Technical report v1.1 2 November 2026 36 min read

Catalogue-Driven Row-Level Security Assurance and Commit-Time Audit-Chain Allocation in a Multi-Tenant PostgreSQL SaaS: A Single-System Case Study

Yousof Selim
Sunway University (Lancaster University dual degree), Malaysia

119× lower median wait for same-customer writes behind slow requests

1. Allocating the audit-chain position at commit cut the median same-tenant wait behind a 5-second transaction from 4,919 ms to 41.3 ms (119-fold; 95% interval 81 to 152).
2. The chain failed verification in all 300 concurrent trials before the fix, 100 of them with true forks, and in none of 600 after it; deleting the newest row or recomputing later hashes still goes undetected.
3. The tests' original tenant scope would now ignore 10 tables; widened tests caught 6 of 8 seeded mistakes and a scope-free check caught the other 2.
4. At 1,122 events, RLS reads took a median 856 to 1,656 times as long as an explicit filter; in a microbenchmark, calling the helper once per statement cut 3,369 ms to 127 ms.

Limits. One pre-launch system on one heavily loaded laptop with synthetic data; the 22,022-event run is thin, and the proposed policy rewrites are untested in the product.

Abstract

The problem When every customer shares one database, a single forgotten protection can expose one customer's data to another, and requests that run at the same time can quietly break an audit log. WayClub's own documents had once claimed protections the database did not have.

What I did	I rebuilt WayClub's database at several points in its history and ran four local experiments: seeded security mistakes against its catalogue-reading tests, concurrent writers against its audit chain, timed audited writes before and after a lock fix, and row-level security against plain filters.
What I found	**The fix cut a same-customer write's median wait behind a slow request from 4,919 ms to 41.3 ms, and the chain went from failing all 300 concurrent trials to failing none of 600. The tests' original scope would now ignore 10 tenant tables. Row-level security made reads hundreds of times slower at 1,122 events.**
Why it matters	Security claims about a database should be queries that can fail, and concurrency claims need concurrent tests. The cost measurements point to a fix before the first large customer arrives, though it is still untested in the product and every number comes from one loaded laptop.

Contributions

- A mutation analysis of five catalogue-driven RLS assertions under old and new tenant-scope predicates, beside a scope-free release check, over eight seeded migration mistakes.
- A rebuilt-from-history measurement of scope drift: documented counts reproduced exactly at migration 022, and the old predicate's blind spot traced from 0 to 6 to 10 tables.
- Concurrency evidence that commit-time chain allocation removed every observed verification failure (0 of 600 trials, against 300 of 300 before) and cut the median same-tenant wait 119-fold.
- A plan-level explanation and microbenchmark of the per-row cost of definer-helper RLS policies, with once-per-statement formulations not yet applied to the product.

Keywords. PostgreSQL · row-level security · multi-tenancy · SECURITY DEFINER · catalogue assertions · mutation analysis · tamper-evident logging · hash chain · concurrency · advisory locks · deferred constraint triggers

1 Introduction

When a shared-schema PostgreSQL product moved its audit-chain lock from the first audited write to commit time, the median wait of a same-tenant write queued behind a 5-second transaction fell from 4,919 ms to 41.3 ms. Its hash-chained audit log, which had failed verification in all 300 concurrent trials before the change, failed in none of 600 after it. This report presents that result and three others from the same system, and states how far each one reaches.

A multi-tenant product that keeps every customer in one shared schema relies on one predicate to keep customers' rows apart[1]. If that predicate lives in application code, one forgotten `WHERE` clause exposes another tenant's data. PostgreSQL row-level security (RLS) moves the predicate into the database: a table with RLS enabled and no applicable policy returns no rows[2]. The risk does not disappear. It changes shape: a table created without RLS, a policy keyed on the wrong column, a helper function that runs with more privilege than its caller, or an integrity mechanism that fails only under concurrency.

The system studied is WayClub, a group-management product for clubs, societies and communities that the author is building. It is pre-launch. There is no live site, no real tenant and no production load, and its cloud database was stopped throughout the study. That limits what can be claimed. It also means every schema state in the project's history can be rebuilt from its migrations and examined locally at no cost.

Two episodes motivate the study. First, the project's security documents once said RLS was "enabled and forced" and that a test asserted coverage. Neither was true. The response was five tests that query PostgreSQL's catalogue directly, and one of them later proved to have a blind spot: its tenant-scope predicate looked only for `institution_id`, so it ignored tables added after `account_id` became the tenant root. Second, the audit log's hash chain forked under concurrency, and its tenant lock made every audited write in a tenant wait for the slowest open request. A serial test suite reported the chain correct throughout.

Research questions

- **RQ1 (catalogue assurance).** Do the catalogue assertions describe the schema accurately across its history, and which plausible migration mistakes do they catch?
- **RQ2 (chain integrity).** Does allocating the chain position at commit prevent forks under concurrent writers, and which kinds of tampering does the chain detect?

- **RQ3 (availability).** How long does a same-tenant audited write wait behind a slow transaction, before and after the fix?
- **RQ4 (cost).** What does the RLS design cost compared with the explicit filter it replaces, and which part of the policies causes the cost?

Each question maps to one experiment in the artifact (Table 1, Section 5).

Contributions

- **A mutation analysis of catalogue-driven RLS assertions.** The five assertions, ported to SQL, are run against eight seeded migration mistakes under the old and new tenant-scope predicates, beside a scope-free release check. The new predicate catches the six mutants inside a tenant scope. The scope-free check catches the two that no scope predicate selects. Neither family catches all eight on its own (Section 6.1).
- **A measured history of scope drift.** Rebuilding the schema at three commits reproduces the documented counts exactly at migration 022. It also shows the old predicate's blind spot growing from 0 to 6 to 10 tables, every one of which nonetheless had RLS enabled (Section 6.1).
- **Concurrency evidence for commit-time chain allocation.** A deferred constraint trigger that allocates the chain position inside the lock removed every verification failure seen before the fix, and cut the median same-tenant wait 119-fold. Two kinds of tampering remain undetected by design (Sections 6.2 and 6.3).
- **A located cost of helper-heavy RLS policies.** At 1,122 events, RLS reads took a median 856 to 1,656 times as long as an explicit filter. Query plans place the cost in helper calls evaluated once per row. A microbenchmark shows that calling the helper once per statement removes most of it, but the rewrite has not been applied to the product (Sections 6.4 and 7.1).

---

2 Background and related work

2.1 Multi-tenant data architectures

Chong, Carraro and Wolter describe a continuum from a separate database per tenant, through a separate schema per tenant, to a shared schema in which every row carries a tenant identifier[1]. The shared schema is the cheapest to run and the easiest to scale to many small tenants, but its isolation is only as good as the code that filters by tenant. WayClub sits at the shared end. Its tenants are many small clubs and a few institutions, and per-tenant databases would cost more than a pre-launch product can carry.

2.2 PostgreSQL row-level security

PostgreSQL RLS attaches policies to a table. A policy's `USING` expression filters the rows a query can see, its `WITH CHECK` expression constrains the rows it can write, and permissive policies combine by OR[2]. Enabling RLS on a table with no policy denies all access. Three exemptions matter here. Superusers and roles with `BYPASSRLS` skip policies. The table owner skips them too, unless the table is marked `FORCE ROW LEVEL SECURITY`. And a `SECURITY DEFINER` function runs with its owner's privileges, so it sees whatever RLS lets the owner see.

PostgreSQL's guidance for definer functions is to pin `search_path` with a `SET` clause[3]. A `SET` clause, and the definer property on its own, each stop the planner inlining a SQL function into the calling query[4]. Vendor guidance for RLS-heavy platforms gives two remedies for per-row function cost: wrap a row-independent call in a scalar sub-select so that it runs once per statement, and use definer functions to look up membership without paying for RLS on the joined table[5]. Section 6.4 measures both.

2.3 Checking RLS configuration

Two families of tools check RLS configuration. Unit-test frameworks such as pgTAP assert the exact set of policies, roles and commands on a named table[6]. That is precise, but the expected values are a hand-kept list that has to be updated with every migration. Schema linters run catalogue queries over a whole schema. Supabase's linter, for example, flags tables in an exposed schema with RLS disabled, tables with RLS enabled but no policy, policies on tables without RLS, and policies that call authentication functions without a sub-select wrapper[7]. WayClub's assertions sit between the two. They are catalogue queries, so they need no list, but they apply only to the tables a tenant-scope predicate selects, and that predicate is where they went wrong.

2.4 Tamper-evident logging

Schneier and Kelsey showed how to make log entries written before a machine's compromise impossible for the attacker to read, or to alter or delete without detection, using hash chains and evolving keys[8]. Crosby and Wallach observed that auditing a hash chain costs linear time, and proposed a Merkle-tree history structure with logarithmic membership and consistency proofs, an untrusted logger, and auditors who hold commitments[9]. Certificate Transparency put that design into production for public certificate logs[10]. WayClub's chain is much more modest: a per-tenant SHA-256 hash chain, unkeyed, inside the same database it protects, with no external commitment. Section 6.2 measures what that does and does not detect.

2.5 Concurrency anomalies in database-backed applications

Berenson et al. showed that the ANSI isolation levels miss several anomalies that real implementations allow, and characterised snapshot isolation[11]. Fekete et al. showed when applications under snapshot isolation remain serialisable[12]. Cahill, Röhm and Fekete made snapshot isolation serialisable at run time[13], and PostgreSQL adopted a version of that method for its `SERIALIZABLE` level[14]. Warszawski and Bailis found 22 concurrency attacks in 12 popular e-commerce applications, arising from application logic that is correct only when its transactions do not overlap[15]. The audit-chain fork in Section 4.4 has the same shape on a smaller scale. A sequence number was allocated outside the lock that was meant to order it, the defect appeared only with two open transactions, and a serial test suite reported the property as holding. The fix uses two PostgreSQL primitives: advisory locks[16] and deferred constraint triggers[17].

2.6 Mutation analysis of tests

Mutation testing judges a test suite by seeding faults and counting how many it detects[18]. Section 6.1 applies the idea to schema-level security assertions. Each mutant is a plausible migration mistake, and a good assertion set should fail on it.

2.7 Positioning

None of the techniques in this report is new. RLS, definer helpers, hash chains, advisory locks and deferred triggers are documented PostgreSQL features, and the sub-select rewrite is published vendor guidance. What the report adds is evidence from one real schema: how a scope predicate drifted when the tenancy model changed, how scoped and scope-free checks complement each other under mutation, how a lock-scope defect and its fix behave under genuine concurrency, and how large the per-row helper cost is at modest data sizes. It is a single-system case study, and its findings should be read as such.

---

3 System and threat model

System. WayClub is a modular monolith: a NestJS API, a Next.js web client and a Flutter mobile client over one PostgreSQL 17 database. At the studied commit (`65a18ff`, 1 November 2026) the database has 68 migration files. The tenant root is the account, which is either an institution that governs many clubs or a standalone club. Every tenant-scoped row carries

`account_id`. Institution-governed rows also carry `institution_id`, tied to the row's own account by a composite foreign key.

Three database roles exist. `wayclub_admin` owns every object and runs migrations and seeds. `wayclub_app` serves every request, owns nothing and lacks `BYPASSRLS`. `wayclub_service` has `BYPASSRLS` and is reserved for narrow scheduled jobs. The API sets the caller's identity for each transaction with `set_config('request.jwt.claims', $1, true)`, and policies read it through helper functions.

Assets. Tenant data (members, events, registrations, money records, handovers), and the audit log, which records approvals, role grants and state changes and has to serve as evidence in a dispute between club officers.

Adversaries in scope. (A1) An authenticated user of one tenant who controls their own requests, including a forged or replayed token for their own identity, and tries to read or write another tenant's rows. (A2) A developer, human or AI assistant, who writes a migration that adds a table or policy and forgets a protection. The catalogue assertions target this adversary. (A3) Concurrent honest traffic. It is not an adversary, but it has broken integrity properties that serial tests reported as holding. (A4) An attacker with database-owner or superuser access who edits the audit log afterwards.

Out of scope. A4 is in scope for detection only: the chain cannot stop an owner rewriting it, and Section 6.2 shows two cases it cannot even detect. Also out of scope are compromise of the API container (which, as the project's security document records, can read the service-role credential), SQL injection (the codebase uses parameterised queries; this report does not test that), side channels, and denial of service beyond the lock-scope effect in Section 6.3.

---

4 Design

4.1 RLS enabled, deliberately not forced

The original architecture decision called for every tenant table to have RLS enabled and forced. Forcing was never implemented, and when the gap was found, turning it on broke the system. `FORCE` changes one thing: whether policies also bind the table owner. Here the owner also owns the `SECURITY DEFINER` helpers that the policies call (`app.is_member_of`, `app.current_account_id` and about thirty more), and those helpers are owner-owned precisely so that evaluating a policy does not recurse into RLS.

We repeated the project's measurement on a seeded copy of today's schema, inside a transaction that was then rolled back. With RLS forced on `public.membership`,

`app.is_member_of()` returned `false` for a genuine member, where it had returned `true`, and the owner's `count(*)` over membership fell from 16 rows to 0.

The adopted design puts the trust boundary in role separation instead. The request role owns nothing and has no `BYPASSRLS`, so RLS binds it unconditionally. Rules that must bind the owner too are written as constraints and triggers, not policies: the composite tenant foreign key, the append-only audit trigger, the `unique (account_id, chain_index)` backstop and several immutability triggers. A catalogue test asserts that no table is forced, and its comment records the measurement, so that nobody reverses the decision without seeing the reason.

4.2 Caller-validated SECURITY DEFINER functions

Definer functions are the audited exceptions to the boundary. They run as the owner, so RLS cannot protect anything a definer writes. An adversarial review found two that authorised on less than they should: `app.write_audit` accepted a forged cross-tenant audit row, and `app.promote_waitlist_head` authorised on tenancy alone. Migrations 012 to 015 made each definer check its caller's claims itself. The request-role audit writer now pins account, institution and actor to the caller's own claims. The unvalidated writer is granted to no role and is reachable only from owner-definer trigger functions. Both regressions are covered by passing tests in the project's database suite.

At head there are 249 `SECURITY DEFINER` functions in the application schemas, all with a pinned `search_path`, and 210 of the 225 public-schema policies call at least one `app.*` helper. That ratio matters in Section 6.4.

4.3 Catalogue assertions

The five assertions in the project's `RLS coverage (mechanical)` test block read `pg_class`, `pg_policy` and `has_table_privilege` instead of a hand-kept list. They first compute a tenant scope, the set of public tables that name a tenant, and then check:

1. **A1**: every in-scope table has `relrowsecurity` set;
2. **A2**: no in-scope table has `relforcerowsecurity` set (Section 4.1);
3. **A3**: no in-scope table is granted to the request role without a policy, with a positive control that more than ten "breadth" tables are both ungranted and without policies;
4. **A4**: the request role has only a SELECT policy on `audit_event`;
5. **A5**: every request-role policy outside the two tenant-root tables mentions `current_account_id`.

The tenant-scope predicate is the weak point. It was first written as "the `institution` table plus every table with an `institution_id` column" (commit `85475d7`). Migration 017 then made `account_id` the tenant root. Tables added after it, including entitlements, club affiliation, institution claims and club migration, carry `account_id` and no `institution_id`, so the predicate silently stopped selecting them. Commit `5096bcd` widened it to "`institution` or `account`, or any table with `institution_id` or `account_id`".

The deployment pipeline also runs a release-time role check. Among other things, it fails on any privilege the request role holds on any relation without RLS, in any non-system schema. That rule needs no tenant predicate. We call it **R1** and include it because it covers exactly the cases the scoped assertions cannot.

4.4 Per-account audit chain with commit-time allocation

`audit_event` is append-only: a `BEFORE UPDATE OR DELETE` trigger rejects changes from every role, owner included. Each row's hash is SHA-256 over the previous row's hash and the row's own fields. Migration 006 computed the hash in a `BEFORE INSERT` trigger, which took `pg_advisory_xact_lock` on the tenant, read the latest row by `seq` (a `generated always as identity` column) and chained onto it. This had two defects.

Integrity. `seq` is allocated when the tuple is formed, before the trigger runs and so before the lock is taken, so allocation order and lock order can differ. Suppose transaction T1 writes a row and holds the lock. T2 allocates `seq` = n and blocks. T1 writes a second row with `seq` = n+1 and commits. T2 then chains its lower-`seq` row onto T1's higher-`seq` row, and a verifier that walks `seq` order sees two broken links. With several rows per transaction it gets worse: a transaction's second row looks up "the latest row by `seq`", which can be another transaction's committed row rather than its own first row, so two rows end up with the same predecessor. That is a true fork.

Availability. An advisory transaction lock lasts until commit. The lock was taken at the first audited write, and API transactions span whole request handlers, so one slow request blocked every audited write in its tenant.

Migration 011 fixed both. A new `chain_index` column carries the authoritative position. Chaining moved into a `DEFERRABLE INITIALLY DEFERRED` constraint trigger, which fires during `COMMIT`. That trigger takes the lock, reads the tenant's highest `chain_index`, and writes `chain_index`, `hash_prev` and `hash` onto the new row in one narrowly exempted update. Chain order and lock order are now the same by construction, and the critical section is the commit, not the rest of the handler. A `unique (account_id, chain_index)` index is the structural backstop: if serialisation were ever defeated, two rows would collide and the commit would fail loudly instead of forking silently.

Migration 019 re-keyed the chain from institution to account, so that standalone clubs, which have no institution, get a chain too. Before re-deriving hashes under the new formula it verified every existing chain under the old one, because re-hashing an unverified chain would launder a forgery. One trade-off remains. A session that runs `SET CONSTRAINTS ALL IMMEDIATE` makes the trigger fire at each insert again, which brings the availability problem back. Integrity is unaffected, because the position is still allocated inside the lock.

---

5 Method

5.1 Experiments and research questions

Table 1 maps each research question to the experiment that answers it. Experiment numbers follow the artifact's file names (`exp1_*` to `exp4_*`); results are reported in research-question order.

Table 1. Research questions, experiments and repetitions. "Seed", "x50" and "x1000" are the seeded head schema with its 22 events copied 0, 50 and 1,000 times (22, 1,122 and 22,022 events).

RQ	Experiment	What is measured	Schemas	Repetitions	Results
RQ1	E3	Catalogue counts; tables selected by the old and new scope predicates; verdicts of A1-A5 and R1 on 8 mutants	Rebuilt at migrations 016 and 022 and at head; seeded head for mutants	One run (catalogue queries are deterministic); 4 states cross-checked through the project's own test runner	Section 6.1
RQ2	E4	Chains failing verification, forks and writer errors; five tampering cases	Before fix, after fix, head; public minimal reproduction	200 interleaving and 100 stress trials per schema; one run per tampering case	Section 6.2
RQ3	E1	Time for a same-tenant and an other-tenant writer to begin, insert and commit behind a 5-second holder	Before fix, after fix, head, head with immediate constraints; public minimal reproduction	30 runs per schema	Section 6.3
RQ4	E2	Server execution time of two reads under RLS and with an explicit filter; pgbench throughput; helper microbenchmark	Seed, x50, x1000	16 personas × 5 at seed and x50; 4 personas × 1 at x1000; pgbench 5 × 20 s per mode; microbenchmark 5 runs	Section 6.4

5.2 Environment

All experiments ran on one Apple M1 Pro laptop (8 cores, 16 GB RAM, macOS 27.0.1) against a dedicated PostgreSQL 17.10 cluster created for the study: port 55499, `fsync` and `synchronous_commit` on, `shared_buffers` 512 MB, other settings at their defaults. Clients were Node.js 26 with `pg` 8.22.0, and pgbench 17.10. No cloud resource was touched. The runs took place on 1 and 2 November 2026 (16:44 to 17:53 UTC on 1 November for the timed experiments).

5.3 Schemas and data

Schemas were rebuilt from the private repository's migrations exactly as they stood at three commits: `85475d7` (migrations up to 010 for the "before" schema and up to 011 for the "after" schema, and up to 016 for the scope history), `940b2ae` (up to 022) and `65a18ff` (head, 68 migration files). The head schema was seeded with the project's deterministic seed: 32 users, 22 events and four accounts. For E2, the seed's events were copied 50 and 1,000 times, keeping each copy's club, tenancy, status and guest policy, and each of the seed's 16 active memberships became a caller identity ("persona"). The audit-chain experiments (E1 and E4) were also run against a public minimal reproduction made of verbatim excerpts of migrations 006 and 011 over a stub table, which needs none of the private source.

5.4 Procedures

E3, catalogue and mutation. The catalogue counting queries and both scope predicates were run on each rebuilt schema. Each of the eight mutants was applied inside a transaction on the seeded head schema, the five assertions (under both predicates) and R1 were evaluated, and the transaction was rolled back. Only violations that are new relative to the unmutated baseline count. The SQL port was checked against the project's own vitest block on the baseline and on mutants M1, M2 and M7.

E4, chain integrity. The "interleave" workload replays the project's regression interleaving (T1 writes, T2 writes and blocks, T1 writes again, T1 commits, T2 commits). The "stress" workload runs 8 concurrent writers that each insert 5 rows with random 0-15 ms pauses. Each trial uses a fresh, empty tenant chain and is then verified by recomputing every hash in the order that version's own verifier walks (`seq` before the fix, `chain_index` after). A verification failure is any recomputed hash or link that does not match; a fork is two rows that share a predecessor hash. For tampering, a superuser disabled triggers for one transaction (`session_replication_role = replica`) to stand in for an owner-level attacker, and edited a fresh 10-row chain.

E1, lock scope. A holder transaction writes one audit row for tenant A and stays open for 5 s. After 100 ms, one writer for tenant A and one for tenant B each run begin, insert, commit, and are timed from the client.

E2, RLS cost. Two reads that the product issues (a club's latest 20 events, and the latest 50 events open for discovery) were run under RLS as the request role with a persona's claims, and as the owner with the explicit filter `account_id = $1` that RLS replaces. Execution time is PostgreSQL's server-side figure from `EXPLAIN (ANALYZE, TIMING OFF)`. pgbench ran a mixed transaction (set claims, both reads, save and unsave an event) in three modes, interleaved. The helper microbenchmark counted persona 1's member

events among one account's 14,014 events at x1000, as superuser with that persona's claims, using seven formulations of the membership test and of a row-independent role check. Plan shapes for both reads and for the role check were archived with `EXPLAIN` on x50.

5.5 Statistics

For timings we report the median and the 95th percentile (linear interpolation) over runs, and the minimum as the least-contended observation. The headline ratio is the ratio of the two 30-run medians, with a percentile-bootstrap 95% interval from 10,000 resamples of each sample (random seed 1). For RLS reads, "ratio" is the median of per-measurement ratios, where each measurement pairs one RLS and one filter execution for the same persona. pgbench figures are the mean over 5 runs with a 95% Student-t half-width. For chain trials we report counts. Zero failures in 300 trials bounds the per-trial failure probability below 1% at 95% confidence (the rule of three), and zero in 600 below 0.5%; this says nothing about failure modes the workloads do not exercise.

5.6 Machine load and deviations from plan

The laptop was heavily loaded by unrelated work throughout. Load logging began at 17:13 UTC, after E1 and E4 had finished; at that moment the 15-minute load average was 106 on 8 cores, so E1 and E4 also ran under heavy load. During E2 and the minimal reproduction the one-minute load average had a median of 84 (range 56-99, 76 samples). Lock-wait results are dominated by the deliberate 5-second hold, but absolute latencies are inflated and are not comparable across schemas run at different times. Execution-time results are inflated and noisy, so we report minima next to medians and draw conclusions only from differences of two orders of magnitude or more.

Four deviations are recorded in the artifact. First, the planned all-persona run at x1000 was stopped after two measurements, because one discovery read took 144 s under RLS on the loaded machine. The reported x1000 figures come from a reduced run: one persona per account (four personas), one repetition each, so four measurements per read. Second, the helper microbenchmark was run three times. The first run failed on a type error, and the second used a bare row-independent condition, which PostgreSQL evaluates once as a one-time filter and which therefore did not represent a policy's OR. Only the third is reported. Third, the project's whole database test directory could not run in parallel on this cluster, because its files reset a shared database, so only the file containing the assertions was run. Fourth, the test-runner mutation runs left one mutant table in the seeded template; it was dropped from every derived database before E1, E2 and E4 used them, and all catalogue counts come from a schema built fresh.

5.7 Replication

E1 and E4 can be reproduced from the public artifact alone, using the minimal excerpts and the commands in its README. E2 and E3 need the private 68-migration schema and seed. Their drivers, SQL and raw outputs are public, so the method can be inspected and re-applied to another schema.

---

6 Results

6.1 RQ1: catalogue assurance

Counts. Table 2 shows the documented figures reproduced exactly when the schema is rebuilt at migration 022: 78 tables with RLS, 126 policies across 47 tables, 48 tables reachable by the request role and 31 deny-all tables. (The rebuilt public schema has 80 tables; the extra one is the migration runner's own ledger, which has neither RLS nor a grant and is excluded throughout.) At head the schema is larger: 102 of 103 tables have RLS, with 225 policies across 75 tables. No table is forced. Only `role`, a global read-only catalogue of role definitions, lacks RLS. The documented counts are therefore correct for the date they give and out of date today.

Table 2. Catalogue counts on rebuilt schemas (public schema; the migration runner's ledger table excluded).

Quantity	Documented (3 August 2026, migration 022)	Measured at migration 022	Measured at head (65a18ff)
Tables	79	79	103
Tables with RLS enabled	78	78	102
Tables without RLS	1 (`role`)	1 (`role`)	1 (`role`)
Tables with RLS forced	0	0	0
Policies	126 across 47 tables	126 across 47 tables	225 across 75 tables (212 for the request role, 13 for the service role)
Tables reachable by `wayclub_app`	48	48	75
Deny-all tables (RLS on, no policy, no grant)	31	31	27

Scope drift. Figure 1 applies the old and new tenant-scope predicates to each rebuilt schema. At migration 016, before account tenancy, both select the same 57 tables. At migration 022 the old predicate misses 6: `account`, `account_entitlement`, `club_affiliation`, `club_migration`, `club_migration_mapping` and `institution_claim`. At head it misses 10, adding `club_activity`, `event_save`, `notification_email_outbox` and `person_avatar_hidden`. Every table in every blind spot does have RLS enabled, so the gap was latent: no table escaped protection, only the assertions' attention. One statement in the project's security document needs correcting. The continuity-domain tables (handover, external tool and others) carry `institution_id` as well as `account_id`, so they were never in the blind spot.

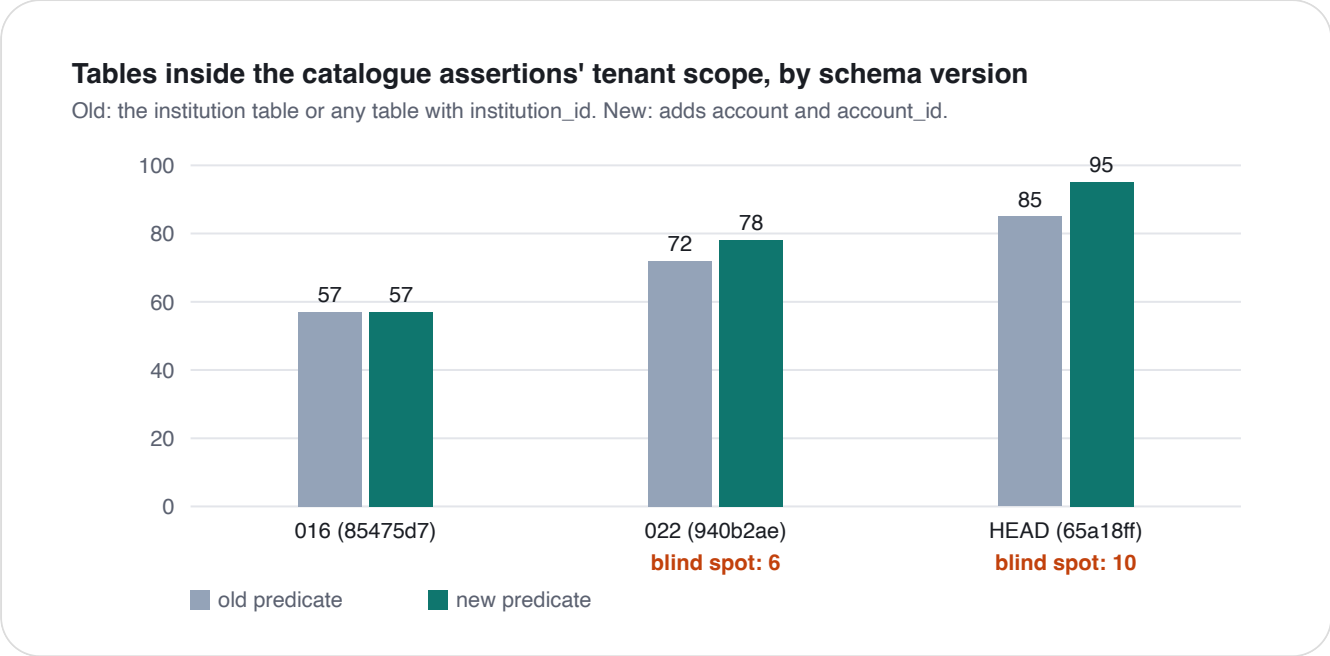


Figure 1. Tables in the tenant scope under the old predicate (the `institution` table or any table with `institution_id`) and the new one (which adds `account` and `account_id`), by schema version.

The project's test file is red at head. Run on its own at the studied commit, the file that holds the assertions passes 105 tests and fails 2. Assertion A5 fails on four policies added by migrations 066 and 067 (on `legal_version`, `reserved_handle`, `person_handle` and `person_avatar`), none of which mentions `current_account_id`. Two of these tables hold global reference data, and the other two are person-scoped and reach the account boundary only through a helper. An unrelated schema-guard test fails on a new column. Whether each A5 exemption is safe is a design question, and the assertion did its job by forcing it to be asked. The developer's in-progress work, since saved on the branch `wip/paused-2026-11-02`, adds an explicit allow-list for exactly these four policies; this study did not use it.

Mutation analysis. Table 3 lists eight mutants, each a mistake a migration author could plausibly make. Under the new predicate the five assertions catch all six mutants that fall inside a tenant scope. Under the old predicate they miss M1 and M2, the two whose only tenant column is `account_id`: the migration-017 gap made concrete. Neither predicate catches M7, a person-scoped table with only `user_id`, or M8, a child table whose tenancy is derivable only through a foreign key. Both are granted to the request role without RLS, which is the worst case, and only the scope-free rule R1 catches them. R1 in turn misses M2, M4, M5 and M6, which need the scoped assertions. Neither family catches everything alone; together they catch all eight. The SQL port's verdicts match the project's own test runner on all four states checked through it (the baseline, M1, M2 and M7).

Table 3. Mutation analysis on the seeded head schema. A verdict counts only violations that are new relative to the unmutated baseline. "Missed" means no new violation; because A5 already fails at head, a mutant caught only by A5 (M6) would add one more entry to a test that was already red.

Mutant	Seeded mistake	A1-A5, new predicate	A1-A5, old predicate	R1 (scope-free release check)
M1	New table with only `account_id`, no RLS, SELECT granted to the request role	caught: A1, A3	missed	caught
M2	New `account_id` table, RLS on, granted, no policy	caught: A3	missed	missed
M3	New table with `institution_id`, no RLS, granted	caught: A1, A3	caught: A1, A3	caught
M4	`FORCE ROW LEVEL SECURITY` on `event`	caught: A2	caught: A2	missed
M5	An INSERT policy for the request role on `audit_event`	caught: A4	caught: A4	missed
M6	A `USING (true)` SELECT policy on `event`	caught: A5	caught: A5	missed
M7	New person-scoped table (`user_id` only), no RLS, granted	missed	missed	caught
M8	New child table (tenancy only through a foreign key to `event`), no RLS, granted	missed	missed	caught

6.2 RQ2: chain integrity and tamper evidence

Table 4 shows that before the fix, all 200 replays of the regression interleaving produced a chain that failed verification in `seq` order, each with 2 broken links in a 3-row chain. In those replays the hash links still formed a single path that simply did not follow `seq` order. The stress workload is worse: all 100 trials produced true forks, in which two rows share a predecessor (median 9 broken links per 40-row chain, range 8-10), so no ordering of the rows can make the chain verify. After the fix, none of 600 trials across the after-fix and head schemas failed verification, had a fork or raised a writer error, and the public

minimal reproduction behaved the same way. The `unique (account_id, chain_index)` backstop never fired, which is consistent with the lock serialising every allocation. A test that runs one transaction at a time cannot observe any of this; the original suite was serial and green while the property was false.

Table 4. Chain integrity under concurrency. Each trial uses a fresh, empty tenant chain and is verified in the order that version's own verifier walks.

Schema and workload	Trials	Chains failing verification	Trials with a fork	Trials with writer errors
Before fix, interleave	200	200	0	0
Before fix, stress	100	100	100	0
After fix, interleave	200	0	0	0
After fix, stress	100	0	0	0
Head, interleave	200	0	0	0
Head, stress	100	0	0	0
Minimal reproduction before, interleave	200	200	0	0
Minimal reproduction before, stress	100	100	100	0
Minimal reproduction after, interleave	200	0	0	0
Minimal reproduction after, stress	100	0	0	0

Table 5 records what the chain detects. An ordinary owner-role update is refused outright by the append-only trigger. With that trigger disabled by a superuser, an edited interior row and a deleted interior row are both detected. Two attacks are not: deleting the newest row, and editing a row and then recomputing every later hash. Both follow directly from an unkeyed chain held in the same database it protects, so the chain is tamper-evident only against an attacker who does not recompute it. Closing the gap needs a commitment held outside the attacker's reach, such as a periodically published chain head, a keyed MAC with an externally held key, or a Merkle-tree log with external

auditors[8][9][10]. The project's documents list externally anchored checkpoints as planned, not built.

Table 5. Tampering cases. Each edit case ran once per schema on a fresh 10-row chain, with triggers disabled by a superuser for one transaction. Results were identical on the before-fix, after-fix and head schemas and on the minimal reproduction, except where noted.

Case	Detected?	How
Owner-role `UPDATE` of an audit row	Refused	Append-only trigger raises an error; no row changes
Edit one interior row's details	Yes	1 broken link
Delete one interior row	Yes	1 broken link, plus a gap in `chain_index` after the fix
Delete the newest row (truncation)	No	The shorter chain verifies
Edit a row and recompute every later hash	No	The rewritten chain verifies

6.3 RQ3: lock scope and availability

Table 6 and Figure 2 show the result. Before the fix, the same-tenant writer waits out the holder: median 4,919 ms, against 21.8 ms for the other tenant. This agrees with the project's original single measurement of 4,780 and 4,792 ms against 43 ms. After the fix the two writers are indistinguishable (medians 41.3 ms and 42.5 ms). The ratio of same-tenant medians is 119 (bootstrap 95% interval 81-152). The separation does not depend on the choice of statistic: the fastest of the 30 before-fix runs (4,906 ms) is 26 times slower than the slowest after-fix run (190 ms).

The head schema behaves the same way (medians 105 ms and 109 ms). Its absolute times are higher, plausibly because the head schema runs more triggers per insert and the machine's load varied between runs; we do not interpret differences of this size. With `SET CONSTRAINTS ALL IMMEDIATE` the old behaviour returns (median 4,972 ms), which confirms that firing at commit is what restores availability. The public minimal reproduction shows the same pattern: 4,968 ms before the fix, and 150 ms against 142 ms after it.

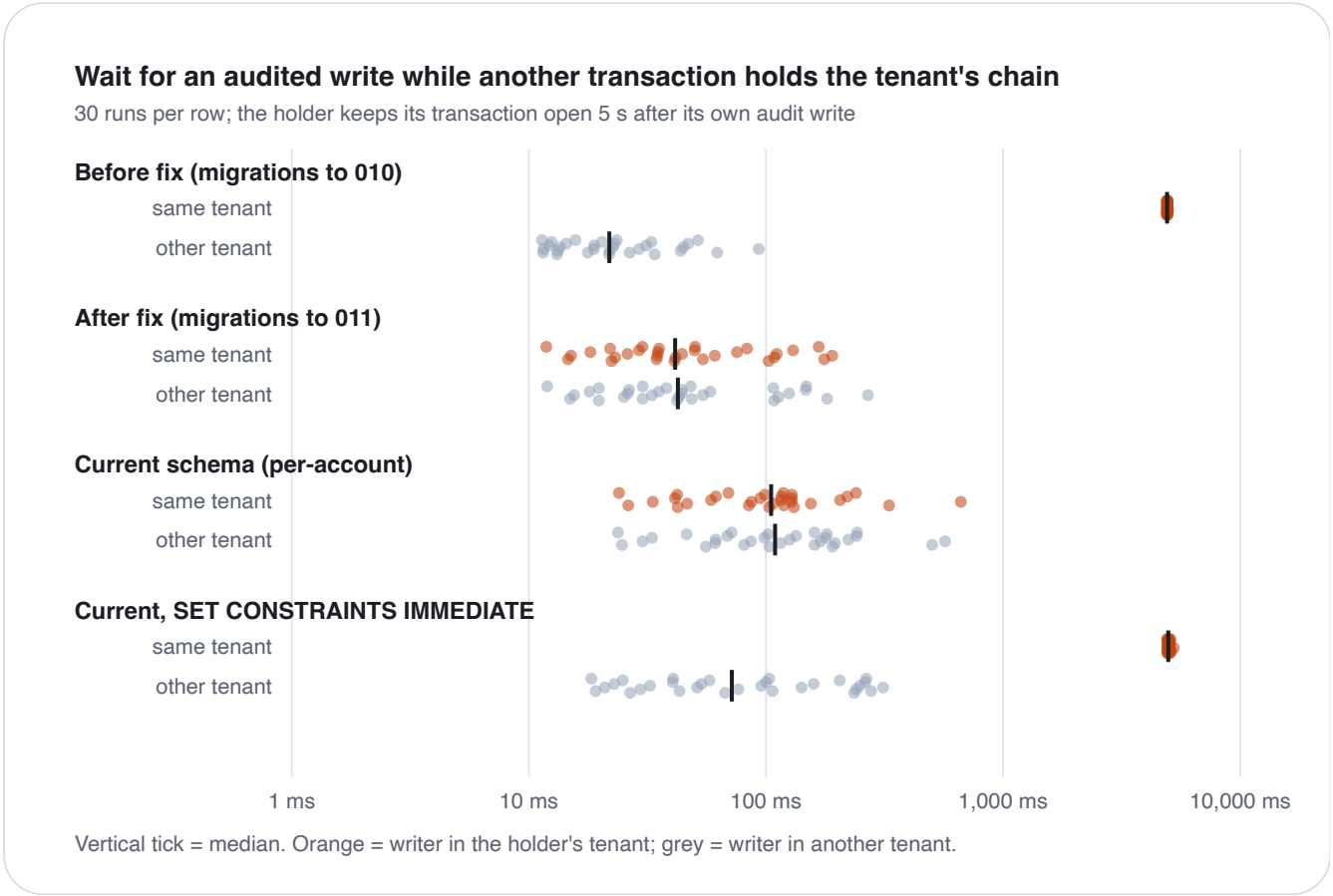


Figure 2. Time for a same-tenant (orange) and an other-tenant (grey) audited write to begin, insert and commit while a third transaction holds the tenant's chain open for 5 s. 30 runs per schema, log scale; ticks mark medians.

Table 6. Time to begin, insert and commit an audited write while a third transaction holds the same tenant's chain open for 5 s after its own audit write. 30 runs per schema. Absolute times are inflated by machine load and are not comparable across rows.

Schema	Same tenant p50	Same tenant p95	Other tenant p50	Other tenant p95
Before fix (migrations 001-010)	4,919 ms	4,945 ms	21.8 ms	57.5 ms
After fix (001-011)	41.3 ms	172 ms	42.5 ms	166 ms
Head, per-account chain	105 ms	290 ms	109 ms	385 ms
Head, with `SET CONSTRAINTS ALL IMMEDIATE`	4,972 ms	5,104 ms	71.8 ms	272 ms
Minimal reproduction, before (excerpt of 006)	4,968 ms	5,230 ms	51.7 ms	241 ms
Minimal reproduction, after (excerpts of 006 and 011)	150 ms	332 ms	142 ms	284 ms

6.4 RQ4: what RLS costs, and why

Reads. Table 7 and Figure 3 compare the two reads under RLS and with the explicit filter. Both modes returned the same number of rows in every measurement. On the 22-event seed the uncontended difference is small: the minima are 0.081 ms under RLS and 0.021 ms with the filter. The seed's median ratio of 84 is mostly load noise acting on very short queries, and we do not rely on it. The cost grows with the number of rows the policy must inspect. At x50, with 80 measurements per read, the median ratios are 856 for the club read and 1,656 for the discovery read, and even the minima differ by factors of 222 and 124. At x1000 the medians are 10.8 s and 5.4 s under RLS against 10.6 ms and 65.8 ms with the filter (ratios 524 and 372). Those figures rest on four measurements per read, whose individual ratios range from 345 to 2,128 (club) and from 56 to 593 (discovery), so they are indicative only. The ratio does not grow monotonically with scale, because the explicit filter's own cost also grows with the account's row count.

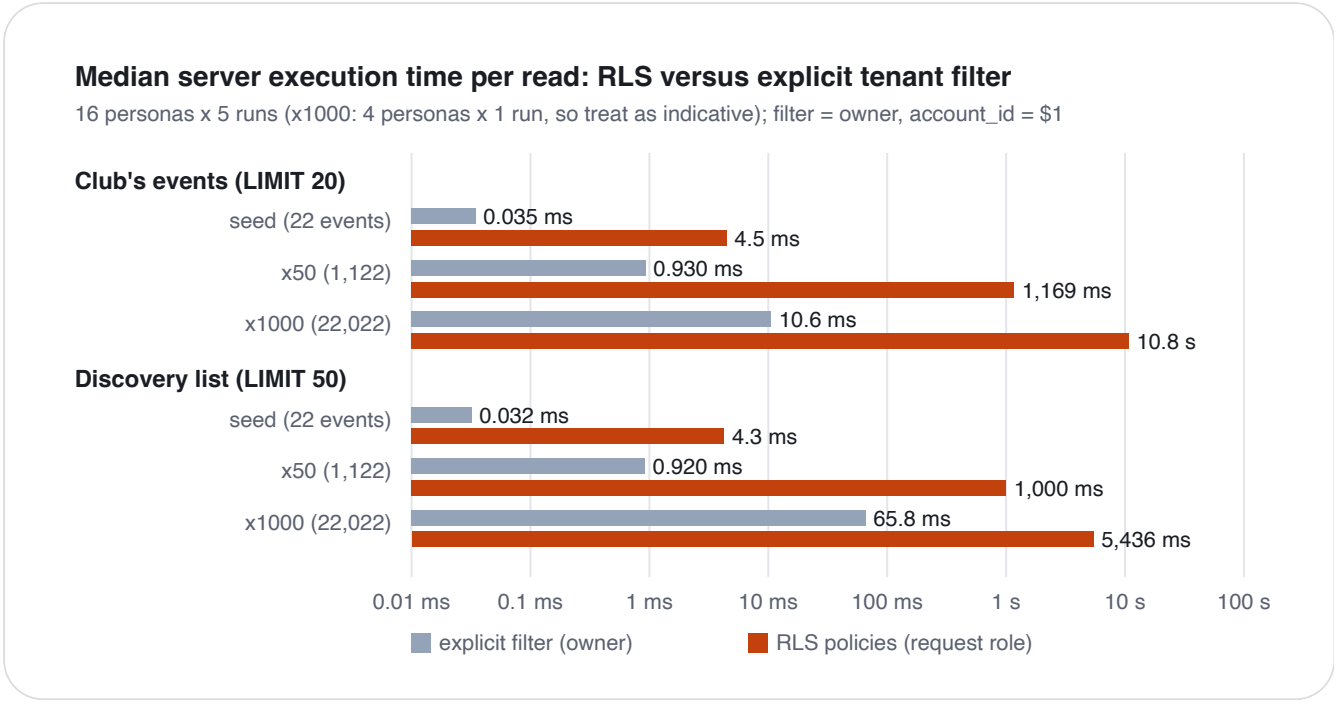


Figure 3. Median server execution time of two reads under RLS (orange) and with an explicit tenant filter (grey), at three data scales. Log scale. The x1000 bars rest on four measurements each.

Table 7. Server execution time (`EXPLAIN ANALYZE`, timing off) of two tenant-scoped reads under RLS and with the explicit tenant filter RLS replaces. "Ratio" is the median of per-measurement ratios. Both modes returned the same number of rows in every measurement.

Read, scale	Measurements	Filter p50	RLS p50	Filter min	RLS min	Ratio
Club events, seed (22 events)	80	0.035 ms	4.5 ms	0.021 ms	0.081 ms	84
Club events, x50 (1,122)	80	0.930 ms	1,169 ms	0.077 ms	17.1 ms	856
Club events, x1000 (22,022)	4	10.6 ms	10.8 s	4.7 ms	1,941 ms	524
Discovery list, seed	80	0.032 ms	4.3 ms	0.019 ms	0.073 ms	97
Discovery list, x50	80	0.920 ms	1,000 ms	0.092 ms	11.4 ms	1,656
Discovery list, x1000	4	65.8 ms	5,436 ms	7.0 ms	2,692 ms	372

Whole transactions. pgbench shows the same pattern (Table 8). On the seed the mixed transaction completes 358 transactions per second with the explicit filter and 65 under RLS. At x50 the figures are 190 and 1.7. Adding the explicit filter on top of RLS changes little (58 and 2.0), because RLS already restricts each read to the account through an index condition, and the cost lies in the remaining policy terms.

Table 8. pgbench, mixed transaction (set claims, two reads, save and unsave an event), 8 clients, 4 threads, 20 s, prepared statements, 5 interleaved runs per mode. Mean ± 95% Student-t half-width over runs. No transaction failed.

Scale, mode	Throughput (tps)	Mean latency (ms)
Seed, explicit filter (owner)	358.3 ± 99.6	23.3 ± 6.7
Seed, RLS	65.5 ± 18.7	127.8 ± 38.2
Seed, RLS plus explicit filter	58.0 ± 20.8	151.2 ± 73.1
x50, explicit filter (owner)	190.0 ± 64.3	45.0 ± 16.8
x50, RLS	1.7 ± 0.8	5,120.8 ± 2,237.3
x50, RLS plus explicit filter	2.0 ± 0.6	4,232.2 ± 1,470.4

Where the cost is. The archived plans locate it. Under RLS, both reads use the account condition as an index condition, and everything else in `event_select` becomes one per-row filter: an OR of calls such as `app.oversees_club(club_id)`, `app.is_committee_of(club_id)`, `app.is_member_of(club_id)`, `app.is_reviewer_for_club(club_id)`, `app.is_university_admin()` and `app.club_is_suspended(club_id)`. Every helper is a `SECURITY DEFINER` SQL function with a `SET search_path` clause, and some call further definers (`is_university_admin` calls `has_institution_role`, which calls `active_assignment`). Such functions cannot be inlined[4], and each call also saves and restores a configuration setting.

Microbenchmark. Our first hypothesis was that this call overhead was the main cost. Table 9 and Figure 4 test it on one account's 14,014 events, and the data largely refute it. Dropping the `SET` clause (median 2,766 ms), or using a plain invoker function with no `SET` that reads the claims inline (2,191 ms), improves little on the shipped helper's 3,369 ms, because the membership sub-query still runs once per row. What works is changing how often the helper runs. A definer called once per statement that returns the caller's club ids, tested with `club_id = any(...)`, brings the median to 127 ms (minimum 40.2 ms, against 2,533 ms), and keeps the definer property the design needs to avoid RLS recursion. A row-independent helper inside an OR is also evaluated per row:

`app.is_university_admin()` costs 39.7 s over these rows. Wrapping it in a scalar sub-select turns it into an InitPlan evaluated once, at 214 ms, as the vendor guidance in Section 2.2 predicts[5]. On its own, outside an OR, PostgreSQL already evaluates such a call once as a one-time filter, which is why the second, discarded run of this microbenchmark showed no effect.

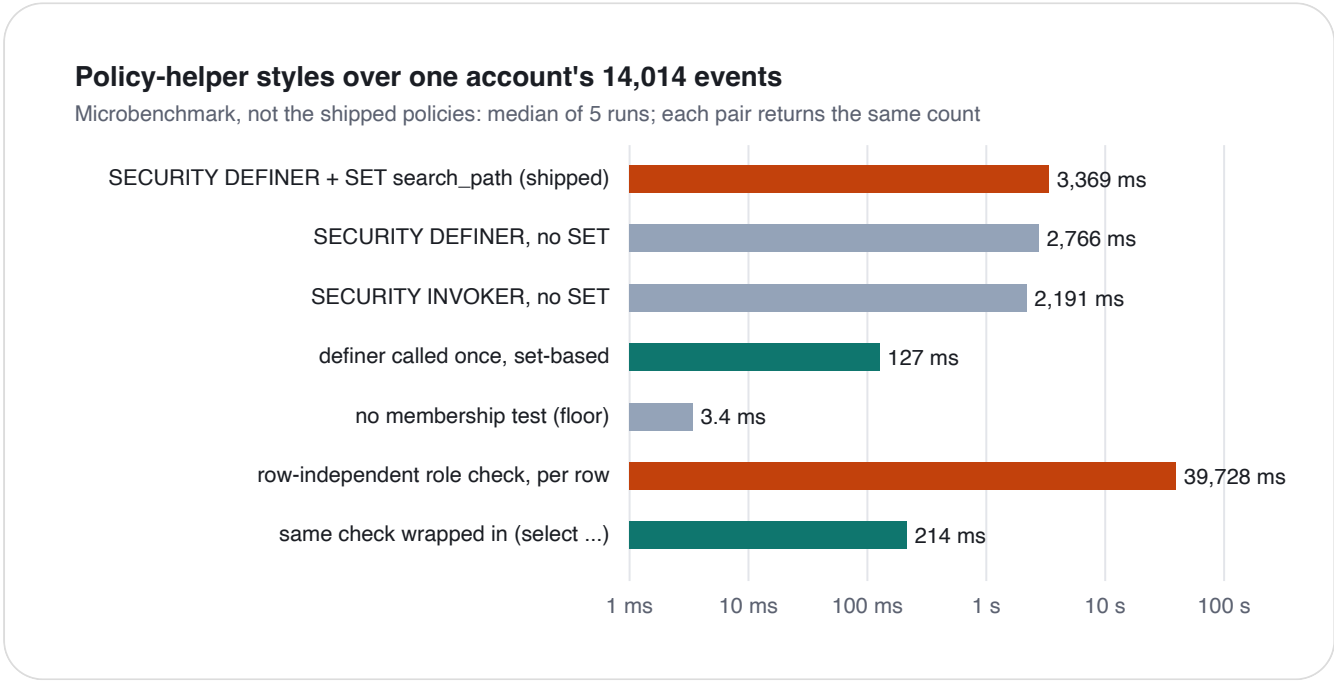


Figure 4. Microbenchmark: the same membership test and the same row-independent role check in different helper formulations, over one account's 14,014 events. Median of 5 runs, log scale. These are not the shipped policies.

Table 9. Microbenchmark of helper formulations over one account's 14,014 events at x1000, as superuser with persona 1's claims. Median and minimum of 5 interleaved runs. Every pair returns the same count. The invoker variant isolates call overhead only: in production it would itself be subject to RLS on `membership`, which is why the design uses definers.

Formulation	p50	min	Count
`SECURITY DEFINER` with `SET search_path` (as shipped)	3,369 ms	2,533 ms	1,001
`SECURITY DEFINER`, no `SET`	2,766 ms	2,062 ms	1,001
`SECURITY INVOKER`, no `SET` (claims read inline)	2,191 ms	1,033 ms	1,001
Definer called once, returns the caller's club ids	127 ms	40.2 ms	1,001
No membership test (floor)	3.4 ms	2.0 ms	14,014
`app.is_university_admin()` inside an OR, per row	39.7 s	26.9 s	0
`(select app.is_university_admin())` inside the same OR	214 ms	98.9 ms	0

7 Discussion and practical implications

7.1 Per-row helper cost, and the once-per-statement fix

The microbenchmark gives a rough per-row price. Over 14,014 rows the shipped membership helper adds about 3.4 s at the median, roughly 0.24 ms per row. The nested role check adds about 39.7 s, roughly 2.8 ms per row. A once-per-statement helper brings the membership test down to under 0.01 ms per row. These absolute figures are inflated by the machine's load, but the structure is not: an RLS read costs roughly the number of rows left after the index condition, times the per-row cost of every helper in the policy's OR. A tenant read that looks cheap on a 22-event seed scales with the tenant's size, and the first large tenant pays for it.

The evidence supports two rewrites, and it is microbenchmark evidence only. First, wrap every row-independent helper in a scalar sub-select, `(select app.is_university_admin())`, so that PostgreSQL evaluates it once per statement as an InitPlan. This is safe only when the helper's result cannot depend on the row, which holds for role checks that read the caller's claims. Second, replace per-row membership calls with a single definer that returns the caller's ids, tested with `= any(...)`. Neither rewrite has been applied to WayClub's 210 helper-calling policies. Before either ships, each rewritten policy needs an equivalence test that compares visible rows per persona under

the old and new policy, and the gain needs to be re-measured on an unloaded machine. A cheap guard against regression is a test that runs `EXPLAIN` on the main reads and fails if a known row-independent helper appears inside a per-row filter. The Supabase linter's `auth_rls_initplan` rule applies the same idea to its own authentication functions[7].

Two non-fixes are worth stating. Dropping the `SET search_path` clause would weaken the definer hardening that PostgreSQL recommends[3] and, in this microbenchmark, saved less than a fifth of the time. Converting helpers to invoker functions would expose them to RLS on the tables they read, which is the recursion the definer design exists to avoid.

7.2 Catalogue assertions and their scope

A query over `pg_class` and `pg_policy` cannot drift from the schema in the way prose did here, where three documents agreed with one another for days and not with the database. But a scope predicate is itself a model of tenancy, and it goes stale when the model changes. The `institution_id` predicate was right until the tenant root moved, then covered fewer tables with every migration. At head it is stale again, now for person-scoped tables. The practical rule is to pair scoped assertions with at least one scope-free invariant, such as "the request role holds no privilege on any relation without RLS". In this study neither family caught every mutant alone, and together they caught all eight.

A red suite is its own hazard. At the studied commit A5 already fails, so a new A5 violation, such as mutant M6's `USING (true)` policy, would arrive as one more line in a failure the team has learned to expect. Keeping the suite green, with each exemption written down and checked for its exact shape, is what makes a catalogue assertion useful as a gate.

7.3 FORCE, definers and the trust boundary

When policies call owner-owned definer helpers, forcing RLS can make those helpers wrong rather than safer: here it made the membership helper deny a genuine member. Rules that must bind every role, owner included, belong in foreign keys, unique indexes and triggers. And because a definer runs with its owner's privileges, every definer is an authorisation decision that must validate its caller's claims itself; unvalidated variants should be granted to no role.

7.4 Concurrency and tamper evidence

Allocate a sequence position inside the lock that orders it, and hold that lock for as short a time as possible. Identity columns are allocated before `BEFORE` triggers run, and a

deferred constraint trigger moves both the allocation and the lock into `COMMIT`. A unique index on the position turns any remaining race into a loud failure. Test such properties with genuinely concurrent transactions: a serial suite was green for the whole life of the forking chain, and two real connections with a deliberate interleaving exposed it in every trial.

An in-database, unkeyed hash chain detects edits and interior deletions by an attacker who does not recompute it, and nothing more. Truncation and a full suffix rewrite pass verification. A threat model should say so, and evidence that must survive a dispute needs an external anchor.

---

8 Threats to validity

Internal validity. Every timed run shared a heavily loaded laptop, and load was logged only from 17:13 UTC, after E1 and E4. Lock-wait conclusions survive the noise, because the fastest before-fix same-tenant run (4,906 ms) is slower than the slowest after-fix run (190 ms); the bootstrap interval for the headline ratio (81-152) is still wide. RLS execution times are inflated and vary widely between repetitions of the same query. We therefore rely only on differences of two or more orders of magnitude and report minima beside medians. The x1000 read timings rest on four measurements per read from a truncated run, and the helper microbenchmark on five runs for one persona, so both are indicative. Different schema variants ran at different times under different load, so absolute latencies are not comparable across rows. The baseline for RLS is the explicit tenant filter, not a full reimplementation of the policies' visibility rules: both modes returned identical row counts in every measurement, but on other data they could differ. The tampering experiment uses a superuser's trigger bypass to stand in for an owner-level attacker. The SQL port of the assertions was checked against the project's own test runner on four of the nine schema states.

Construct validity. "Fork" means two rows that share a predecessor hash. "Verification failure" means any recomputed hash or link that does not match in the version's own order. The pre-fix interleaving produces failures without forks, and the stress workload produces both, so the two are reported separately. "Ratio" for RLS reads compares server

execution time only; it excludes planning, network and client time, which matter to a user but are not affected by the policy terms studied.

External validity. One system, one developer, one PostgreSQL version and synthetic data. The x1000 scale copies the seed's 22 events a thousand times, which preserves the seed's mix of status and visibility but not a real tenant's distribution. The mutants are hand-chosen to represent plausible mistakes; they are not a sample of real ones, and eight mutants cannot estimate a detection rate.

Conclusion validity. Zero failures in 600 trials bounds the per-trial failure probability under these two workloads below 0.5% at 95% confidence. It is not a proof that forks are impossible, although the design in Section 4.4 is intended to rule them out by construction, and the unique backstop would turn any that occurred into a failed commit.

---

9 Limitations

- **No real tenants and no real load.** WayClub is pre-launch, with no live site, and its cloud database was stopped throughout. Nothing here shows how the system behaves in production, and the RLS cost has not been measured at the scale real tenants would reach.
- **One heavily loaded laptop.** All absolute timings are inflated by unrelated work on the same machine, and only large ratios should be read from them.
- **A truncated large-scale run.** The 22,022-event read timings come from four personas, one repetition each, after the full run was stopped.
- **The proposed RLS rewrites are microbenchmarks.** They have not been applied to the product's policies or tested for equivalence there.
- **The head test file is red.** At the studied commit, A5 fails on four new policies and an unrelated schema-guard test fails on a new column. Both fixes were in progress at the time (since saved on the branch `wip/paused-2026-11-02`); this study deliberately did not use them.
- **Undetected tampering.** The audit chain does not detect deletion of its newest row or a rewrite that recomputes later hashes. External anchoring is planned, not built.
- **A known gap that is enforced but not proven.** The project's security document records that the negative side of the two-party handover attestation (an ordinary member being refused an access verification) is enforced only by a policy, with no test asserting it. This study did not close that gap.
- **Single developer, AI-assisted.** One person designed the system, and much of the implementation and of this report was produced with AI coding assistants under his

direction. No independent party has reviewed either, and the adversarial review cited in Section 4.2 was itself AI-assisted.

- **Private source.** Only the audit-chain experiments can be reproduced from the public artifact alone.

10 Conclusion

In one pre-launch system, measured on one loaded laptop, four results hold. Allocating the audit-chain position inside its lock at commit time removed every chain failure the concurrent workloads had produced, and a same-tenant writer behind a 5-second holder waited a median of 41.3 ms instead of 4,919 ms. Catalogue assertions caught every seeded mistake inside their tenant scope, but the scope predicate is itself a model of tenancy that drifted, and a scope-free check was needed to catch the rest. The chain still cannot detect truncation or a full rewrite without an external anchor. And the policy-helper style that the RLS design relies on costs two to three orders of magnitude over an explicit filter at modest data sizes; a microbenchmark points to a fix that has not yet been applied or tested in the product. The broader lesson is modest: security claims about a database are worth most when they are written as queries that can fail, and concurrency claims when they are tested with concurrent transactions.

Artifact availability

The artifact is at /research/catalogue-driven-rls/artifact/. It contains the experiment drivers (Node.js and shell), the SQL ports of the catalogue assertions and the release check, the eight mutation files, the benchmark transactions and plan queries, every raw result file, the analysis, bootstrap and figure scripts, and a README with exact commands, software versions and the run dates (1-2 November 2026). WayClub's source is private. The artifact includes only verbatim excerpts of the two audit-chain migrations (006 and 011), enough to reproduce E1 and E4 on an empty database, plus

test-runner logs trimmed to result lines. E2 and E3 need the private schema and are provided as scripts and raw outputs.

References

1. F. Chong, G. Carraro and R. Wolter. Multi-tenant data architecture. Microsoft Corporation (MSDN), June 2006. citus-docs.readthedocs.io/en/v8.0/_static/mt-data-arch.pdf
2. PostgreSQL Global Development Group. PostgreSQL 17 documentation, section 5.9: Row Security Policies. postgresql.org/docs/17/ddl-rowsecurity.html
3. PostgreSQL Global Development Group. PostgreSQL 17 documentation: CREATE FUNCTION, section "Writing SECURITY DEFINER Functions Safely". postgresql.org/docs/17/sql-createfunction.html
4. PostgreSQL wiki. Inlining of SQL functions. wiki.postgresql.org/wiki/Inlining_of_SQL_functions
5. Supabase. Row Level Security: performance recommendations (documentation page, accessed 2 November 2026). supabase.com/docs/guides/database/postgres/row-level-security
6. pgTAP: unit testing for PostgreSQL. Documentation, policy assertions policies_are(), policy_roles_are() and policy_cmd_is() (accessed 2 November 2026). pgtap.org/documentation.html
7. Supabase. Performance and security advisors (database linter "splinter"), lints 0003 auth_rls_initplan, 0007 policy_exists_rls_disabled, 0008 rls_enabled_no_policy and 0013 rls_disabled_in_public (documentation page, accessed 2 November 2026). supabase.com/docs/guides/database/database-linter
8. B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. ACM Transactions on Information and System Security 2(2):159-176, 1999. doi.org/10.1145/317087.317089
9. S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In Proceedings of the 18th USENIX Security Symposium, pp. 317-334, 2009. usenix.org/conference/usenixsecurity09/technical-sessions/presentation/efficient-data-structures-tamper-evident
10. B. Laurie, A. Langley and E. Kasper. Certificate Transparency. RFC 6962, IETF, June 2013 (obsoleted by RFC 9162). rfc-editor.org/rfc/rfc6962

11. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil. A critique of ANSI SQL isolation levels. In Proceedings of ACM SIGMOD 1995, pp. 1-10. doi.org/10.1145/223784.223785

12. A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha. Making snapshot isolation serializable. ACM Transactions on Database Systems 30(2):492-528, 2005. doi.org/10.1145/1071610.1071615

13. M. J. Cahill, U. Röhm and A. D. Fekete. Serializable isolation for snapshot databases. In Proceedings of ACM SIGMOD 2008, pp. 729-738. doi.org/10.1145/1376616.1376690

14. D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. Proceedings of the VLDB Endowment 5(12):1850-1861, 2012. doi.org/10.14778/2367502.2367523

15. T. Warszawski and P. Bailis. ACIDRain: concurrency-related attacks on database-backed web applications. In Proceedings of ACM SIGMOD 2017, pp. 5-20. doi.org/10.1145/3035918.3064037

16. PostgreSQL Global Development Group. PostgreSQL 17 documentation, section 13.3: Explicit Locking (advisory locks). postgresql.org/docs/17/explicit-locking.html

17. PostgreSQL Global Development Group. PostgreSQL 17 documentation: CREATE TRIGGER (constraint triggers, DEFERRABLE INITIALLY DEFERRED). postgresql.org/docs/17/sql-createtrigger.html

18. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5):649-678, 2011. doi.org/10.1109/TSE.2010.62

Cite and reuse

Authorship and AI assistance. Yousof Selim designed the system and its security model and directed the work; much of the implementation and parts of this report were produced with AI coding assistants (Claude, Codex) under his direction and review. The experiments are reproducible from the artifact. This is a technical report and has not been peer reviewed.

```
@techreport{selim2026catalogue,
  author      = {Yousof Selim},
  title       = {Catalogue-Driven Row-Level Security Assurance and Commi
  institution = {Sunway University (Lancaster University dual degree), M
  type        = {Technical report},
  number      = {v1.1},
  year        = {2026},
  url         = {https://yousofselim.com/research/catalogue-driven-rls/}
}
```