves typing issues. Also, some engines, like SQLite have "flexible" type systems. In Programming Languages, type systems are a central topic, but SQL and databases have received little attention. Both areas have been sources of inspiration and techniques for our work.

Classical database literature. There are many works formalizing SQL [8–10, 27]. Guagliardo and Libkin [19] developed a comprehensive formal semantics for SQL whose core we follow here. Following the classic framework in the area, they assume that all comparisons and operations are applied to arguments of the right types. Therefore essentially they do not deal with typing issues. Regarding errors in SQL, based on previous work [2, 33, 39], Taipalus *et al.* [35] review SQL errors to build a unified error categorization. In further work, Taipalus *et al.* [34] compares the error messages of the four most popular relational database management systems (MySQL, Oracle, PostgreSQL, and SQL Server) in terms of error message effectiveness, effects, and usefulness, and error recovery confidence. Our work does not deal with error messages, but instead with detecting errors. Finally, regarding formalization, Benzaken et al. Benzaken and Contejean [4] provide a Coq mechanised, executable, formal semantics for a realistic fragment of SQL. Their coq formalization covers null values, functions, aggregates, quantifiers and nested potentially correlated sub-queries. Ricciotti and Cheney [31] complement and deepens the work of Guagliardo and Libkin [19] by making the notions of their semantics and proof precise and formal using Coq. [5] propose the first mechanically verified compiler (DBCert, using Coq) for SQL queries. These works assume the precise matching of types and therefore there are no implicit casts.

Flexible typing databases. A distinctive example is SQLite, the most widely deployed database engine [21], which enjoys flexible typing. Data of any type may be stored in any column of an SQLite table (except an INTEGER PRIMARY KEY column, in which case the data must be integral) and columns can be declared without any data type [12]. SQL queries are viewed as strings and little error checking is done for dynamically-generated SQL query strings.

Wassermann *et al.* [38] propose a static program analysis technique to verify that dynamically-generated query strings do not contain type errors. Similar to TRAF, they employ a type system to reject invalid dynamically-generated queries. However, their focus does not lie in the formalism of dynamic semantics or casts, and the evaluation uses the grammar of Oracle.

Strongly-typed queries. The development of programming language libraries and tools for type-checking queries has been extensively explored [3, 13–16, 24–26, 32]. However, the formalization of implicit and explicit type casts has not been addressed. Additionally, these studies lack a practical exploration of the varied behaviors induced by typing in industry-standard database engines. From a formal perspective, significant progress has been made in the area of type inference for relational algebra [7, 28, 36] and SQL [11, 23]. In TRAF, we presume the existence of a typed schema, and consider the definition of type inference as an area for future exploration.

8 Conclusion

In this paper, we identify some discrepancies in behavior regarding the handling of types both statically and dynamically in current SQL engines. This presents practical problems (e.g. when porting queries among engines) that are challenging to address.

We demonstrated that addressing this issue is feasible by integrating a lightweight typing system. Indeed, we present TRAF, a formal framework for a typed relational algebra with support for implicit and explicit type casts. TRAF permits to formally understand the behavior of different database engines; we validate this expressiveness by providing five different instantiations.

Our framework highlights the necessary requirements for any concrete instantiation to satisfy formal properties such as type safety and soundness, among others. The typing discrepancies addressed shed light that certain apparently minor design decisions of engines may lead to major changes in behavior. As future work, we believe this initial step that constitutes our work should be extended to deal with discrepancies under query optimizations performed by many practical engines. It would also be valuable to develop a technique that standardizes query behavior according to a specific database semantics model, by inserting sufficient casts or other forms of disambiguation so that the query runs correctly across different databases. Additionally, we could extend the scope to encompass the casting behavior of various types; for example, how one system handles casting an integer or float to a less precise decimal type may differ from the approach of another system.

References

1. Converting mysql to postgresql (2020), `https://en.wikibooks.org/wiki/Converting_MySQL_to_PostgreSQL`

2. Ahadi, A., Behbood, V., Vihavainen, A., Prior, J., Lister, R.: Students' syntactic mistakes in writing seven different types of sql queries and its application to predicting students' success. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education. p. 401–406. SIGCSE '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2839509.2844640, https://doi.org/10.1145/2839509.2844640
3. Augustsson, L., Ågren, M.: Experience report: Types for a relational algebra library. SIGPLAN Not. **51**(12), 127–132 (sep 2016). https://doi.org/10.1145/3241625.2976016, https://doi.org/10.1145/3241625.2976016
4. Benzaken, V., Contejean, E.: A coq mechanised formal semantics for realistic sql queries: Formally reconciling sql and bag relational algebra. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 249–261. CPP 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293880.3294107, https://doi.org/10.1145/3293880.3294107
5. Benzaken, V., Contejean, E., Hachmaoui, M.H., Keller, C., Mandel, L., Shinnar, A., Siméon, J.: Translating canonical sql to imperative code in coq. Proc. ACM Program. Lang. **6**(OOPSLA1) (apr 2022). https://doi.org/10.1145/3527327, https://doi.org/10.1145/3527327
6. Bhandari, H., Chitrakar, R.: Comparison of data migration techniques from sql database to nosql database. J Comput Eng Inf Technol 9 **6**, 2 (2020)
7. Buneman, P., Ohori, A.: Polymorphism and type inference in database programming. ACM Trans. Database Syst. **21**(1), 30–76 (mar 1996). https://doi.org/10.1145/227604.227609, https://doi.org/10.1145/227604.227609
8. Ceri, S., Gottlob, G.: Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. IEEE Transactions on Software Engineering **SE-11**, 324–345 (1985), https://api.semanticscholar.org/CorpusID:22717180
9. Chu, S., Wang, C., Weitz, K., Cheung, A.: Cosette: An automated prover for sql. In: Conference on Innovative Data Systems Research (2017), https://api.semanticscholar.org/CorpusID:12408033
10. Chu, S., Weitz, K., Cheung, A., Suciu, D.: Hottsql: proving query rewrites with univalent sql semantics. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (2016), https://api.semanticscholar.org/CorpusID:644867
11. Colazzo, D., Sartiani, C.: Precision and complexity of xquery type inference. In: Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming. p. 89–100. PPDP '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2003476.2003490, https://doi.org/10.1145/2003476.2003490
12. Gaffney, K.P., Prammer, M., Brasfield, L.C., Hipp, D.R., Kennedy, D.R., Patel, J.M.: Sqlite: Past, present, and future. Proc. VLDB Endow. **15**, 3535–3547 (2022), https://api.semanticscholar.org/CorpusID:252066674
13. Gould, C., Su, Z., Devanbu, P.T.: JDBC checker: A static analysis tool for SQL/JDBC applications. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) 26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom. pp. 697–698. IEEE Computer Society (2004). https://doi.org/10.1109/ICSE.2004.1317494, https://doi.org/10.1109/ICSE.2004.1317494
14. Group, T.D.D.: Typechecking queries (2019), https://tpolecat.github.io/doobie/docs/06-bChecking.html

15. Group, T.K.D.: Kysely (2021), https://kysely.dev/
16. Group, T.P.D.: Pgtyped (2020), https://github.com/adelsz/pgtyped
17. Group, T.P.G.D.: Postgresql documentation (1996), https://www.postgresql.org/docs/current/typeconv-boverview.html
18. Group, T.S.D.: Sqlines (2010), http://www.sqlines.com/online
19. Guagliardo, P., Libkin, L.: A formal semantics of sql queries, its validation, and applications. Proc. VLDB Endow. **11**(1), 27–39 (sep 2017). https://doi.org/10.14778/3151113.3151116, https://doi.org/10.14778/3151113.3151116
20. Haas, S.W.: Erik peter bansleben . database migration : A literature review and case study (2004), https://api.semanticscholar.org/CorpusID:17518212
21. Hipp., D.R.: Most widely deployed and used database engine. https://www.sqlite.org/mostdeployed.html
22. Khan, S., Kalia, A., Dastjerdi, H.M., Nizamuddin, N.: Automated tool for nosql to sql migration. In: Proceedings of the 7th International Conference on Information Systems Engineering. p. 20–23. ICISE '22, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3573926.3573931, https://doi.org/10.1145/3573926.3573931
23. Lin, W.: Type inference in SQL. Ph.D. thesis, Concordia University (2004)
24. Marlow, S., et al.: Haskell 2010 language report. Available online http://www.haskell. org/(May 2011) (2010)
25. MIT: Ts-sql-query (2019), https://ts-bsql-bquery.readthedocs.io/
26. Necco, C.M., Nuno Olivera, J.: Toward generic data processing. In: XI Congreso Argentino de Ciencias de la Computación (2005)
27. Negri, M., Pelagatti, G., Sbattella, L.: Formal semantics of sql queries. ACM Trans. Database Syst. **16**(3), 513–534 (sep 1991). https://doi.org/10.1145/111197.111212, https://doi.org/10.1145/111197.111212
28. Ohori, A., Buneman, P.: Type inference in a database programming language. In: Proceedings of the 1988 ACM Conference on LISP and Functional Programming. p. 174–183. LFP '88, Association for Computing Machinery, New York, NY, USA (1988). https://doi.org/10.1145/62678.62700, https://doi.org/10.1145/62678.62700
29. Park, D., Stefanescu, A., Rosu, G.: KJS: a complete formal semantics of javascript. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 346–356. ACM (2015). https://doi.org/10.1145/2737924.2737991, https://doi.org/10.1145/2737924.2737991
30. Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: the full monty. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013. pp. 217–232. ACM (2013). https://doi.org/10.1145/2509136.2509536, https://doi.org/10.1145/2509136.2509536
31. Ricciotti, W., Cheney, J.: A formalization of sql with nulls. J. Autom. Reason. **66**(4), 989–1030 (nov 2022). https://doi.org/10.1007/s10817-b022-b09632-b4, https://doi.org/10.1007/s10817-b022-b09632-b4
32. Silva, A., Visser, J.: Strong types for relational databases. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell. p. 25–36. Haskell '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1159842.1159846, https://doi.org/10.1145/1159842.1159846

33. Smelcer, J.B.: User errors in database query composition. Int. J. Hum.-Comput. Stud. **42**(4), 353–381 (apr 1995). https://doi.org/10.1006/ijhc.1995.1017, https://doi.org/10.1006/ijhc.1995.1017
34. Taipalus, T., Grahn, H., Ghanbari, H.: Error messages in relational database management systems: A comparison of effectiveness, usefulness, and user confidence. Journal of Systems and Software **181**, 111034 (2021). https://doi.org/https://doi.org/10.1016/j.jss.2021.111034, https://www.sciencedirect.com/science/article/pii/S016412122100131X
35. Taipalus, T., Siponen, M., Vartiainen, T.: Errors and complications in sql query formulation. ACM Trans. Comput. Educ. **18**(3) (aug 2018). https://doi.org/10.1145/3231712, https://doi-borg.eproxy.lib.hku.hk/10.1145/3231712
36. Van den Bussche, J., Waller, E.: Polymorphic type inference for the relational algebra. Journal of Computer and System Sciences **64**(3), 694–718 (2002). https://doi.org/https://doi.org/10.1006/jcss.2001.1812, https://www.sciencedirect.com/science/article/pii/S0022000001918124
37. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science **2**, 461–493 (1992)
38. Wassermann, G., Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. ACM Trans. Softw. Eng. Methodol. **16**(4), 14–es (sep 2007). https://doi.org/10.1145/1276933.1276935, https://doi.org/10.1145/1276933.1276935
39. Welty, C.: Correcting user errors in sql. International Journal of Man-Machine Studies **22**(4), 463–477 (1985)