Trust at the Seams: Differential, Decentralized Compliance for Composable Query Engines

Ranjan Sinha
IBM
San Jose, USA
rsinha@us.ibm.com

ABSTRACT

Composable data systems connect independently built plan producers, optimizers, and execution engines through portable standards such as Substrait and Apache Arrow. A standard fixes the syntax of the seam between parts, not its semantics: an engine can accept a plan and silently return different rows than its neighbor. We argue that conformance testing should itself be a composable component, and present a decentralized compliance substrate for Substrait built from three reusable pieces: a versioned, AI-augmented corpus (5,041 function tests plus serialized TPC-H and TPC-DS plans); capability contracts that let a producer check, before composition, whether a downstream engine covers a plan at an adequate fidelity tier; and engine-local execution with federated reporting. On this substrate we report a cross-stack differential study of two engines (464 plan pairs, 98.1% agreement, nine divergences each traced to a named layer), an LLM-driven remediation loop that lifts translation-fidelity rates by 9.6 points on matched categories, and a dispatch study showing capability-guided routing measurably reduces the silent errors blind routing incurs.

VLDB Workshop Reference Format:
Ranjan Sinha. Trust at the Seams: Differential, Decentralized Compliance for Composable Query Engines. VLDB 2026 Workshop: Fourth International Workshop on Composable Data Management Systems.

VLDB Workshop Artifact Availability:
The source code, data, and artifacts for the Substrait Compliance Framework have been made available at https://github.com/IBM/substrait-compliance.

1 INTRODUCTION

The composable data management movement rests on a simple bet: that a database is better built from a modular stack of reusable parts than as another monolith [11, 19]. The parts are by now familiar: Arrow [1] for in-memory data, Substrait [23] for plan interchange, embeddable execution engines [8], Calcite [3] and dataframe front ends for plan production. The appeal is real: specialization without reinvention, and a path out of the siloed landscape [20]. The bet pays off only if the parts actually fit together, and a standard is supposed to make them fit.

But a standard fits parts together at the level of bytes and grammar. It says how a plan is serialized and what a valid plan looks like. It does not, on its own, make two independently built consumers compute the same answer for that plan. SQL has shown this for forty years. A shared language has never implied shared semantics, because casting rules, null handling, function definitions, and optimizer assumptions all drift between engines that nominally speak it [6, 20]. Substrait moves the contract down from query text to the plan, which removes much parsing-level ambiguity, but the harder problem rides along: an engine can accept and run a plan without error and still return the wrong rows. In a monolith these divergences never surface; composability is what exposes them.

Consider an interactive client that builds a plan with a dataframe library acting as a Substrait producer [7]; a rewrite service that pushes a predicate down and re-emits the plan; and a dispatcher that sends the rewritten plan to whichever execution engine sits closest to the data. That is three independently built components and two seams. Suppose the plan computes `avg(x)` over a group the pushdown step can leave empty: one engine returns `NULL`, another raises on empty input. Suppose it divides two `DECIMAL(38,s)` columns: engines disagree on the result scale and the rounding mode. The SQL surface never appears, so SQL conformance suites do not apply; each component parses the plan cleanly, so nothing is flagged; and the error becomes visible only when something downstream acts on a wrong number. The seam is where composability earns its value, and where the guarantees are weakest.

This is a compliance problem, and the usual instinct is to solve it the way protocol compliance is solved, by standing up a central authority that runs a canonical suite and stamps each engine conformant. For composable systems that instinct is wrong. Engine teams cannot easily ship proprietary builds, internal traces, or customer data to an external validator; a central service lags continuous engine development; a compliance failure can expose implementation details an organization would rather not disclose; and some engines only run inside infrastructure that is awkward to reproduce off-site. A companion short paper made this argument from the standpoint of AI agents that route plans across engines [17]. We take the composability standpoint and ask a different question. Not which engine an agent should route to, but whether a given composition of parts is semantically safe in the first place, and how to make the machinery that answers it a reusable part of the stack rather than a service bolted on the side.

Contributions. This paper makes five contributions, and we are explicit about which extend the earlier framework [17] and which are new.

(1) **Compliance as a composable component.** We reframe conformance testing as a module in the modular stack [11],

Proceedings of the VLDB Endowment. ISSN 2150-8097.

with a clean separation between a shared corpus, engine-local execution, and federated reporting (Section 3). The architectural skeleton is shared with [17]; the framing and interfaces here are organized around reuse across components rather than around a single agent consumer.

(2) **Composition-time capability contracts.** We define a typed capability-contract schema with per-category *fidelity tiers* and a checking procedure that, given a plan and a chain of components, decides whether the composition is safe or returns the offending seams (Section 3.2). This elevates the earlier capability declaration, which was used only to skip tests, into a static check at the boundary between parts. *(New.)*

(3) **Differential cross-engine testing.** We turn the shared corpus into a divergence oracle: run each plan on every engine that declares support and flag disagreement (Section 4). Unlike expected-output testing, this needs no golden answer and localizes blame to a seam. The earlier paper listed differential testing as future work; here it is the central method, with an initial cross-stack study. *(New.)*

(4) **An expanded, AI-augmented substrate and a study.** The corpus has grown to 5,041 tests across 14 categories, with an eight-language SDK whose C++ binding lets the runner embed natively in C++ engines.[1] We report a cross-stack differential study across Engine A and Engine E and the lessons it surfaced (Sections 3, 4).

(5) **LLM-driven translation remediation.** For engines reached through a Substrait-to-native-SQL executor rather than a native consumer, we add an iterative, LLM-in-the-loop procedure that diagnoses clustered translation failures and repairs the executor (leaving engines, fixtures, and scoring untouched) and reports its effect across five engines, where it lifts per-category translation-fidelity rates until a dialect ceiling (Section 5). *(New.)*

2 THE SEAM PROBLEM

We use *seam* for the boundary at which a plan, or a fragment of one, passes from a component that produced or transformed it to a component that consumes it. A composable pipeline is a chain of seams: a dataframe producer hands a plan to a pushdown rewriter, which hands a transformed plan to an executing engine, and the intended meaning must survive each hand-off. The standard guarantees that each component can parse what it receives; it does not guarantee that the components agree on meaning across a seam.

What goes wrong at a seam is not random. From building and running the corpus, the recurring classes are type coercion (when implicit casts fire, and the result's precision and scale); null propagation under three-valued logic; overflow and rounding, most painfully for `DECIMAL` result scale; aggregate edge cases (empty groups, all-null inputs, `count(*)` versus `count(x)`); window framing; and string semantics (collation, Unicode normalization, and the regex dialect behind functions like `regexp_count`). None is exotic, and all are invisible to a check that only asks whether a plan parses.

[1]We anonymize the five execution engines under study as Engine A through E and report only per-category behavior, to keep results from being read as an engine-by-name leaderboard.

The reason composability makes this acute, rather than merely annoying, is feedback. A human analyst who runs a query and sees implausible revenue will second-guess it against domain knowledge. A pipeline that hands a plan fragment from one service to another has no such reflex; a divergence at seam 1 propagates through seam 2 and lands in a result set that no one inspects. The point is not that engines are buggy. It is that "valid Substrait" is a syntactic property, and composition needs a semantic one.

3 COMPLIANCE AS A COMPOSABLE COMPONENT

Our response is to build the compliance machinery as another reusable part, with the same three-layer shape the modular stack favours.

Layer 1: a shared corpus. A versioned, language-neutral set of artifacts: 5,041 function-level test cases across 14 categories (Section 3.1): arithmetic with decimal, rounding, and logarithmic subfamilies; string; comparison, which includes boolean predicates; datetime; aggregate; array; map; struct; set; window; conditional; JSON; cast; and geospatial. These are joined by TPC-H (22 queries) and TPC-DS (99 queries) serialized as Substrait plans in binary and JSON. Each function test fixes input columns with explicit types, a plan exercising one function under defined edge conditions (nulls, boundaries, overflow, empty groups), and an expected result with an explicit floating-point tolerance. These are correctness fixtures, not performance benchmarks. We measure whether an engine returns the right rows, not how fast [12, 24].

Layer 2: engine-local execution. A small SDK in eight languages (Java, Python, C++, Go, TypeScript/JavaScript, C#/.NET, Scala, and Rust) exposes one contract an engine implements: declare identity and capabilities, validate a plan, and execute it returning tabular results. Everything runs inside the engine team's environment; no data leaves it. The earlier framework shipped Java, Python, and Rust [17]; the C++ binding is what lets us drive C++-native engines through their own in-process Substrait paths rather than through a foreign-language bridge, which matters because the bridge can itself introduce coercions we are trying to measure.

Layer 3: federated reporting. Results aggregate into a report (per-category pass rates, versions, declared capabilities) that an engine may publish through a REST API. The separation that matters is between private diagnostics, the failing inputs and traces that stay local, and a public summary of pass/fail counts, features, and versions. That split makes participation compatible with proprietary engines, since an organization contributes comparable evidence without disclosing why a test failed.

3.1 AI-Augmented Corpus Construction

Hand-writing edge cases is where coverage quietly rots. Authors test the function they have in mind and forget the empty group, the all-null column, the boundary value, the cross-type combination. We used a large language model as a proposer in a generate-and-gate loop. A gap analyzer reads an existing category and proposes candidate cases targeting missing null, boundary, overflow, and cross-type behavior. A generator emits them in the corpus format. An automated validator rejects any case that is malformed, that

Table 1: Fidelity tiers, assigned per category from corpus results.

Tier	Meaning
NONE	Not supported, or the engine declares it skips the category.
BASIC	Passes nominal inputs; null, boundary, or overflow cases fail.
EDGE	Passes nominal and the standard edge cases (nulls, boundaries, empty groups).
VERIFIED	Passes the full category, including overflow and cross-type cases, above threshold τ.

lacks a deterministic expected value, or whose expected value the proposer could not justify against the Substrait function definition. The model never has the last word: the validator does. Proposals that pass its checks enter automatically; one that regresses the aggregate pass rate is held for review, since a regression usually signals a malformed test rather than a new engine gap.

We are deliberately cautious about what this buys. The loop expands breadth (the arithmetic category grew substantially, with most of the growth in null and overflow handling) but it cannot manufacture ground truth where the specification is silent, and several proposed cases were discarded for that reason. We treat the model as a cheap way to enumerate the corners a human skips, not as an oracle.

3.2 Capability Contracts

The piece that makes the substrate composable, rather than just a test runner, is the contract. A component does more than pass or fail a suite. It advertises what it can be trusted to do, and at what level of fidelity, so a producer can decide before composing whether a downstream component is safe for a particular plan.

Fidelity tiers. A binary verdict is too coarse. An engine that handles nominal inputs but mishandles nulls is fine for some plans and dangerous for others. We summarize each category at one of four tiers, derived from measured pass rates over the corpus (Table 1). The tiers describe which kinds of inputs an engine is trustworthy for, rather than a single percentage. In the current implementation, tiers are assigned from corpus pass rates as a proxy for the input-class semantics of Table 1: VERIFIED at or above $\tau = 95\%$, EDGE at 80–94%, BASIC at 60–79%, and NONE below 60%; a sub-threshold rate indicates the tier's bar was not met, not necessarily absent support, and per-input-class tier assignment is future work.

The contract. For a component c we record

$$\mathrm{Cap}(c) = \big(R_c,\ F_c,\ T_c,\ \mathrm{tier}_c(\cdot) \big),$$

where R_c is the set of supported relational operators, F_c the set of supported function signatures (by Substrait anchor and argument types), T_c the supported types, and tier_c maps each category to a tier in Table 1. A plan P induces a demand $\mathrm{Dem}(P)$: the operators, function signatures, and types it uses, together with a *required tier* per category that the caller chooses (the default is EDGE, since analytical correctness usually depends on null and boundary behavior; a caller who knows its data is dense may relax it). A contract binds to a component configuration rather than a binary; an engine may publish one contract per execution profile, and Algorithm 1 treats each as a distinct candidate. Listing 1 shows a fragment.

Listing 1: A capability-contract fragment (abridged).

```
{ "engine": "engine-a", "version": "1.x",
  "substrait": "0.x",
  "relations": ["read","filter","project","aggregate","join"],
  "types": ["i32","i64","fp64","decimal","string","date"],
  "categories": {
     "arithmetic":  {"tier":"verified"},
     "comparison":  {"tier":"verified"},
     "string":      {"tier":"basic", "notes":"regex dialect differs"},
     "datetime":    {"tier":"edge"},
     "aggregate":   {"tier":"edge"} } }
```

Algorithm 1 CONTRACTCHECK

Require: plan P, chain $\langle c_1, \ldots, c_k \rangle$, required tiers $\rho(\cdot)$
Ensure: SAFE, or a set V of violations

$V \leftarrow \emptyset$
$D \leftarrow \textsc{Demand}(P)$ ▷ operators, signatures, types, per category
for each component c_i that executes or forwards part $D_i \subseteq D$ **do**
  for each $(op,\ cat) \in D_i$ **do**
    if $op \notin R_{c_i} \cup F_{c_i}$ **or** $\mathrm{tier}_{c_i}(cat) \prec \rho(cat)$ **then**
      $V \leftarrow V \cup \{(op,\ c_i,\ \mathrm{tier}_{c_i}(cat))\}$
    end if
  end for
end for
return $V = \emptyset$? SAFE : V

Checking a composition. Algorithm 1 decides whether a chain $c_1 \to \cdots \to c_k$ is safe for P. It walks the components that must faithfully execute or forward each part of the plan and reports the seams where a component's tier falls below what the plan demands, rather than a single yes/no. The output is actionable: it names the operator, the component, and the observed tier, which is the information a planner or a human integrator needs to decide whether to substitute a component, lower the required tier deliberately, or refuse the composition.

The check is intentionally cheap and conservative. It does not prove a composition correct; it certifies that every part of the plan meets a declared, measured bar, and otherwise points at the weak seam. False alarms are possible (a category can be marked BASIC because of one stubborn edge case the plan never triggers), and the caller can override per category. We prefer this failure mode to the alternative, in which a seam silently degrades a result.

4 CROSS-STACK DIFFERENTIAL TESTING

Expected-output testing answers "is this engine right?" given a curated answer. It is essential, but it costs human effort to maintain golden answers, and it cannot catch the case where the curated answer is itself wrong because the specification is ambiguous. Differential testing answers a different and, for composition, more useful question: do the engines I might compose agree? It needs no golden output, since disagreement is the signal, and it is a classic technique for this situation, where several systems implement a common language and any difference between them exposes a fault

in at least one [9, 18]. Recent database-testing work has used the same idea with great success to find logic bugs within a single engine, by comparing an engine against transformed versions of a query or a deliberately non-optimizing reference [14–16]. We apply it *across* independently built engines, through a shared portable intermediate representation, to certify the seam rather than to fuzz one engine.

4.1 Method

For each plan in the corpus we execute it on every engine whose contract declares support, then compare result sets pairwise under typed equality: same schema up to nullability, same multiset of rows, with floating-point columns compared to the test's declared tolerance and decimals compared at the stricter of the two declared scales. Each plan is then labelled AGREE (all participating engines equal), DIVERGE (at least two disagree on values), PARTIAL (an engine declines via its contract), or ERROR (an engine fails to execute a plan it claimed to support). Differential testing alone cannot say which engine is right when two disagree and the specification is silent, so we run it alongside the expected-output oracle. DIVERGE cases then split into "one engine contradicts the curated answer" (a likely bug) and "engines disagree and the corpus has no confident answer" (a likely specification gap).

4.2 Setup

The study is cross-stack: Engine E is the Substrait consumer under test, ingesting plan bytes natively over Arrow inputs in both pathways; Engine A serves as an independent SQL reference, executing SQL the harness renders from the same corpus fixtures, and consumes no plan. Both run on one machine, removing hardware as a variable. Plans reach Engine E two ways. The first generates them from SQL with the Isthmus/Calcite producer [22]: 337 of 736 attempted scalar function cases are eligible pairs. The second authors plans directly from protobuf bindings for functions Isthmus cannot compile to Substrait (bitwise operations, factorial, the `regexp` family, string-representation functions): of 233 cases, 127 are eligible; the other 106 are PARTIAL, one stack lacking support. Where Engine A lacks a function outright (`regexp_count`) the harness substitutes a `regexp_extract_all` proxy. Against the expected-output oracle the stacks pass at 64.3% (Engine A) and 62.9% (Engine E) on the Isthmus slice, yet agree on 99.1% of those pairs: where they miss the curated answer, they miss it the same way.

4.3 Results

Table 2 reports agreement over the 464 eligible pairs: 455 agree (98.1%), with comparison, boolean, datetime, and pathway-I string in full agreement. The nine divergences cluster into three root causes, and the oracle splits them the way the method intends.

Two of the three pathway-I divergences are the toolchain's doing, not the engines': the corpus types the case fp32, the Isthmus producer promotes it to an fp64 plan for Engine E, and the harness rendering independently maps fp32 to SQL `FLOAT` for Engine A, so the two stacks hand their engines different numeric types. Each engine answers its input correctly; the cross-stack comparison exposed the producer and the harness at the first seam. The third is a difference of one unit in the last place (ULP) in fp64 `atanh` on identical input, the one genuinely bilateral disagreement. The six pathway-D divergences are all Engine E `regexp_count` nonconformances against the oracle: four return 0 instead of `NULL` on null input, and two count a greedy optional pattern, `(.*)?c`, one match short where its regex engine skips the zero-length match at end of string. Engine A's proxy agrees with the oracle incidentally: these six are oracle findings, not bilateral disagreements.

Table 2: Cross-stack agreement between Engine E's native Substrait execution and Engine A's SQL execution of the same cases, over eligible pairs (both stacks executed), by plan-generation pathway: I = Isthmus/Calcite-generated; D = directly authored protobuf plans for functions Isthmus cannot compile to Substrait. The nine divergences trace to three root causes: toolchain type mismatch (I, 2), last-bit fp64 rounding (I, 1), and Engine E `regexp_count` nonconformance against the oracle (D, 6).

Category	Path	Pairs	DIV.	AGREE
Arithmetic (non-dec.)	I	126	3	97.6%
Arithmetic (bitwise, fact.)	D	36	0	100%
Comparison	I	97	0	100%
Boolean	I	16	0	100%
Datetime	I	46	0	100%
String	I	52	0	100%
String (regexp, repr.)	D	91	6	93.4%
Total		464	9	98.1%

Two lessons follow. Every divergence is attributable to a named layer: two to the toolchain, six to one consumer's `regexp_count` nonconformance (the regex-counting pair where the specification is silent on dialect), and one to fp64 rounding. And the larger hazard is not silent disagreement but coverage: the 106 PARTIAL pairs outnumber divergences by an order of magnitude. These are agreement rates over a curated corpus on one environment, a lower bound on divergence rather than its true frequency [18].

5 TRANSLATION REMEDIATION WITH AN LLM LOOP

The differential study (Section 4) had one engine consume Substrait natively; most engines in the wild have no consumer at all. For them the executor compiles each plan to the engine's native SQL dialect and submits that text, inserting a translation layer between the portable plan and the engine. A failure then has two causes: the engine genuinely disagrees with the plan's semantics, or the executor mistranslated it. The second is a translation artifact, neither an engine fault nor a specification gap, and counting it as either pollutes every signal the substrate produces, including the differential oracle. This section adds a procedure that drives translation artifacts down, so that what remains is closer to true semantic divergence.

Why not an off-the-shelf converter? A fair objection is that this translation should already exist. The ecosystem's main SQL/Substrait bridge, Isthmus, runs in the producer direction (SQL to a Substrait plan via Calcite), the inverse of what a consumer needs [22]. A reverse path exists but emits SQL for whole relational plans and

Table 3: Per-category function-test pass rate (%) at the final remediation iteration, for the five engines over a common corpus; the per-engine test count is in parentheses and Avg is the across-engine mean. These are translation-fidelity rates, the translation layer composed with each engine's SQL semantics, not a measure of engine quality.

	Post-remediation pass rate (%)					
Category (tests/engine)	A	B	C	D	E	Avg
aggregate (250)	61.2	64.0	53.6	42.0	39.6	52.1
arithmetic (1,497)	64.7	70.4	82.2	71.2	82.5	74.2
cast (78)	66.6	96.1	100	94.8	92.3	90.0
comparison (780)	46.1	95.3	91.1	89.6	89.3	82.3
conditional (93)	100	100	100	81.7	100	96.3
datetime (415)	57.3	89.1	52.0	44.5	37.1	56.0
string (768)	56.3	51.6	64.8	61.5	62.1	59.3
window (233)	91.4	88.8	90.9	89.2	71.6	86.4

inherits Calcite's under-specifications, decimal among them. Our corpus is single typed function fixtures whose metadata lives outside the plan, and each engine needs dialect-specific handling a generic converter does not supply. The step the loop remediates is thus irreducible, and its contribution is making that step faithful, per engine and per category.

Scope and loop. The loop operates on the executor's translation layer and the per-category test harness; the engines and the corpus fixtures are unchanged. It runs per category as a six-step cycle, three steps LLM-driven (Claude Opus 4.8, temperature 0.1) and the rest mechanical or human-gated: *run* the full suite so cross-category regressions surface; *analyze* (LLM) by clustering failures on error signature, since one root cause explains dozens at once; *propose* (LLM) a generic, cluster-scoped edit rather than a per-fixture hack; *implement* a new executor version; *test and compare* against the prior version through a regression gate; and *document* (LLM) the fix and residual gaps for the next round.

Knowing when to stop. The stopping rule is explicit: when a cluster stops yielding, that is a real limit, not a cue for more effort. A regression gate reverts edits that lose ground, and categories converged in one to several rounds. Some plateaued against a ceiling no edit could clear: the dialect-bound categories (string, datetime) settle short of the standardized ones across engines, the same limit the differential study predicts.

Setup. We ran the full corpus on five engines (Engine A through E) and applied the remediation loop to all five over one common corpus; Engine E has a native consumer (Section 4) but is driven through the translation path here like the rest, so all five are measured on one comparable axis. The test set is the AI-augmented function corpus of Section 3. Each execution is scored PASSED, FAILED (ran but wrong rows), or ERROR (never executed), which route to different fixes. The same fixtures and scoring apply to every engine, so cross-engine differences are signal, not corpus variation.

Results. Table 3 reports per-category pass rates for the five engines at the final iteration over one common corpus. Conditional, cast, and window sit near the top, arithmetic and comparison in the middle, and aggregate, datetime, and string lower. The signal we read is the cross-engine spread within a category, not the absolute

Table 4: Pre- to post-remediation lift for the four categories with matched runs under identical harness, fixtures, and scoring (Baseline, Remediated). Values are across-engine mean pass rates (%).

Category	Baseline	Remediated	Δ
arithmetic	69.1	74.2	+5.1
comparison	67.0	82.3	+15.3
datetime	42.9	56.0	+13.1
string	54.4	59.3	+4.9

height. It is widest on datetime, which runs from 37.1% to 89.1% as interval and timezone handling diverge, and on comparison, where Engine A sits at 46.1% against 89–95% for the rest. That single-engine gap isolates Engine A's SQL semantics rather than the translator's competence, since the same framework reaches the high end elsewhere. Arithmetic spans 64.7% to 82.5%, and string holds a narrow 51–65% band. The wide-spread categories are the seams a composition checker must refuse to cross blindly.

Lift. For the four categories with matched pre- and post-remediation runs under one harness, fixtures, and scoring rule (Table 4), the loop adds 9.6 points on average (58.4% to 68.0%). The gains concentrate where an engine started low: Engine D climbs 44 points on comparison and 20 on string, engines already near the top move far less, and string slips a point or two on Engines A, B, and C. The baseline layer was authored by Claude Sonnet 4.5 and remediated by a Claude Opus 4.8 loop, with the model held fixed from the first remediated iteration onward. Because the two runs share harness, fixtures, and scoring, the lift is not a relaxed bar.

6 DISCUSSION

What the contract buys. The capability contract is the part we would most like other composable stacks to borrow. It converts compliance evidence from a leaderboard into a precondition. A producer or planner can ask, before execution, whether a downstream chain meets the bar a plan needs, and get back the specific weak seam if not. Unknown coverage is treated as unsafe rather than guessed, so the check degrades gracefully.

The contract in action. To test whether the contract does more than describe, we simulated capability-guided dispatch over the four categories that execute on every engine (3,460 tests), routing each category to its best-tier engine, which reaches 82.3% accuracy against 70.6% for random selection: a gain of 11.7 points, or about 406 fewer silent errors, rising to 27.9 points (~967 fewer) against worst-case routing. Restricting the study to the fully-executing categories keeps the gain from being inflated by categories an engine cannot yet run.

Specification gaps versus bugs. Differential testing keeps surfacing disagreements that are nobody's bug, places where Substrait, like SQL before it [6, 20], has not committed to one interpretation. Some cross-engine disagreements trace to the specification leaving a behavior undefined rather than to any engine being wrong, and differential testing surfaces them as a by-product, with the failing cases as the evidence to file upstream (the regex-dialect cases in Section 4). Some are invisible while only two engines are compared

and appear once strict-typed engines join, which motivates a wider oracle. A substrate that can hand the specification an evidence-backed list of these ambiguities is healthier than one that only grades engines against it.

Translation artifacts versus semantics. Section 5 draws the practical line the rest of the paper argues for in principle. Once the executor is repaired, the categories that stay low do so for a reason no translation edit can fix. The string floor, datetime, and the wide arithmetic spread reflect dialect and type semantics the engines disagree on, not SQL that was mistranslated. The loop thus acts as a filter, separating fixable translation noise from the semantic divergence a composition checker must respect, and giving a cheap signal for which seams warrant a specification fix rather than more engineering.

Limitations. Several caveats bound the claims. Contracts are self-advertised, and we have no attestation yet that a declared tier was honestly measured (cryptographic attestation is on the roadmap, with its own key-management cost). Environment variation (compiler, OS, host regex and ICU versions) can confound cross-engine comparison; pinning one machine controls the variable at the cost of realism, since a real composition spans machines. Coverage is bounded by a curated corpus, so the agreement rates under-approximate divergence; generating differential cases automatically (Section 8) is the right long-term answer. Finally, the ambiguity-versus-bug triage rests on manual root-cause judgement. And the Section 5 pass rates measure our translation layer composed with each engine's SQL semantics, not native Substrait execution.

7 RELATED WORK

Composable systems. The modular-stack argument and its open-source instantiation motivate this work [10, 11], as does the broader "one size does not fit all" lineage [19, 20]. Some designs embed a single shared execution library across systems; others are embeddable engines that independently consume Substrait [8, 13]; Photon [4] and Calcite [3] round out the components. Lakehouse architectures [2] created the multi-engine topology that makes plan-level interoperability a practical, not hypothetical, concern, and learned cost models now route queries across its engines for performance, predicting per-engine execution time over Substrait plans [21]; the capability contract routes for correctness, a complementary constraint on the same engine-selection decision. Our contribution is orthogonal to all of these. It is the compliance substrate they need in order to be composed safely.

Standards and interoperability. Substrait [23] and Arrow [1] standardize plans and data; the recurring lesson, from SQL onward, is that a shared surface does not imply shared semantics [6, 20]. We add the methodology that makes the standard's semantic edges measurable.

Database and differential testing. Differential testing is an old and effective idea for systems that share a language [9], and RAGS [18] applied massive stochastic comparison to SQL engines decades ago. The modern SQLancer line of pivoted query synthesis, non-optimizing reference engines, and ternary logic partitioning [14–16] finds logic bugs largely within an engine by comparing it against a transformed self or a reference. We reuse the oracle-free comparison but aim it across independently built engines through a portable intermediate representation, to certify a seam rather than to fuzz one engine; the synthesis machinery of that line is the natural way to generate our differential cases automatically. Conformance programs in other ecosystems [5] show federated self-certification at scale, which we extend from API surface to query-plan semantics.

8 FUTURE DIRECTIONS AND CONCLUSION

The next step is to search for differential cases rather than curate them. The loop of Section 3.1 fills anticipated coverage gaps and requires a justified expected answer per case; a query synthesizer [15, 16] coupled to a Substrait producer would compose plans no checklist anticipates and needs no expected answer at all, since disagreement is the oracle, turning the agreement rates of Table 2 into lower bounds. Widening the oracle to more consumers on shared plans would turn two-stack agreement into majority-vote consensus that names the outlier rather than only flagging it, provided translation artifacts are tiered apart from true divergence. Attestation would address self-reporting bias, and a fidelity-aware planner could treat the contract as a least-cost tier constraint, pairing measured fidelity with the learned per-engine cost estimates cross-engine optimizers already supply [21]. The compliance framework will be released at https://github.com/IBM/substrait-compliance.

The argument is short. Composability bets that independently built parts will fit together, and a standard makes them fit syntactically but not semantically, and the fit that matters lives at the seams, where it must be measured rather than assumed. The measurement apparatus should itself be a composable part: a corpus, an engine-local runner, federated reporting, and a capability contract that turns evidence into a composition-time precondition, with differential cross-engine testing as the oracle that exposes seam divergence without a golden answer. The evidence is consistent. Mature consumers agree at the plan level except where the specification is silent, the wide spreads live on the translation path, and a real share of both gaps is the specification's to close.

ACKNOWLEDGMENTS

We thank the CDMS workshop organizers for the opportunity to discuss these ideas. We are grateful to colleagues at IBM Research for feedback on the compliance framework design, and to the Substrait and Apache Arrow communities for ongoing collaboration on interoperability standards. We also thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] Apache Arrow. 2024. Apache Arrow: A Cross-Language Development Platform for In-Memory Analytics. https://arrow.apache.org. Accessed 2026-06-01.

[2] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org.

[3] 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)*. ACM, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[4] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa

Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, USA, 2326–2339. https://doi.org/10.1145/3514221.3526054

[5] Cloud Native Computing Foundation. 2024. Certified Kubernetes Software Conformance. https://www.cncf.io/training/certification/software-conformance/. Accessed 2026-06-01.

[6] Alon Halevy. 2005. Why Your Data Won't Mix. *ACM Queue* 3, 8 (2005), 50–58. https://doi.org/10.1145/1103822.1103836

[7] Ibis Project. 2024. Ibis: The Portable Python Dataframe Library. https://ibis-project.org. Accessed 2026-06-01.

[8] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion)*. ACM, New York, NY, USA, 5–17. https://doi.org/10.1145/3626246.3653368

[9] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[10] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a Query Engine From Spare Parts. In *1st International Workshop on Composable Data Management Systems (CDMS@VLDB)*.

[11] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R. Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (2023), 2679–2685. https://doi.org/10.14778/3603581.3603604

[12] Meikel Poess and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Record* 29, 4 (2000), 64–71. https://doi.org/10.1145/369275.369291

[13] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[14] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 1140–1152. https://doi.org/10.1145/3368089.3409710

[15] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (2020), 30 pages. https://doi.org/10.1145/3428279

[16] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA, USA, 667–682.

[17] Ranjan Sinha. 2026. Parsing Is Not Executing: Decentralized Compliance for Agentic Query Plan Routing. In *Proceedings of the Supporting Our AI Overlords Workshop (SAO)*.

[18] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, San Francisco, CA, USA, 618–622.

[19] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Washington, DC, USA, 2–11. https://doi.org/10.1109/ICDE.2005.1

[20] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *SIGMOD Record* 53, 2 (2024), 21–37. https://doi.org/10.1145/3685980.3685984

[21] András Strausz, Niels Pardon, and Ioana Giurgiu. 2025. A Learned Cost Model-based Cross-engine Optimizer for SQL Workloads. In *VLDB 2025 Workshop: Third International Workshop on Composable Data Management Systems (CDMS)*. arXiv:2506.02802.

[22] Substrait Community. 2024. Substrait Java: Isthmus (SQL-to-Substrait) and the Calcite-based Substrait-to-SQL path. https://github.com/substrait-io/substrait-java. Accessed 2026-06-01.

[23] Substrait Project. 2024. Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io. Accessed 2026-06-01.

[24] Transaction Processing Performance Council. 2024. TPC Benchmark H Standard Specification. http://www.tpc.org/tpch/. Accessed 2026-06-01.