A Formal Framework for Typing and Cast Semantics in SQL Engines

WENJIA YE, The University of Hong Kong, China and University of Bristol, United Kingdom
MATÍAS TORO, PLEIAD Lab, Computer Science Department, University of Chile & IMFD, Chile
CLAUDIO GUTIERREZ, Computer Science Department, University of Chile & IMFD, Chile
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China
ÉRIC TANTER, PLEIAD Lab, Computer Science Department, University of Chile & IMFD, Chile

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 address 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 systematically model the type and cast behavior of different SQL engines (PostgreSQL, MS SQL Server, MySQL, SQLite, and Oracle).

We show that this formalism provides formal guarantees for the handling of types. We identify practical conditions under which engine instantiations satisfy type safety and type soundness. In this regard, TRAF can serve as an explicit and testable account of typing in existing engines, potentially guide their evolution, and provide a formal basis to study type-aware query optimizations and design provably-correct query translators. We also test the adequacy of the formalism by implementing TRAF in Python for these five engines and testing it with synthetic queries, SQLancer++-generated queries, and adapted Spider and Calcite benchmark queries.

CCS Concepts: • **Theory of computation → Type structures**; • **Information systems → Query languages**; • **Software and its engineering → Semantics**.

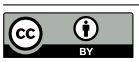
Additional Key Words and Phrases: SQL, Typing Semantics, Databases

ACM Reference Format:
Wenjia Ye, Matías Toro, Claudio Gutierrez, Bruno C. d. S. Oliveira, and Éric Tanter. 2026. A Formal Framework for Typing and Cast Semantics in SQL Engines. *ACM Trans. Program. Lang. Syst.* 1, 1 (January 2026), 43 pages. https://doi.org/10.1145/3834861

1 Introduction

Query translation between different SQL engines is common in different scenarios, such as database migration (to reduce costs, maintenance effort, software changes, etc.) [Bansleben 2004; Bhandari and Chitrakar 2020; Khan et al. 2023] and prototyping (developing code against lightweight databases such as SQLite and then porting it to a more robust database). Today there are many tools addressing this task [Mysql 2020; RebaseData 2023; SQLines 2010].

Authors' Contact Information: Wenjia Ye, The University of Hong Kong, Hong Kong, China and University of Bristol, Bristol, United Kingdom, yewenjia@outlook.com; Matías Toro, PLEIAD Lab, Computer Science Department, University of Chile & IMFD, Santiago, Chile, mtoro@dcc.uchile.cl; Claudio Gutierrez, Computer Science Department, University of Chile & IMFD, Santiago, Chile, cgutierr@dcc.uchile.cl; Bruno C. d. S. Oliveira, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk; Éric Tanter, PLEIAD Lab, Computer Science Department, University of Chile & IMFD, Santiago, Chile, etanter@dcc.uchile.cl.



ACM 1558-4593/2026/1-ART
https://doi.org/10.1145/3834861

Table 1. Examples of discrepant behaviors of different database engines, considering the table `R(A,B) = {('Bob',10),('1',20),('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.

	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 FROM R) 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`	✗	↛	✗	∅	∅

Translation between SQL engines can yield surprising semantic discrepancies. In this paper, we focus on those related to typing behavior. These discrepancies are mainly due to differences in datatypes, typechecking, when checks are performed, and explicit and implicit type casts that may or may not be applied 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.[1] This issue is not merely a corner of the SQL standard: it directly affects porting and query translation, because a syntactically valid translation can still change whether a query is rejected statically, fails at runtime, or silently returns a different table. A type-aware formalization can therefore be used both diagnostically, to explain which engine-specific casts are being relied on, and constructively, to guide where an explicit cast must be inserted or where behavior cannot be preserved in the target engine.

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 second yields 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 treat types differently, following different design models: some are more statically typed while others embrace dynamic typing, and they also differ in how they overload basic arithmetic and comparison operations.

Understanding and addressing these anomalies is not a simple task. A first issue is that SQL standards either do not cover many typing-related issues or leave their interpretation rather open.[2]

[1] Some examples: Wikibooks MySQL-to-PostgreSQL guide, SQLines online converter, and RebaseData MySQL-to-PostgreSQL converter.

[2] 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 TD [target value] that does not lose any leading

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 [Guagliardo and Libkin 2017; Ricciotti and Cheney 2022], they assume that all comparisons and operations apply to the right types. Typing in SQL has been explored [Augustsson and Ågren 2016; Gould et al. 2004; Marlow et al. 2010; Necco and Nuno Olivera 2005; Silva and Visser 2006]. 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 type-related discrepancies in 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 [Ceri and Gottlob 1985; Guagliardo and Libkin 2017; Negri et al. 1991]) 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, nulls, selections, cross-products, nested queries, and set operations. We also add support for arithmetic operators and type casts. Although features such as aggregation 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 (Section 3.6), 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).[3] This typechecker is responsible for identifying mismatches between types and the number of columns of subqueries in set operations, and ensuring 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 is the bridge between static and dynamic behavior: it records the implicit casts selected by typechecking as explicit casts in the target relational algebra term. This makes the subsequent evaluator simple and exposes the remaining data-dependent failures, such as a column cast that is permitted statically but fails on a particular string value at runtime.

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

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))

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

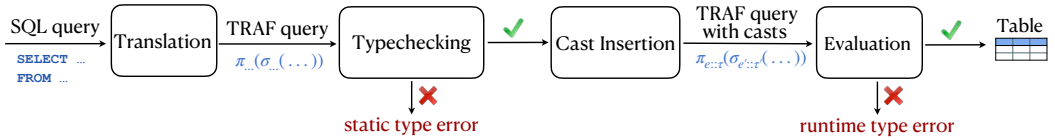


Fig. 1. Overview of TRAF. Some abstract operators in the typechecking, cast insertion, and evaluation phases must be instantiated to implement a particular engine. We provide sample instantiations for PSQL, MSSQL, Oracle, MySQL and SQLite.

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.

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 formulate and prove properties satisfied by all or some engines (Section 5). First, we prove a type safety result ensuring 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 strengthen this result by proving that the resulting table can be typed with the same type as the query. Third, the instantiations of the model for PSQL, MySQL, and SQLite satisfy a theorem stating that, if the programmer does not use explicit casts in well-typed queries, then the queries 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 validate semantics by testing them against real-world implementations [Guagliardo and Libkin 2017; Park et al. 2015; Politz et al. 2013], we developed multiple database interpreters and tested them with synthetic queries generated by our grammar, SQLancer++-generated queries [Zhong and Rigger 2026], and adapted Spider [Yu et al. 2018] and Calcite [Begoli et al. 2018] benchmark queries, comparing their results with those obtained from actual database engines. In addition, we highlight the impact and difficulties posed by engine-specific query optimizations, especially those that break behavioral equivalence in the presence of errors.

In summary, our contributions include the identification 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 appendix. We also made a prototype implementation, PyTRAF, available[4].

Prior publication. This article is an extension of a prior conference article [Ye et al. 2025b]. The extended version expands and refines the contributions in the following ways:

(1) The TRAF framework now supports existential expressions (`EXISTS`), membership expressions (`IN`), and support for `NULL` values, clarifying their behavior in the operational semantics. These extensions are not isolated syntax additions: `EXISTS` and `IN` require subquery typing and evaluation under outer environments, while `NULL` interacts with three-valued logic and with optimizer rewrites that can hide or expose runtime type errors. These features were missing in

[4]https://github.com/YeWenjia/Elucidating-Type-Conversions-in-SQL-Engines

the conference version, and now TRAF supports the same feature set considered by Guagliardo and Libkin [2017].
(2) We present the complete set of relations used in the formal development, which were only partially included in the conference version. Likewise, this version includes the full translation rules from the surface language to the core calculus, providing a complete and systematic description of the compilation process.
(3) We add an Oracle instantiation of the framework to illustrate the applicability of TRAF to an additional real-world database engine.
(4) This version provides a more detailed discussion of the limits of the experimental validation, including clearer statements of the effect of engine optimizations and a table that illustrates these limits. It also broadens the workload with adapted Spider and Calcite queries and SQLancer++-generated tests, complementing the original TRAF-specific random generator. We also identify several optimizations that break query behavioral equivalence in the presence of errors.
(5) Finally, the article expands on several explanations and includes additional examples to improve clarity and accessibility.

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-related semantic discrepancies

In this section we explain the discrepancies in Table 1, which serve as a starting point for the development and justification of TRAF. For readability, we restate the relevant query in the heading of each example paragraph. For clarity, we categorize the examples of discrepancies into three groups: arithmetic, boolean, and set operations.

Throughout the paper, "real" refers to the SQL `REAL`/floating-point family rather than to mathematical real numbers; concrete DBMSs choose implementation-dependent precision and often expose related types such as `FLOAT`, or `DECIMAL`. Our core type language intentionally abstracts from width and precision parameters such as `VARCHAR(n)` or `INTEGER(n)`, because the discrepancies studied here arise from the coarse type families and from implicit casts between them. PostgreSQL also has an internal unknown type for literals, which helps resolve overloaded functions and casts but is not a table-column type; SQLite is different again because it uses dynamic typing, allowing values whose runtime type need not match the declared column affinity.

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: `SELECT 1.1 + 1 FROM R` and `SELECT '1' + 1 FROM R`**.** 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: `SELECT '1.1' + 1 FROM R`**.** 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: `SELECT '1.1' + 1.1 FROM R`**.** 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: `SELECT '1' + '1' FROM R`**.** PSQL reports a 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: `SELECT 'a' + '2b' FROM R`**.** 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 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: `SELECT 1+A FROM R WHERE B=20`**.** 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: `SELECT 1+A FROM R WHERE B=10`**.** 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 MSSQL and Oracle.

E9: `SELECT 1 + A FROM (SELECT '2' AS A FROM R) B`**.** Unlike 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 FROM R) B`.

2.2 Boolean operations

The following examples illustrate differences in behavior related to comparison operators in conditions.

E10: `SELECT 1 FROM R WHERE '1'<2`**.** Almost every engine is able to cast `'1'` to `INT`, returning 1. SQLite, on the other hand, returns an 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 greater than any number.

E11: `SELECT 1 FROM R WHERE '1.1'<2`**.** 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 arguments (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 deserves special attention because it illustrates distinctive cases related to the comparison operator, as exemplified in Table 2. Notably, operations `'0' < 1` and `'1' < 0` yield `0`. This behavior is attributed to the fact that strings are considered larger than integers, as previously explained. However, the effect differs by expression: for `'0' < 1`, adding arithmetic to the left operand (`'0'+0 < 1`) or casting the right operand (`'0' < CAST(1 AS INT)`) forces a numeric comparison and yields `1`; for `'1' < 0`, the analogous numeric comparison would be `1 < 0` and therefore remains false. Since `'0' < CAST(1 AS INT)` performs a numeric comparison, it is tempting to expect the arithmetically equivalent-looking expression `'0' < 1+0` to behave the same way. Nevertheless, the result is actually `0`, because SQLite does not assign the comparison operand the same static type information in the arithmetic expression case.

Table 2. Behavior of the comparison operator in SQLite

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

2.3 Set operations

E12: `SELECT '1.1' FROM R INTERSECT SELECT 1.1 FROM R`**.** PSQL and MSSQL yield the numeric value `1.1`, while MySQL yields the string value `'1.1'` from the left operand. In all three engines, the result is non-empty because, during intersection, two conditions are checked: (1) the cardinality of both subqueries must match, and (2) the types of the columns must also be consistent. To achieve the second condition, an implicit conversion is used to compare the two operands. 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: `SELECT '1.1' FROM R INTERSECT SELECT 1 FROM R`**.** Since `'1.1'` cannot be implicitly cast to an integer (the column type of the right subquery), this query is rejected by PSQL, MSSQL and Oracle. Both MySQL and SQLite return an empty result.

These examples illustrate the variety of typing discrepancies that arise across SQL engines. We now turn to the formalism of TRAF to model them systematically.

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 TRAF is presented in Figure 2. The formalization is inspired by the work of Guagliardo and Libkin [2017], 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). We use the symbol ? for the unknown type, used by PSQL to type string literals [PostgreSQL 1996], and to model flexible typing in SQLite. Thus, in the PSQL instantiation, ? corresponds to PostgreSQL's internal unknown type for literals, whereas in the SQLite instantiation it approximates SQLite's dynamic typing discipline: a column declaration gives an affinity, but the value stored in a column may still have a different runtime type. A relation type T is an ordered list of pairs of column names and its corresponding value type. A schema Γ is a list of pairs of relation names and their relation type. We assume that both T and

Types and	$\tau ::=$	$\mathbb{Z} \mid \mathbb{B} \mid \mathsf{String} \mid ? \mid \mathbb{R}$	(value types)
Schemas	$T ::=$	$N \mapsto \tau \mid T, N \mapsto \tau$	(relation types)
	$\Gamma ::=$	$\emptyset \mid \Gamma, R \mapsto T$	(schema)
Values and	$0_\mathsf{A} \in$	$\{+\}$	(arithmetic ops)
Expressions	$0_\mathsf{C} \in$	$\{<, =\}$	(comparison ops)
	$0_\mathsf{B} \in$	$\{\wedge, \vee\}$	(boolean ops)
	$w ::=$	$d \mid n \mid b \mid s \mid \mathsf{null}$	(simple values)
	$v ::=$	$w \mid w :: ?$	(values)
	$e ::=$	$N \mid v \mid e\ 0_\mathsf{A}\ e \mid e :: \tau$	(general expressions)
	$\theta ::=$	$e\ 0_\mathsf{C}\ e \mid \theta\ 0_\mathsf{B}\ \theta \mid \neg\theta \mid \mathsf{empty}(Q) \mid$ $(e_1, \ldots, e_k) \in Q \mid e\ \mathsf{is\ null}$	(boolean expressions)
	$\beta ::=$	$e\ \mathsf{as}\ N \mid \beta, e\ \mathsf{as}\ N$	(aliased expressions)
Queries	$0_\mathsf{S} \in$	$\{\cup, \cap, \times, -\}$	(set query ops)
	$Q ::=$	$R \mid \pi_\beta(Q) \mid \sigma_\theta(Q) \mid Q\ 0_\mathsf{S}\ Q \mid \varepsilon(Q)$	(queries)

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

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*. Each row is a list of *values*. A *simple value* w represents atomic data (integers, booleans, strings, and null). 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 are standard comparison operations or logical combinations of such operations. In addition, they include membership test expressions $(e_1, \ldots, e_n) \in Q$, which check whether the tuple of expressions $(e_1, \ldots, e_n)$ belongs to the result of subquery Q, the function $\mathsf{empty}(Q)$, which tests whether a query result is empty, and the predicate $\mathsf{isnull}(e)$, which tests whether an expression evaluates to the null value.[5] 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, as usual, is either a relation R, a projection $\pi_\beta(Q)$, a selection $\sigma_\theta(Q)$, or a bag operation, that is, cross product $(Q \times Q)$, intersection $(Q \cap Q)$, union $(Q \cup Q)$, difference $(Q - Q)$, and the removal of duplication $(\varepsilon(Q))$. 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 $\pi_{(1+1)\ \mathsf{as}\ C}(R)$.

3.2 Abstract operators

Some key operators in TRAF are left abstract, as they depend on the specific engine being used.

[5]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 in the Oracle version used in our validation, `SELECT 1 < 2 FROM R` is a syntactically invalid query, while it is valid in PSQL, MySQL and SQLite; recent Oracle versions have added support for this form).

Bidirectional Implicit Cast. Operator $biconv(e_1, \tau_1, e_2, \tau_2) = \tau_3$ 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. The pointwise lifting used for lists of aliased expressions is:

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

Explicit Cast. Operator $cast(v, \tau) = v'$ 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 $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4$ determines which specific operation among a set of overloaded ones should be called based on the provided arguments during invocation. More specifically, given a set of function types $\overline{\tau_3} = \overline{\tau_5 \times \tau_6 \to \tau_7}$, it tries to find the best candidate type for expressions e_1 and e_2, typed as τ_1 and τ_2 respectively. A function type of the form $\tau_5 \times \tau_6 \to \tau_7$, denotes a function that accepts two arguments, of types τ_5 and τ_6 respectively, and produces a result of type τ_7.

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 whether it is possible to explicitly cast expression e of type τ', to an expression of type τ.

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

Candidate Types. Operation $ty(\mathtt{0}) = \overline{\tau}$ determines the set of possible function types available for operation 0. The output is a list of function types $\overline{t}$ given that many arithmetic and comparison operators are overloaded with more than one type.

Unknown Type Removing. Operator $clean(T) = T'$ performs post-processing for relation type T returning a new relation type T'. In some engines, this operation replaces occurrences of the unknown type with another type.

Annotation Insertion. Operator $insert(e, \tau, \mathtt{0}) = e'$ attempts to explicitly cast expression e to type τ in a context within the 0 operation. It returns a new expression e' depending on whether the expression was cast or not.

Value Operation Application. Operator $apply(\mathtt{0}, v_1, v_2) = v_3$ performs arithmetic or comparison operation 0 to values v_1 and v_2, yielding value v_3.

As a concrete example, in the PSQL instantiation string literals have type unknown ($ty(\texttt{'s'}) = ?$), projection output types replace unknowns by strings ($clean(T) = T[\mathsf{String}/?]$), and *biconv* accepts a pair such as $(\texttt{'1'}, ?, 1, \mathbb{Z})$ by choosing $\mathbb{Z}$, but rejects $(\texttt{'1.1'}, ?, 1, \mathbb{Z})$ because the literal cannot be implicitly cast to an integer. This illustrates why these operators take both expressions and their static types: the type alone does not determine whether a literal cast is feasible.

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

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). The rules are syntax directed: for each syntactic form there is a unique rule schema, except for the engine-specific abstract operators in the premises. Consequently, for a fixed engine instantiation in which the abstract operators are computable, typechecking follows the query syntax and is linear in the number of syntax nodes, up to the cost of abstract operator calls.

$\boxed{\Gamma; T \vdash e : \tau}$

$$(\mathrm{T}v)\frac{}{\Gamma; T \vdash v : \; ty(v)} \qquad (\mathrm{Tis})\frac{\Gamma; T \vdash e : \tau}{\Gamma; T \vdash e \text{ is null} : \mathbb{B}} \qquad (\mathrm{TN})\frac{(N \mapsto \tau) \in T}{\Gamma; T \vdash N : \tau}$$

$$(\mathrm{T}\neg)\frac{\Gamma; T \vdash \theta : \mathbb{B}}{\Gamma; T \vdash \neg\theta : \mathbb{B}} \qquad (\mathrm{T}::)\frac{\Gamma; T \vdash e : \tau' \qquad e : \tau' \lessapprox \tau}{\Gamma; T \vdash (e :: \tau) : \tau}$$

$$(\mathrm{T}0)\frac{\Gamma; T \vdash e_1 : \tau_1 \qquad \Gamma; T \vdash e_2 : \tau_2 \qquad 0 \in 0_{\mathsf{A}} \cup 0_{\mathsf{C}} \qquad resolve(e_1, \tau_1, e_2, \tau_2, ty(0)) = \tau_3 \times \tau_4 \to \tau_5}{\Gamma; T \vdash e_1 \; 0 \; e_2 : \tau_5} \qquad (\mathrm{T}0_{\mathsf{B}})\frac{\Gamma; T \vdash \theta_1 : \mathbb{B} \qquad \Gamma; T \vdash \theta_2 : \mathbb{B}}{\Gamma; T \vdash \theta_1 \; 0_{\mathsf{B}} \; \theta_2 : \mathbb{B}}$$

$$(\mathrm{T}\exists)\frac{\Gamma; T \vdash Q : T'}{\Gamma; T \vdash \mathsf{empty}(Q) : \mathbb{B}}$$

$$(\mathrm{T}\in)\frac{\overline{\Gamma; T \vdash e_i : \tau_i} \qquad \overline{resolve(e_i, \tau_i, N_i', clean(T'(N_i)), ty(=)) = \tau_i^l \times \tau_i^r \to \mathbb{B}} \qquad \Gamma; T \vdash Q : T' \qquad \ell(\Gamma, Q) = \overline{N_i} \qquad \overline{N_i' \text{ fresh}}}{\Gamma; T \vdash (e_1, \ldots, e_k) \in Q : \mathbb{B}}$$

$\boxed{\Gamma; T \vdash \beta : T'}$

$$(\mathrm{T}\beta)\frac{\forall i. \Gamma; T \vdash e_i : \tau_i \qquad unique(\overline{N_i})}{\Gamma; T \vdash \overline{e_i \text{ as } N_i} : \overline{N_i \mapsto \tau_i}}$$

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

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

$$(\mathrm{T}\times)\frac{\ell(\Gamma, Q_1) \cap \ell(\Gamma, Q_2) = \emptyset \qquad \Gamma; T \vdash Q_1 : T_1 \qquad \Gamma; T \vdash Q_2 : T_2}{\Gamma; T \vdash Q_1 \times Q_2 : T_1, T_2} \qquad (\mathrm{T}R)\frac{\Gamma(R) = T}{\Gamma; T' \vdash R : T}$$

$$(\mathrm{T}0_{\mathsf{S}})\frac{\Gamma; T' \vdash \pi_{\beta_1}(Q_1) : T_1 \qquad \Gamma; T' \vdash \pi_{\beta_2}(Q_2) : T_2 \qquad biconv^*(\beta_1, T_1, \beta_2, T_2) = T \qquad 0_{\mathsf{S}} \in \{\cup, \cap, -\}}{\Gamma; T' \vdash \pi_{\beta_1}(Q_1) \; 0_{\mathsf{S}} \; \pi_{\beta_2}(Q_2) : T} \qquad (\mathrm{T}\varepsilon)\frac{\Gamma; T' \vdash Q : T}{\Gamma; T' \vdash \varepsilon(Q) : T}$$

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

Expressions. The judgment $\Gamma; T \vdash e : \tau$ states that the expression e has type τ under schema Γ and relation type T. Intuitively, T is needed to typecheck column names N, and Γ to typecheck subqueries Q inside empty 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 (Tis) types the e is null expression as a boolean. 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 (T0), (T0_B) and (T$\neg$) type operations. (T0_B) and (T$\neg$) type boolean operations as usual. The interesting case is rule (T0) that types arithmetic and string operations. Based on the types of e_1 and e_2 and operation 0, this 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 (T$\exists$) checks whether the subquery Q is well typed under the schema Γ and relation type T. The relation type is threaded through (T$\exists$) and (T$\in$) because nested queries may be correlated: the subquery must still be able to typecheck references to columns bound by the surrounding query. Rule (T$\in$) first typechecks the subquery and keeps its output names N_i. For each component, the outer expression e_i is typechecked under the original outer relation type T, while the corresponding inner column contributes only its cleaned type $clean(T'(N_i))$. Equality resolution is then performed on the pair (e_i, N_i'), where N_i' stands for the fresh renamed copy of the corresponding subquery column used by the implementation. The key point is that the implementation does not typecheck a single equality under a combined scope $T, clean(T')$. Instead, it first computes the left type from the outer scope and the right type from the renamed-and-cleaned subquery output, and only then calls *resolve* to obtain the argument types expected by =. This avoids changing the resolution of an outer unqualified name when the subquery contains a column with the same name. Finally, each expression is typechecked in isolation to ensure that it does not reference any column introduced by the subquery.

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. The judgment $\Gamma; T \vdash Q : T'$ denotes that query Q has relation type T' under schema Γ and ambient relation type T. 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 contexts Γ and 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×) 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:

$$\begin{aligned}
\ell(\Gamma, R) &= \overline{N_i} \text{ where } \Gamma(R) = \overline{N_i \mapsto \tau_i} \\
\ell(\Gamma, Q_1 \times Q_2) &= \ell(\Gamma, Q_1), \ell(\Gamma, Q_2) \\
\ell(\Gamma, Q_1\ \mathsf{0_S}\ Q_2) &= \ell(\Gamma, Q_1) \text{ where } \mathsf{0_S} \in \{\cup, \cap, -\};\ (*) \\
\ell(\Gamma, \sigma_\theta(Q)) &= \ell(\Gamma, Q) \\
\ell(\Gamma, \pi_\beta(Q)) &= \ell(\Gamma, \beta) \\
\ell(\Gamma, \beta, e \text{ as } N) &= \ell(\Gamma, \beta), \ell(\Gamma, e \text{ as } N) \\
\ell(\Gamma, e \text{ as } N) &= N
\end{aligned}$$

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

Rule ($\mathsf{T0_S}$) deals with set operations (union, intersection, and difference) which require special attention. Usually, it is assumed that the names, cardinality, and types of the columns in the subqueries match. In practice, the column names do not necessarily match, but cardinality 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. For instance, `(SELECT '1.1' AS A FROM P) UNION (SELECT 1.1 AS A FROM P)` is encoded as $\pi_{\texttt{'1.1'} \text{ as } \mathsf{A}}(P) \cup \pi_{1.1 \text{ as } \mathsf{A}}(P)$. 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 introduced in Section 3.2.

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 $\pi_{\texttt{'1'}+1 \text{ as } \mathsf{A}}(R)$ to $\pi_{(\texttt{'1'}::\mathbb{Z})+1 \text{ as } \mathsf{A}}(R)$.

Figure 4 presents the explicit cast insertion rules. 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,

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

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

$$(\mathrm{E}0)\frac{\Gamma; T \vdash e_1 : \tau_1 \rightsquigarrow e_1' \qquad \Gamma; T \vdash e_2 : \tau_2 \rightsquigarrow e_2' \qquad 0 \in \{<, =, +\} \qquad \mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \mathit{ty}(0)) = \tau_3 \times \tau_4 \to \tau_5}{\Gamma; T \vdash e_1\ 0\ e_2 : \tau_4 \rightsquigarrow \mathit{insert}(\mathit{insert}(e_1', \tau_3, 0)\ 0\ \mathit{insert}(e_2', \tau_4, 0), \tau_5, 0)}$$

$$(\mathrm{E}::)\frac{\Gamma; T \vdash e : \tau' \rightsquigarrow e' \qquad e : \tau' \not\approx \tau}{\Gamma; T \vdash e :: \tau : \tau \rightsquigarrow e' :: \tau} \qquad (\mathrm{E}N)\frac{T(N) = \tau}{\Gamma; T \vdash N : \tau \rightsquigarrow N}$$

$$(\mathrm{E}0_\mathsf{B})\frac{\Gamma; T \vdash \theta_1 : \mathbb{B} \rightsquigarrow \theta_1' \qquad \Gamma; T \vdash \theta_2 : \mathbb{B} \rightsquigarrow \theta_2' \qquad 0_\mathsf{B} \in \{\wedge, \vee\}}{\Gamma; T \vdash \theta_1\ 0_\mathsf{B}\ \theta_2 : \mathbb{B} \rightsquigarrow \theta_1'\ 0_\mathsf{B}\ \theta_2'} \qquad (\mathrm{E}\neg)\frac{\Gamma; T \vdash \theta : \mathbb{B} \rightsquigarrow \theta'}{\Gamma; T \vdash \neg\theta : \mathbb{B} \rightsquigarrow \neg\theta'}$$

$$(\mathrm{E}\in)\frac{\Gamma; T \vdash Q : T' \rightsquigarrow Q' \qquad \ell(\Gamma, Q) = \overline{N_i} \qquad \overline{N_i' \text{ fresh}} \qquad \overline{\mathit{resolve}(e_i, \tau_i, N_i', \mathit{clean}(T'(N_i)), \mathit{ty}(=)) = \tau_i' \times \tau_i'' \to \mathbb{B}} \qquad \overline{\Gamma; T \vdash e_i : \tau_i \rightsquigarrow e_i'} \qquad \overline{\theta_i' = \mathit{insert}(\mathit{insert}(e_i', \tau_i', =) = \mathit{insert}(N_i', \tau_i'', =), \mathbb{B}, =)} \qquad \overline{e_i'' = \mathsf{lhs}(\theta_i')} \qquad \overline{e_i''' = \mathsf{rhs}(\theta_i')}}{\Gamma; T \vdash (e_1, \ldots, e_k) \in Q : \mathbb{B} \rightsquigarrow (e_1'', \ldots, e_k'') \in \pi_{e_1''' \mathsf{as} N_1', \ldots, e_k''' \mathsf{as} N_k'}(\rho_{\overline{N_i} \mapsto \overline{N_i'}} Q')}$$

$$(\mathrm{E}\exists)\frac{\Gamma; T \vdash Q : T' \rightsquigarrow Q'}{\Gamma; T \vdash \mathsf{empty}(Q) : \mathbb{B} \rightsquigarrow \mathsf{empty}(Q')} \qquad (\mathrm{Eis})\frac{\Gamma; T \vdash e : \tau \rightsquigarrow e'}{\Gamma; T \vdash e\ \mathsf{is\ null} : \mathbb{B} \rightsquigarrow e'\ \mathsf{is\ null}}$$

$\boxed{\Gamma; T \vdash \beta : T' \rightsquigarrow \beta'}$

$$(\mathrm{E}\beta)\frac{\overline{\Gamma; T \vdash e : \tau \rightsquigarrow e'} \qquad \mathit{unique}(\overline{N})}{\Gamma; T \vdash \overline{e\ \mathsf{as}\ N} : \overline{N \mapsto \tau} \rightsquigarrow \overline{e'\ \mathsf{as}\ N}}$$

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

$$(\mathrm{E}0_\mathsf{S})\frac{\Gamma; T \vdash \pi_{\beta_1}(Q_1) : T_1 \rightsquigarrow \pi_{\beta_1'}(Q_1') \qquad \Gamma; T \vdash \pi_{\beta_2}(Q_2) : T_2 \rightsquigarrow \pi_{\beta_2'}(Q_2') \qquad \mathit{biconv}^*(\beta_1, T_1, \beta_2, T_2) = T \qquad 0_\mathsf{S} \in \{\cup, \cap, -\} \qquad e_1 = \mathit{insert}^*(\beta_1', T, 0_\mathsf{S}) \qquad e_2 = \mathit{insert}^*(\beta_2', T, 0_\mathsf{S})}{\Gamma; T \vdash \pi_{\beta_1}(Q_1)\ 0_\mathsf{S}\ \pi_{\beta_2}(Q_2) : T \rightsquigarrow \pi_{e_1}(Q_1')\ 0_\mathsf{S}\ \pi_{e_2}(Q_2')}$$

$$(\mathrm{E}R)\frac{\Gamma(R) = T}{\Gamma; T' \vdash R : T \rightsquigarrow R} \qquad (\mathrm{E}\pi)\frac{\Gamma; T'' \vdash Q : T \rightsquigarrow Q' \qquad \Gamma; T'', \mathit{clean}(T) \vdash \beta : T' \rightsquigarrow \beta'}{\Gamma; T'' \vdash \pi_\beta(Q) : T' \rightsquigarrow \pi_{\beta'}(Q')}$$

$$(\mathrm{E}\sigma)\frac{\Gamma; T' \vdash Q : T \rightsquigarrow Q' \qquad \Gamma; T', T \vdash \theta : \mathbb{B} \rightsquigarrow \theta'}{\Gamma; T' \vdash \sigma_\theta(Q) : T \rightsquigarrow \sigma_{\theta'}(Q')}$$

$$(\mathrm{E}\times)\frac{\ell(\Gamma, Q_1) \cap \ell(\Gamma, Q_2) = \emptyset \qquad \Gamma; T \vdash Q_1 : T_1 \rightsquigarrow Q_1' \qquad \Gamma; T \vdash Q_2 : T_2 \rightsquigarrow Q_2'}{\Gamma; T \vdash Q_1 \times Q_2 : T_1, T_2 \rightsquigarrow Q_1' \times Q_2'}$$

Fig. 4. TRAF Cast Insertion. Abstract operators are highlighted in gray.

elaboration rules are defined inductively and grouped in three categories: for general and aliased expressions, and for queries. Most elaboration rules directly follow their corresponding typing rule. The expression judgment $\Gamma; T \vdash e : \tau \rightsquigarrow e'$ states that expression e, typed as τ under schema Γ and relation type T, elaborates to e'. The aliased-expression judgment $\Gamma; T \vdash \beta : T' \rightsquigarrow \beta'$ states that aliased expressions elaborate to β' with relation type T'. The query judgment $\Gamma; T' \vdash Q : T \rightsquigarrow Q'$ states that query Q, under schema Γ and relation type T', elaborates to Q' with relation type T.

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. For example, the expression `'0'` < 1 evaluates to 0 (which represents false in that engine). To facilitate this special case, the expression is elaborated to `'0'` $::$? $< 1 ::$?.

Rule (E0) 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. For instance, in SQLite, expression `'1'` $+ 1$ is elaborated to $(\texttt{'1'} :: ?) :: \mathbb{Z} + (1 :: ?) :: \mathbb{Z}$.

Rule (E::) elaborates an ascription by first elaborating the subexpression and then inserting an explicit cast to the target type. Rule (EN) leaves column names unchanged during elaboration. Rules (E0_B) and (E$\neg$) elaborate boolean operations by elaborating each subexpression without introducing further elaboration steps.

Rule (E$\in$) elaborates membership test expressions by first elaborating the subquery Q and renaming its output columns to fresh names N'_i. For each component, the outer expression e_i is elaborated under the original outer relation type T, while the right-hand side contributes only the cleaned type $\mathit{clean}(T'(N_i))$ of the corresponding renamed subquery column. Equality resolution then computes the two argument types expected by $=$, and explicit casts are inserted on both sides before wrapping the whole equality with the result cast. Since the elaboration of an equality may introduce a casted equality that produces a boolean, the rule extracts the left-hand side and right-hand side using auxiliary functions $\mathsf{lhs}(e)$ and $\mathsf{rhs}(e)$ defined as follows:

$$\mathsf{lhs}((e_1 = e_2) :: \tau) = e_1 \qquad \mathsf{rhs}((e_1 = e_2) :: \tau) = e_2$$
$$\mathsf{lhs}(e_1 = e_2) = e_1 \qquad \mathsf{rhs}(e_1 = e_2) = e_2$$

The right-hand-side expressions are then used to project from the renamed elaborated subquery the elaborated (and potentially casted) columns of the subquery. For instance, in the case of PSQL, the query

```
SELECT 1 FROM R WHERE ('1') IN (SELECT 1 AS N1 FROM R)
```

is elaborated to

```
SELECT 1 FROM R WHERE (CAST('1' AS INT)) IN
  (SELECT CAST(N1 AS INT) FROM (SELECT 1 AS N1 FROM R))
```

Rules (E$\exists$) and (Eis) elaborate each subquery and each subexpression respectively, without introducing further elaboration steps.

Rule (E0_S) elaborates set operations by inserting casts to the output type of biconv^*. This is done using the insert^* operation defined as $\mathit{insert}^*(\overline{e \text{ as } N}, \overline{N \mapsto \tau}, 0_S) = \overline{\mathit{insert}(e, \tau, 0_S) \text{ as } N}$. For instance, for PSQL, query

```
(SELECT '1' FROM R) INTERSECT (SELECT 1 FROM R)
```

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

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

$$[\![\pi_\beta(Q)]\!]_{\eta',\mathtt{D},\Gamma} = \mathrm{do}\{t \leftarrow [\![Q]\!]_{\eta',\mathtt{D},\Gamma}; \lceil \{\!\{ \underbrace{[\![\beta]\!]_{\eta'\oplus\eta^r_{\ell(\Gamma,Q)},\mathtt{D},\Gamma}, \ldots, [\![\beta]\!]_{\eta'\oplus\eta^r_{\ell(\Gamma,Q)},\mathtt{D},\Gamma}}_{\text{k times}} \mid r \in_\mathsf{k} t \}\!\} \rceil\} \tag{Rπ}$$

$$[\![\sigma_\theta(Q)]\!]_{\eta',\mathtt{D},\Gamma} = \mathrm{do}\{t \leftarrow [\![Q]\!]_{\eta',\mathtt{D},\Gamma}; \lceil \{\!\{\mathbf{ok}\ r \mid r \in_\mathsf{k} t \wedge \phi_r = \mathsf{true}\}\!\} \cup \{\!\{\mathbf{error} \mid r \in_\mathsf{k} t \wedge \phi_r = \mathbf{error}\}\!\} \rceil\} \tag{Rσ}$$
$$\text{where } \phi_r = [\![\theta]\!]_{\eta'\oplus\eta^r_{\ell(\Gamma,Q)},\mathtt{D},\Gamma}$$

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

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

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

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

$$[\![v]\!]_{\eta,\mathtt{D},\Gamma} = \mathbf{ok}\ v \tag{Rv}$$

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

$$[\![e :: \tau]\!]_{\eta,\mathtt{D},\Gamma} = \mathrm{do}\{v \leftarrow [\![e]\!]_{\eta,\mathtt{D},\Gamma} \quad e \neq v; [\![v :: \tau]\!]_{\eta,\mathtt{D},\Gamma}\} \tag{Re ::}$$

$$[\![\theta_1\ \mathtt{O_B}\ \theta_2]\!]_{\eta,\mathtt{D},\Gamma} = \mathrm{do}\{v_1 \leftarrow [\![\theta_1]\!]_{\eta,\mathtt{D},\Gamma}; v_2 \leftarrow [\![\theta_2]\!]_{\eta,\mathtt{D},\Gamma}; \mathbf{ok}\ v_1\ \mathtt{O_B}\ v_2\} \tag{RO$_\mathtt{B}$}$$

$$[\![\neg\theta]\!]_{\eta,\mathtt{D},\Gamma} = \mathrm{do}\{v_1 \leftarrow [\![\theta_1]\!]_{\eta,\mathtt{D},\Gamma}; \mathbf{ok}\ \neg v_1\} \tag{R$\neg$}$$

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

$$[\![(e_1,\ldots,e_k) \in Q]\!]_{\eta,\mathtt{D},\Gamma} = \mathrm{do}\{t \leftarrow [\![Q]\!]_{\eta,\mathtt{D},\Gamma}; r \leftarrow ([\![e_1]\!]_{\eta,\mathtt{D},\Gamma}, .., [\![e_k]\!]_{\eta,\mathtt{D},\Gamma}); r\ \tilde{\in}\ t\} \tag{R$\in$}$$

$$\text{where } (v_1,\ldots,v_k)\ \tilde{\in}\ t = \begin{cases} \mathbf{error} & \exists (v'_1,..,v'_k) \in t, \exists i, [\![v_i = v'_i]\!]_{\eta,\mathtt{D},\Gamma} = \mathbf{error} \\ \mathbf{ok}\ \mathsf{true} & \exists (v'_1,..,v'_k) \in t, \forall i, [\![v_i = v'_i]\!]_{\eta,\mathtt{D},\Gamma} = \mathsf{true} \\ \mathbf{ok}\ \mathsf{false} & \forall (v'_1,..,v'_k) \in t, \exists i, [\![v_i = v'_i]\!]_{\eta,\mathtt{D},\Gamma} = \mathsf{false} \\ \mathbf{ok}\ \mathsf{null} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{empty}(Q)]\!]_{\eta,\mathtt{D},\Gamma} = \mathrm{do}\{t \leftarrow [\![Q]\!]_{\eta,\mathtt{D},\Gamma}; \mathbf{ok}\ t = \emptyset\} \tag{R$\exists$}$$

$$[\![e\ \mathsf{is\ null}]\!]_{\eta,\mathtt{D},\Gamma} = \mathrm{do}\{v \leftarrow [\![e]\!]_{\eta,\mathtt{D},\Gamma}; \mathbf{ok}\ v = \mathsf{null}\} \tag{Ris}$$

Fig. 5. Dynamic Semantics of TRAF for queries and expressions.

is elaborated to

```
(SELECT CAST('1' AS INT) FROM R) INTERSECT (SELECT CAST(1 AS INT) FROM R)
```

The rest of the elaboration rules for queries directly follow their corresponding typing rules without introducing further elaboration steps.

3.5 Dynamic Semantics

Figure 5 presents the dynamic semantics of TRAF, which differs from the ones of Guagliardo and Libkin [2017] (GL) as follows. First, TRAF dynamic semantics are defined over a relational algebra, whereas GL uses SQL syntax to support 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 expression `'s'` :: $\mathbb{Z}$ evaluates to an error, so a query such as `SELECT CAST('s' as INT) FROM R` fails during evaluation.

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 [Wadler 1992]. 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. 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 : $[\![Q]\!]_{\eta,\mathtt{D},\Gamma}$ to reduce queries, $[\![e]\!]_{\eta,\mathtt{D},\Gamma}$ to reduce expressions, $[\![\beta]\!]_{\eta,\mathtt{D},\Gamma}$ to reduce aliased expressions, and $[\![\theta]\!]_{\eta,\mathtt{D},\Gamma}$ to reduce boolean expressions. The evaluation of a query is parametrized by a database $\mathtt{D}$, which maps relation names R to tables t, and a schema Γ. We need Γ to extract the name of queries during evaluation. A table t is a bag of *rows* $\{\!\{r_i\}\!\}$, where a row r is an ordered list of values $\overline{v_i}$. We use $r \in_{\mathsf{k}} t$ to denote that r appears k times in t. All four evaluation categories are also parameterized by an environment η, which maps column names N to values v. This environment is empty for top-level query evaluation and is threaded into subquery evaluation for `IN` and `EXISTS`, so that correlated subqueries can refer to columns bound by the surrounding row. For expressions, aliased expressions, and boolean expressions, the same environment is used to extract the value associated with a column name for a given row. For instance, consider a relation of persons $\mathtt{P}(\mathtt{Name} \mapsto \mathtt{String}, \mathtt{Age} \mapsto \mathbb{Z})$, and a table $\{(\mathtt{'Bob'}, \mathtt{10}), (\mathtt{'Alice'}, \mathtt{20})\}$. Then, for the first row, $\eta(\mathtt{Name}) = \mathtt{'Bob'}$ and $\eta(\mathtt{Age}) = \mathtt{10}$. When a subquery is evaluated inside a predicate over that row, this same environment is passed to the subquery and then combined with the subquery row environment whenever the subquery evaluates one of its own projections or filters.

Queries. The basic case is Rule (RR), which evaluates the name of a relation yielding the table associated to that name in database $\mathtt{D}$. Rule (R$\mathtt{O}_\mathtt{S}$) 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 Guagliardo and Libkin [2017], $\eta^{r,v}_{\overline{N_i},N} = \eta^r_{\overline{N_i}}, N \mapsto v$ 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 $\mathtt{S}$ to an optional bag of rows using the $\lceil\cdot\rceil$ function defined as: $\lceil\mathtt{S}\rceil$ = **error**, when **error** $\in \mathtt{S}$; and **ok** $\{\!\{r \mid (\textbf{ok}\ r) \in \mathtt{S}\}\!\}$ 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. The rule stores the result of each condition evaluation in ϕ_r, propagates any error using $\lceil\cdot\rceil$, and otherwise keeps exactly the rows for which ϕ_r is true.

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 (R0_B), (R$\neg$), and (R0) operate in a similar fashion. First, each subexpression is reduced, then the resulting values are combined using the specific operation at hand. For boolean operations, the standard three-valued logic is used [Bolc and Borowik 1992]. For arithmetic and comparison operations, the exact operation is performed using the *apply* operation. We assume *apply* is total on well-formed runtime values for the operations modeled here; failures due to arithmetic overflow, division by zero, or other non-cast execution errors are outside the core calculus. Runtime errors in the metatheory therefore arise from explicit casts and are otherwise propagated by the monadic rules. For instance, in PSQL, the expression `'0'` $<$ 1 evaluates to true, whereas in SQLite, it yields false.

Rule (R$\in$) first reduces subquery Q. If the result is successful (yielding table t), then it reduces each subexpression e_i to a value v_i. Then, it checks if there exists a row $r' = (v'_1, .., v'_k)$ in t such that for each i, the equality between v_i and v'_i evaluates to true. If the evaluation of any equality produces an error, this error is propagated. If there exists such a row r' in t for which all equalities hold, the rule returns true. If for all rows in t there exists an i such that the equality between v_i and v'_i evaluates to false, then it returns false. Otherwise, it returns null. Rule (R$\exists$) evaluates subquery Q. If the result is successful (yielding table t), then it returns true if t is empty, and false otherwise. Rule (Ris) first reduces subexpression e to a value v, then it returns true if v is null, and false otherwise.

Example. Suppose $\Gamma = P : (\mathsf{Name} \mapsto \mathsf{String}, \mathsf{Age} \mapsto \mathbb{Z})$, database D maps P to the table $\{(\texttt{'Bob'}, 10), (\texttt{'Alice'}, 20)\}$, and we want to evaluate the query $\pi_{\mathsf{Name\ as\ }N}(P)$. The query first evaluates P to

$$\mathbf{ok}\ \{(\texttt{'Bob'}, 10), (\texttt{'Alice'}, 20)\}$$

Since the result is successful, we proceed to evaluate the aliased expression for each row. To do this, we compute $\ell(\Gamma, P) = (\mathsf{Name}, \mathsf{Age})$. Then for the first and second row we evaluate $[\![\mathsf{Name\ as\ } N]\!]_{\eta_1,\mathrm{D},\Gamma}$, and $[\![\mathsf{Name\ as\ } N]\!]_{\eta_2,\mathrm{D},\Gamma}$ where

$$\eta_1 = \eta^{(\texttt{'Bob'},10)}_{(\mathsf{Name},\mathsf{Age})} = \mathsf{Name} \mapsto \texttt{'Bob'}, \mathsf{Age} \mapsto 10$$
$$\eta_2 = \eta^{(\texttt{'Alice'},20)}_{(\mathsf{Name},\mathsf{Age})} = \mathsf{Name} \mapsto \texttt{'Alice'}, \mathsf{Age} \mapsto 20$$

Continuing the example and using Rule (RN):

$$[\![\mathsf{Name\ as\ } N]\!]_{\eta_1,\mathrm{D},\Gamma} = [\![\mathsf{Name}]\!]_{\eta_1,\mathrm{D},\Gamma} = \mathbf{ok}\ \eta_1(\mathsf{Name}) = \mathbf{ok}\ \texttt{'Bob'}$$

and similarly, $[\![\mathsf{Name\ as\ } N]\!]_{\eta_2,\mathrm{D},\Gamma} = \mathbf{ok}\ \texttt{'Alice'}$. Then the result is:

$$\begin{aligned}[\![\pi_{\mathsf{Name\ as\ }N}(P)]\!]_{\eta,\mathrm{D},\Gamma} &= \lceil\{\mathbf{ok}\ (\texttt{'Bob'}), \mathbf{ok}\ (\texttt{'Alice'})\}\rceil \\ &= \mathbf{ok}\ \{(\texttt{'Bob'}), (\texttt{'Alice'})\}\end{aligned}$$

3.6 Translation from SQL to TRAF

The syntax of SQL queries considered in this work is presented in Figure 6. Expressions E include column names N, values v, binary expressions with arithmetic operators $\mathrm{O_A}$, and type casts. Boolean expressions C include comparisons, logical combinations, null checks, and membership tests with subqueries. Since aliasing is optional, aliased columns G can be either an expression E or an expression with an alias. A list of columns B is a comma-separated list of aliased columns to be used within a **SELECT** clause. From queries F can be either a table R, a table with an alias, a subquery with an alias, or a comma-separated list of from queries. Select queries S can be either a **SELECT-FROM-WHERE** query, a **SELECT-FROM** query, or the result of applying bag or set operators, or **EXCEPT** between two select queries. **EXCEPT** is written separately because SQL's plain **EXCEPT** has set semantics, while **EXCEPT ALL** is the corresponding bag operator. The core language already contains

$$
\begin{array}{rcll}
E & ::= & N \mid v \mid E\ \mathsf{O_A}\ E \mid \texttt{CAST}(E\ \texttt{AS}\ \tau) & \text{(expressions)} \\
C & ::= & E\ \mathsf{O_C}\ E \mid C\ \mathsf{O_B}\ C \mid E\ \texttt{IS NULL} \mid (E, ..., E)\ \texttt{IN}\ S \mid \texttt{EXISTS}\ S & \text{(boolean expressions)} \\
G & ::= & E \mid E\ \texttt{AS}\ N & \text{(aliased column)} \\
B & ::= & G \mid B, G & \text{(columns)} \\
O_b & ::= & \texttt{UNION ALL} \mid \texttt{INTERSECT ALL} \mid \texttt{EXCEPT ALL} & \text{(bag operators)} \\
O_s & ::= & \texttt{UNION} \mid \texttt{INTERSECT} & \text{(set operators)} \\
S & ::= & \texttt{SELECT}\ B\ \texttt{FROM}\ F\ \texttt{WHERE}\ C & \\
 & \mid & \texttt{SELECT}\ B\ \texttt{FROM}\ F \mid S\ O_b\ S \mid S\ O_s\ S \mid S\ \texttt{EXCEPT}\ S & \text{(select queries)} \\
F & ::= & R \mid R\ N \mid S\ N \mid F, F & \text{(from queries)}
\end{array}
$$

Fig. 6. SQL syntax

duplicate elimination, so a query with **SELECT DISTINCT** can be translated as $\varepsilon(\pi_\beta(Q))$; we omit it from the grammar only to keep the presentation focused on casts and set operators.

Figure 7 presents the translation rules from SQL to TRAF. The rules are defined inductively, and divided into three categories: translation rules for expressions e, aliased columns G, and queries S. The rules are syntax directed: each source syntactic form is handled by exactly one translation rule, and recursive premises translate only its immediate subcomponents. The only auxiliary choices, such as generated aliases and the SQL set-operator mapping, are deterministic functions fixed by the definition.

Expressions. Rule (TrN) translates a column name N to the same column name in TRAF. Rule (Tr::) translates a type cast by recursively translating the expression being cast, and then applying the $\cdot :: \cdot$ operator. Rule (Trv) translates a value v to the same value in TRAF. Rules (TrO), and (Tr$\mathsf{O_B}$) translates by recursively translating its subexpressions. Rule (Tris) is straightforward, translating an `IS NULL` check into the corresponding $\cdot$ is null operator. Rule (Tr$\in$) translates an **IN** expression with a list of expressions and a subquery by recursively translating each expression in the list and the subquery, and then applying the membership operator $\in$ between the tuple of translated expressions and the translated subquery. Rule (Tr$\exists$) translates an **EXISTS** expression by recursively translating the subquery and then applying the negated existential operator $\neg\mathsf{empty}(\cdot)$.

Aliased columns. Rule (TrE) transforms expressions without an alias, as seen in `SELECT 1+1 FROM R`. In these cases, we utilize the $alias(\cdot)$ function, which generates an appropriate name for the subexpression. For example, $alias(1 + 1)$ = `"1+1"`, and $alias(\texttt{NAME})$ = `NAME`. Rules (TrEas) and (TrB) translate aliased columns and lists of aliased columns by recursively translating their subcomponents.

Queries. Rule (TrR) translates a table R to the same table in TRAF. Rules (Tr$Fasr$) and (TrFas) translate aliased tables and subqueries by projecting the translated subcomponent, and renaming each column to include the table or subquery alias as a prefix. Rule (TrSW) translates a **SELECT-FROM-WHERE** query by recursively translating each subcomponent and then generating a projection over a selection of the from query. Rule (TrS) translates a **SELECT-FROM** query similarly, but without a selection. Rule (Tr$\times$) is straightforward, translating a comma-separated list of from queries into a Cartesian product. Rule (TrFas) transforms aliased subqueries by creating a projection of the translated subquery, where each column is aliased using the table name plus the column name. Rules (TrO_b) transforms a bag operation, by transforming each subquery and then projecting the columns of them. By doing this we ensure that the resulting bag operations are made between projections to facilitate typing. Both rules use the map $toRA(\cdot)$ from SQL set-operator tokens to

$\boxed{\Gamma \vdash E \rightsquigarrow_e e, \Gamma \vdash C \rightsquigarrow_e \theta}$

$$(\mathrm{Tr}N)\frac{}{\Gamma \vdash N \rightsquigarrow_e N} \qquad (\mathrm{Tr}::)\frac{\Gamma \vdash E \rightsquigarrow_e e}{\Gamma \vdash \texttt{CAST}(E\ \texttt{AS}\ \tau) \rightsquigarrow_e e :: \tau} \qquad (\mathrm{Tr}v)\frac{}{\Gamma \vdash v \rightsquigarrow_e v}$$

$$(\mathrm{Tr}0)\frac{\Gamma \vdash E_1 \rightsquigarrow_e e_1 \quad \Gamma \vdash E_2 \rightsquigarrow_e e_2}{\Gamma \vdash E_1\ 0\ E_2 \rightsquigarrow_e e_1\ 0\ e_2} \qquad (\mathrm{Tr}0_{\mathsf{B}})\frac{\Gamma \vdash C_1 \rightsquigarrow_e \theta_1 \quad \Gamma \vdash C_2 \rightsquigarrow_e \theta_2}{\Gamma \vdash C_1\ 0_{\mathsf{B}}\ C_2 \rightsquigarrow_e \theta_1\ 0_{\mathsf{B}}\ \theta_2}$$

$$(\mathrm{Tris})\frac{\Gamma \vdash E \rightsquigarrow_e e}{\Gamma \vdash E\ \texttt{IS}\ \texttt{NULL} \rightsquigarrow_e e\ \mathsf{is\ null}} \qquad (\mathrm{Tr}\in)\frac{\overline{\Gamma \vdash E_i \rightsquigarrow_e e_i} \quad \Gamma \vdash S \rightsquigarrow Q}{\Gamma \vdash (E_1, ..., E_k)\ \texttt{IN}\ S \rightsquigarrow_e (e_1, ..., e_k) \in Q}$$

$$(\mathrm{Tr}\exists)\frac{\Gamma \vdash S \rightsquigarrow Q}{\Gamma \vdash \texttt{EXISTS}\ S \rightsquigarrow_e \neg\mathsf{empty}(Q)}$$

$\boxed{\Gamma \vdash B \rightsquigarrow_b \beta}$

$$(\mathrm{Tr}E)\frac{\Gamma \vdash E\ \texttt{AS}\ alias(E) \rightsquigarrow_b e}{\Gamma \vdash E \rightsquigarrow_b e} \qquad (\mathrm{Tr}Eas)\frac{\Gamma \vdash E \rightsquigarrow_e e}{\Gamma \vdash E\ \texttt{AS}\ N \rightsquigarrow_b e\ \mathsf{as}\ N}$$

$$(\mathrm{Tr}B)\frac{\Gamma \vdash B \rightsquigarrow_b \beta \quad \Gamma \vdash G \rightsquigarrow_b e}{\Gamma \vdash B, G \rightsquigarrow_b \beta, e}$$

$\boxed{\Gamma \vdash F \rightsquigarrow Q, \Gamma \vdash S \rightsquigarrow Q}$

$$(\mathrm{Tr}R)\frac{}{\Gamma \vdash R \rightsquigarrow R} \qquad (\mathrm{Tr}SW)\frac{\Gamma \vdash B \rightsquigarrow_b \beta \quad \Gamma \vdash F \rightsquigarrow Q \quad \Gamma \vdash C \rightsquigarrow_e \theta}{\Gamma \vdash \texttt{SELECT}\ B\ \texttt{FROM}\ F\ \texttt{WHERE}\ C \rightsquigarrow \pi_\beta(\sigma_\theta(Q))}$$

$$(\mathrm{Tr}S)\frac{\Gamma \vdash B \rightsquigarrow_b \beta \quad \Gamma \vdash F \rightsquigarrow Q}{\Gamma \vdash \texttt{SELECT}\ B\ \texttt{FROM}\ F \rightsquigarrow \pi_\beta(Q)} \qquad (\mathrm{Tr}\times)\frac{\Gamma \vdash F_1 \rightsquigarrow Q_1 \quad \Gamma \vdash F_2 \rightsquigarrow Q_2}{\Gamma \vdash F_1, F_2 \rightsquigarrow Q_1 \times Q_2}$$

$$(\mathrm{Tr}Fas)\frac{\Gamma \vdash S \rightsquigarrow Q \quad \ell(\Gamma, Q) = \overline{N_i}}{\Gamma \vdash S\ N \rightsquigarrow \pi_{\overline{N_i \mathsf{as} N.N_i}}(Q)} \qquad (\mathrm{Tr}Fasr)\frac{\ell(\Gamma, R) = \overline{N_i}}{\Gamma \vdash R\ N \rightsquigarrow \pi_{\overline{N_i \mathsf{as} N.N_i}}(R)}$$

$$(\mathrm{Tr}O_b)\frac{\Gamma \vdash S_1 \rightsquigarrow Q_1 \quad \Gamma \vdash S_2 \rightsquigarrow Q_2 \quad \ell(\Gamma, Q_1) = \overline{N_{1i}} \quad \ell(\Gamma, Q_2) = \overline{N_{2i}}}{\Gamma \vdash S_1 O_b S_2 \rightsquigarrow \pi_{\overline{N_{1i}}}(Q_1)\ toRA(O_b)\ \pi_{\overline{N_{2i}}}(Q_2)}$$

$$(\mathrm{Tr}O_s)\frac{\Gamma \vdash S_1 \rightsquigarrow Q_1 \quad \Gamma \vdash S_2 \rightsquigarrow Q_2 \quad \ell(\Gamma, Q_1) = \overline{N_{1i}} \quad \ell(\Gamma, Q_2) = \overline{N_{2i}}}{\Gamma \vdash S_1 O_s S_2 \rightsquigarrow \varepsilon(\pi_{\overline{N_{1i}}}(Q_1)\ toRA(O_s)\ \pi_{\overline{N_{2i}}}(Q_2))}$$

$$(\mathrm{Tr}-)\frac{\Gamma \vdash S_1 \rightsquigarrow Q_1 \quad \Gamma \vdash S_2 \rightsquigarrow Q_2 \quad \ell(\Gamma, Q_1) = \overline{N_{1i}} \quad \ell(\Gamma, Q_2) = \overline{N_{2i}}}{\Gamma \vdash S_1\ \texttt{EXCEPT}\ S_2 \rightsquigarrow \varepsilon(\pi_{\overline{N_{1i}}}(Q_1)) - \pi_{\overline{N_{2i}}}(Q_2)}$$

Fig. 7. SQL to RA translation

TRAF operators:

$$toRA(\texttt{UNION ALL}) = \cup \qquad toRA(\texttt{INTERSECT ALL}) = \cap \qquad toRA(\texttt{EXCEPT ALL}) = -$$
$$toRA(\texttt{UNION}) = \cup \qquad toRA(\texttt{INTERSECT}) = \cap.$$

Rule (TrO_s) is similar to (TrO_b), but it adds a duplicate elimination operator $\varepsilon(\cdot)$ over the bag operation to model set operations. The map $toRA(\cdot)$ fixes only the binary operator symbol: `UNION` and `UNION ALL` both translate to $\cup$, and likewise `INTERSECT` and `INTERSECT ALL` to $\cap$. The set-versus-bag distinction is not lost, because (TrO_s) wraps the operator with $\varepsilon(\cdot)$: `UNION` becomes $\varepsilon(\cup)$ while `UNION ALL` stays $\cup$, so evaluating the translation preserves the intended semantics. Finally, rule (Tr$-$) translates the `EXCEPT` operator by translating each subquery, projecting their columns, duplicate-eliminating the left operand, and applying $-$. Duplicate-eliminating the right operand is not needed: removing the same value once or many times has the same effect on a set-valued left operand.

In addition, we prove that every query S can be translated to a unique query in TRAF. The theorem is a well-definedness result for the source-to-core translation, not a claim that all DBMSs will execute the translated query with identical observable behavior. As Section 6 shows, practical optimizers can skip or reorder expressions and thereby hide or expose runtime cast errors. Those optimizer-induced differences are orthogonal to the syntactic uniqueness of the translation and are treated as validation mismatches rather than as nondeterminism in the translation relation.

Theorem 3.1. *For any SQL query S there exists a unique relational algebra query Q, such that $\Gamma \vdash S \rightsquigarrow Q$.*

4 Instantiating TRAF

In this section we illustrate how to instantiate TRAF for PSQL, Oracle and SQLite. The complete rules and the instantiations for other engines (MSSQL and MySQL) can be found in the appendix. 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 8 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 and null to ?. The operator for removing unknown types replaces occurrences of ? with String, while leaving other types unchanged.[6] 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, operations return null if any argument is null; otherwise, they are executed by the underlying implementation unchanged.

Bidirectional implicit cast. Operator *biconv* is defined using the auxiliary *implicit 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 a singleton numeric result (printed as `1` or `1.0` depending on the client and selected common type), since $\mathbb{Z}$ (the type of `1`) can be implicitly cast to $\mathbb{R}$ (the type of `1.0`). Implicit value cast $icast(v, \tau) = v'$ is defined for extracting numbers from strings, but not viceversa. Conversions from integer to string are only allowed in explicit casts. For instance, `(SELECT '-1.0' FROM R) INTERSECT (SELECT -1.0 FROM R)` is accepted and evaluates to a set containing -1.0. In these rules,

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

Basic definitions

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

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

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

$\boxed{e : \tau \rightsquigarrow \tau'}$

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

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

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

$\boxed{e : \tau \approx\!\!> \tau'}$

$$\frac{e \neq v}{e : \tau \approx\!\!> \tau'} \qquad \frac{cast(v, \tau') = v'}{v : \tau \approx\!\!> \tau'}$$

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

$$cast(n_1.n_2, \mathbb{Z}) = n_1$$
$$cast(v, \mathsf{String}) = \text{`}v\text{'} \quad v \neq s$$
$$cast(v, \tau) = icast(v, \tau)$$

$\boxed{bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau}$

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

$\boxed{cost(\tau_1, \tau_2) = n}$

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

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau}$

$$\frac{\tau_1 = ? \qquad \tau_2 = ? \qquad bestCandidate(\mathsf{String}, \mathsf{String}, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$$

$$\frac{\tau_1 \neq ? \qquad \tau_2 = ? \qquad icast(e_2, \tau_5) = v_2' \qquad bestCandidate(\tau_1, \tau_1, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$$

$$\frac{\tau_1 = ? \qquad \tau_2 \neq ? \qquad icast(e_1, \tau_4) = v_1' \qquad bestCandidate(\tau_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$$

$$\frac{\tau_1 \neq ? \qquad \tau_2 \neq ? \qquad bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$$

Fig. 8. The TRAF/PSQL Instantiation

metavariables such as 'n' and 'r' range over strings that the corresponding engine parses unambiguously as the integer n or real r; if parsing fails, the conversion relation is undefined.

Explicit cast feasibility. If the expression tested is not a value, the relation vacuously holds. Conversely, if e is a value v, it is eagerly checked whether the explicit cast to τ' fails or not, using the explicit cast relation.

Explicit cast. In addition to what an implicit cast can do, explicit casts 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 DECIMAL(10,1)) FROM R` is accepted and evaluates to 1.1, but `SELECT CAST('1.1' AS INT) FROM R` is rejected by the typechecker.

Overload resolution. This rule is divided in four cases. If both expressions are unknown then the best candidate is searched assuming both types are strings. If any of the types is unknown, the best candidate is searched using the known type in both positions. The general case arises when the types of the two expressions are not unknown. Function *bestCandidate* is used to determine the best candidate. We model this function by initially calculating the sum of the type conversion costs between the corresponding types of the expression and the domain of each candidate. The function $cost(\tau, \tau')$ 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 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/Oracle

Oracle handles types and casts in a middle ground between PSQL and the more permissive engines such as SQLite. Figure 9 presents the TRAF/Oracle instantiation. For the basic function definitions, TRAF/Oracle behaves similarly to PSQL except for string literals, which are typed directly as String. The operator for removing unknown types acts as the identity function, and the operator for inserting annotations introduces annotations for every operator. The operator for inserting annotations and the $apply(0, \cdot, \cdot)$ operation work exactly as in PSQL.

Bidirectional implicit cast. The operator *biconv* is defined as in PSQL, but the *implicit cast* relation differs. There is a special case that allows casting from the unknown type to any other type without performing any additional implicit-cast check.

Explicit cast feasibility. Contrary to PSQL, this operator is more flexible: every expression can be explicitly cast to any type.

Explicit cast. The explicit cast operator is similar to PSQL, but string literals can be directly cast to integers even when they represent decimal numbers.

Overload resolution. This rule is simplified in contrast to PSQL. There is a single rule that always selects the best candidate using the type-difference operator. The type difference behaves as in PSQL, except that there is a special case for the unknown type that assigns a difference of 1 when casting from the unknown type to any other type.

Basic definitions

$$ty(d) = \mathbb{R} \quad ty(n) = \mathbb{Z} \quad ty(s) = \text{String} \quad ty(\text{null}) = ?$$
$$insert(e, \tau, 0) = e :: \tau \;\; 0 \in \{+, <, =\} \qquad clean(T) = T$$
$$ty(0_C) = \{\mathbb{R} \times \mathbb{R} \to \mathbb{B}, \text{String} \times \text{String} \to \mathbb{B}\} \cup \{? \times ? \to \mathbb{B}\}$$
$$ty(+) = \{\mathbb{R} \times \mathbb{R} \to \mathbb{R}, ? \times ? \to ?\}$$
$$apply(0, v_1, v_2) = \text{null}\ (v_1 = \text{null} \vee v_2 = \text{null}) \qquad apply(0, v_1, v_2) = v_1\ 0\ v_2 \text{ otherwise}$$

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

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

$\boxed{e : \tau \rightsquigarrow \tau'}$

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

$\boxed{e : \tau \not\rightsquigarrow \tau'}$

$$\frac{}{e : \tau \not\rightsquigarrow \tau'}$$

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

$$\begin{array}{rlrl} cast(\text{null}, \tau) & = \text{null} & cast('n', \mathbb{Z}) & = n \\ cast('r', \mathbb{R}) & = r & cast(v, ty(v)) & = v \\ cast('n', \mathbb{R}) & = n & cast('n_1.n_2', \mathbb{Z}) & = n_1 \\ cast(n_1.n_2, \mathbb{Z}) & = n_1 & cast(n, \mathbb{R}) & = n \\ cast(v, \text{String}) & = \text{`}v\text{'} & cast(v, ?) & = v \end{array}$$

$\boxed{cost(\tau_1, \tau_2) = n}$

$$cost(\tau, \tau) = 0 \;\; cost(\mathbb{Z}, \mathbb{R}) = 0 \;\; cost(\text{String}, \mathbb{R}) = 1 \;\; cost(?, \_) = 1 \;\; cost(\_, \_) = 2$$

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$

$$\frac{\{m\} = \arg\min_i((cost(\tau_1, \tau_{1i})) + (cost(\tau_2, \tau_{2i})))}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_{1m} \times \tau_{2m} \to \tau_{3m}}$$

Fig. 9. The TRAF/Oracle Instantiation

4.3 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 10 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.

Basic definitions

$$ty(v) = ? \quad dty(w :: ?) = ? \quad dty(r) = \mathbb{R} \quad dty(s) = \text{String}$$

$$insert(e, \tau, 0_C) = e \quad insert\ (e, \tau, +) = e :: \tau \qquad clean(T) = T$$

$$ty(0_C) = \{? \times ? \to ?, \mathbb{R} \times \mathbb{R} \to ?, \text{String} \times \text{String} \to ?\} \cup \{? \times ? \to ?, \mathbb{B} \times \mathbb{B} \to ?\}$$

$$ty(+) = \{\mathbb{R} \times \mathbb{R} \to ?\}$$

$$apply(0, v_1, v_2) = \text{null} \quad (v_1 = \text{null} \vee v_2 = \text{null}) \qquad apply(+, v_1, v_2) = v_1 + v_2 \text{ otherwise}$$

$$apply(0_C, v_1, v_2) = compare(|v_1'|, |v_2'|) :: ?$$
$$\text{where } resolve(v_1, dty(v_1), v_2, dty(v_2), ty(0_C)) = \tau_1 \times \tau_2 \to \tau_3,$$
$$icast(v_1, \tau_1) = v_1', icast(v_2, \tau_2) = v_2'$$

$$|w :: ?| = w \quad |w| = w$$

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

$$\frac{}{biconv(e_1, \tau_1, e_2, \tau_2) = ?}$$

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

$$number(\text{`}n\text{'}) = n \qquad number(\text{`}r\text{'}) = r$$
$$icast(v, ty(v)) = v \qquad icast(n_1.n_2, \mathbb{Z}) = n_1$$
$$icast(n, \mathbb{R}) = n \qquad icast(v, \text{String}) = \text{`}v\text{'} \quad v \neq s$$
$$icast(w :: ?, \tau) = icast(w, \tau) \qquad icast(\text{null}, \tau) = \text{null}$$
$$icast(s, \tau) = icast(number(s), \tau) \quad \tau \in \{\mathbb{Z}, \mathbb{R}\} \qquad icast(s, \tau) = s \quad number(s) \text{ is not def.}$$

$\boxed{e : \tau \rightsquigarrow \tau'}$

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

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

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

$\boxed{cost(\tau_1, \tau_2) = n}$

$$cost(\tau, \tau) = 0 \qquad cost(\mathbb{Z}, \mathbb{R}) = 0 \qquad cost(\text{String}, \mathbb{R}) = 1 \qquad cost(?, \tau) = 1 \qquad cost(\_, \_) = 2$$

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau}$

$$\frac{\{m\} = \arg\min_i((cost(\tau_1, \tau_{1i})) + (cost(\tau_2, \tau_{2i})))}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_{1i} \times \tau_{2i} \to \tau_{3i}}) = \tau_{1m} \times \tau_{2m} \to \tau_{3m}}$$

Fig. 10. The TRAF/SQLite Instantiation

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

$$compare(w_1, w_2) = \begin{cases} 1 & (dty(w_1) = dty(w_2) \wedge w_1 < w_2) \vee dty(w_1) < dty(w_2) \\ 0 & \text{otherwise} \end{cases}$$

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 a fixed ordering on dynamic type tags rather than on value sets: numeric values, represented by integer and real tags, are ordered before text values, such that $\mathbb{Z} < \mathbb{R}, \mathbb{R} < \mathbb{Z}$ and $\mathbb{R} <$ String.

To illustrate the comparison behavior in SQLite, we provide the following examples:

```
SELECT '0' < 1 FROM R;
SELECT '0' < CAST(1 AS INT) FROM R;
SELECT '0hi' < 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 is $? \times ? \longrightarrow ?$, and the implicit cast leaves them untouched. Finally, as the dynamic type of the 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 $\mathbb{R} \times \mathbb{R} \longrightarrow ?$, 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`.

The third example evaluates to `0`. It follows a pattern similar to the second example, but with the implicit cast to a numeric type failing on the left expression, resulting in the same string as the output. Consequently, since the dynamic types of both operands differ (string and real), a type comparison is carried out, leading to the final result of `0`.

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 identical 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' AS INT) FROM R` evaluates to $(12, 0)$.

Overload resolution. Similar to Oracle, 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 going from integer to real, and 2 from unknown to any other type.

4.4 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 $ty(+) =$

$\{\mathbb{R}\times\mathbb{R}\to ?\}$, $resolve(\texttt{'1.1'}, ?, 1, ?, ty(+)) = \mathbb{R}\times\mathbb{R}\to ?$, and

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

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 $resolve(\texttt{'1.1'}, ?, 1, \mathbb{Z}, ty(+))$ requires $icast(\texttt{'1.1'}, \mathbb{Z})$ to be defined, but it is not. What is defined is the implicit cast to a real $icast(\texttt{'1.1'}, \mathbb{R})$. We can force such a cast with the following translation that preserves the behavior: `SELECT CAST('1.1' AS DECIMAL(10,1)) + 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. Recent PostgreSQL versions provide polymorphic pseudo-types such as `anycompatible` for declaring functions, but these pseudo-types do not give SQL expressions a first-class runtime type argument that can be passed to `CAST`. They therefore do not remove the need for either case-splitting translations or generated engine-specific helper functions.

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 FROM R WHERE CAST('1' AS INT) < 2`, or surprisingly, by casting the right operand: `SELECT 1 FROM R 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.

5 Properties

Based on the core relational algebra of TRAF, we can establish metatheoretical results for each engine considered, consisting of lemmas and theorems regarding queries and their evaluation. Specifically, we can articulate the formal distinctions among various engines, and pinpoint the exact requirements that abstract operators (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.

R1: Every operator must be deterministic and total for their right types.
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 $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6$, then
$[\![e_1 :: \tau_4]\!]_{\eta,\mathtt{D},\Gamma} \neq$ **error** and $[\![e_2 :: \tau_5]\!]_{\eta,\mathtt{D},\Gamma} \neq$ **error**.
R6: If $biconv(e_1, \tau_1, e_2, \tau_2) = \tau$, then
$[\![e_1 :: \tau]\!]_{\eta,\mathtt{D},\Gamma} \neq$ **error** and $[\![e_2 :: \tau]\!]_{\eta,\mathtt{D},\Gamma} \neq$ **error**.
R7: If $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6$ then $e_1 : \tau_1 \lessapprox \tau_4$ and $e_2 : \tau_2 \lessapprox \tau_5$.
R8: If $biconv(e_1, \tau_1, e_2, \tau_2) = \tau$ then $e_1 : \tau_1 \lessapprox \tau$ and $e_2 : \tau_2 \lessapprox \tau$.
R9: Either $clean(T) = T[\mathsf{String}/?]$ or $clean(T) = T$.

Fig. 11. Requirements that abstract operators must meet to satisfy the properties.

Table 3. Requirements satisfied by the DBMS formalizations used in this paper.

Req.	PSQL	MSSQL	Oracle	MySQL	SQLite
R1	✓	✓	✓	✓	✓
R2	✓	✓	✓	✓	✓
R3	✓	✓	✓	✓	✓
R4	✓	✓	✓	✓	✗
R5	✓	✗	✗	✓	✓
R6	✓	✗	✓	✓	✓
R7	✓	✓	✓	✓	✓
R8	✓	✓	✓	✓	✓
R9	✓	✓	✓	✓	✓

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

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

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

A row is well-typed if every value has the type specified for it in T. Note that null is assigned any type τ, rather than the unknown type ?. For example, `CAST(NULL AS INT)` evaluates to `NULL`, and to preserve the type of the expression, the resulting `NULL` is typed as `INT`. 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. The proofs require some properties about the implementation of abstract operators shown in Figure 11.

Intuitively, these requirements isolate the engine-dependent facts used in the proofs. R1 states that the abstract operators behave functionally. R2 and R3 say that bidirectional conversion and overload resolution do not invent types outside the candidate types already considered by the type system. R4 connects runtime conversion with typing: a successful explicit cast to τ must produce a value whose cleaned type is τ. R5 and R6 are the cast-success assumptions used for cast-free source queries. Although a cast-free source query contains no explicit cast, cast insertion can add casts after

overload resolution (R5) or bidirectional conversion in set operations (R6); these requirements state that those inserted casts cannot fail at runtime. R7 and R8 are the corresponding static consistency assumptions for cast insertion: when overload resolution or bidirectional conversion selects a target type, the explicit cast-feasibility relation accepts that cast. Finally, R9 constrains the post-processing function $clean(\cdot)$ to the two forms used by the engines considered here: either it is the identity, or it replaces unknown columns by strings. This restriction is needed in the cast-free theorem because the proof relates source and target relation types modulo $clean(\cdot)$; without R9, an instantiation could normalize unknown columns to arbitrary types, making the casts inserted by overload resolution or bidirectional conversion fall outside the cases covered by R5 and R6.

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. In other words, evaluation of well-typed queries does not get stuck. For instance, an ill-type query, or `SELECT Foo FROM P`, where `Foo` is not defined in `P`, gets stuck as $[\![\texttt{Foo}]\!]_{\eta,\mathtt{D},\Gamma}$ does not evaluate further.

Theorem 5.1 (Type Safety). *If R1, R2 and R3 hold, then* $\forall T\, T'\, Q\, \mathtt{D}$, *if* $\Gamma; T' \vdash Q : T$ *and* $\vdash \mathtt{D} : \Gamma$ *then* $(\exists t, [\![Q]\!]_{\eta,\mathtt{D},\Gamma} = t) \vee ([\![Q]\!]_{\eta,\mathtt{D},\Gamma} = \textbf{error})$.

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

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 $A \mapsto$ String evaluates to $\{$`'hi'`, ...$\}$, typed as $A \mapsto ?$ (literal strings are typed ?), but $clean(\text{String}) = clean(?) = \text{String}$.

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 $\mathtt{CF}(Q)$, which is defined when Q does not have explicit casts of the form $e :: \tau$. The definition of $\mathtt{CF}(Q)$ defined inductively and is given in Figure 12.

Theorem 5.3 (Cast-free queries do not fail). *If R1, R5, R6 and R9 hold, and* $\mathtt{CF}(Q)$, $\vdash \mathtt{D} : \Gamma$ *and* $\Gamma; Q \vdash T : Q' \rightsquigarrow$ *then* $[\![Q']\!]_{\eta,\mathtt{D},\Gamma} \neq \textbf{error}$.

This theorem requires R1, R5, R6 and R9. Both MSSQL and Oracle do not satisfy this property. Specifically, MSSQL does not satisfy R5 and R6, and Oracle does not satisfy R5. The failure of R5 can be seen with table $R = \{$(`'1'`), (`'hi'`)$\}$, schema $R \mapsto (A \mapsto \text{String})$, and query `SELECT A + 1 FROM R`. For MSSQL and Oracle, overload resolution accepts an addition signature that requires converting A to a numeric type, but the conversion fails on `'hi'`. Thus the source query is cast-free, but the cast inserted because of overload resolution can still raise a runtime error. For MSSQL, R6 fails for the analogous reason in set operations: bidirectional conversion can select a target type for which an inserted cast later fails on some runtime value.

Regarding cast insertion, the translation of a well-typed query typechecks and is unique:

Theorem 5.4 (Cast insertion is a type-preserving function). *If R1, R2, R4, R7, R8 and R9 hold, and* $\Gamma; T' \vdash Q : T$ *then there exists a unique* Q' *such that* $\Gamma; T' \vdash Q : T \rightsquigarrow Q'$ *and* $\Gamma; T' \vdash Q' : T$.

$$\boxed{\mathsf{CF}(Q), \mathsf{CF}(\beta), \mathsf{CF}(e), \mathsf{CF}(\theta)}$$

$$\frac{}{\mathsf{CF}(R)} \qquad \frac{\mathsf{CF}(\beta) \quad \mathsf{CF}(Q)}{\mathsf{CF}(\pi_\beta(Q))} \qquad \frac{\mathsf{CF}(\theta) \quad \mathsf{CF}(Q)}{\mathsf{CF}(\sigma_\theta(Q))} \qquad \frac{\mathsf{CF}(Q_1) \quad \mathsf{CF}(Q_2)}{\mathsf{CF}(Q_1\ \mathsf{O_S}\ Q_2)}$$

$$\frac{\mathsf{O} \in \{+, <, =\} \quad \mathsf{CF}(e_1) \quad \mathsf{CF}(e_2)}{\mathsf{CF}(e_1\ \mathsf{O}\ e_2)} \qquad \frac{\mathsf{CF}(\theta_1) \quad \mathsf{CF}(\theta_2)}{\mathsf{CF}(\theta_1\ \mathsf{O_B}\ \theta_2)} \qquad \frac{}{\mathsf{CF}(N)} \qquad \frac{}{\mathsf{CF}(v)}$$

$$\frac{\mathsf{CF}(e)}{\mathsf{CF}(e\ \mathsf{as}\ N)} \qquad \frac{\mathsf{CF}(e) \quad \mathsf{CF}(\beta)}{\mathsf{CF}(\beta, e\ \mathsf{as}\ N)} \qquad \frac{\mathsf{CF}(e)}{\mathsf{CF}(e\ \mathsf{is\ null})} \qquad \frac{\mathsf{CF}(Q)}{\mathsf{CF}(\mathsf{empty}(Q))}$$

$$\frac{\mathsf{CF}(e_1) \ldots \mathsf{CF}(e_k) \quad \mathsf{CF}(Q)}{\mathsf{CF}((e_1, \ldots, e_k) \in Q)}$$

Fig. 12. Cast-free Queries.

Table 4. Main meta-theoretic properties, the operator requirements they depend on, and which DBMS formalizations satisfy them.

Property / Theorem	**Reqs.**	PSQL	MSSQL	Oracle	MySQL	SQLite
Type Safety	R1, R2, R3	✓	✓	✓	✓	✓
Type Soundness	R1, R2, R3, R4	✓	✓	✓	✓	✗
Cast-free queries do not fail	R1, R5, R6, R9	✓	✗	✗	✓	✓
Cast insertion is type-preserving	R1, R2, R4, R7, R8, R9	✓	✓	✓	✓	✗

This theorem requires R1, R2, R4, R7, R8 and R9, and is satisfied by all the engines we studied except SQLite.

Table 4 summarizes the main meta-theoretic properties established in this section, the operator requirements each property depends on, and which TRAF formalization of each studied DBMS satisfies it.

6 Experimental Validation

In order to experimentally validate TRAF, we developed a Python reference implementation, PyTRAF, with one engine instance per target DBMS. The main objective is to study the semantic alignment between PyTRAF instances and the real-world engines they are meant to represent.[7] Given a SQL query, a *validation against a target engine* consists in comparing the outcome of (a) typechecking the query in PyTRAF and evaluating it if well typed, and (b) submitting the query to the target engine. The two outcomes are compared as row sets, errors, or rejections.

We report on three complementary validations. First, we validate 100,000 random queries against each engine, in four configurations, and report how many are statically rejected, how many agree

[7]This evaluation is concerned with semantic alignment, not with the actual performance of PyTRAF. However, we did validate that typechecking adds negligible overhead relative to query execution. Specifically, we measured the typechecker in isolation on two workloads, timing only the typechecking procedure. On the 3,472 Spider queries that the interpreter parses (across 133 schemas), typechecking averages 57.1 μs per query (min 7.5 μs, max 3.0 ms, standard deviation 62.6 μs), and the suite typechecks in 0.198 s. On 100,000 randomly generated queries from the PyTRAF generator, typechecking averages 45.4 μs per query (min 11.0 μs, max 0.89 ms, standard deviation 23.7 μs), and the batch typechecks in 4.54 s. The means remain in the tens of microseconds across both workloads, consistent with a syntax-directed type system whose cost is linear in query size.

with the engine, and how disagreements split across error kinds (Section 6.2). Second, we validate the Spider [Yu et al. 2018] and Calcite [Begoli et al. 2018] benchmarks against the five engines (Section 6.3). Third, we use SQLancer++ [Zhong and Rigger 2026] as a SQL fuzzer with an adaptive statement generator that constructs database states while maintaining an internal schema model, and then generates random queries whose feature choices are refined through feedback from the target DBMS. This surfaces corner cases that the fixed-schema generator of the first axis cannot reach (Section 6.4).

Across all three axes, the same overall pattern emerges: PSQL, MySQL, and SQLite agree with PyTRAF after the simplifications described in Section 6.1, while disagreements manifest with both MSSQL and Oracle. The disagreements reduce to a small set of cataloged cases (Section 6.5): optimizer rewrites that hide runtime errors, engine-specific behaviors (reported to vendors), integer-overflow artifacts mitigated by representing integers as `BIGINT`, and the remaining open cases listed in Table 7.

6.1 Experimental Setup

PyTRAF is composed of a query generator, a parser, a typechecker, and an interpreter. The PyTRAF interpreter is roughly 7.4k lines of Python, and the validation suite (query generator, benchmark execution, result tabulation) around 19k lines. We use the following engine versions: PostgreSQL 14.13, MySQL 9.4.0, SQLite 3.31.1 (Python `sqlite3`), Microsoft SQL Server 2022, and Oracle Database 21c XE. All experiments are run on an Apple M3 Pro machine with 18 GB of memory.

For random fuzzing, we use three tables `TA`, `TB`, `TC`, each with three columns covering strings, integers, and real numbers. We keep the tables small so that each validation run is short. The generator combines arithmetic and comparison operators, casts, `IS NULL` checks, `IN` predicates, selections, projections, cross products, unions, intersections, and set differences. The generator applies two engine-agnostic adjustments in every configuration. MSSQL and Oracle lack a native boolean type, so we represent booleans using integers, both in PyTRAF and in the engines. Real numbers are represented as `DECIMAL(10,1)` in the generated schemas and as decimal values in PyTRAF to avoid floating-point precision mismatches.[8] Comparing real numbers with a tolerance becomes problematic once the value is cast to a string. The generator also rewrites the SQL submitted to MySQL by inserting explicit `DECIMAL(10,1)` casts for arithmetic operands that would otherwise follow MySQL's floating-point evaluation path. This is a target-specific rewrite of the engine-side SQL query, not a change to the PyTRAF term, which already uses decimal semantics for these values. The rewrite avoids spurious mismatches such as `select 'hi' + 5.1 - 9 from R` evaluating to -3.9 in PyTRAF but to -3.9000000000000004 in MySQL.

Note that the PyTRAF interpreter goes beyond the formalism presented in this article, with basic support for aggregation, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`, `LIKE`, and additional operators required by Spider and Calcite. These extensions allow us to validate real-world queries, but are not formalized in TRAF. Claims about type-conversion behavior in this section only apply to the subset covered by the formalism.

6.2 Random Query Fuzzing

We generate 100,000 random queries per engine in four configurations, obtained by independently switching two options: *simplification* on or off, and *nulls* on or off. Without simplification the generator emits queries that the optimizations of the engines may rewrite; with simplification, the generator avoids the constructs that we have shown trigger those rewrites (see Section 6.5).

[8] In Python, $1.9 - 0.1 = 1.7999999999999998$.

The outcome a query validation falls into five exclusive cases: both sides return the same row set (*match*); both raise an error (*both failed*); only the interpreter raises (*interp-only err*); only the engine raises (*engine-only err*); both succeed but return different rows (*value mismatch*). We treat *match* and *both failed* as agreement.

Table 5 reports, for each (configuration, engine) pair, the total number of queries, how many were statically rejected by the typechecker, how many remained well-typed, how many agreed with the engine, and how many disagreed. The *statically rejected* column gives the count of queries that TRAF rejects before evaluation.

The corpus includes a large fraction of ill-typed queries (under *no-sim, no-null*, 77% for PSQL and 44% for Oracle), because agreement on rejection is itself a positive result: when PSQL refuses to typecheck a query and PyTRAF rejects it under the same rules, the formalism is validated on that case. The experiment therefore measures two forms of agreement. The well-typed subset measures agreement on *evaluation*; the rejected subset measures agreement on static typing.[9] Static agreement matters most for PSQL, whose type system is among the strictest of the five engines, so that a substantial fraction of random queries are ill-typed by construction. The *agree* column of Table 5 captures both forms, since *both failed* subsumes queries that both sides reject at typecheck time. Restricting the corpus to well-typed queries would hide this form of agreement.

Table 5 supports three observations. SQLite shows 0 disagreements in every configuration, MySQL shows at most 7, and PSQL has 0 disagreements once simplification is applied. The five MySQL disagreements under *sim, no-null* correspond to behaviors the MySQL team has reviewed (see Section 6.5). Simplification has a large effect on MSSQL (disagreement drops from 6,986 to 746 in the no-null case) and on Oracle (from 4,852 to 1,098). This drop comes from the generator avoiding constructs known to trigger optimizations; the formalism does not resolve these cases on its own. Nulls increase disagreement for MSSQL and Oracle specifically, because the arithmetic-with-null behavior of those engines is the dominant remaining source of divergence (see Section 6.5.4).

Table 6 partitions the *disagree* column into the three outcome subcategories. This analysis clarifies two aspects of the residual mismatches. Most MSSQL and Oracle disagreements without simplification are *interp-only err*: the interpreter raises while the engine returns a table, indicating that the engine has skipped the failing subexpression. These are cataloged in Section 6.5. The *value mismatch* column isolates the cases where both sides succeed but disagree on the result; the small MySQL entries in this column across all four configurations are the UNION/INTERSECT behaviors discussed in Section 6.5.5. PSQL, SQLite, and MSSQL record zero *value mismatch* entries in every configuration; the remaining non-zero values in the column belong to Oracle and grow under simplification. The simplified generator emits more well-typed queries that Oracle evaluates without error but on whose result sets the interpreter and the engine disagree (Section 6.5.4).

6.3 Real-World Benchmark Validation

While random queries cover the constructs the generator can produce, they do not resemble the queries people write against real databases, with realistic schemas and idioms. To cover those, we validate PyTRAF with two real-world SQL benchmarks.

SPIDER [Yu et al. 2018]. SPIDER contains 3,678 test queries, each with its expected result, over 137 database schemas. We exclude 206 queries (related to the four schemas `baseball_1`, `bike_1`, `college_2`, and `flight_4`) because PyTRAF does not terminate in a reasonable amount of time: these databases hold tables of up to 170,000 rows, and PyTRAF joins tables with a naive nested-loop algorithm, so a single two-table join can take on the order of 10^{10} Python-level operations. As we

[9] This includes basic syntax checking: for instance, SQLite does not perform typechecking per se, but checks the syntax of a query prior to running it.

Table 5. Top-level fuzzing results, 100,000 queries per (configuration, engine). *stat. rej.* counts queries rejected statically by the typechecker (the *ill-typed* count); *w.t.* is well-typed (total − stat. rej.); *agree* counts agreement (*match* or *both failed*); *disagree* is the complement.

Configuration	Engine	total	stat. rej.	w.t.	agree	disagree
no-sim, no-null	PSQL	100,059	77,553	22,506	99,649	410
	MySQL	100,033	33,335	66,698	100,027	6
	SQLite	100,025	33,317	66,708	100,025	0
	MSSQL	100,022	8,982	91,040	93,036	6,986
	Oracle	100,057	43,851	56,206	95,205	4,852
no-sim, with null	PSQL	100,059	74,710	25,349	99,508	551
	MySQL	100,033	33,205	66,828	100,026	7
	SQLite	100,025	32,933	67,092	100,025	0
	MSSQL	100,022	8,739	91,283	88,764	11,258
	Oracle	100,057	42,040	58,017	93,859	6,198
sim, no-null	PSQL	100,059	54,846	45,213	100,059	0
	MySQL	100,033	0	100,033	100,028	5
	SQLite	100,025	0	100,025	100,025	0
	MSSQL	100,022	3,432	96,590	99,276	746
	Oracle	100,057	20,747	79,310	98,959	1,098
sim, with null	PSQL	100,059	48,771	51,288	100,059	0
	MySQL	100,033	0	100,033	100,027	6
	SQLite	100,025	0	100,025	100,025	0
	MSSQL	100,022	2,766	97,256	98,315	1,707
	Oracle	100,057	16,787	83,270	98,762	1,295

discuss in Section 6.6, this join performance aspect is unrelated to the type-conversion semantics we are studying. Importantly, PyTRAF and all five engines agree on the result of each the remaining 3,472 queries.

Calcite [Begoli et al. 2018]. This corpus comes from Calcite's optimizer test suite: 397 pairs of queries—794 queries in total—where each pair consists of a query and a rewritten version that Calcite considers equivalent. We do not use the pairing: we validate each query separately against each engine, so the corpus serves as a set of individual realistic queries rather than as equivalence checks. We skip queries that use `OVER`, `PARTITION`, or `FULL OUTER JOIN`, which PyTRAF does not yet support; 250 queries remain and 544 are skipped. On the queries we run, PyTRAF agrees with every engine on every query.

Spider Adoption: Preprocessing. Spider was curated against SQLite and includes idioms that violate standard SQL or that depend on SQLite-specific semantics. Validating the corpus against five engines required the following adaptations. Each is a syntactic or data change that preserves the intended meaning of the query and yields a result on which all five engines agree.

- *Permissive* `GROUP BY`. Queries select a non-aggregate column without listing it in `GROUP BY`, as in

Table 6. Analysis of disagreement. The table shows the same rows and configurations as Table 5, with the *disagree* count split into *interp-only err* (interpreter raises while engine succeeds), *engine-only err* (engine raises while interpreter succeeds), and *value mismatch* (both succeed but return different rows).

Configuration	Engine	disagree	interp-only err	engine-only err	value mismatch
no-sim, no-null	PSQL	410	391	19	0
	MySQL	6	0	0	6
	SQLite	0	0	0	0
	MSSQL	6,986	6,395	591	0
	Oracle	4,852	4,364	420	68
no-sim, with null	PSQL	551	514	37	0
	MySQL	7	0	0	7
	SQLite	0	0	0	0
	MSSQL	11,258	10,969	289	0
	Oracle	6,198	5,790	300	108
sim, no-null	PSQL	0	0	0	0
	MySQL	5	0	0	5
	SQLite	0	0	0	0
	MSSQL	746	383	363	0
	Oracle	1,098	131	316	651
sim, with null	PSQL	0	0	0	0
	MySQL	6	0	0	6
	SQLite	0	0	0	0
	MSSQL	1,707	1,486	221	0
	Oracle	1,295	175	338	782

```
SELECT T1.aircraft FROM aircraft AS T1
JOIN match AS T2 ON T1.aircraft_id = T2.winning_aircraft
GROUP BY T2.winning_aircraft
ORDER BY COUNT(*) DESC LIMIT 1;
```

PSQL rejects this with *column must appear in the* `GROUP BY` *clause*, while the SQLite/MySQL relaxed rule that Spider was written against accepts it. We rewrote the queries to standard SQL.

- **`UNION` *with* `GROUP BY` *column clash.*** Queries of the form

```
SELECT t1.name, t1.id FROM station AS t1
JOIN status AS t2 ON t1.id = t2.station_id
GROUP BY t2.station_id
HAVING avg(t2.bikes_available) > 14
UNION
SELECT name, id FROM station
WHERE installation_date LIKE '12/%';
```

select different non-grouped columns on either side of `UNION` after a `GROUP BY`. We rewrote these queries to standard SQL.

- *Joins without* `ON`. In queries of the form `FROM A JOIN B JOIN C ON ...` the second `JOIN` has no `ON` clause; we rewrote them to attach the correct condition to each `JOIN`.
- *Subqueries without alias.* We wrapped each such subquery in an explicit alias.
- `ORDER BY` *on a non-selected column under* `DISTINCT`. PSQL rejects these; we added the column to the `SELECT` list.
- *Scalar subquery returning multiple rows.* Queries such as

```
SELECT name FROM city
WHERE county_id =
  (SELECT county_id FROM county_public_safety
   ORDER BY police_officers DESC);
```

compare a column against a subquery that returns more than one row. SQLite silently returns the first row, while the formalism rejects the query as ill-formed. We added an explicit `LIMIT 1` to the subquery so that every engine agrees on a single-row result.

- `LIMIT` *without* `ORDER BY`. Some Spider queries use `LIMIT` without an `ORDER BY`, or with an `ORDER BY` that leaves ties. The kept rows are then whichever rows the engine happens to produce first, so the expected result recorded by Spider is not reproducible in general. Where the intended ranking was clear we added an explicit `ORDER BY`; otherwise we dropped the `LIMIT`. Both rewrites make the result deterministic.
- *Non-integer strings in* SQLite *integer columns.* Source data uses values such as `'94040-1724'`, `'nil'`, and `''` in integer columns, relying on SQLite's type-affinity rules to take the leading integer part or treat the value as `NULL`. We applied the same translation explicitly: `'94040-1724'` → `94040`, `'nil'` → `NULL`, `''` → `NULL`.

6.4 SQLancer++ Fuzzing Validation

SQLancer++ [Zhong and Rigger 2026] constructs database states and adapts each generated query to feedback from the target DBMS, in order to exercise code paths that our fixed-schema generator (Section 6.2) cannot reach. The corpus contains 76,808 generated cases, each consisting of a schema, a data instance, and a query to validate. When a case crashed the PyTRAF interpreter or returned a different result from the engine, we fixed the interpreter and kept the query as a regression test. After applying these fixes, the PyTRAF interpreter agrees with PSQL, MySQL, and SQLite on the entire corpus: zero *value mismatch* and zero *interp-only err* cases remain on those three engines. On MSSQL and Oracle the residual mismatches required two adjustments to the suite (the boolean adaptations and the dual-semantics evaluation for short-circuit, both described next) and exposed one new optimization, the `(p) IS NULL` → `(1=0)` rewrite, cataloged as row 11 of Table 7 (Section 6.5).

Boolean adaptations for MSSQL *and* Oracle. Neither dialect exposes a first-class boolean datatype, so every position where the generated query uses a predicate as a scalar operand must be rewritten before the engine parses it. We wrap each such predicate as `(CASE WHEN p THEN 1 WHEN NOT (p) THEN 0 ELSE NULL END)` in numeric contexts and as `(CASE WHEN p THEN 'True' WHEN NOT (p) THEN 'False' ELSE NULL END)` in string contexts, preserving three-valued logic on both sides. The rewrite families cover comparison and arithmetic operands, `LIKE` match and pattern positions, `CAST(<bool> AS INT|VARCHAR)`, `(p) IS [NOT] NULL`, and the symmetric set of Oracle wrappers—numeric `CASE` for comparison operands, string `CASE` for `LIKE` arguments, `CAST AS NUMBER` for `CAST AS INT`, `CAST AS VARCHAR2` via `CASE` for `CAST(<bool> AS VARCHAR)`. We replace the `TRUE` and

`FALSE` literals with `(1=1)` and `(1=0)` respectively. Furthermore, we add schema qualifiers to table references for MSSQL, and case-quote them for Oracle.

Dual-semantics evaluation for short-circuit. The SQL standard does not specify any evaluation order for the operands of `AND` and `OR`. Engines exploit this freedom: a plan may evaluate either operand first and skip the other when the first already decides the result. Skipping is not behavior-preserving when the skipped operand raises an error, so a query whose left side raises and whose right side is constantly `FALSE` can succeed or fail depending on the chosen plan. To avoid spurious mismatches, we evaluate every MSSQL and Oracle SQLancer++ query through the interpreter twice: once with strict left-to-right semantics, and once with logical short-circuit enabled A query validation is successful if each evaluation order agrees with the engine.

Residual mismatches. MSSQL produces 348 unresolved mismatches and Oracle produces 26. The MSSQL total splits into 176 *interp-only err* cases driven by the eager string-to-`INT` cast (row 10 of Table 7), 168 cases where the engine rewrites `(p) IS NULL` to `(1=0)` ahead of the cast that fails in the interpreter (row 11), and 4 *engine-only err* cases driven by `INT32` overflow. The Oracle total splits into 24 *interp-only err* cases from the same row-level `NULL`-before-cast as row 10 (elaborated under Section 6.5.4) and 2 cases where the rewriter places an invalid boolean literal in an `ON` clause and the engine rejects the query. These 4 cases are the ones the `BIGINT` mitigation (row 16) cannot fix. Declaring the integer columns as `BIGINT` removes the overflow on the comparison side, where the `CASE` wrapper's default 32-bit `INT` literals would otherwise force any string operand compared against them to be cast to 32-bit `INT` as well. It does not help when a `VARCHAR` column stores a number too large for a 32-bit integer, such as `'9223372036854775807'`: to compare that string with an integer, MSSQL converts it to a 32-bit `INT` and overflows before doing any arithmetic. Our mitigation only changes how integer values are typed, not the declared type of a text column, so MSSQL raises an error on these cases while PyTRAF succeeds.

6.5 Catalog of Mismatches

Table 7 lists every mismatch we identified across these validation experiments, grouped into six categories. Each case is tagged with the direction of the divergence: FN when PyTRAF raises an error on a query the engine accepts, and FP when PyTRAF accepts a query the engine rejects. Two cases are engine bugs rather than divergences from the formalism: EB+ marks a bug confirmed by the vendor, and EB- a behavior we reported that the vendor considers intended. The *Status* column records what we did about each case: some behaviors are now adopted in PyTRAF; some disappear when the simplified generator stops producing the construct that triggers them (written *sim: <avoided construct>*); some are resolved by the Spider preprocessing; the rest remain open.

6.5.1 Subquery dead-code elimination. Engines skip evaluating expressions inside a subquery when the outer query does not depend on them, hiding the runtime errors the inner expressions would raise if evaluated. The canonical PSQL example (row 1) is:

```
SELECT 1 FROM (SELECT CAST(Name AS INT) AS c FROM TA) AS S
```

It returns a table even though the inner cast is invalid, while `SELECT S.c FROM (...) AS S` raises. The same phenomenon occurs in two other subquery positions. Under `EXISTS` (row 2) the engine only checks whether the inner result set is non-empty, so the projection list is never materialized:

```
SELECT 1 FROM TA WHERE EXISTS
  (SELECT CAST(Name AS INT) FROM TA)
```

Table 7. Catalog of observed mismatches.
FN: PyTRAF rejects, engine accepts. FP: PyTRAF accepts, engine rejects. EB+: vendor-confirmed engine bug. EB-: reported to the vendor and classified as intended behavior.

#	Case	Engines	Tag	Status
Section 6.5.1. Subquery dead-code elimination				
1	Projection-insensitive skipping (`FROM` subquery)	all	FN	sim: no `FROM` subquery
2	`EXISTS` projection skipping	all	FN	adopted in PyTRAF
3	`IN` -subquery projection skipping	PSQL	FN	sim: column-only projection
Section 6.5.2. Non-deterministic boolean short-circuiting				
4	`AND` /`OR` reorder hides cast errors	PSQL, MSSQL, Oracle	FN	sim: no conjunctions
Section 6.5.3. Column-pruning in set operators				
5	First-column pruning in `INTERSECT`	MSSQL	FN	sim: one-column projection
6	Empty-left-operand skipping	Oracle	FN	adopted in PyTRAF
Section 6.5.4. `NULL` semantics divergences				
7	`NULL` -comparison elimination	all	FN	adopted in PyTRAF
8	Partial `NULL` arithmetic	MSSQL	FN/FP	open
9	Inconsistent `NULL` behavior, operand-order	Oracle	FN/FP	open
10	Row-level `NULL`-before-cast in comparison	MSSQL, Oracle	FN	open
11	`(p) IS NULL` $\rightarrow$`(1=0)` over cast-failing p	MSSQL	FN	open
Section 6.5.5. Engine-specific inconsistencies				
12	`UNION` `'0'` vs `-5.0+5.0`	MySQL	EB+	confirmed (#119414) [Ye 2025b]
13	`INTERSECT` operand-order dependence	MySQL	EB-	reported (#119413) [Ye 2025a]
Sections 6.5.6, 6.5.7, 6.5.8. Engine rejects, PyTRAF *accepts (FP)*				
14	Static literal-cast rewrite of `CAST('foo' AS INT)`	PSQL	FP	sim: no `FROM` subquery
15	Aggressive arithmetic rewrite in `WHERE`	Oracle	FP	open
16	`INT32` overflow on `VARCHAR` column values	MSSQL	FP	mitigated on literals via `BIGINT`
17	Runtime eager cast before `WHERE` filtering	MSSQL	FP	open

The engine returns rows because `EXISTS` short-circuits on the first row of `TA` without evaluating the cast. PyTRAF adopts the same rewrite at runtime (the `EXISTS` subquery's projection is replaced by `SELECT 1` before evaluation), so the interpreter agrees with the engine. Under `IN` (row 3) the

typechecker unifies the outer operand with the inner projection by inserting a cast on the outer side; the engine can skip both the cast and the inner subquery when the inner `WHERE` is provably empty, while PyTRAF runs the cast eagerly on every outer row. For instance:

```
SELECT 1 FROM TA WHERE Name IN
  (SELECT Age FROM TA WHERE 1=2)
```

This returns no rows on the engine: the inner `WHERE 1=2` rewrites to an empty set and `Name IN ()` is `FALSE` for every row without consulting the value of `Name`. PyTRAF types `Name` as `CAST(Name AS INT)` for the comparison and runs that cast on every row of the outer `TA` before evaluating the inner subquery, raising on the first non-numeric `Name`. The simplified generator avoids the remaining cases by construction: no `FROM` subqueries and only column-reference projections inside `IN` subqueries.

6.5.2 Non-deterministic boolean short-circuiting. Engines reorder conjuncts. In PSQL, `WHERE 5 = CAST('hello' AS FLOAT) AND 2 < 1` fails on the cast, but `WHERE 5 = CAST(Name AS FLOAT) AND 2 < 1` returns empty because the engine evaluates `2 < 1` first. Because evaluation order is not deterministic across engines, the simplified generator avoids conjunctions (row 4).

6.5.3 Column-pruning in set operators. MSSQL detects an empty `INTERSECT` by examining only the first column (row 5); we restrict simplified projections to one column. Oracle skips evaluating the right operand of an `INTERSECT` when the left operand is empty (row 6); we adopt the same rule in PyTRAF:

```
SELECT 1 FROM TA WHERE FALSE INTERSECT SELECT CAST(Name AS FLOAT) FROM TA
```

returns empty.

6.5.4 `NULL` semantics divergences. In engines, a comparison with `NULL` evaluates to `NULL`, so a `WHERE x < NULL` clause drops every row and the query returns no rows (row 7); PyTRAF behaves the same way. MSSQL accepts `realage + fullname < NULL` as empty but raises on `realage + fullname< NULL + 1`, except inside an `IN` subquery, where the inner `NULL + 1` is again rewritten (row 8). Oracle accepts `age + name FROM TA WHERE (1 + age IS NULL)` but raises on the operand-flipped `name + age FROM TA WHERE (1 + age IS NULL)` and on `age + name FROM TA WHERE (1 + NULL IS NULL)` (row 9).

The next divergence is row-level `NULL`-before-cast in comparison operators (row 10), the dominant remaining cause of MSSQL/Oracle disagreement. Consider a table `T(s VARCHAR, n INT)` with the single row `('0E', NULL)` and the query `SELECT * FROM T WHERE s <> n`. PyTRAF typechecks the predicate as `CAST(s AS INT) <> n`, evaluates the cast on `'0E'` before `<>` ever inspects the right operand, and raises *cast fails 0E to Int*. MSSQL and Oracle evaluate `<>` row by row: with `n IS NULL` the comparison returns `NULL`, the surrounding `WHERE` drops the row, and the engine never applies the implicit `CAST(s AS INT)` cast to `'0E'`. The cast rules themselves agree; the divergence comes from evaluation order. Evaluating the query `SELECT * FROM dual WHERE '0E' < 1` confirms that Oracle does raise an error when the cast is unavoidable. Internally, PyTRAF inserts a cast on each operand of the `Op` node and runs both sides before `apply` has any chance to propagate `NULL`, so the string-to-`INT` cast fires before the `NULL` short-circuit does.

Unlike rows 8–10, row 11 is a difference in the semantics of `NULL` itself, independent of operand position. In MSSQL, predicates and ordinary values are separate categories: a predicate evaluates to `TRUE`, `FALSE`, or `UNKNOWN`. `UNKNOWN` is not the value `NULL`, so a predicate can never be `NULL`. It follows that `(p) IS NULL` is `FALSE` for every predicate `p`. MSSQL exploits this when planning a query that wraps a predicate in `IS NULL`: it rewrites the whole `WHERE` to a constant before the subexpressions are evaluated. Consider `T(s VARCHAR)` with the single row `('abc')` and the query

`SELECT * FROM T WHERE (s = 5) IS NULL`. MSSQL executes it as `WHERE (1=0)` and returns no rows, while PyTRAF evaluates the inner predicate, runs the eager `CAST(s AS INT)` cast on `'abc'`, and raises before the `IS NULL` test is reached. This rewrite accounts for 168 of the 348 MSSQL mismatches in our corpus.

6.5.5 Engine-specific inconsistencies. Two MySQL behaviors fall outside the optimization categories above and do not derive from any rule in TRAF. We reported both to the vendor. The `UNION` case (row 12) was confirmed as a bug [Ye 2025b]: `SELECT '0' FROM TC UNION SELECT -5.0 + 5.0 FROM TC` returns `'0'` alone, while `SELECT '0' FROM TC UNION SELECT 0.0 FROM TC` returns both `'0'` and `'0.0'`. The `INTERSECT` operand-order case (row 13) was reviewed by the vendor and classified as not a bug [Ye 2025a], leaving the divergence in MySQL: `SELECT NULL FROM TA INTERSECT SELECT 1 FROM TA` returns empty, while the operand-swapped query returns 1.

6.5.6 Engine evaluates casts that PyTRAF *skips.* Rows 14 and 17 collect FP cases where the engine evaluates a cast on values PyTRAF never reaches. PyTRAF waits for the `WHERE` (or the inner `WHERE` of an `IN` subquery) to filter the row set first, and then runs the projection or the `IN` comparison only on surviving rows; the cast on a non-numeric value never fires. The engine performs the cast in two different phases, described next.

Static rewrite path (row 14, PSQL*).* PSQL rewrites

```
SELECT CAST(R.A AS INT) AS col0
FROM (SELECT 'foo' AS A FROM TC WHERE 1=2) R
```

by pushing the literal `'foo'` into the outer `CAST`. The resulting `CAST('foo' AS INT)` is a constant expression that PSQL evaluates during typechecking, so the query is rejected before evaluation considers the `WHERE 1=2` clause. PyTRAF does not perform this transformation, so the cast remains dynamic: the inner `WHERE 1=2` empties the subquery, and no row reaches the outer projection. The countermeasure on the generator side is to avoid `FROM`-subqueries—the same restriction considered for row 1.

Runtime plan path (row 17, MSSQL*).* MSSQL's chosen plan materializes the implicit cast on the cartesian product before the `WHERE` filter prunes the failing rows. We identify two forms of this pattern in the corpus. The first is an `IN`-subquery whose inner `WHERE` empties the subquery, while the inner projection's type forces an implicit cast against the outer operand:

```
SELECT '-4' FROM TB
WHERE TB.realage IN (SELECT TB1.fullname FROM TB TB0, TB TB1
                     WHERE TB0.fullheight = 40.0)
```

No row of `TB0` has `fullheight = 40.0`, so PyTRAF returns no rows; MSSQL raises because its plan casts `fullname` on every row of the inner cartesian product before applying the `WHERE` filter.

The second is a top-level projection whose implicit cast fires on rows the outer `WHERE` would discard:

```
SELECT TB1.realage + TA0.name FROM TA TA0, TB TB1
WHERE CAST(TA0.height AS INT) = TB1.fullheight
```

The `realage + name` sum forces an implicit `VARCHAR`⟶ numeric cast on `TA0.name`. No pair satisfies the join condition, so PyTRAF returns no rows; MSSQL raises similarly because its plan materializes the projection on the cartesian product before the `WHERE` prunes it.

Row 3 covers the symmetric FN case (engine skips, interpreter runs the cast eagerly). Row 17 cannot be avoided by restricting the generator: the cast comes from implicit type unification, not from any syntactic construct the generator could omit without losing coverage.

6.5.7 Aggressive arithmetic rewrite in `WHERE` *(*Oracle*).* Row 15 denotes a different kind of static rewriting. Oracle rejects `SELECT CAST(name AS INT) FROM TA WHERE age + age < age` (a query whose `WHERE` clause is always false), but accepts `SELECT CAST(name AS INT) FROM TA WHERE name = '1'`. The behavior suggests an arithmetic-rewrite path in Oracle: when the `WHERE` arithmetic fits a pattern the rewriter normalizes, it appears to push `CAST(name AS INT)` into a context that the rewritten query then rejects.

6.5.8 Overflow artifacts. Row 16 is an FP case that does not arise from any rule in TRAF. MSSQL uses `INT32` integer columns by default and raises on large constants generated by SQLancer++. We mitigate this by representing integers as `BIGINT` on both the interpreter and the engine side, which is enough for the constants the SQLancer++ fuzzer emits in literal positions but not when the overflowing value lives inside a string column. When a case compares a `VARCHAR` column holding a string like `'9223372036854775807'` against an integer, MSSQL still tries the implicit `INT` conversion on the column value before `BIGINT` arithmetic begins, and the engine raises while the interpreter succeeds.

6.6 Threats to Validity

Random-query bias. The fuzzer of Section 6.2 is parametrized by a fixed three-table schema; it cannot produce the schemas and queries found in real applications. Spider and Calcite (Section 6.3) and the database states and SQL fuzz queries generated by SQLancer++ (Section 6.4) address this directly.

Naive extensions for benchmark validation. The interpreter's support for aggregation, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`, and `LIKE` is implemented outside the formal model. For queries that use these constructs we claim only that the interpreter and the engines return the same results on the benchmarks; the theorems of Section 5 do not cover them. Extending TRAF to cover them is future work.

Interpreter join performance. The unoptimized nested-loop join in PyTRAF times out on the four Spider databases listed in Section 6.3. This implementation issue however does not affect the analysis of the typing and cast semantics.

Engine version drift. Optimizer behaviors and dialect rules change across releases. We pinned the engine versions used in this paper in Section 6.1. Re-running the experiments against different versions may affect the exact disagreement counts, but is unlikely to change the per-engine asymmetry. Indeed, the cataloged cases are based on the engines' published semantics rather than on release-specific bug fixes.

7 Related Work

Traditionally, SQL engines have relied either on static typing or on syntactic checking, yet type errors and typing disciplines themselves have received relatively little direct attention. Nonetheless, there are two lines of work that relate to this work. In the databases literature, corner cases such as NULLs and dynamically generated queries raise typing issues. In addition, some engines, like SQLite, have "flexible" type systems. In the programming-languages literature, type systems are a central topic, but SQL and databases have received comparatively little attention. Both areas have provided inspiration and techniques for our work.

Classical Database literature. There are many works formalizing SQL [Ceri and Gottlob 1985; Chu et al. 2017, 2016; Negri et al. 1991]. Guagliardo and Libkin [2017] developed a comprehensive

formal semantics for SQL whose core we follow here. Following a classic framework in the area, they assume that all comparisons and operations are applied to arguments of the right types. Consequently, they do not deal with typing issues directly. Regarding errors in SQL, based on previous work [Ahadi et al. 2016; Smelcer 1995; Welty 1985], Taipalus et al. [2018] review SQL errors to build a unified error categorization. In further work, Taipalus et al. [2021] 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. [2022] propose the first mechanically verified compiler (DBCert, using Coq) for SQL queries. Ricciotti and Cheney [2022] complement and deepen the work of Guagliardo and Libkin [2017] by making the notions in their semantics and proofs precise and formal in Coq. These works assume 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 [Hipp. [n. d.]], which supports 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 [Gaffney et al. 2022]. In such settings, SQL queries are often treated as strings, and little error checking is performed on dynamically generated query strings. Wassermann et al. [2007] 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 is not on the formalization of dynamic semantics or casts, and their evaluation uses the grammar of Oracle. Closer to our setting, Ye et al. [2025a] develop FPPC, a formal type system for a core fragment of GQL, the ISO standard for querying property graphs. Following gradual typing, they admit imprecise types in both the schema and the query, and prove that well-typed queries do not fail at runtime and an emptiness property: whenever their analysis flags a query, a type mismatch forces it to return no results. Unlike our work, they neither insert casts nor consider runtime errors.

Strongly-typed queries. The development of programming-language libraries and tools for type-checking queries has been extensively explored [Augustsson and Ågren 2016; Gould et al. 2004; Group 2019, 2021, 2020; Marlow et al. 2010; MIT 2019; Necco and Nuno Olivera 2005; Silva and Visser 2006]. 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 [Buneman and Ohori 1996; Ohori and Buneman 1988; Van den Bussche and Waller 2002] and SQL [Lin 2004]. Related type-inference techniques have also been studied for XQuery [Colazzo and Sartiani 2011]. In TRAF, we assume the existence of a typed schema and consider the definition of type inference 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 light-weight 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. We validated our formalization

via prototype implementations, comparing synthetic, SQLancer++-generated, Spider-derived, and Calcite-derived queries. 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, and to cover additional SQL constructs used pervasively in realistic workloads, such as aggregation, `GROUP BY`, `ORDER BY`, and `OUTER JOIN`s. Table 7 separates the remaining discrepancies by status: several optimizer- and `NULL`-related cases remain open, one MySQL behavior has been confirmed by the vendor as a bug, and another reported MySQL behavior was classified by the vendor as intended.

Acknowledgments

We thank Suyang Zhong for helping us customize SQLancer++ and generate test cases, and the anonymous reviewers for detailed feedback and suggestions to enhance this work. This work was partially funded by ANID–Millennium Science Initiative Program–Code ICN17_002, ANID FONDECYT Iniciación 11250054 and Hong Kong Research Grant Council project number 17213525.

References

Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Prior, and Raymond Lister. 2016. 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* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 401–406. doi:10.1145/2839509.2844640

Lennart Augustsson and Mårten Ågren. 2016. Experience Report: Types for a Relational Algebra Library. *SIGPLAN Not.* 51, 12 (sep 2016), 127–132. doi:10.1145/3241625.2976016

Erik Peter Bansleben. 2004. Database Migration: A Literature Review and Case Study. https://api.semanticscholar.org/CorpusID:17518212

Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD 2018)*. ACM, 221–230. doi:10.1145/3183713.3190662

Véronique Benzaken, Évelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2022. Translating Canonical SQL to Imperative Code in Coq. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 83 (apr 2022), 27 pages. doi:10.1145/3527327

HL Bhandari and R Chitrakar. 2020. Comparison of Data Migration Techniques from SQL Database to NoSQL Database. *J Comput Eng Inf Technol 9* 6 (2020), 2.

Leonard Bolc and Piotr Borowik. 1992. *Many-Valued Logics. 1: Theoretical Foundations.* Springer Berlin Heidelberg. doi:10.1007/978-b3-b662-b08494-b6

Peter Buneman and Atsushi Ohori. 1996. Polymorphism and Type Inference in Database Programming. *ACM Trans. Database Syst.* 21, 1 (mar 1996), 30–76. doi:10.1145/227604.227609

Stefano Ceri and Georg Gottlob. 1985. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering* SE-11 (1985), 324–345. https://api.semanticscholar.org/CorpusID:22717180

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:12408033

Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2016. 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

Dario Colazzo and Carlo Sartiani. 2011. Precision and Complexity of XQuery Type Inference. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming* (Odense, Denmark) *(PPDP '11)*. Association for Computing Machinery, New York, NY, USA, 89–100. doi:10.1145/2003476.2003490

Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15 (2022), 3535–3547. https://api.semanticscholar.org/CorpusID:252066674

Carl Gould, Zhendong Su, and Premkumar T. Devanbu. 2004. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 697–698. doi:10.1109/ICSE.2004.1317494

The Doobie Development Group. 2019. Typechecking Queries. https://tpolecat.github.io/doobie/docs/06-bChecking.html.
The Kysely Development Group. 2021. Kysely. https://kysely.dev/.
The PgTyped Development Group. 2020. PgTyped. https://github.com/adelsz/pgtyped.
Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proc. VLDB Endow.* 11, 1 (sep 2017), 27–39. doi:10.14778/3151113.3151116
D. Richard Hipp. [n. d.]. Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html
Shafaq Khan, Ajish Kalia, Haleh Mehdipour Dastjerdi, and Nishara Nizamuddin. 2023. Automated Tool for NoSQL to SQL Migration. In *Proceedings of the 7th International Conference on Information Systems Engineering* (, Charleston, SC, USA,) *(ICISE '22)*. Association for Computing Machinery, New York, NY, USA, 20–23. doi:10.1145/3573926.3573931
Weisheng Lin. 2004. *Type inference in SQL*. Ph. D. Dissertation. Concordia University.
Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)* (2010).
MIT. 2019. TS-SQL-Query. https://ts-bsql-bquery.readthedocs.io/.
Mysql. 2020. Converting MySQL to PostgreSQL. https://en.wikibooks.org/wiki/Converting_MySQL_to_PostgreSQL.
Claudia Mónica Necco and J Nuno Olivera. 2005. Toward generic data processing. In *XI Congreso Argentino de Ciencias de la Computación*.
M. Negri, G. Pelagatti, and L. Sbattella. 1991. Formal Semantics of SQL Queries. *ACM Trans. Database Syst.* 16, 3 (sep 1991), 513–534. doi:10.1145/111197.111212
Atsushi Ohori and Peter Buneman. 1988. Type Inference in a Database Programming Language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (Snowbird, Utah, USA) *(LFP '88)*. Association for Computing Machinery, New York, NY, USA, 174–183. doi:10.1145/62678.62700
Daejun Park, Andrei Stefanescu, and Grigore Rosu. 2015. KJS: a complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 346–356. doi:10.1145/2737924.2737991
Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: the full monty. In *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*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 217–232. doi:10.1145/2509136.2509536
PostgreSQL. 1996. PostgreSQL Documentation. https://www.postgresql.org/docs/current/typeconv-boverview.html.
RebaseData. 2023. RebaseData MySQL to PostgreSQL Online Converter. Online converter.
Wilmer Ricciotti and James Cheney. 2022. A Formalization of SQL with Nulls. *J. Autom. Reason.* 66, 4 (nov 2022), 989–1030. doi:10.1007/s10817-b022-b09632-b4
Alexandra Silva and Joost Visser. 2006. Strong Types for Relational Databases. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell* (Portland, Oregon, USA) *(Haskell '06)*. Association for Computing Machinery, New York, NY, USA, 25–36. doi:10.1145/1159842.1159846
John B. Smelcer. 1995. User Errors in Database Query Composition. *Int. J. Hum.-Comput. Stud.* 42, 4 (apr 1995), 353–381. doi:10.1006/ijhc.1995.1017
SQLines. 2010. SQLines. http://www.sqlines.com/online.
Toni Taipalus, Hilkka Grahn, and Hadi Ghanbari. 2021. Error messages in relational database management systems: A comparison of effectiveness, usefulness, and user confidence. *Journal of Systems and Software* 181 (2021), 111034. doi:10.1016/j.jss.2021.111034
Toni Taipalus, Mikko Siponen, and Tero Vartiainen. 2018. Errors and Complications in SQL Query Formulation. *ACM Trans. Comput. Educ.* 18, 3, Article 15 (aug 2018), 29 pages. doi:10.1145/3231712
Jan Van den Bussche and Emmanuel Waller. 2002. Polymorphic Type Inference for the Relational Algebra. *J. Comput. System Sci.* 64, 3 (2002), 694–718. doi:10.1006/jcss.2001.1812
Philip Wadler. 1992. Comprehending Monads. *Mathematical Structures in Computer Science* 2 (1992), 461–493.
Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. 2007. Static Checking of Dynamically Generated Queries in Database Applications. *ACM Trans. Softw. Eng. Methodol.* 16, 4 (sep 2007), 14–es. doi:10.1145/1276933.1276935
Charles Welty. 1985. Correcting User Errors in SQL. *International Journal of Man-Machine Studies* 22, 4 (1985), 463–477.
Wenjia Ye. 2025a. Bug #119413: Intersection order. MySQL Bugs Database, File ID 119413. https://bugs.mysql.com/bug.php?id=119413 Submitted 17 Nov 23:52 UTC. "SELECT NULL INTERSECT SELECT 1 returns empty but SELECT 1 INTERSECT SELECT NULL gives the result 1.".
Wenjia Ye. 2025b. Bug #119414: 0.0 is treated differently. MySQL Bugs Database, File ID 119414. https://bugs.mysql.com/bug.php?id=119414 Accessed online via https://bugs.mysql.com/bug.php?id=119414.
Wenjia Ye, Matías Toro, Tomás Díaz, Bruno C. d. S. Oliveira, Manuel Rigger, Claudio Gutierrez, and Domagoj Vrgoč. 2025a. Flexible and Expressive Typed Path Patterns for GQL. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 283 (Oct. 2025), 27 pages. doi:10.1145/3763061

Wenjia Ye, Matías Toro, Claudio Gutierrez, Bruno C. d. S. Oliveira, and Éric Tanter. 2025b. Elucidating Type Conversions in SQL Engines. In *Proceedings of the 34th European Symposium on Programming Languages and Systems (ESOP 2025) (Lecture Notes in Computer Science, Vol. 15694)*, Viktor Vafeiadis (Ed.). Springer-Verlag, 408–435.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP 2018)*. Association for Computational Linguistics, 3911–3921. https://aclanthology.org/D18-b1425/

Suyang Zhong and Manuel Rigger. 2026. Scaling Automated Database System Testing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 1677–1692. doi:10.1145/3779212.3790215