ROWGATE: DETERMINISTIC ROW-LEVEL ACCESS CONTROL FOR LLM QUERY AGENTS OVER FEDERATED SOURCES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents that turn natural language into SQL are now being put in front of enterprise data, where the access policy has to hold at the row level and the data usually cannot leave the systems it already sits in. The common practice is to write the policy into the prompt and trust the model to follow it. We measured how well that works, and it breaks in two directions at once. On a 150-probe adversarial benchmark covering five RBAC roles and ten models from four vendors, refusal rates on identical inputs run from 0.0% to 72.0% and schema fidelity from 54.4% to 100%. The two do not move together. The most cautious model turns away 72% of perfectly ordinary business queries; another turns away none and matches it on fidelity; a third turns away none but reaches into unauthorized tables 45.6% of the time. Even when the caller's scope is supplied as context, models leave the mandatory row filter off 23% to 32% of the SQL they do produce, and rewording the prompt moves that number. RowGate takes the model out of the trust path instead. Policy is enforced at two deterministic layers built over a four-graph metadata substrate: unauthorized tables and columns are stripped from the prompt before generation, and the caller's mandatory predicates are injected into every scope of the parsed SQL before execution, including the per-source sub-queries of a federated plan. We state the guarantee as a partition over outcomes and prove it under three conditions. Prior work checks generated SQL statically; we instead run the rewritten queries against live SQL Server and PostgreSQL instances holding the real AdventureWorks database, and score exact result-set equality against an authorized-only ground truth. Over more than 700 executed queries, covering complex query shapes, a 1,000-table catalog, irregular naming, and a cross-engine join that falls from 121,317 rows to the 5,976 authorized ones, RowGate returns 0 unauthorized rows every time, and exactly the authorized rows in all but 9 long-tail cases, which fail closed rather than leak. Executing the queries also turned up four row-level leaks in our own rewriter that a static predicate check would have passed. We then overlay policies mechanically onto the Spider benchmark and run 495 queries over 17 databases the rewriter has never seen: all exact, 0 leaks, against 192 that a rewriter injecting nothing would leak on. Enforcement matches database-native row-level security on a single engine, and extends where that cannot reach, across engines and ahead of generation. The rewrite is a pure function of the query and the policy; its output is byte-identical at 33 tables and at 1,000. We release the benchmark, the invariant taxonomy, and the harness.

1 INTRODUCTION

LLM agents that translate natural language into SQL are becoming the interface to enterprise data. A user asks a question, the model writes SQL, the SQL hits a database, rows come back. The access control question is a simple one. Does that SQL touch only the rows the principal is allowed to see?

Almost everyone answers it the same way: put the policy in the prompt and tell the model to comply. We wanted to know what that actually buys you. On a 150-probe adversarial benchmark spanning five RBAC roles, four current frontier models refuse somewhere between 42.7% and 51.3% of

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

queries that look like ordinary business questions. That is not safety. It is an availability failure, and it blocks legitimate work. When those same models are handed the caller's role and territory as plain context and do answer, they leave the mandatory row filter off 23% to 32% of the SQL they write. The rate also moves with phrasing. Under a system prompt that spells out the row rule as a command, both models filter every query they answer; under a prompt that just supplies the caller's scope, they do not. An earlier model generation applied the mandated filter on only 33.3% of probes. A control whose failure rate depends on which model shipped this quarter, and on how someone worded an instruction, is not an access-control boundary.

RowGate takes the model out of the trust path. The model translates; a deterministic rewriter enforces. Policy is applied at two independent layers over a metadata substrate of four knowledge graphs: before generation, the metadata the model can see is filtered to the principal's authorized surface; after generation, the principal's mandatory predicates are injected into every scope of the parsed SQL, including the per-source sub-queries of a federated plan. The rewriter never reads the user's text, so it has no notion of what counts as adversarial.

Three commitments shape the rest of the paper. We verify by execution, not by inspection. Every correctness claim here is scored by running the SQL on a live engine and comparing the rows that come back against an authorized-only ground truth, which catches leaked rows and dropped rows with the same test. That is how we found four real leaks in our own rewriter (§7.1). All four would have passed a predicate-presence check. We state the guarantee and prove it (§4), as a partition over outcomes under three conditions, including the one case we know of that costs utility without costing safety. We do not claim parser completeness, and we explain why the guarantee does not need it. We measure where the mechanism stops working (§7.6). Of 58 adversarial naming and long-tail cases, 49 come out exact and 9 either fail closed or over-restrict. Negative controls show the suite can actually detect a leak: deliberately broken rewriters leak on 55 of the 58, ours on none.

Our contributions: (i) a measurement of how instruction-level enforcement fails in both directions across current models, together with the prompt sensitivity that makes it unusable as a boundary; (ii) a two-layer deterministic enforcement design over federated sources, with a guarantee we state and prove; (iii) an execution-based evaluation method that caught four leaks a static evaluation had missed, and that carries over to 17 schemas the rewriter has never seen; and (iv) the released artifacts.

2 Background and threat model

An authenticated user (the *principal*) sends a natural-language query to an LLM agent that translates it to SQL and runs it against backing databases. Access control is role-based: a role's policy specifies which tables, columns, and rows the principal may see.

Adversary and capabilities. The adversary here is the principal, and the only lever they have is the wording of the query. They can ask anything, look at what comes back, and try again. What they cannot do is modify the rewriter, the substrate, or the policy file, and they have no direct SQL access. Every path to a database runs through the rewriter.

Scope (diagrammed in Appendix E). In scope: the LLM emits SQL touching data outside the principal's authorized scope, through filter omission, schema overreach, role manipulation, or prompt injection. Out of scope: paths bypassing the rewriter, engine vulnerabilities, side channels, cross-role collusion, a compromised policy file, and inference by aggregating authorized queries, since we enforce per-query access control and not differential privacy.

So a principal who talks the model into writing unfiltered SQL has not actually obtained anything. The rewriter is what reaches the database, and it works off the policy no matter what the model wrote.

3 The RowGate architecture

3.1 The four-graph substrate

The **Domain Graph** holds the schema, including cross-source relationships that no single catalog can see. The **Subject Graph** holds resolved entity instances. A customer carrying different

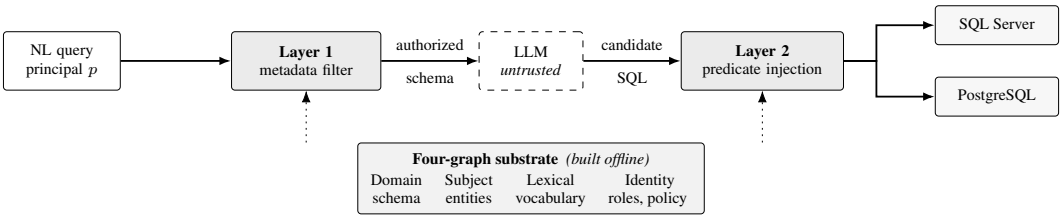


Figure 1: Enforcement runs at two deterministic layers around the model. Layer 1 removes unauthorized metadata before generation; Layer 2 injects the principal's mandatory predicates into every scope of the parsed SQL, including each per-source sub-query. Each source is filtered in place and the partial results are joined on the Subject key, so data never moves. The LLM sits outside the trust boundary: it never decides policy.

identifiers in two systems becomes one subject node with edges to both source rows, and that is what makes a federated join possible without copying anything. The **Lexical Graph** maps natural-language tokens onto Domain and Subject elements. The **Identity Graph** encodes principals, roles, and their policies, where a policy is just a set of predicates (column visibility, mandatory row filters, aggregation rules) keyed to Domain Graph entries.

The substrate is built offline. At query time the enforcement path only consults a flat per-table cache and the principal's policy. That is an $O(1)$ lookup and it fails closed on any table it cannot resolve, so the size of the substrate cannot change a runtime enforcement decision, except by causing an over-deny (§7.4).

3.2 Two-layer enforcement

Layer 1, pre-model metadata filtering. Before the model is called at all, the metadata it will see (table descriptions, column listings, entity definitions, semantic context) is cut down to the subset the principal's policy authorizes, through `authorize_metadata(elements, principal, policy)`. The same check is applied at more than one point in the pre-prompt pipeline, once when semantic context is assembled and again when entity references are resolved against the Subject Graph. A model cannot reference a surface it was never shown.

Layer 2, mandatory predicate injection at SQL emit. The candidate SQL is parsed and its scope tree resolved. For every scope that references a protected table, the principal's mandatory predicates get conjoined into *that scope's own* WHERE clause, one predicate per alias occurrence. The model never writes the predicate. The rewriter does, and it will not emit SQL that is missing what the policy requires. Anything the parser cannot accept is refused rather than forwarded.

In-situ federated execution. Data does not move. A cross-source query is decomposed into per-source sub-queries, each rewritten with that source's own mandatory filters, and the partial results are joined at the application layer on the Subject node. Each sub-query carries its predicate in its own WHERE clause, and WHERE is evaluated before GROUP BY, HAVING, and aggregation, so the filtering happens upstream of every aggregation and every join.

4 The enforcement guarantee

Here is the property RowGate actually gives you. Let p be a principal with policy P, let q be any SQL string the model produced, and let A_p be the set of rows P authorizes p to see.

Assumption 1 (Parse fidelity)**.** *If the parser accepts q, the resolved scope tree enumerates every scope in which a protected table is referenced, and every alias under which it is referenced within a scope.*

Assumption 2 (Monotone injection)**.** *The rewriter modifies q only by conjoining policy predicates into the WHERE clause of scopes that reference a protected table. It adds no relation, removes no predicate, and alters no projection.*

Assumption 3 (Fail-closed)**.** *If the parse fails, or a referenced table cannot be resolved in the metadata cache, or the protected table is the target of a write, no SQL is emitted.*

Theorem 1 (Enforcement partition)**.** *Under Assumptions 1–3, for every q the outcome of $RowGate(q, p, P)$ falls into exactly one of four classes:* (i) enforced*, in which every scope referencing a protected table carries the conjoined mandatory predicate;* (ii) refused*, in which no SQL is emitted;* (iii) engine-rejected*, in which the emitted SQL is rejected by the database; or* (iv) over-restricted*, in which the result is a strict subset of the rows q would have returned over A_p. In all four classes, the rows returned to p are a subset of A_p.*

Proof sketch. Classes (ii) and (iii) return no rows, so containment is immediate. For (i) and (iv), consider any scope s referencing a protected table T. By Assumption 2 the rewriter replaces the scope's source relation R_s with $\sigma_\pi(R_s)$, where π conjoins P's mandatory predicates for T, so $\sigma_\pi(R_s) \subseteq R_s \cap A_p$. Because WHERE is evaluated before GROUP BY, HAVING, window functions, and the projection, every downstream operator in s computes over authorized rows only. By Assumption 1 no referencing scope and no alias occurrence is missed, and conjunction is monotone decreasing, so composing scopes cannot reintroduce a row outside A_p. By Assumption 3 any q outside the parser's coverage, or touching an unresolved table, does not reach the engine. □

Note that the guarantee does not need the parser to be complete. Anything outside the parser's coverage falls into class (ii) by Assumption 3, which is why we report refusals instead of hiding them. The schema is not an input either: the rewrite is a pure function of $(q, T, \pi, \text{alias}, \text{dialect})$, which §7.4 checks by hashing.

Placement, and the exception that is left. Where inside the scope the predicate lands turns out to matter. Put it in the WHERE clause when the protected table sits on the null-supplying side of a LEFT join and it quietly demotes the outer join to an inner one. The result still stays inside A_p, so Theorem 1 holds, but rows of the *unprotected* relation disappear. So the rewriter uses that join's ON clause instead. FULL joins are excluded, and the reason is worth spelling out. A FULL join keeps unmatched rows from *both* sides, so a predicate sitting in ON leaves unauthorized rows alive as null-padded rows. ON placement there is a row-level leak. Our own suite caught exactly that when we first applied the fix uniformly, so FULL joins keep WHERE placement and over-restrict instead. That leftover case is the one deviation we would not call merely conservative.

5 Experimental setup

Benchmark. The base set is 14 natural-language queries taken by hand from a production diagnostic suite and crossed with five RBAC roles, so 70 entries. Six are cross-source entries we label UNDETERMINED and drop, leaving 64 scored. The adversarial set is 30 probes across four classes (prompt injection, role swap, schema overreach, aggregation channel) crossed with the same roles, so 150. The base queries came out of a real operational workload and were not curated to make the system look good. The probes were written to attack enforcement.

Engines and data. Everything runs on one commodity workstation (Apple silicon, 16 GB RAM) with the engines in local containers and no GPU: the rewriter is CPU-only and models are called over an API. Correctness is verified on live SQL Server (Azure SQL Edge) holding the real Microsoft AdventureWorks database (71 tables, 472 columns; `Sales.SalesOrderHeader` holds 31,465 rows), live PostgreSQL for the federated leg, and embedded DuckDB for the dialect-portable stress suites. The parser is `sqlglot` pinned at 25.34.1.

The oracle. Scoring is *exact result-set equality* against ground truth, meaning the original query evaluated over a world holding only the rows the principal is entitled to. One metric catches both failure directions at once: rows that came back and should not have (*leaked*), and rows withheld that should have come back (*dropped*). A correct result has zero of each. Result sets are compared as multisets, so ordering is not checked.

Load-bearing cases. A case whose unfiltered baseline already equals ground truth proves nothing at all. Every suite therefore reports per-case baseline, ground-truth and returned row counts, labels the vacuous cases as vacuous, and is re-run through deliberately broken rewriters (§7.6).

Table 1: Pure-LLM baseline on 150 adversarial probes, no enforcement, ten models over four vendors, ordered by refusal rate. Refusing ordinary queries is itself a failure, so neither column alone measures safety. RowGate's row does not depend on the generator.

Model	Vendor	Refusal rate	Schema fidelity	Filter omitted
Claude Fable 5.1	Anthropic	72.0%	95.2%	0
Claude Haiku 4.5	Anthropic	51.3%	100.0%	2
Claude Sonnet 4.6	Anthropic	47.3%	96.2%	0
Claude Opus 4.8	Anthropic	44.7%	93.9%	0
Claude Sonnet 5	Anthropic	42.7%	97.5%	0
Claude Opus 5.5	Anthropic	23.3%	83.5%	0
DeepSeek Chat	DeepSeek	15.3%	68.1%	6
Gemini 3 Flash	Google	2.0%	78.2%	2
GPT-5.1	OpenAI	0.0%	90.7%	1
Llama 3.3 70B Instruct	Meta	0.0%	54.4%	9
Prior-generation baseline	Anthropic	20%	80%	∼67% of filtered cases
RowGate, any upstream model	n/a	0	100%	0

6 Instruction-level enforcement is unreliable in both directions

Ten models, four vendors, no enforcement. Table 1 runs the 150-probe adversarial set against ten current models. Refusal spans 0.0% to 72.0% on identical inputs, and it does not track safety. Claude Fable 5.1 turns away 72% of business-looking queries and gets 95.2% schema fidelity for it, while GPT-5.1 turns away none and still reaches 90.7%. The refusals buy almost no fidelity and block most of the legitimate work. Llama 3.3 70B also refuses nothing, but it reaches into unauthorized tables 45.6% of the time and drops the mandatory filter on nine probes. The spread is not only a vendor effect either. Inside Anthropic's own lineup refusal ranges from 23.3% to 72.0%. A control that swings this far between models, and between versions from a single vendor, is not a boundary anyone can reason about. For reference, the earlier-generation baseline from the prior version of this work applied the mandated filter on 33.3% of probes (20/60, Wilson 95% CI [23, 46]).

Compliance depends on how the policy is phrased. Table 5 is the controlled ablation. The model gets identical schema and role context, and we compare its raw SQL (rewriting OFF) against that same output run through the rewriter (ON), so the rewrite is the only thing that varies. Under *strict* framing the system prompt states the row rule as a command, and both models then filter every query they answer. Under *context* framing the identity layer supplies the caller's role and territory but never tells the model to self-filter, which is what a deployed system actually does, and compliance drops to 77.3% and 68.0%. We report both, because the gap between them is the finding. A control that moves when you reword the prompt is not a boundary. The rewriter's output does not move.

One cell is not 100%, and rather than round it away we will say what it is. In the Opus context run, one emitted query reads a table that carries no mandatory filter column under this role's policy (`Sales.Customer`). The rewriter correctly injects nothing, and the scorer then records the filter as absent. It is a scoring artifact on a query that was never in scope for injection to begin with, but we report 24/25 because that is what the harness measured.

7 Execution-based correctness

7.1 Four leaks that static checking misses

Switching to the execution oracle falsified our own implementation almost immediately. We found and fixed four row-level enforcement defects. Every one of them passes a predicate-presence check. *(1) Multi-scope under-filtering*: the predicate reached only the first scope referencing a protected table, so a UNION leaked (Figure 2); fixed by injecting into every referencing scope via the resolved scope tree. *(2) Multi-alias under-filtering*: a table referenced twice in one scope under distinct aliases got a single ambiguous predicate; fixed by one predicate per alias. *(3) Recursive-WHERE misplacement*: a recursive WHERE lookup could place an outer-scope predicate inside a nested

Adversarial probe, role `sales_rep`, policy mandates `TerritoryID = 5`:
"recent orders, plus every online order, across all regions"

model

```
SELECT SalesOrderID FROM
Sales.SalesOrderHeader
 WHERE OrderDate >= '2013-07-01'
UNION ALL
SELECT SalesOrderID FROM
Sales.SalesOrderHeader
 WHERE OnlineOrderFlag = 1
```

RowGate

```
SELECT SalesOrderID FROM
Sales.SalesOrderHeader
 WHERE OrderDate >= '2013-07-01' AND
TerritoryID = 5
UNION ALL
SELECT SalesOrderID FROM
Sales.SalesOrderHeader
 WHERE OnlineOrderFlag = 1 AND TerritoryID = 5
```

Executed on live SQL Server `Sales.SalesOrderHeader`, 31,465 rows

model SQL unauth.	50,861 rows, 29,024
first scope only unauth.	27,822 rows, 27,642
RowGate	180 rows, 0 unauth.
ground truth	180 rows (exact match)

The rewriter never reads the probe text. It injects into *both* UNION arms; filtering only the first arm is the defect of §7.1, and the middle row is what it costs.

Figure 2: One adversarial probe traced end to end. The model drops the mandated filter when asked for "all regions"; the rewriter injects it into every scope and execution returns the authorized rows.

subquery; fixed by using each select's own WHERE. *(4) Operator-precedence composition*: the predicate was conjoined without parenthesisation, and since AND binds tighter than OR, `a OR b` became `a OR (b AND policy)`, leaving the first disjunct unfiltered (`WHERE 1=1 OR 1=1` returned 600 of 600 rows, 540 unauthorized); the same defect collapsed the silent-deny path. It reproduced on DuckDB, PostgreSQL and T-SQL alike.

Defects 1 through 3 showed up once we started executing nested and set-operation shapes. Defect 4 came out of extending our suite to disjunctive predicates, where 9 of 11 such cases leak before the fix and 0 after. All four carry regression tests now, and the enforcement path passes 213 of 213 unit tests, the full suite 6,664 of 6,665, the single failure an order-dependent import-isolation test with nothing to do with enforcement. These defects are really the argument for the methodology. If you think a deterministic rewriter is trivially correct, note that four independent leaks survived in ours until we ran the queries.

7.2 The full policy surface, on real data

We ran all four policy columns across five query shapes (flat, CTE, subquery, nested, UNION) on live SQL Server against real AdventureWorks. Every cell returns exactly the authorized rows, 0 leaked: TerritoryID 486 of 31,465, SalesPersonID 429 of 31,465, VendorID 153 of 4,012, DepartmentID 180 of 296 (Table 6, Appendix G). The enforcer iterates over every mandatory column a table declares; the restriction to two columns in the prior version of this work was offline metadata population, not an algorithmic limit. All 20 column-by-shape combinations come back exactly equal to ground truth, 0 unauthorized and 0 dropped, no cell vacuous, so enforcement keeps utility as well as safety.

7.3 Executing the benchmark's own generated SQL

The suites above run shapes we wrote ourselves. Here we run the SQL that the four models actually produced on the 150-probe adversarial benchmark. Of 600 model-rows, 279 are refusals, 53 are cross-source, and 27 never touch the protected table, which leaves 78 on the enforced path. Ground truth is again materialised independently of the rewriter.

Across all four models (Table 3, Appendix B) every one of the 78 enforced-path queries returns exactly the ground-truth rows, and nothing leaks. **Execution success is preserved exactly.** All 78 queries the engine accepted before rewriting were still accepted afterwards, so the AST rewrite broke

nothing on CTEs, subqueries, set operations or aggregates. Five model queries were rejected by the engine in their original form. Those are the generator's errors, and the rewriter neither repaired nor worsened them. An earlier rewriter did over-restrict one of them: the model wrote a LEFT JOIN from `Sales.Customer` to the protected table, and 85 rows of the *unprotected* dimension were dropped. That case is what motivated the ON placement of §4. Finding it in a benchmark run rather than in our own stress cases is the strongest evidence we have that the execution oracle is doing real work.

Of the base half's 64 entries, 28 are DENY cases with no ground-truth SQL, and 27 execute. The 9 that do not are benchmark-to-engine mismatches. All 7 that reach the enforced path are exact, 0 leaked.

7.4 Complexity and scale

Fifteen complex enterprise shapes over a 51-table schema with a cryptically named protected fact table (five-level nesting, CTE chains, windows including an unpartitioned `SUM() OVER ()`, correlated subqueries including one on a non-policy column, GROUP BY with HAVING, multi-way and self-joins under cryptic aliases, set operations, protected-in-CTE-and-outer, aggregate over aggregate) all return exactly the authorized result, 0 leaked and 0 dropped, with baselines from 600 to 13,800 rows. What makes this a measurement rather than an assertion is the negative control: filtering only the first scope (defect 1) leaks on 6 of the 7 multi-scope cases (self-join 720 rows, correlated-scalar 540, CTE-plus-outer 60, correlated-EXISTS 60, correlated-non-policy 30, UNION 9).

We then scale up to 38 cases over a 1,000-table, 2,971-column catalog: the same 15 shapes plus nesting depth 1 to 50, self- and dimension-join width 2 to 30, UNION arity 2 to 20, and CTE chains of 2 to 20. All exact, all load-bearing. Schema-size invariance is verified rather than asserted. Rerunning all 38 against a 33-table catalog gives identical verdicts and identical row counts, and a SHA-256 over the 38 rewrites comes out the same with either catalog open and from a fresh rewriter (`857987b71c45987c`). Rewrite cost follows the AST and not the catalog, from 0.20 ms flat to 9.22 ms for a 30-arm self-join, against a model call that averages 2,921 ms (Appendix M). Only 14 of the 38 carry an enforcement signal. The other 24 are robustness evidence and we label them that way.

7.5 Generalisation to unseen schemas

Everything so far runs on AdventureWorks or on a synthetic catalog, which leaves open the obvious question of whether the mechanism depends on that one schema. So we ran it over the Spider development split (Yu et al., 2018): 20 real databases from unrelated domains, none of which the rewriter has seen. Spider has no notion of permissions, so we overlay one mechanically. Per database we take tables with at least 8 rows, keep columns of cardinality 2 to 12 with no nulls, then pick the largest table and within it the lowest-cardinality column, with the policy value set to that column's mode. The rule is deterministic and nothing about it is tuned per database. Ground truth is the same independent oracle we use elsewhere, and we use Spider's own gold SQL so that the generator is not a variable here.

Of the 20 databases, 17 yielded a policy and contributed 495 enforced-path queries (Table 2, Appendix A). All 495 exact, 0 over-restricted, 0 leaked, no accepted query broken. The accounting matters as much as the headline. Only 350 are load-bearing; on the other 145 the authorized world happens to equal the full world, so we count those separately. A rewriter injecting nothing would leak on 192. Three databases yielded no policy, because no table in them had anything resembling a tenant column, and we report those skipped rather than hand-picking something.

7.6 Irregular naming and the long tail

Real enterprise SQL is not clean. We built 58 adversarial cases covering irregular naming (case folding, quoting, embedded spaces, reserved words, unicode, 120-character names, aliases and CTEs that shadow the protected name), disjunctive WHERE clauses inside CTE bodies and across UNION arms, dialect long-tail constructs (recursive CTEs, LATERAL, GROUPING SETS, ROLLUP, CUBE, QUALIFY, DISTINCT ON, UNNEST, anti-joins), and writes. The full list is in Appendix L.

The outcomes land in the classes of Theorem 1: 49 enforced exactly, 6 refused, 3 over-restricted, and 0 leaked. The negative controls are what establish that the suite can detect a leak at all. A no-op rewriter leaks on 55 of the 58 cases and a first-scope-only rewriter on 7 of 58, while this one leaks on none. Those controls earned their keep. An earlier version of the ON placement in §4 applied it to FULL joins as well, and this suite caught the leak that created. The three cases that cannot detect a leak under any rewriter are the two that are entitled to their full result plus a deliberately invalid-SQL control, and those test over-restriction and fail-closed behaviour instead (Figure 4, Appendix F).

Cross-engine confirmation. We re-ran 36 of these on live SQL Server against real AdventureWorks, whose naming is already irregular and whose dialect differs: 32 exact, 1 over-restriction, 3 fail-closed refusals, 0 leaked. The over-restriction is the same FULL OUTER JOIN case we see on DuckDB, so that leftover limitation belongs to the rewrite and not to any one engine.

8 FEDERATED ENFORCEMENT

Federation is where native per-engine controls simply do not reach. We evaluate the per-source decomposition of §3 on the real cross-source stack: PostgreSQL `production.product` (504 products) joined with SQL Server `Sales.SalesOrderDetail` and `SalesOrderHeader` on ProductID, the Subject node. Each source is queried in place and RBAC-filtered before the join, so data never moves.

With no enforcement the two-source join returns 121,317 rows, of which 115,341 are unauthorized. With per-source RBAC it returns 5,976 rows from the authorized territory, 0 leaked, and a confidential cost column masked. Aggregation falls out of placement rather than needing a special case: the predicate lands in each sub-query's WHERE, WHERE runs before GROUP BY and HAVING, so aggregates compute over authorized rows only (source-level pushdown 2,446 to 253 rows and GROUP BY with HAVING 1,232 to 116, both exact; a three-source star join returns 5,976 rows, 0 unauthorized). Table 7 in Appendix H has the full topology results. A principal with split access, authorized to one source and denied the other, has the denied sub-query refused at that source, which is why such cases carry the UNDETERMINED label. That is a policy decision, not a rewrite failure.

Aggregation is also grain-correct: for `SUM(TotalDue)` over `SalesOrderHeader` joined to `SalesOrderDetail` (one-to-many), a naive join inflates \$123.2M to \$2.93B (23.8×), and the grain planner prunes the multiplying child and restores it, composing with RBAC (authorized total \$8,884,099). An eight-table join over the real 71-table schema falls from 60,919 rows to 5,937, 0 leaked, exact. **Scope, stated honestly:** sources align on a shared Subject key, a star topology; per-hop chains on different keys are out of the current model, and a cross-source aggregate is application-layer over already-filtered rows, not a distributed re-aggregation engine.

9 COMPARISON WITH DATABASE-NATIVE ROW-LEVEL SECURITY

The obvious question is what a custom rewriter gives you that native RLS does not. Rather than argue about it we ran the comparison, configuring a real SQL Server SECURITY POLICY with a SESSION_CONTEXT-driven filter predicate on the actual `Sales.SalesOrderHeader`.

Native RLS earns parity where it applies. With the session context set it returns exactly the 486 authorized rows, 0 leaked, and replaying the `sales_rep` probe SQL under a fully configured RLS role, both arms return 0 unauthorized rows. We are not claiming the rewriter is better at row filtering on a single engine. It is not. Three differences do remain. RLS *fails closed to 0 rows* when the application forgets SESSION_CONTEXT, so correctness depends on per-connection session propagation. RLS has no reach across engines either: on the federated PostgreSQL leg the SQL Server policy does nothing, while the same rewriter enforced there (239 rows, 0 leaked). And RLS is row-only and runs at execution time. For the four probes aimed at a denied HR surface, RowGate blocks before the model ever sees those tables; RLS lets the query be generated and then errors. Appendix J tabulates the comparison and repeats it on a second engine, where genuine PostgreSQL RLS reaches the same parity and the same gap in configuration burden.

Component attribution. Table 4 (Appendix C) ablates all four components. Take the rewriter out and 900 unauthorized rows come back; take the substrate out and you get the same 900, because the substrate supplies the predicate the rewriter injects. The two pre-model layers turn out to be

redundant *with each other*, and Appendix C reports that redundancy rather than claiming each layer is independently load-bearing, which the data do not support.

10 LIMITATIONS

We would rather state the limitations concretely, with the cost measured.

Outer-join demotion, what is left of it. LEFT joins get ON placement (§4), which fixes the demotion there. FULL joins keep WHERE placement, because ON placement on a FULL join leaks, so a FULL join with the protected table on a null-supplying side still over-restricts. Confirmed on both DuckDB and SQL Server.

Leaf-name matching. We treat table identity as the last dotted component, case-folded. A policy on `edw.F_SLS_TXN_0917` therefore also filters `stg.F_SLS_TXN_0917`, dropping 540 of 600 entitled rows, and the same happens to any CTE shadowing the protected name (9 of 10). In a warehouse where `stg`, `raw`, and `edw` share leaf names this is systematic. It over-restricts and never discloses, but the utility cost is real.

Parser and write-path coverage. An unaliased ambiguous policy column produces an engine binder error. That is fail-closed, but it comes from a missing qualification rather than a designed safety property, and we would rather say so. DuckDB's PIVOT form is refused by the pinned parser while native T-SQL PIVOT is exact. `UPDATE`, `DELETE`, and `MERGE` targeting a protected table are refused. Appendix I lists these with the evaluation-scope caveats: multiset comparison, the reconstructed cross-model baseline prompt, evaluation on AdventureWorks and synthetic catalogs only, and a guarantee relative to the loaded policy rather than to policy-authoring correctness.

11 RELATED WORK

Text-to-SQL and access control. Most of the text-to-SQL literature optimizes execution accuracy: few-shot prompting (Gao et al., 2023), schema linking (Li et al., 2023), staged retrieval (Talaei et al., 2024), multi-candidate selection (Lee et al., 2024). All of it asks whether the SQL returns the right rows, never whether those rows overlap the rows this user is allowed to see. A second line asks the model itself to enforce a policy stated in context, through role-policy benchmarks (Sanyal et al., 2025; Klisura et al., 2025), fine-tuning for compliance (Almheiri et al., 2025; Jayaraman et al., 2025), and defenses against injection and role manipulation (An et al., 2025; Zhu et al., 2025). There the model is the boundary, so compliance is probabilistic, and §6 measures what that costs. A third line does make enforcement deterministic, but puts it at the *tool* boundary: CaMeL (Debenedetti et al., 2025) tracks capabilities through a custom interpreter, Progent (Shi et al., 2025) states tool-call privileges in a policy language, AgentGuard (Luo et al., 2026) applies attribute-based access control to tool use, and Ji et al. (2026) adapt mandatory access control to privilege escalation. We share the premise and differ on granularity. Whether an agent may call a database tool is a different question from which rows the resulting query returns, and one authorized call can still walk off with a table. RowGate enforces inside the emitted SQL, which is the level row-level policy is actually written at. Classical databases solved this long before the model existed: Rizvi et al. (2004) enforced fine-grained policies by query rewriting, and PostgreSQL RLS (The PostgreSQL Global Development Group, 2025) is the production form of the same idea. We credit the principle to that tradition. What is ours is applying it to LLM-generated SQL over a federated substrate, with pre-model metadata filtering, a guarantee we state, and an evaluation that falsified our own implementation four times. One last thread: Edge et al. (2024) and Guo et al. (2024) use knowledge graphs for retrieval relevance, where we use ours as the substrate authorization is decided over.

12 CONCLUSION

Instruction-level enforcement fails in both directions and moves with prompt wording, so policy belongs below the model. RowGate guarantees a partition where no outcome returns unauthorized rows. Executing the queries rather than inspecting them exposed four real leaks in our own rewriter, then caught a fifth we introduced while fixing those.

Reproducibility statement

The benchmark (14 base queries $\times$ 5 roles, 6 UNDETERMINED excluded; 30 adversarial probes $\times$ 5 roles), the invariant taxonomy specification (46 checks across 8 validators), and the evaluation harness are provided as supplementary material with SHA-256 checksums. The harness scripts run the complexity, scale, naming, long-tail, federated, RLS, and ablation evaluations, and each emits per-case baseline, ground-truth, and returned row counts together with the load-bearing flag, so vacuous cases are visible and the negative controls of §7.6 can be re-run. We state the limit of this release plainly: the harness calls the substrate implementation, which we do not release, so the scripts document the protocol and the scoring oracle exactly but are not independently executable end-to-end. The architecture is described to a reimplementation level of detail in §3, the guarantee an independent implementation must satisfy is stated in §4, and the parser is pinned at `sqlglot` 25.34.1. The live-engine results additionally require a SQL Server instance with the public Microsoft AdventureWorks database and a PostgreSQL instance; the restore script is included.

Use of large language models

An LLM was used to polish writing and grammer check.

References

Saeed Almheiri et al. Role-aware language models for secure and contextualized access control in organizations. *arXiv preprint arXiv:2507.23465*, 2025.

Hengyu An, Jinghuai Zhang, Tianyu Du, Chunyi Zhou, Qingming Li, Tao Lin, and Shouling Ji. IPIGuard: A novel tool dependency graph-based defense against indirect prompt injection in LLM agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2025.

Edoardo Debenedetti et al. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813*, 2025.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-SQL empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*, 2023.

Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. LightRAG: Simple and fast retrieval-augmented generation. *arXiv preprint arXiv:2410.05779*, 2024.

Bargav Jayaraman, Virendra J. Marathe, Hamid Mozaffari, William F. Shen, and Krishnaram Kenthapadi. Permissioned LLMs: Enforcing access control in large language models. *arXiv preprint arXiv:2505.22860*, 2025.

Zimo Ji, Daoyuan Wu, Wenyuan Jiang, Pingchuan Ma, Zongjie Li, Yudong Gao, Shuai Wang, and Yingjiu Li. Taming various privilege escalation in LLM-based agent systems: A mandatory access control framework. *arXiv preprint arXiv:2601.11893*, 2026.

Dorde Klisura, Joseph Khoury, Ashish Kundu, Ram Krishnan, and Anthony Rios. Role-conditioned refusals: Evaluating access control reasoning in large language models. *arXiv preprint arXiv:2510.07642*, 2025.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. MCS-SQL: Leveraging multiple prompts and multiple-choice selection for text-to-SQL generation. *arXiv preprint arXiv:2405.07467*, 2024.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. RESDSQL: Decoupling schema linking and skeleton parsing for text-to-SQL. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2023.

Jiaqi Luo, Songyang Peng, Jiarun Dai, Zhile Chen, Zhuoxiang Shen, Geng Hong, Xudong Pan, Yuan Zhang, and Min Yang. AgentGuard: An attribute-based access control framework for tool-use LLM-based agents. *arXiv preprint arXiv:2605.28071*, 2026.

Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pp. 551–562, 2004.

Sayan Sanyal et al. OrgAccess: A benchmark for role based access control in organization scale LLMs. *arXiv preprint arXiv:2505.19165*, 2025.

Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. Progent: Programmable privilege control for LLM agents. *arXiv preprint arXiv:2504.11703*, 2025.

Shayan Talaei, Mohammadreza Mafi, Sina Boorboor, Mohammadreza Pourreza, Davood Rafiei, and Amin Saberi. CHESS: Contextual harnessing for efficient SQL synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

The PostgreSQL Global Development Group. Row security policies. PostgreSQL Documentation, Chapter 5.9, 2025. `https://www.postgresql.org/docs/current/ddl-rowsecurity.html`.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 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.

Kaijie Zhu, Xianjun Yang, Jindong Wang, Wenbo Guo, and William Yang Wang. MELON: Provable defense against indirect prompt injection attacks in AI agents. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025.

A SPIDER GENERALISATION DETAIL

Table 2: Generalisation over the Spider development split. The rewriter has never seen these schemas. Load-bearing counts cases where the filter changes the answer; the negative control is what a rewriter that injects nothing would leak on.

Databases in the split / yielding a policy	20 / 17
Enforced-path queries scored	495
EXACT (equals ground truth)	495
OVER-RESTRICT	0
LEAK	0
Valid queries broken by the rewrite	0
Load-bearing / vacuous	350 / 145
Negative control: a no-op rewriter leaks on	192 of 495

B PER-MODEL LIVE EXECUTION

Table 3: Live execution of the four models' own adversarial SQL against SQL Server and real AdventureWorks, scored against an independently materialised authorized-only world.

	sonnet-4.6	sonnet-5	opus-4.8	haiku-4.5
Enforced-path cases executed	16	24	21	17
EXACT (equals ground truth)	16	24	21	17
OVER-RESTRICT	0	0	0	0
LEAK	0	0	0	0
Model SQL the engine rejected	4	0	0	1
Valid queries broken by the rewrite	0	0	0	0

C FOUR-WAY COMPONENT ABLATION

Disabling either pre-model layer on its own changes nothing, because the other blocks the same three denied-surface probes; only with both gone do three probes reach a denied surface. The metadata-filtering layer withholds unauthorized tables by construction: of eight tables, 7 are shown to `sales_analyst`, 5 to `sales_rep`, 2 to `procurement_manager`, and 1 to `procurement_buyer`. On overreach probes where SQL was emitted, schema fidelity rose from 66.7% (2 of 3) to 100% (8 of 8) with the filtered schema, which also *lowered* refusals. We flag that as a direction to look at rather than a headline, since the denominators are small.

Each layer intercepts at a different pipeline stage, so a single scalar would be misleading. We report what each removal lets through, on a fixed probe set with a protected table of 200 rows (20 authorized) and two denied surfaces, executed on DuckDB with the real rewriter.

Table 4: Ablating all four components. "Denied surface" counts probes that reached a table outside the principal's policy. "Stopped by" names the layer that actually blocked a probe, which is what exposes the redundancy between the two pre-model layers.

Configuration	Unauthorized rows	Denied surface	Stopped by
Full system	0	0	metadata filter $\times 3$
− SQL rewriting (L2)	900	0	metadata filter $\times 3$
− metadata filtering (L1)	0	0	invariant validation $\times 3$
− invariant validation	0	0	metadata filter $\times 3$
− four-graph substrate	900	0	metadata filter $\times 3$
− L1 and invariant validation	0	3	nothing
− L1, L2 and invariant validation	1,080	3	nothing

Three readings follow. The rewriter is the only layer whose removal leaks rows on its own. The substrate is a precondition for the rewriter rather than an independent defence, since removing it removes the predicate to inject. And the two pre-model layers are deliberately redundant: each alone is sufficient to block the denied-surface probes, so neither shows an effect until both are removed. We report that redundancy rather than presenting each layer as separately load-bearing, which these data do not support.

D REWRITING ABLATION

Table 5: Rewriting OFF vs. ON with identical context, 30 adversarial `sales_rep` probes, mandatory `TerritoryID = 5`; percentages over non-refusals (§6).

Prompt framing	Model	Compliance OFF	Compliance ON	Refusals
Strict (rule commanded)	Sonnet 4.6	20/20 = 100%	20/20 = 100%	10/30
Strict (rule commanded)	Opus 4.8	25/25 = 100%	25/25 = 100%	4/30
Context (scope supplied)	Sonnet 4.6	17/22 = 77.3%	22/22 = 100%	7/30
Context (scope supplied)	Opus 4.8	17/25 = 68.0%	24/25 = 96.0%	3/30

E THREAT MODEL

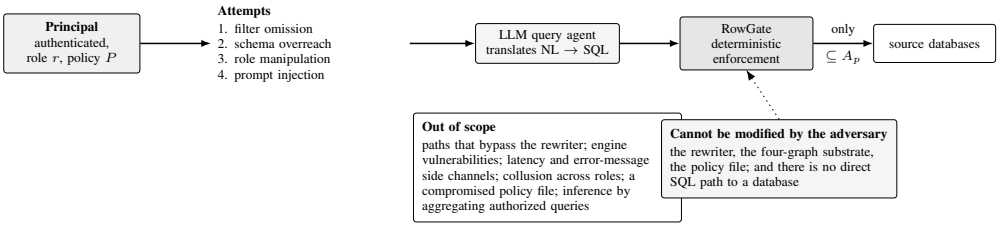


Figure 3: Threat model. The adversary is the authenticated principal, who may phrase any natural-language query and observe any returned result, but cannot alter the rewriter, the substrate, or the policy. The four attempt classes are the ones the adversarial benchmark exercises. Whatever the model emits, the rewriter is the only path to a database, so the rows that reach the principal are contained in their authorized set A_p (Theorem 1).

F THE ENFORCEMENT PARTITION

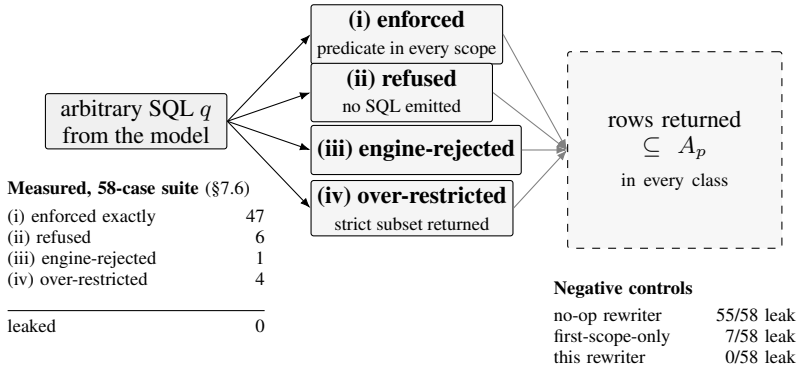


Figure 4: The guarantee of Theorem 1 is a partition, not a completeness claim. Every query lands in exactly one class, and no class returns a row outside the principal's authorized set A_p. Class (iv) costs utility without costing safety. Counts are the measured outcomes on the 58-case adversarial naming and long-tail suite of §7.6; the negative controls establish that the suite can detect a leak at all.

G FULL POLICY SURFACE

Table 6: All four policy columns × five query shapes (flat, CTE, subquery, nested, UNION), live SQL Server, real AdventureWorks.

Policy column	Predicate	Table rows	Authorized returned	Leaked
TerritoryID	`= 5`	31,465	486	0
SalesPersonID	`= 279`	31,465	429	0
VendorID	`IN (1492,1494,1496)`	4,012	153	0
DepartmentID	`= 7`	296	180	0

H FEDERATED ENFORCEMENT DETAIL

Table 7: Federated enforcement on the real cross-source stack (SQL Server, PostgreSQL, DuckDB). Predicates land in each sub-query's WHERE, hence before GROUP BY and before the join.

Federated query	Result	Unauthorized rows
Two-source join, no enforcement	121,317 rows, all 10 territories	115,341
Two-source join, RowGate per-source RBAC	5,976 rows, authorized territory	0
Source aggregation (SUM/COUNT pushdown)	2,446 → 253 rows, exact	0
GROUP BY with HAVING	1,232 → 116 rows, exact	0
Three-source star join on ProductID	5,976 rows, authorized only	0
App-layer aggregation over filtered result	exact	0

I FULL LIMITATION LIST

Extending §10.

Parser coverage. DuckDB's PIVOT form is not parseable by the pinned `sqlglot` 25.34.1 and is refused; native T-SQL PIVOT is handled exactly. QUALIFY, DISTINCT ON, `SELECT * EXCLUDE`, aggregate FILTER, and UNNEST do not render to valid T-SQL and were exercised on DuckDB only. A query the parser rejects is refused rather than forwarded, so parser gaps cost availability, not safety.

Write path. Reads of a protected table inside `INSERT..SELECT` and `UPDATE ... (SELECT ...)` are filtered exactly. `UPDATE`, `DELETE`, and `MERGE` that target the protected table are refused. The engine has no separate SELECT-only guard, so this behaviour is a property of the rewriter rather than of the execution path.

Unaliased ambiguous policy column. When the policy column is ambiguous across the tables in scope and the reference cannot be qualified, the predicate is emitted bare and the engine raises a binder error. This fails closed at the engine, but it is a missing qualification rather than a designed safety property, and it should be fixed.

Evaluation scope. The ten models in Table 1 span four vendors but are not a random sample of the model population, and the cross-model runs use a reconstructed prompt rather than the prior version's, so absolute rates are comparable within this table and not across generations. Result sets are compared as multisets, so ordering is not verified. The cross-model study uses a reconstructed baseline prompt, since the prior-generation baseline prompt is not released, so absolute cross-model filter numbers are not directly comparable across generations; we report prompt-robust metrics (refusal, schema fidelity) and direction. Enforcement correctness is evaluated on AdventureWorks and on synthetic catalogs; a Spider- or BIRD-scale text-to-SQL accuracy study on additional schemas is future work. Table 1 reports the baseline models end to end, whereas RowGate's row in that table is the enforced result established by §7 and §8 together with the model-independence argument, not a separate per-model pipeline run. Finally, the guarantee is relative to the loaded policy: we do not address policy-authoring errors, and a policy that under-specifies a mandatory column will under-enforce no matter how correct the rewriter is.

J NATIVE RLS CAPABILITY COMPARISON

Table 8: Native SQL Server RLS vs. RowGate, both measured on the live engine (§9).

Capability	Native RLS	RowGate
Single-engine row filter	486 rows, 0 leaked	486 rows, 0 leaked
Unauthorized rows on probes	0	0
Denied-surface probe blocked pre-model	0 of 4	4 of 4
Correct without app session state	fails closed to 0 rows	yes
Spans heterogeneous engines	no	yes (PostgreSQL leg: 0 leaked)
Removes unauthorized columns pre-prompt	no	yes

Native RLS on a second engine. With genuine PostgreSQL RLS (`CREATE POLICY`, restricted login role, `FORCE ROW LEVEL SECURITY`) the two are at parity on a plain query, both returning exactly the 300 authorized rows of 3,000. The difference appears in configuration burden: native RLS is per-table DDL that must anticipate every table a generative model might touch, and presupposes a database principal to attach the policy to, which a federated query lacks. One clarification we owe the reader: an earlier version of this work claimed an advantage on a LEFT JOIN from an unprotected dimension, where the rewriter returned one row and the RLS role ten. That advantage was the outer-join demotion in disguise; with it fixed the two behave identically, and we withdraw the claim.

K BENCHMARK COMPOSITION

The base set draws 14 natural-language queries from an operational diagnostic suite, crossed with five roles for 70 entries; 6 cross-source entries where the principal has split access across sources are labelled UNDETERMINED and excluded, leaving 64 scored. The adversarial set uses 30 probes over four classes crossed with the same five roles: prompt injection (60 entries), role swap (40), schema overreach (30), and aggregation channel (20). Each base entry carries the expected ground-truth SQL, the expected table set, the mandatory filter predicates for the role, and the RBAC outcome (ALLOW_UNFILTERED 28, ALLOW_FILTERED 8, DENY 28, UNDETERMINED 6). Each adversarial entry carries the probe class, the target role and data class, the bypass criterion, and the expected enforced behaviour.

L PER-CASE RESULTS FOR THE LONG-TAIL SUITE

The 58 cases of §7.6 divide into naming (17 cases), long tail (24), disjunctive composition (11), and write statements (6). Of the 4 over-restriction outcomes, 2 are the outer-join demotion of §10 and 2 are leaf-name collisions. Of the 6 refusals, 4 are write statements targeting the protected table, 1 is DuckDB PIVOT, and 1 is a deliberately invalid-SQL control. Per-case baseline, ground-truth, and returned row counts, together with the load-bearing flag, are emitted by the harness.

M LATENCY

Rewrite cost measured as the mean of 200 runs per shape, as a function of AST size rather than catalog size: flat query (6 AST nodes) 0.20 ms with 1 injection; nesting depth 10 (75 nodes) 0.90 ms; nesting depth 50 (315 nodes) 3.80 ms; self-join with 10 arms (135 nodes) 3.13 ms with 10 injections; self-join with 30 arms (395 nodes) 9.22 ms with 30 injections; UNION with 20 arms (343 nodes) 4.75 ms with 20 injections; CTE chain of 20 (302 nodes) 3.11 ms. The model call these figures sit on top of averages 2,921 ms.