Reifying the Error: Metamorphic and Differential Logic-Bug Testing for SPARQL Engines

Jesse Wright · the sparq project

DRAFT — in progress. *This is a first draft written against the merged testing instrument only. The cross-engine bug-hunting campaign has* not *run: no third-party bug is claimed anywhere in this document, every campaign-dependent table is an explicitly marked placeholder, and §9 states the honest publishability condition this draft does not yet meet. The evaluation methodology (§7.2) is fixed* before *the campaign so results cannot steer the method.*

Abstract

Logic bugs — a query silently returning a wrong result — are the worst class of database engine defect: nothing crashes, and the user has no oracle. Metamorphic testing closed this oracle gap for SQL (SQLancer's TLP and NoREC found 77 logic bugs and 51 optimization bugs respectively [1], [2]), and the approach has since been re-derived for Datalog [3], [4] and for property-graph query languages [5], [6], [7], [8]. SPARQL — a W3C-standardised query language with multiple independent production engines — has no dedicated logic-bug testing work at all; the closest published evidence is a translation-pipeline study that reported wrong results in one engine *incidentally* [9]. We show the gap is not an accident of neglect but of semantics: SQL's Ternary Logic Partitioning splits on a third truth *value* (`NULL`, testable in-language with `IS NULL`), whereas SPARQL's third truth state is a third evaluation *outcome* — a type error — that is not a value, cannot be bound, and has no `IS ERROR` test. We re-derive TLP for SPARQL by reifying the error outcome with `COALESCE`, the language's only error-to-value form, wrapped around an EBV-normalising `IF`: the partition `FILTER(c) ⊎ FILTER(!c) ⊎ FILTER(COALESCE(IF(c, false, false), true))` provably recomposes the unpartitioned query under the SPARQL 1.1 specification's own clauses (§17.2, §17.3, §17.4.1.2–3, §18.5), including the type-error, unbound-variable-under-`OPTIONAL`, and error-absorbing-connective cases. We adapt NoREC by moving the predicate into projection position — where an error observably yields an *unbound* flag rather than a dropped row — and state honestly that its "non-optimizable" premise is a structural heuristic, not a theorem. We implement both oracles plus a cross-engine differential oracle in an open-source harness with a seeded deterministic generator, fail-closed verdicts that never conflate wrong results with engine failures, and a found-bug ledger whose schema makes an entry impossible without an upstream issue link and a developer-confirmation status. Instrument validation: a deliberately injected wrong-result mutant is flagged by all three oracles against a real engine, and the metamorphic relations hold on every generated case in the pinned self-test suite. The cross-engine campaign is the next step, with its methodology fixed in this draft before execution; this draft contains no campaign result and claims no found bug.

1. Introduction

A database engine that crashes is annoying; an engine that silently returns the wrong answer is dangerous. This class — *logic bugs* — evades both crash-oriented fuzzing and ordinary regression testing, because the test author is exactly as likely to mispredict the correct result as the engine is to miscompute it. The breakthrough for relational systems was to replace the missing oracle with *metamorphic relations* [10]: derive from one query a set of related queries whose results must recompose the original, and flag any engine where they do not. SQLancer's Pivoted Query Synthesis, NoREC, and Ternary Logic Partitioning (TLP) found, per their papers, 121 total, 51 optimization-class, and 77 logic-class previously unknown bugs in mature production DBMSs [1], [2], [11] — distinct counting bases the field separates (§6.3): PQS reports total unique bugs, while NoREC and TLP report the bug class their oracle targets. The recipe has since been transplanted to Datalog engines — queryFuzz (13 bugs [3]) and DLSmith (16 bugs [4]) — and to graph database systems: randomized differential testing of

Gremlin engines [5], GDsmith for Cypher (28 reported bugs [6]), graph-aware metamorphic relations in Gamera (39 reported logic bugs, 15 developer-confirmed [7]), and — the closest structural precedent to this paper — a port of TLP itself to Cypher engines [8].

SPARQL is conspicuously absent from this list. It is a W3C-standardised language [12] with a formal algebra [13] and at least seven independently implemented production engines (Apache Jena, OpenLink Virtuoso, Blazegraph, GraphDB, QLever, Oxigraph, MillenniumDB), yet — to our knowledge, and per the systematic venue search behind this project's paper-selection record — **no dedicated metamorphic or differential logic-bug testing work targets SPARQL engines**. The closest published evidence is indirect: SparqLog [9], while benchmarking a SPARQL-to-Datalog translation pipeline, found that Virtuoso deviated from the standard semantics on a portion of its benchmark queries (14 of the 77 examined in that study) — wrong-result evidence reported *incidentally*, without a dedicated oracle, a generator, or an upstream bug-filing methodology.

Why the gap? The tempting answer — "TLP transfers trivially, someone just has to run it" — is wrong, and the reason it is wrong is the technical core of this paper. TLP for SQL partitions on a predicate `p` into `WHERE p`, `WHERE NOT p`, and `WHERE p IS NULL`: three branches for SQL's three-valued logic, where the third truth state is a *value* (`NULL`) that flows through expressions and is testable in-language. The Cypher port [8] kept this shape, because Cypher also models the third state as a `NULL` value. SPARQL is different in kind: its effective-boolean-value (EBV) semantics has the trichotomy **true / false / error**, and the third state is an evaluation *outcome*, not a value ([12] §17.2.2). An error cannot be bound to a variable, does not flow as data, and has no `IS ERROR` predicate; `FILTER` eliminates the false *and* error rows indistinguishably (§17.2). A naive third branch therefore cannot be written at all — the error partition must be *reconstructed* from the language's own error-absorbing forms. That reconstruction, with a spec-clause-by-spec-clause case analysis precise enough for a reviewer to check against the recommendation text, is Contribution 1.

This paper contributes:

1. **TLP re-derived for SPARQL's three-valued error semantics** (§3). The partition `FILTER(c)` / `FILTER(!c)` / `FILTER(COALESCE(IF(c, false, false), true))` — the third branch reifies the error outcome as a total boolean — with a case analysis grounded in the SPARQL 1.1 recommendation (§17.2, §17.2.2, §17.3, §17.4.1.1–3, §17.4.1.7, §18.2.1, §18.5), covering type errors from bound values, unbound variables under `OPTIONAL`, and the non-strict error-absorbing connectives. The relation needs only a single engine's internal consistency — no reference implementation and no cross-engine agreement.
2. **NoREC adapted to SPARQL** (§4): the predicate moved to projection position, where the trichotomy stays observable (`true` / `false` / *unbound*), with the cardinality cross-check done harness-side — and an explicit honesty caveat that "non-optimizable" is an architectural heuristic about how engines are built, not a semantic theorem.
3. **An open-source instrument** (§6): the two metamorphic oracles plus a cross-engine differential oracle over an engine-independent comparator; a seeded, wall-clock-free deterministic case generator whose grammar provably excludes every construct that voids the relations (including `EXISTS`, excluded because its substitution semantics is a known specification-level defect, not an engine property); SPARQL-protocol drivers; fail-closed verdicts that never conflate a wrong result with an engine failure; and a found-bug ledger whose schema structurally requires an upstream issue URL and a developer-confirmation status per entry.
4. **A cross-engine campaign** (§7): methodology fixed in this draft *before* execution; results pending. **No bug count is claimed in this draft.**

2. Background: SPARQL evaluation and the three-valued EBV

We assume the SPARQL 1.1 recommendation [12] and its algebra (formalised in [13]). Evaluating a group graph pattern `P` over an RDF dataset yields a multiset of *solution mappings* μ: partial functions from variables to RDF terms. Partiality is load-bearing — `OPTIONAL` produces solutions in which some variables are *unbound* — and `SELECT *` projects all in-scope variables, leaving unbound ones absent from the row.

The EBV trichotomy. For a filter expression `c` and a solution μ, write $\mathrm{ebv}(c, \mu)$ for the effective boolean value of `c` under μ. Section 17.2.2 of the recommendation defines EBV coercion for `xsd:boolean`, numeric, and string operands, and rules that "all other arguments, including unbound arguments … produce a type error". So $\mathrm{ebv}(c, \mu)$ has exactly one of three outcomes — **true**, **false**, or **error** — and the trichotomy is total and exclusive provided `c` is deterministic and a function of μ alone (a precondition we return to in §3.4).

`FILTER` keeps only true. Filter evaluation (§17.2) eliminates the solutions for which the expression "either result[s] in an effective boolean value of false or produce[s] an error"; the algebra operator `Filter` (§18.5) keeps exactly the solutions where the expression's EBV is true. Two of the three outcomes — false and error — are therefore dropped *indistinguishably*. This is the crux: SQL can ask `p IS NULL`; SPARQL cannot ask "did `c` error", because the error is not a value.

Where errors come from. Three SPARQL-specific sources matter for oracle design:

- *Type errors from bound values*: comparing incomparable operands (`"abc" < 5` has no operator mapping, §17.3), arithmetic on non-numerics, `RDFterm-equal` between literals of incomparable types ("produces a type error", §17.4.1.7).
- *Unbound variables*: evaluating a variable unbound in μ — typically introduced by `OPTIONAL` — is a type error in every operator and function *except* the unbound-tolerant forms: `BOUND` returns false (§17.4.1.1), `COALESCE` skips erroring arguments (§17.4.1.3), and `IF`/`||`/`&&` absorb rather than evaluate.
- *Error absorption by the connectives*: `&&` and `||` are non-strict in errors (§17.2, "Truth Table for && and ||"): `false && error = false` and `true || error = true`, but `true && error = error` and `false || error = error`. A sub-expression error can be absorbed into a top-level true or false — and short-circuit implementations that are accidentally strict are precisely the historical divergence surface.

3. TLP re-derived for SPARQL

3.1. The partition

For a group graph pattern `P` and a deterministic filter expression `c`, build four queries: the unpartitioned base and three branches, each appending one filter at the top level of the group.

branch	filter appended to `P`	keeps μ iff $\mathrm{ebv}(c, \mu)$ is
base	—	(all of `eval(P)`)
true	`FILTER( c )`	**true**
false	`FILTER( !( c ) )`	**false**
error	`FILTER( COALESCE( IF( c, false, false), true) )`	**error**

Table 1: The SPARQL TLP partition. The error branch is the re-derivation: SPARQL has no `IS ERROR`, so the error outcome is *reified* into a total boolean by wrapping an EBV-normalising `IF` in `COALESCE`, the language's only error-to-value form (`&&`/`||` also absorb errors, but only when the other operand already fixes the result, so they cannot isolate the error case).

Metamorphic relation (partition recomposition). Writing $\uplus$ for multiset union:

$$\text{eval}(\text{base}) \equiv \text{eval}(\text{true branch}) \uplus \text{eval}(\text{false branch}) \uplus \text{eval}(\text{error branch})$$

A reviewer can check each branch against the recommendation text directly:

- **True branch.** `FILTER(c)` keeps exactly the solutions with $\text{ebv}(c, \mu) = \text{true}$, by the filter-evaluation rule quoted in §2 (§17.2; algebra `Filter`, §18.5). No rewrite machinery is involved.
- **False branch.** `!` is XPath `fn:not` applied to the operand's EBV (operator mapping, §17.3): `!true = false`, `!false = true`, and an erroring operand *propagates the error* — `fn:not` never converts an error into a boolean. Hence `FILTER(!(c))` keeps exactly $\text{ebv}(c, \mu) = \text{false}$; the error rows stay eliminated and are *not* folded into this branch.
- **Error branch.** `IF(c, false, false)` evaluates `c`, takes its EBV, and returns `false` on both true and false; "if evaluating the first argument raises an error, then an error is raised for the evaluation of the IF expression" (§17.4.1.2). Wrapping in `COALESCE(x, true)` — which returns the RDF-term value of the first expression that evaluates without error (§17.4.1.3) — yields `false` whenever `c` evaluated either way and `true` exactly when `c` errored. The composite is a *total* boolean expression that is true precisely on the error rows: the error outcome, reified as a value.

Exhaustiveness and exclusivity follow from the EBV trichotomy being total and exclusive (§2), and the recomposition is *multiset* equality because `Filter` neither alters a solution mapping nor changes the in-scope variables — a `FILTER` contributes no bindings (§18.2.1), so `SELECT *` projects identically in all four queries.

3.2. Why the SPARQL-specific cases hold

The three error sources of §2 are exactly where a naive port would break, so we treat them explicitly:

- **Unbound under `OPTIONAL` (exhaustiveness).** Take `P` with an `OPTIONAL` clause binding `?w` on only some solutions, and `c = ?w >= 2`. On a solution where `?w` is unbound, `c` errors (§17.2.2), `!(c)` errors (§17.3), and `IF(c, false, false)` errors ⟶ `COALESCE` returns `true` — the solution lands in the error branch, once. By contrast `BOUND(?w)` *evaluates* (to false) on the same solution and lands in the false branch, and `COALESCE(?w, 0) >= 2` lands in true or false. The partition is on the evaluation *outcome*, not the syntactic cause, and stays exhaustive and exclusive in every such case.
- **Type errors from bound values.** `"twenty" < 25` has no operator mapping (§17.3) and errors; the same chain routes it to the error branch.
- **Connective absorption.** For `c = (?v < 5 || ?w < 5)` with `?v = 3` bound and `?w` unbound: the left disjunct is true, so `c` is true by the non-strict truth table (§17.2) — the solution belongs to the true branch, and the reification agrees, because `IF` sees EBV true. Only the *top-level* outcome decides the branch; errors absorbed below the top level never surface. Engines with accidentally strict connective implementations diverge exactly here, and the partition tests this surface for free.

Two properties make this oracle unusually deployable. First, it requires **no cross-engine agreement and no reference implementation**: only one engine's internal consistency across four queries. Implementation-defined extensions (e.g. extra comparable datatypes, §17.3.1) do not break the relation so long as $\text{ebv}(c, \mu)$ is deterministic within the engine. Second, the base query is unconstrained in its pattern *shape*: `P` may contain `OPTIONAL`, `UNION`, and subqueries — the preconditions below constrain the partition level and additionally require the whole base query (`P` as well as `c`) to be *deterministic* (precondition 1).

3.3. Attribution caveat (what a violation does and does not localise)

A TLP violation is evidence of a wrong result **somewhere in** the engine's evaluation of {`P`, `FILTER`, EBV, `!`, `IF`, `COALESCE`, the operators of `c`} — the rewrite machinery is part of the tested surface. It is a genuine logic bug (some query among the four returned a wrong result set), but the violation alone

does not localise *which* construct is wrong; campaign triage reduces the failing case before it enters the found-bug ledger.

3.4. Scope preconditions (violating any voids the relation)

The generator (§6.1) enforces all four by construction:

1. **Deterministic base query — both `P` and `c`**: no `RAND()`, `NOW()`, `UUID()`, `STRUUID()`, `BNODE(…)` anywhere in the base query, and no `eval(P)`-destabilising construct inside `P` itself (a subquery with `LIMIT` and no `ORDER BY`, or `REDUCED` in a subquery, can return a different multiset on each evaluation). The relation compares four separate query executions, so `eval(P)` must be a fixed multiset and `c` a function of μ alone; a nondeterministic base can land the same solution in different branches across the four evaluations — a false positive independent of any engine bug. The generator's fixed pattern (§6.1) contains no subquery, `LIMIT`, or nondeterministic function, and so satisfies this by construction.
2. **No `EXISTS` / `NOT EXISTS` in `c`**: the substitution semantics of `EXISTS` is a known specification-level defect of SPARQL 1.1 — acknowledged in the errata and under revision for SPARQL 1.2 [14] — so a "violation" involving `EXISTS` would measure spec ambiguity, not an engine logic bug. Excluding it is an honesty decision: the oracle must not manufacture bugs out of a defect in the standard itself.
3. **Plain `SELECT *` at the partition level**: no `DISTINCT`/`REDUCED` (breaks multiset additivity), no `ORDER BY`/`LIMIT`/`OFFSET` (partitioning does not commute with slicing), no aggregates. The pattern `P` underneath is unrestricted in shape (but must be deterministic — precondition 1).
4. **Top-level filter placement**: a constraint applies to the whole group regardless of position (§17.2), so the branch filter is appended at the top level of the group and never pushed into `OPTIONAL`/ subquery scope, where filter scoping rules differ.

4. NoREC adapted to SPARQL

NoREC for SQL [2] rewrites `SELECT * FROM t WHERE p` so that `p` is evaluated per row in a position the optimizer does not treat as a filter, then cross-checks cardinalities: a mismatch means the optimized and non-optimized evaluation paths of the *same engine* disagree. Our SPARQL rewrite moves the predicate into projection position:

- **optimized**: `SELECT * WHERE { P FILTER( c ) }` — the form every engine's optimizer targets (filter push-down, index-driven evaluation, join reordering around the selective predicate);
- **rewrite**: `SELECT ( IF( c, true, false) AS ?flag ) WHERE { P }` — the engine must materialise `eval(P)` in full and evaluate `c` once per solution to bind `?flag`.

The relation, under the recommendation's semantics: `FILTER(c)` keeps exactly the $\mathrm{ebv}(c, \mu) = \mathrm{true}$ solutions (§17.2); `IF(c, true, false)` returns `true` on exactly the same EBV (§17.4.1.2); and when `c` *errors*, the projection-expression semantics differs observably from the filter semantics in precisely the way that makes the rewrite faithful — an error in a `SELECT` expression leaves the variable *unbound while keeping the row* (algebra `Extend`, §18.5: when the expression evaluates to an error, `Extend` returns the solution mapping unchanged, so the variable stays unbound; a `SELECT` expression is `Extend` + `Project`, §18.2.4.4). The rewrite therefore preserves the full trichotomy observably as `true` / `false` / unbound, and since `Project` without `DISTINCT` preserves bag cardinality, the rewrite returns exactly $|\mathrm{eval}(P)|$ rows, of which the boolean-true-flagged ones must number exactly $|\mathrm{eval}(\mathrm{optimized})|$ — which presupposes, as in §3.4 precondition 1, a *deterministic* base query: the optimized and rewrite forms are two separate executions of `P`, so a nondeterministic `P` (e.g. a `LIMIT` subquery without `ORDER BY`) would break the cardinality identity independently of any engine bug. The harness counts the true-flagged rows *on the client side* (accepting both valid `xsd:boolean` lexical forms, `true` and `1`) rather than wrapping the rewrite in a `COUNT` aggregate — a nested aggregate would hand the expression straight back to the optimization machinery the oracle is trying to bypass.

Honesty note: "non-optimizable" is a structural heuristic, not a theorem. A semantics-preserving optimizer *may* legally evaluate `c` early even in projection position. The claim — the same one the original NoREC paper makes for SQL — is architectural: in surveyed engine implementations, filter position feeds the plan-level optimization paths while projection expressions are evaluated by a generic per-row expression interpreter after pattern matching, so the two forms exercise different code paths, and a divergence exposes a bug in one of them. The oracle detects the divergence; it does not prove which path is wrong, and it would lose (not corrupt) its bug-finding power on an engine that normalises both forms into one plan.

5. The differential oracle

The third oracle runs the same query and data on two or more engines and compares results. Differential testing [15] needs an agreement standard, and ours is deliberately **not any engine's own value code**: results are compared through an engine-independent comparator library (the project's `sparq-difftest` crate, which by a documented dependency constraint shares no code with the engine under test), using multiset equality over value-canonical solution keys for `SELECT` — order-insensitive, duplicate-preserving — and boolean equality for `ASK`. A shape mismatch (one engine answers solutions, another a boolean) is classified as a driver/protocol failure to triage, never as a wrong-result claim.

Two scope limits are documented and generator-enforced. *Blank nodes*: SPARQL does not require cross-engine agreement on blank-node labels, so label-level comparison is meaningful only within one engine; the generator emits no blank nodes, and cross-engine blank-node isomorphism is left to future work. *Implementation-defined latitude*: engines may legitimately differ where the recommendation leaves room (extended-datatype comparisons, some canonical lexical choices — value-canonical keying absorbs the known lexical variance). A divergence is therefore a *candidate* bug; campaign triage attributes it to an engine, or to spec latitude, before it can enter the ledger.

The three oracles are complementary: TLP and NoREC need one engine and catch internal-consistency violations including those shared by every engine's filter path; the differential oracle needs two engines and catches consistent-but-wrong evaluation that no single-engine relation can see.

6. The instrument

The harness is an open-source Rust crate (`sparq-metamorph`, merged; opt-in, outside every shipping dependency graph) with the three oracles, a generator, drivers, and the ledger. Design decisions with methodological weight:

6.1. Seeded deterministic generation

Cases are generated by a self-contained SplitMix64 PRNG [16] — no wall clock, OS randomness, or external RNG dependency — so a ledger entry's seed reproduces its exact test case bit-for-bit on any machine, for the generator version that produced it (the PRNG is a fixed published mixing function, pinned by a unit test against its reference output, so the reproducibility contract survives dependency upgrades; a grammar extension shifts the seed-to-case mapping, which is why ledger entries also carry the reduced query and data inline). One case is (data, pattern, predicate): N-Triples data mixing integers, decimals, doubles, plain and language-tagged strings, booleans, and IRIs under one predicate — so comparisons hit both comparable values and type errors — plus a pattern with an `OPTIONAL` clause whose object is present on only some subjects (the unbound-error fuel), plus a filter predicate drawn from a grammar spanning comparisons, arithmetic, `BOUND`/`COALESCE`, the accessors `STR`/`LANG`/`DATATYPE`/`isIRI`/`isLiteral` (the `LANG` atom is itself three-outcome: tag / empty string / a type error on an IRI), and the error-absorbing connectives `&&`/`||`/`!` at bounded nesting depth.

The grammar *excludes by construction* everything §3.4 forbids: no `RAND`/`NOW`/`UUID`/`STRUUID`/`BNODE` (nondeterminism), no `EXISTS`/`NOT EXISTS` (the spec-defect exclusion), no blank nodes in data (the

differential scope limit). A pinned test sweeps generated predicates across a fixed seed range and asserts the exclusion list holds.

6.2. Drivers and validated presets

External engines are driven over the standard SPARQL 1.1 Protocol (GET, POST-form, and POST-direct query methods), with per-endpoint configuration for the engines whose protocol behaviour needs it — e.g. the Virtuoso preset uses form-encoded POST plus an explicit results-format output parameter, because its content negotiation historically ignores a bare `Accept` header on some deployments. Presets exist only for endpoints whose behaviour was validated against the driver (generic, Fuseki, Oxigraph, Virtuoso); presets for Blazegraph, GraphDB, QLever, and MillenniumDB are deliberately deferred to campaign bring-up rather than shipped unvalidated — an unvalidated preset would risk misattributing protocol quirks as engine failures. The engine under test also includes the project's own engine in-process: finding our own bugs is part of the method, not an embarrassment to be excluded.

6.3. Fail-closed verdict discipline

Every oracle returns one of three verdicts: *pass*, *violation* (wrong result), or *engine failure* — and the latter two are never conflated. If any of a TLP instance's four queries fails to evaluate, the verdict is an engine failure, not a violation and not a pass; a differential run with fewer than two engines is a harness failure, never a vacuous pass; a result-shape mismatch is a driver problem to triage. This mirrors the counting discipline of the field (the SQLancer-line papers count wrong-result bugs separately from crashes and errors) and makes the wrong-result class — the only class the headline claim of a logic-bug paper may count — structurally uninflatable by harness or protocol noise.

6.4. The found-bug ledger: confirmed-only counting, machine-enforced

The ledger is the paper's evidence artifact, and its schema enforces the counting methodology structurally rather than by author discipline. One entry is one JSON record with: engine and exact version; the flagging oracle; the reproducing generator seed (or null for hand-reduced cases); the reduced query and data; a strict classification (`wrong-result` vs `engine-error`, mirroring §6.3); a one-line summary; an **upstream issue URL, required** — the validator rejects an entry without a well-formed link, so an unfiled observation *cannot be a ledger entry*; and a **developer-confirmation status, required with no default**, over the lifecycle {reported, developer-confirmed, fixed, rejected, duplicate}. Rejected reports are retained deliberately: they are part of the method's error rate, and deleting them would silently inflate precision. Serialisation fails closed — an invalid entry aborts the write; a malformed line aborts the read. A paper claim about "logic bugs found" counts only `wrong-result` entries, and a headline count cites only the `developer-confirmed`-or-better subset, following the field norm [1], [7].

7. Evaluation

7.1. Instrument validation (evidence that exists today)

A metamorphic-testing instrument earns trust by demonstrating non-vacuity (it flags a known bug) and fidelity (its relations hold on a correct engine). Both are pinned as CI self-tests against the *real* engine of this project — not a mock evaluator — and are the only empirical claims this draft makes:

- **The case analysis, concretely.** On a fixed dataset covering all three EBV outcomes via both error causes — ages straddling the predicate `?age < 25`, one age bound to the string `"twenty"` (type error), one subject with no age triple under `OPTIONAL` (unbound error) — the pinned test asserts the base query returns 4 solutions and the branches return exactly 1 (true), 1 (false), and 2 (error): the §3.2 case analysis, checked row-for-row on a real engine (`tests/oracle_self_tests.rs::tlp_branches_partition_the`
- **Non-vacuity (mutation check).** A deliberately injected wrong-result mutant — a wrapper that silently removes one row from any `FILTER` query's result — must be flagged as a violation by TLP,

NoREC, **and** the differential oracle; a suite that cannot flag its own seeded bug proves nothing. All three flag it.

- **Fidelity on generated cases.** Across the pinned seed range (seeds 0–49, 50 generated cases), TLP and NoREC hold on the pristine engine — every verdict is a pass, and an engine failure would also fail the test, so silent grammar/driver rot is caught rather than skipped. A separate 200-seed sweep asserts the generator's exclusion list (§6.1).
- **Fail-closed behaviour.** An always-failing driver and a syntactically invalid query each yield engine-failure verdicts on all oracles — never a pass, never a violation.

These validate the *instrument*. They are deliberately not presented as a bug-finding result: passing self-tests on one engine is the entry ticket to a campaign, not a finding.

7.2. Campaign methodology (fixed before execution)

The campaign design is committed in this draft, before any campaign result exists, so that results cannot steer the method. Targets: Apache Jena/Fuseki, OpenLink Virtuoso, Blazegraph, GraphDB, QLever, Oxigraph, MillenniumDB — and the project's own engine as a first-class target. Procedure, per engine version pinned in a container matrix: iterate seeds through the generator; run TLP and NoREC per engine and the differential oracle across engines on identical data; deduplicate violations by (engine, oracle, reduced shape); *reduce* each surviving case (shrink data and predicate while the violation persists); manually triage against the recommendation text, discarding anything attributable to documented implementation latitude (§5); file an upstream issue; only then create a ledger entry, which the schema forces to carry the issue URL and a confirmation status. Headline counting rule, fixed now: **previously-unknown, wrong-result-class, developer-confirmed bugs in third-party engines**, reported per engine and per oracle, with reported-but-unconfirmed and rejected counts disclosed alongside. Engine-error-class findings (crashes, rejections of valid queries) are reported separately and never folded into the logic-bug count. The generator grammar may be extended during the campaign (UNION, subqueries, more datatypes) — extensions are versioned, and every ledger entry records the seed and grammar version that reproduce it.

7.3. Campaign results

PLACEHOLDER — campaign not yet run (bead `sq-gum8.11`). This section intentionally contains no numbers. It will report, per engine and per oracle: previously-unknown developer-confirmed wrong-result bugs (the headline count, cross-checked 1:1 against the public ledger — each with its upstream issue link); reported-awaiting-confirmation, rejected, and duplicate counts (the method's full precision accounting); engine-error-class findings, separately; and per-oracle overlap (which bugs only one oracle could see). Bugs found in our own engine will be reported in the same tables, marked as self-found. If the campaign yields no substantive third-party confirmations, this paper will be honestly re-framed as an experience/negative-result note — see §9.

engine	TLP	NoREC	differential	confirmed total
(per-engine rows pending the campaign)	—	—	—	—

Table 2: Placeholder: confirmed wrong-result bugs by engine and flagging oracle. Every cell will be machine-cross-checked against the found-bug ledger (one upstream issue link per counted bug); the table is generated from the ledger, not hand-typed.

7.4. Threats to validity

Internal: the rewrite machinery is part of the tested surface (§3.3) — a violation localises to a set of constructs, not one; triage-by-reduction addresses, but cannot fully eliminate, misattribution within that set. The oracles' preconditions are enforced by the generator's grammar; hand-written cases

bypass that enforcement and must be checked manually. *External*: the generated fragment is narrow (one pattern shape with `OPTIONAL`; filter predicates over two variables; no property paths, aggregates at the oracle level, or updates), so absence of violations says nothing beyond the fragment; this bounds claims, not correctness. *Construct*: NoREC's premise is architectural, not semantic (§4) — on engines that canonicalise both forms to one plan it finds nothing, which weakens coverage claims but cannot produce false alarms. *Conclusion*: confirmed bug counts depend on upstream maintainer responsiveness; the ledger's reported/confirmed/rejected split keeps that dependency visible instead of hiding it.

8. Related work

SQL. SQLancer established practical logic-bug oracles for relational engines: PQS synthesises a query guaranteed to contain a pivot row (121 total bugs across SQLite, MySQL, PostgreSQL [11]); NoREC compares optimized against non-optimizing evaluation forms (51 optimization bugs [2]); TLP partitions on a predicate over SQL's ternary logic (77 logic bugs [1]). Our §3 is a re-derivation of TLP for a language whose third truth state is not a value, and our §4 adapts NoREC's rewrite through SPARQL's projection-error semantics.

Datalog. queryFuzz applies metamorphic transformations to Datalog programs (13 bugs in Soufflé, μZ, and ddlog [3]); DLSmith composes dependency-aware metamorphic relations (16 bugs [4]). Datalog's two-valued semantics sidesteps exactly the trichotomy that makes the SPARQL derivation non-trivial.

Graph query languages. Randomized differential testing found bugs across Gremlin implementations [5]; GDsmith generates semantics-respecting Cypher (28 reported bugs [6]); Gamera contributes graph-aware metamorphic relations over Gremlin systems (39 reported logic bugs, 15 developer-confirmed [7]). Closest to us, Kamm et al. ported TLP to Cypher engines [8] — but Cypher, like SQL, models its third truth state as a `NULL` *value* with an `IS NULL` test, so the port keeps SQL's partition shape. To our knowledge no prior work derives a logic-bug oracle for a language whose third truth state is a non-value evaluation outcome; the `COALESCE∘IF` reification and its case analysis (§3) do not appear in prior work.

SPARQL. Engine correctness work has centred on the W3C test suites (shared, finite, and known to every implementer — valuable for conformance, structurally unable to find bugs outside their fixed cases) and on benchmark studies. SparqLog [9] reported wrong or non-conforming results in Virtuoso on 14 of 77 studied queries — found incidentally while validating a SPARQL-to-Datalog translation, without a dedicated oracle, generator, or filing methodology. We found no dedicated metamorphic or differential logic-bug testing publication for SPARQL engines in the venue search behind this project's paper-selection record (top software-engineering, database, and semantic-web venues); we state this as a search-bounded claim, not an absolute one.

Delta. Against all of the above, this paper's delta is: (i) the first dedicated logic-bug testing method for SPARQL engines; (ii) the TLP partition re-derived for a three-*outcome* (not three-value) semantics, with a recommendation-clause-level case analysis; (iii) an oracle-validity honesty layer (spec-defect exclusion for `EXISTS`, spec-latitude triage, fail-closed verdict classes) that keeps the method's claims inside what the standard actually pins down; and (iv) a machine-enforced confirmed-only counting methodology (§6.4).

9. Limitations and honest status

This is a draft without campaign results. The field's publishability bar for a testing paper is previously-unknown, developer-confirmed bugs in third-party systems (13–121 across the cited papers); this draft claims *zero* such bugs, because the campaign has not run. If the campaign yields too few substantive confirmations, the honest disposition — decided in advance — is to re-frame this work

as an experience/negative-result note reporting the re-derivation and what a bug-free (or latitude-dominated) result field says about SPARQL engine maturity. The methodology of §7.2 is fixed either way.

Fragment scope. The oracles cover `SELECT` queries under the §3.4 preconditions; `EXISTS` is excluded on spec-defect grounds (a limitation of the standard we inherit deliberately); property paths, aggregates-at-the-oracle-level, updates, and federated queries are out of scope. The generated grammar is deliberately narrow at first (§7.4) and grows only as engines prove stable on the current fragment.

Oracle power. TLP and NoREC test consistency of an engine with itself; a uniformly-wrong evaluator satisfies both (the differential oracle exists precisely for that case, and conversely needs a second engine). NoREC additionally rests on an architectural heuristic (§4).

Instrument-stage evidence. Everything empirical in this draft is CI-pinned self-test evidence on one engine (§7.1) — deliberately so, and deliberately labelled as instrument validation rather than findings.

10. Conclusion

SPARQL engines have never had a dedicated logic-bug testing method, and the reason turns out to be semantic, not sociological: the field's strongest metamorphic oracle partitions on a third truth *value* that SPARQL does not have. We re-derived the partition for SPARQL's three-valued error semantics by reifying the error outcome through `COALESCE` and `IF` — a construction checkable clause-by-clause against the recommendation — adapted NoREC through SPARQL's projection-error semantics with its heuristic status stated plainly, and built the surrounding instrument so that its verdicts fail closed and its evidence ledger cannot structurally contain an unfiled or unconfirmed "bug". The instrument is validated; the campaign, whose methodology is fixed in this draft, is the next step — and its results, whatever they are, will be reported against the counting rules committed here.

References

[1] M. Rigger and Z. Su, "Finding Bugs in Database Systems via Query Partitioning," *Proceedings of the ACM on Programming Languages*, vol. 4, 2020.

[2] M. Rigger and Z. Su, "Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[3] M. N. Mansur, M. Christakis, and V. Wüstholz, "Metamorphic Testing of Datalog Engines," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, 2021.

[4] M. N. Mansur, V. Wüstholz, and M. Christakis, "Dependency-Aware Metamorphic Testing of Datalog Engines," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*, 2023.

[5] Y. Zheng *et al.*, "Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*, 2022.

[6] Z. Hua *et al.*, "GDsmith: Detecting Bugs in Cypher Graph Database Engines," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*, 2023.

[7] Z. Zhuang, P. Li, P. Ma, W. Meng, and S. Wang, "Testing Graph Database Systems via Graph-Aware Metamorphic Relations," *Proceedings of the VLDB Endowment*, vol. 17, no. 4, 2023.

[8] M. Kamm, M. Rigger, C. Zhang, and Z. Su, "Testing Graph Database Engines via Query Partitioning," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*, 2023.

[9] R. Angles, G. Gottlob, A. Pavlović, R. Pichler, and E. Sallinger, "SparqLog: A System for Efficient Evaluation of SPARQL 1.1 Queries via Datalog," *Proceedings of the VLDB Endowment*, vol. 16, no. 13, 2023.

[10] T. Y. Chen *et al.*, "Metamorphic Testing: A Review of Challenges and Opportunities," *ACM Computing Surveys*, vol. 51, no. 1, 2018.

[11] M. Rigger and Z. Su, "Testing Database Engines via Pivoted Query Synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[12] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language." W3C, Mar. 21, 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[13] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and Complexity of SPARQL," *ACM Transactions on Database Systems*, vol. 34, no. 3, 2009.

[14] "SPARQL 1.2 Query Language." W3C, 2025. [Online]. Available: https://www.w3.org/TR/sparql12-query/

[15] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, 1998.

[16] G. L. Steele Jr., D. Lea, and C. H. Flood, "Fast Splittable Pseudorandom Number Generators," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2014)*, 2014.

---

sparq project · DRAFT, in progress — the cross-engine campaign (tracked as bead `sq-gum8.11`) has not run; no third-party bug is claimed. Evidence in §7.1 traces to the merged instrument crate `crates/sparq-metamorph` (bead `sq-gum8.6`): the TLP case analysis and self-tests in `src/tlp.rs` + `tests/oracle_self_tests.rs`, the NoREC honesty note in `src/norec.rs`, the generator exclusions in `src/generate.rs`, and the ledger schema in `src/ledger.rs`. Positioning per `research/paper-selection.md` §3.7 + §5-P2. Campaign-dependent numbers will flow through the paper-factory evidence layer (cross-checked against the public found-bug ledger) when they exist; none appear here.