SQL/PGQ in PostgreSQL 19: coverage and latency on the LSQB benchmark

Report version: 1.0 **Date:** 15 September 2026 **Authors:** Dmitriy Eremeev, Ivan Khramkov, Aleksey Teplov (T-Technologies, R&D Center)

> **Status of the feature, September 2026.** After this work was carried out, SQL/PGQ was reverted from PostgreSQL and **will not ship in version 19**. The last pre-release to carry it is beta 3, which is the code measured throughout this report: `src/backend/rewrite/rewriteGraphTable.c` and `src/backend/commands/propgraphcmds.c` are present at tag `REL_19_BETA3` and absent from `master`.
>
> This report therefore documents a feature that was withdrawn, not one that shipped. Section 6.5 sets out which of its findings are tied to the reverted code and which are properties of the engine underneath it, and so apply to whatever implementation arrives next.

TL;DR

The PostgreSQL 19 betas carried SQL/PGQ; the release will not (see above). What they carried had **no graph executor**: a property graph stores nothing, `GRAPH_TABLE` rewrites into an ordinary relational plan, the usual planner runs it (§5.1). Measured on LSQB, against the same queries written as explicit joins. The engine-level findings outlive the revert; §6.5 says which.

Works. 6 of 9 queries run; Q1, Q4, Q5, Q7, Q8 at every scale factor of the controlled series, SF 0.1–30. A separate SF 100 run, on different hardware and a longer timeout (§3.5), returns five of the nine at 2.1 G edges. All 438 returned runs matched LDBC's published answers (§4.5). Latency premium, for the formulations and physical layouts tested, 1.25–2.55× on each size's common set and 1.85–2.90× on the fixed pair Q5, Q7 (§4.2). Q1 is 1.60–2.20× *faster* as a pattern, so the pattern form is not inherently slower (§6.1).

Doesn't. Eight pattern constructions rejected at analysis time; no quantifiers, hence **no variable-length paths and no transitive closure**, and non-walk patterns need manual decomposition (§5.1). Q3, Q6, Q9 never complete in any of 21 configurations, on packaged defaults — the three LSQB attributes to WCOJ and factorisation, which PG implements for no query. Q2 completes at SF 1 and SF 3 only. `ANALYZE` rescued nothing and cost five completing cells: the estimate that fails is at the join, not the table (§5.2, §5.3). Perfect discriminator on these nine queries, and on nothing wider: patterns matching ≥1 `knows` edge complete in 6 or 0 of 18 configurations, patterns matching none in 16–18 (§6.1.1).

Choose. Derived relations as PT or MV, never plain views — 1.89–3.28× slower, worst coverage at scale, for 38–42 % disk that a few query passes erase (§6.2). PT vs MV: 0.97–1.44, no trend, identical failure rates — no stable latency advantage either way, on definitions that differ in indexing as well as in materialisation (§6.2).

Trust. Relational queries are LSQB's reference texts, ours are not; that bound is unmeasured (§6.3). Relational side has no indexes at all, SQL/PGQ has primary keys — so the gap is a lower bound, and an indexed baseline was not measured. Beta 1, 2 and 3 agree: identical coverage, same plans in 160 of 162 combinations, latency within 0.81–1.17 (§5.4).

Abstract

Why we did this. Our team works on graph data analytics and graph query processing. A mainstream relational engine picking up native graph pattern matching is, to us, one of the more significant developments in this space, and the version 19 betas were the first place the SQL/PGQ standard found a home in PostgreSQL. It has since been withdrawn, which makes the measurement a record of a first attempt rather than of a shipped feature. We wanted to measure early what the first implementation can actually do. This report is that measurement — a snapshot of a fast-moving feature, not a verdict on where it will end up.

It is worth saying at the outset what is being asked of the database, because it frames every number below. PostgreSQL implements SQL/PGQ with no graph executor at all: the rewrite turns each pattern into an ordinary relational plan and hands it to the same planner and the same executor as any other query (Section 5.1). A cost-based optimiser built for relational joins is therefore being asked to do work that dedicated graph engines address with purpose-built algorithms. That it does most of it is the first thing these measurements show.

The problem. A graph pattern over relational data can be written two ways: as a chain of relational joins, or as a pattern the database is asked to match. In PostgreSQL, until version 19, only the first was available. It spells out one `JOIN` per edge and grows with the pattern: nine edges mean a nine-way join written out in full, and the next pattern starts again. That is the form every graph-shaped query over a PostgreSQL database has had to take.

What changed. SQL/PGQ, Part 16 of the SQL standard since 2023 [6], offers the second way. A property graph is declared once as a view over existing tables, and thereafter a pattern is written as a pattern. `GRAPH_TABLE` matches it and returns the matches as an ordinary relation, which the rest of the query consumes normally. The same rows, the same database, a shorter query. Version 19 is the first PostgreSQL release to implement it, so for anyone already on that database the choice is now a real one. What it costs is not yet documented: we searched for published measurements of SQL/PGQ in PostgreSQL and found none. The standard fixes what the operator means, not which patterns a given database will execute, or how fast.

In this work we price that choice. The benchmark is LSQB, the Labelled Subgraph Query Benchmark of the Linked Data Benchmark Council [1, 2]. Every query is written twice over the same data: as a `GRAPH_TABLE` pattern, and as the explicit joins a team would write today. Both forms run across six dataset sizes and three ways of supplying the derived relations the schema needs, under a single controlled protocol.

What we found is an implementation bounded first by pattern syntax, then by planner coverage, and only then by speed. `GRAPH_TABLE` accepts a subset of the standard's patterns: element pattern quantifiers are rejected, so there are no variable-length paths and no transitive closure, and a pattern that is not a single walk has to be decomposed by hand (Section 5.1). Of the nine benchmark queries six run. At the two middle dataset sizes SQL/PGQ answered everything except Q3, Q6 and Q9 — precisely the three the

benchmark designed to require worst-case optimal joins or factorisation, which PostgreSQL implements for no query, graph-shaped or not. LSQB is a demanding place to start. Its authors built it to separate graph workloads from relational ones, and the queries it loses here are exactly the ones they name as needing those techniques. Coverage measured this way therefore sits nearer a worst case than a representative one. Every value either form returned matched the benchmark's published answers, so the operator is correct wherever it answers. Where both forms work the explicit joins lead by 1.25 to 2.55 times on the set each size has in common, or 1.85 to 2.90 times on the fixed pair Q5 and Q7. On one query the pattern form is the faster of the two, by 1.60 to 2.20 times. For a first release that is a narrow margin, and that it reverses at all shows the graph form is not inherently the slower one.

What this settles. A reader choosing between the two forms learns which patterns PostgreSQL 19 will execute in each, what the shorter form costs where both work, and which of three ways to supply the schema's derived relations. Across three pre-releases those answers do not move. Beta 1, beta 2 and beta 3 agree query for query on coverage and within 0.81–1.17 of one another on latency, so what is measured here is settled behaviour rather than a moving target. What the report does not settle is whether the patterns that currently time out can be made to run. One of them, Q3, returned in 1.504 s in a single beta-1 configuration under the earlier uncontrolled protocol, and did not reproduce under the controlled one. That is suggestive of a planner ceiling rather than an operator one, but it does not establish it. Nor does the report settle whether a better SQL/PGQ formulation of these nine exists than ours. The harness, schemas, query texts, raw results and plans are published [5].

What to do on PostgreSQL 19 today

Five practical conclusions, each established below and each qualified where it is established.

1. **Check the pattern syntax first.** PostgreSQL 19's `GRAPH_TABLE` accepts a subset of the standard's patterns. Element pattern quantifiers are rejected, so there are no variable-length paths and no transitive closure. A comma-separated list of path patterns sharing element variables — the standard's way of writing a pattern that is not a single walk — is rejected too, so such patterns must be decomposed into several `GRAPH_TABLE` expressions by hand. Eight constructions are refused, all of them during analysis rather than at run time, and Table 16 lists them with their diagnostics.
2. **Expect the pattern to run unless it traverses a many-to-many edge repeatedly.** On these nine queries — a sample far too small to generalise from — the number of `knows` edges a pattern matches separates completion from timeout exactly. The five queries that match none complete in 16 to 18 of 18 configurations; the four that match one or more complete in 6 or 0 (Section 6.1). The relational form answers all four, though Q9 only at two sizes of six. This is a nine-query correlation, not a law, but it is the cheapest pre-flight check available.
3. **In this benchmark configuration, budget a latency premium of roughly 1.25–2.55× on the varying common set, or 1.85–2.90× on the fixed pair Q5 and Q7 — not an order of magnitude.** Expect it to reverse on patterns that span many separately stored edge relations. That is the one case measured here where the pattern form wins outright (Sections 4.2 and 6.1). The two forms compose freely in one query: `GRAPH_TABLE` returns an ordinary relation, and six of the nine SQL/PGQ texts measured here join it to base tables (Table 4). What composition does not do is rescue a stalled pattern — Q3 written as two `GRAPH_TABLE` expressions joined five ways times out exactly as the undecomposed pattern does (Table 17).
4. **Supply the schema's derived relations as plain tables or materialised views, not as plain views.** Plain views cost 1.89–3.28× in aggregate latency. A median of 2.81× of that is attributable to view expansion itself, once a control group of queries touching no derived relation is subtracted (Section 6.1). They also carry the worst coverage at the larger sizes and reach 1.17 GiB of peak memory on one query. What that buys is a disk saving that a handful of passes over the query set already outweighs (Sections 4.3 and 6.2). For this workload and these physical definitions, no stable latency advantage of plain tables over materialised views was observed. A controlled comparison — the two modes equalised on indexing and element keys — remains future work (§6.4), so the honest reading is that speed did not separate them here, rather than that it cannot.
5. **Run plain `EXPLAIN` before running the query — both failure modes are visible without executing it.** An estimated row count orders of magnitude above anything the data could yield marks an estimate that view expansion has defeated. Under plain views at SF 100 the plan for Q5 estimates 4.2×10^{20} rows against 165 million over plain tables, and Q4 2.5×10^{45} against 1.9×10^{8} (Section 6.3). Three of the four queries carrying that signature timed out, but Q7 completed despite a 9.6×10^{32}

estimate, so treat it as a warning rather than a verdict. At the other extreme, a join estimated at one row feeding a nested loop over an unindexed relation marks a tie the cost model cannot see through; that one an index fixes (Section 5.3).

1. Terminology and abbreviations

Abbreviation	Definition
SQL/PGQ	SQL Property Graph Queries — the graph-pattern extension standardised as ISO/IEC 9075-16:2023, surfaced through the `GRAPH_TABLE` table function
PGQ	Used in this report as a shorthand for "the SQL/PGQ approach of a benchmark query"
SQL	The reference relational approach of the same query, written with explicit `JOIN` operations and no property-graph objects
`JOIN`	The relational operation that combines two tables on a predicate. Matching a graph pattern in SQL requires one `JOIN` for every edge of the pattern, written out explicitly in the query text. This report calls such a query a *join formulation*, in contrast with the *graph formulation* written through `GRAPH_TABLE`, where the joins are generated by the rewrite rather than written
PT	Plain Tables — the derived union relations materialised as ordinary base tables
RV	Regular Views — the same relations defined as non-materialised views
MV	Materialised Views — the same relations defined as materialised views
SF	Scale Factor — nominal dataset size multiplier (SF 1 ≈ 2.57 GiB on disk in PT mode)
LSQB	Labelled Subgraph Query Benchmark, published by the Linked Data Benchmark Council [1, 2] at github.com/ldbc/lsqb. Its stated aim is to exercise the query optimiser, in particular `JOIN` ordering and the choice between binary and n-ary joins, and the execution engine
RSS	Resident Set Size — how much of a process's address space is actually held in RAM at a given moment, as opposed to swapped out or merely reserved. Reported by the kernel in `/proc/[pid]/status`: `VmRSS` is the current figure and `VmHWM` the highest it has reached since the process started
Approach	One of the two things compared: the SQL/PGQ implementation or the relational one, each being a schema together with a query formulation over the projection it requires (Section 3.3). Used in preference to "arm" throughout
Configuration	One approach × one storage strategy × one scale factor, on one PostgreSQL build. The relational approach has 18 (3 strategies × 6 scale factors, beta 3 only). SQL/PGQ has 18 per build at SF 0.1–30, plus 3 more at SF 100 on beta 3 alone, which is where the count of 21 comes from; the plan corpus of Section 5.2 spans 72, being 54 SQL/PGQ configurations across three builds plus the 18 relational ones
Cell	One query in one configuration — the unit the latency and coverage tables report
GiB / MiB	Binary multiples (2^{30} / 2^{20} bytes). All storage figures in this report use binary units

2. Introduction

2.1 Motivation

Consider a query that finds, say, every person who has commented on a post created by a friend of a friend. Over relational tables that is a chain of joins: one per edge of the pattern, each with its own join predicate, written out in full, growing with the pattern rather than with the data. LSQB's patterns run from three edges to nine [1], so the longest is a nine-way join spelled out in the query text. Over a property graph the same computation over the same rows is a pattern, matched by the database, and shorter at every length. Until version 19 only the first form was available. **The subject of this report is what the second one costs.**

SQL/PGQ, standardised in 2023 as Part 16 of SQL [6], replaces the chain with two things: a declaration, made once, and a pattern, written per query. Other systems already implement it, DuckPGQ over DuckDB among them [7]; this report measures PostgreSQL's implementation and does not compare it with theirs. The version 19 betas were the first place it appeared in PostgreSQL, which is what made the question worth measuring when we did. To the best of our knowledge no measurements of it have been published, so the cost of the choice is at present a matter of expectation rather than record.

Two properties of the feature decide how it can be measured at all, and both matter for reading Section 3. First, a property graph holds no data of its own: it is declared over base tables that already exist, and `GRAPH_TABLE` returns its matches as a relation that ordinary SQL consumes. A graph query and a `JOIN` query therefore compete over the same stored rows, which is what makes a controlled comparison possible. Second, the standard says nothing about how a schema's derived relations should be supplied when the source data does not provide them as base tables. The LSQB schema [2] needs five such relations, so the choice has to be made and nothing guides it. This report treats it as a third dimension of the measurement rather than settling it by fiat.

2.2 Contributions

Pricing that choice comes down to four questions. The contributions of this report are the answers, in decreasing order of how well the evidence supports them.

1. **Which patterns does PostgreSQL 19 run in each form, and how fast?** A measurement on LSQB across six dataset sizes, three pre-releases and three ways of supplying the derived relations, under one protocol throughout (Section 3.5). Every returned value was checked against the benchmark's reference answers.
2. **How far does the pattern form reach?** Which queries it executes, at which sizes and at what latency, where it stops instead, and a reproduction of the Q3 timeout that fits on one page (Section 5.1).

3. **Would the obvious remedy have helped?** PostgreSQL's documentation states that accurate statistics help the planner choose the most appropriate query plan [8]. Collecting them rescues no query anywhere in the matrix and causes five previously completing cells to time out (Sections 5.2 and 5.4).
4. **Why do the failures happen?** A diagnostic account of three mechanisms: the query the `GRAPH_TABLE` rewrite generates, the cardinality estimates the planner forms over it, and the access paths available to it. This is the weakest of the four, because plans establish what the planner chose rather than why the choice was open to it.

No claim of priority is made, and none about SQL/PGQ implementations other than PostgreSQL's: DuckPGQ [7] and others exist and are not measured here. The authors are not aware of another published evaluation of SQL/PGQ in PostgreSQL on this benchmark, but no systematic survey of the literature was carried out.

3. Experimental setup

3.1 The LSQB benchmark: nine subgraph-matching queries

The benchmark comprises the nine LSQB queries Q1 to Q9 [1, 2], each a subgraph-matching query returning an aggregate count. All eighteen texts as executed, nine per approach, are published with the harness [5]. This report identifies queries by number alone and attaches no informal characterisation to them, since none is derivable from the measurement data.

Listing 1 shows one of the nine in both approaches, to fix what the two look like. Q6 counts the ways a person can be reached in two `knows` steps from someone with a given interest. It is one of the three SQL/PGQ queries that consist of a single graph pattern and nothing else, which makes the contrast with the `JOIN` form as direct as the benchmark offers.

Listing 1. Q6 as executed, in the SQL/PGQ approach and in the relational approach.

```
-- SQL/PGQ: the pattern is written once, and the engine chooses how to traverse it
SELECT count(*)
FROM GRAPH_TABLE (lsqb
    MATCH (person1 IS Person)-[k IS knows]-(person2 IS Person)-[k2 IS knows]-(person3 IS Person)-[hi IS
    hasInterest]->(T IS Tag)
    WHERE person1.id <> person3.id
    COLUMNS(1 AS c)
);
```

```
-- relational: one JOIN per edge, written explicitly
SELECT count(*)
FROM Person_knows_Person pkp1
JOIN Person_knows_Person pkp2
  ON pkp1.Person2Id = pkp2.Person1Id
 AND pkp1.Person1Id != pkp2.Person2Id
JOIN Person_hasInterest_Tag
  ON Person_hasInterest_Tag.PersonId = pkp2.Person2Id;
```

The graph form names three edges and one inequality. The `JOIN` form names the same three edges as two self-joins of `Person_knows_Person` plus one join to `Person_hasInterest_Tag`, and it fixes an order in which to combine them. That difference is the whole subject of the comparison: the graph form leaves the order to the planner, and the `JOIN` form does not. Q6 is also the query that never completes under SQL/PGQ at any scale factor, while the `JOIN` form beneath it answers in 3.0 s at SF 0.1 (Sections B.1 and B.2).

Note that the SQL/PGQ text is a single pattern only in the sense that one `MATCH` covers it. Six of the nine combine `GRAPH_TABLE` with ordinary relational joins (Table 4), which matters when they fail. A single-pattern failure like Q6 is evidence about the operator; a composed one like Q3 is not, until the composition has been ruled out (Section 5.1).

LSQB targets the query optimiser, in particular join ordering and the choice between binary and n-ary joins, together with the execution engine [1, 2]. Date and string operations, query composition and complex aggregates and filters are declared out of scope [2], and every query ends in a bare `count(*)`. Nothing in a measurement of LSQB is therefore attributable to projection, ordering or aggregation.

There are nine because a single query cannot separate the three operations the benchmark's authors identify as distinguishing graph workloads: many-to-many joins, cyclic joins and long acyclic joins [1]. The set turns one knob at a time. Q1 is the long acyclic case with no cycle in it. Q2 adds a cycle but only one many-to-many edge label, so binary joins remain optimal. Q3 closes a cycle over three `knows` edges, where cyclicity and many-to-many cardinality coincide. Q4 changes the shape rather than the length, with four edges meeting at one hub. Q5 is the smallest pattern in the set. Q6 puts two `knows` steps in front of a third edge, and the authors name it and Q9 the most challenging of the nine [1]. Q7, Q8 and Q9 change no shape at all: they repeat Q4, Q5 and Q6 with one operator added, and so measure the cost of that operator.

Table 1 sets the nine out along both divisions, the algebraic one from the benchmark and the structural one visible in the patterns. Q1 to Q6 are select-project-join-group-by queries; Q7 to Q9 add an antijoin or an outer join. The figure's notation carries over into this report: «neg» marks an edge that must not exist, «opt» an edge that need not, and thicker lines an edge label whose cardinality is many-to-many [1]. All nine are evaluated under homomorphism, so a vertex or edge may be matched more than once unless a filter forbids it [1].

Table 1. The nine LSQB queries. Grouping and pattern counts follow the benchmark's own account and pattern figure [1]. The relational text uses a different number of joins, in either direction. It uses fewer where the `merged` projection stores an edge as a foreign-key column, and more where an edge is a junction relation of its own.

Query	Algebra	Pattern	Edges	Extends
Q1	SPJG	Long acyclic chain through eight vertex labels, `Country` to `TagClass`	7	—
Q2	SPJG	Cyclic, but with one many-to-many edge label only	4	—
Q3	SPJG	Cyclic: a `knows` triangle whose three people share a country	9	—
Q4	SPJG	Star: four edges meeting at one `Message`	4	—
Q5	SPJG	Two distinct tags on a message and a reply to it	3	—
Q6	SPJG	Two `knows` edges and one `hasInterest` edge	3	—
Q7	SPOJG	Q4 with `likes` and `replyOf` made «opt»	4	Q4
Q8	SPOJG	Q5 with a «neg» `hasTag` edge	4	Q5
Q9	SPOJG	Q6 with a «neg» `knows` edge closing the pattern	4	Q6

That account anticipates the coverage result of Section 4.1 almost exactly. The four queries SQL/PGQ cannot execute, or executes at two scale factors only, are Q3, Q6, Q9 and Q2. Three of them are the ones the benchmark identifies as requiring worst-case optimal joins or factorisation, neither of which PostgreSQL's planner and executor implement. Q2 is not among those: it needs only binary joins, and fails for a different reason (Section 6.3, item 7). What all four share is that their patterns traverse `knows`, the

schema's largest many-to-many edge; Section 6.1.1 develops that as the sharper discriminator. The five SQL/PGQ does execute, in at least 16 of 18 configurations each, are the chain, the star, the tag comparison and their extensions. Optional and negative edges, which the benchmark separates out as advanced, cost little by comparison: Q7 completes in all 18 configurations against Q4's 17, and Q8 in 16 against Q5's 18, both Q8 losses being under regular views.

3.2 PostgreSQL version, configuration and test hardware

Table 2. Database under test.

Parameter	Value
Version	PostgreSQL 19 beta 3
Container image	`postgres:19beta3`, the official Docker Hub image; Dockerfile pinned at commit `4a1f78f`, variant `19/trixie`
Server package	`19~beta3-1.pgdg13+1`, from the `trixie-pgdg` repository at `apt.postgresql.org`
Base image	`debian:trixie-slim`; locale `en_US.UTF-8`, `LANG=en_US.utf8`
Architecture	`x86_64` (amd64), for which upstream publishes official builds
JIT	The provider package `postgresql-19-jit` (`19~beta3-1.pgdg13+1`) is installed, JIT being a separate package from PostgreSQL 18 onwards. It was **not used**: `jit` is `off`, which is also its boot value in this build (Table A2)
Configuration changes made by the image	The Dockerfile edits one line of the sample configuration, `listen_addresses = '*'`. The generated `postgresql.conf` additionally sets `shared_buffers` and `max_wal_size` explicitly, at values equal to their boot defaults (Table A2). Every other setting is the built-in default
Data directory	`/var/lib/postgresql/19/docker`
Bulk load method	`COPY` from CSV

Leaving the server configuration unmodified was deliberate, and establishes an untuned baseline. The image itself alters only `listen_addresses`, so the effective configuration is that of the packaged `postgresql.conf.sample`. Defaults change between major versions, and the Debian packaging may differ from an upstream build, so the effective values were read from the running server rather than assumed. They are listed in full in Appendix A, and every one of them equals its boot default. Five bear directly on the interpretation of the results.

`shared_buffers` is 128 MB. Against 32 GiB of memory on the instance that measured SF 0.1 to SF 30, the buffer pool holds half the database at SF 0.1 and 0.16 % of it at SF 30; at SF 100 it holds 0.05 %. The measurements are therefore dominated by the operating-system page cache and, at SF 30 and SF 100, by storage.

`effective_cache_size` is 4 GB at every scale factor. The planner costed its choices against an assumed cache eight times smaller than the 32 GiB actually available at SF 0.1 to SF 30, and eighty times smaller than the 320 GiB at SF 100. This misinformation applies uniformly to both approaches. It is nonetheless a plausible contributor to the estimate collapse under view expansion documented in Section 6.3, where the RV plan for Q5 at SF 100 estimates 4.2×10^{20} rows against PT's 165 million.

`jit` is off. That removes just-in-time compilation from the set of possible explanations for the Q3, Q6 and Q9 timeouts, notwithstanding that the provider package is installed.

`work_mem` is 4 MB. It applies to every hash table and every sort in every plan. LSQB is a benchmark of multi-way joins over many-to-many edges, whose intermediate results reach billions of rows at the larger scale factors [1]. A hash join given 4 MB on inputs of that size spills to disk almost at once. It would be easy to conclude from this that the queries which never complete are simply short of memory, and the measurements do not support that conclusion.

Three observations argue against it. The `GRAPH_TABLE` formulation of Q3 once completed in 1.504 s with a peak working set of 40 MiB under this same 4 MB setting (Section 5.4). The same triangle expressed as explicit `JOIN` operations completed at SF 0.1 in 1.459 s, again under this setting (Section 5.1). And the one failure this report diagnoses at the level of plans is Q9 in the relational approach: a nested loop over an unindexed relation, which no amount of working memory would help and which an index resolved (Section 5.3). A plan that chooses a product where a join would do does not become fast when given more memory to hold the product in.

`work_mem` can in principle change which plan is chosen. It enters the cost of a hash join through the number of batches, and the cost of a sort through the choice between an in-memory and an external one. But it enters through *estimated* volumes, not actual ones, and that limits what it can do here. Take the one cell this report diagnoses at the level of plans. There the anti-join feeding the top join is estimated at one row against an actual five million, and the outer input to that join is estimated at one row as well (Section 5.3). Estimates of that size sit far below 4 MB and equally far below any larger value, so no setting of `work_mem` alters the comparison between the competing plans in that cell. The lever there is cardinality estimation and access paths, not memory: an index changed the outcome, and collecting statistics did not.

Whether the same holds for the SQL/PGQ queries that never complete cannot be said. Those queries reach the timeout, so only estimate-only plans exist for them (Section 5.2). A re-run at a higher `work_mem` would test the reasoning rather than remedy anything, and Section 6.4 asks for it on that footing.

`max_parallel_workers_per_gather` is 2. A query may therefore use three processes on instances with 8 or 20 vCPU, so most of the machine is idle for most of the run. This applies equally to both approaches and to all three storage strategies, so it does not distort the comparisons, but it does mean that no figure here should be read as what the hardware can do.

3.3 The two implementations compared: SQL/PGQ schema vs relational JOIN schema

What this report compares is two complete PostgreSQL implementations, not two query languages in isolation. They differ in four respects beyond the query language, and every cross-approach statement should be read with these in mind:

1. **Projection.** SQL/PGQ reads a normalised layout with a separate relation per edge type; the relational approach reads a merged one with edges folded into foreign-key columns.
2. **Indexing.** Every SQL/PGQ table carries a primary key and its index; the relational schema declares none, following the benchmark's own reference configuration.
3. **`knows`.** One row per undirected edge in the first, two in the second.
4. **Provenance.** The relational query texts are the benchmark's reference implementations; the SQL/PGQ texts were written for this study (Section 3.1). This is the strongest of the four, and Section 6.3 treats it as a named threat.

The wording throughout follows suit: it is the relational implementation that outperforms the SQL/PGQ implementation here, not relational SQL that is faster than SQL/PGQ.

Both approaches were measured with the same three storage strategies across the six shared scale factors, giving 18 relational configurations against 21 for SQL/PGQ. Latency, coverage and failure rate are compared; memory, disk footprint and load time are not, for the reasons below. SF 100 is excluded, since the relational approach was not measured there.

- **Dataset:** the pre-generated LSQB datasets published in the SURF data repository [3], obtained through the download scripts supplied with the benchmark [2] rather than generated locally. They come from the LDBC social-network generator with all property values removed, leaving vertex identifiers, and the schema has nine vertex types and eleven edge types [1].
- **Format:** CSV
- **Projections used:** LSQB publishes each scale factor in two layouts, and both were used:
- `social-network-sf{SF}-projected-fk`, referred to below as the `projected` projection, in which foreign keys remain separate relations. Used for the SQL/PGQ runs.
- `social-network-sf{SF}-merged-fk`, referred to below as the `merged` projection, in which foreign keys are folded into the entity tables. Used for the relational SQL runs.

LDBC publishes the size of each dataset, and Table 3 gives them for the seven scale factors used here. The benchmark requires at least one of its datasets to reach a billion edges, which SF 100 does [1].

Table 3. Graph size by scale factor, as published by LDBC [1, 3]. These count the graph: vertices and edges, not the relations that hold them.

Scale factor	Vertices	Edges
SF 0.1	432 235	2 080 404
SF 0.3	1 179 535	6 183 839
SF 1	3 955 790	22 196 676
SF 3	11 257 915	66 267 918
SF 10	35 496 603	217 044 821
SF 30	103 177 340	650 542 290
SF 100	326 042 268	2 124 678 502

These are graph sizes, and the number of rows the database ends up holding is a different quantity. The `projected` layout stores some edges twice, because four of the five derived relations repeat rows already present in `Comment_hasTag_Tag`, `Comment_hasCreator_Person` and their `Post` counterparts; and it omits four edge types that the `merged` layout carries as foreign-key columns. Section 4.3 counts what was actually loaded and sets the two side by side.

The three differences listed above beyond the projection are documented in the DDL, and each is worth setting out in full.

First, `Person_knows_Person`. The two DDL scripts show the difference directly [5]. The SQL/PGQ approach loads one row per undirected edge, because the `pgq` pattern `-[k:knows]-` expands in both directions (Section 6.3, item 4). The relational approach loads the same CSV file twice, in both column orders, and therefore holds twice as many `knows` rows. Both approaches return the reference answer for every query (Section 4.5), so each convention is consistent with its own query formulation. The relational approach nevertheless scans twice as many edges wherever `knows` appears, and still completes those queries faster (Section 4.2).

Second, the two approaches differ in indexing, and it is the SQL/PGQ side that departs from the benchmark. The relational schema contains no primary key, unique constraint or index of any kind [5]. That follows the benchmark's own reference configuration. Its authors loaded with `COPY` and state that no indexes, integrity constraints or uniqueness constraints were defined [1]; the `sql/schema.sql` in the repository declares none either [2]. Every table in the SQL/PGQ approach, by contrast, carries a `BIGSERIAL` primary key and the associated btree index. The relational approach is thus the less physically supported of the two, and also the one that follows the benchmark as published.

Third, the relation sets are not congruent. Four relations exist only in the relational schema: `Forum_hasTag_Tag`, `Person_studyAt_University`, `Person_workAt_Company` and `Message_isLocatedIn_Country`. Eight exist only in the SQL/PGQ schema under PT, and nine under RV and MV, which additionally keep `Comment_replyOf_Comment` as a base table. The reason is that the relational approach folds the corresponding foreign keys into the entity tables: `Comment.hasCreator_PersonId` replaces a `Comment_hasCreator_Person` relation, and so on. The SQL/PGQ schema also carries a `Message` vertex relation for which the relational schema has no use.

Given these differences, three quantities are compared across approaches: query latency, coverage and failure rate. Both approaches answer the same nine questions over the same source dataset at the same scale factor, and part of any latency difference is properly attributed to the schema rather than to the query language.

Three quantities are not compared across approaches. Disk footprint and load time are properties of the projection and of the bulk-load path. Memory is excluded because its metric does not separate query-private allocation from shared-buffer pages faulted into the backend (Section B.3). These three are compared only within each approach, across PT, RV and MV.

The two approaches draw their query texts from different places, and the difference bounds what Section 6.1 establishes. The relational approach uses the formulations in the `sql/` directory of the LSQB repository unchanged [2]. These are long-standing reference implementations, exercised by the other systems the benchmark supports. The SQL/PGQ approach uses formulations written by the present authors, adapted from the `pgq/` directory of the same repository to the syntax PostgreSQL accepts; none of the nine is textually identical to its counterpart there. Every query in both approaches returns the reference answer (Section 4.5), so the SQL/PGQ query texts are demonstrably correct; the returned values are in the raw result files [5]. What cannot be demonstrated is that they are the best available expression of these nine patterns, and a cross-approach deficit may therefore reflect the formulation as well as the engine.

A second consequence is more specific. Only three of the nine SQL/PGQ queries consist of a single graph pattern; the remaining six combine `GRAPH_TABLE` with ordinary relational machinery, as Table 4 sets out.

Table 4. Structure of the SQL/PGQ query texts. The final column counts relational `JOIN` operations applied outside the graph pattern.

Query	`GRAPH_TABLE` invocations	`knows` edges matched	Joins outside the pattern	Shape
Q1	1	0	0	single pattern
Q2	2	1	1	two patterns joined
Q3	2	1, reused	5	two patterns as CTEs, joined five ways
Q4	3	0	2	three patterns as CTEs, joined
Q5	1	0	0	single pattern
Q6	1	2	0	single pattern
Q7	1	0	2	pattern with outer joins to base tables
Q8	1	0	1	pattern with an outer join to a base table
Q9	1	2	1	pattern with an outer join to a derived closure

The distinction matters where these queries fail. Q6 is a single pattern and never completes, which is evidence about `GRAPH_TABLE` itself. Q3 also never completes here, and because its formulation is a decomposition it might be objected that the decomposition, rather than the engine, is at fault. That objection has been tested directly and does not hold; Section 5.1 sets out the experiment.

3.4 Storage strategies for the derived relations: plain tables, views, materialised views

The LSQB property-graph schema is defined over a vertex type `Message`, which the source dataset does not supply. What the dataset supplies are its two subtypes, `Comment` and `Post`, in separate files. Every relation that the schema attaches to `Message` therefore has to be assembled from the corresponding pair of `Comment` and `Post` relations. Five relations are needed:

- `Message` — union of `Comment` and `Post`
- `Comment_replyOf_Message` — union of `Comment_replyOf_Comment` and `Comment_replyOf_Post`
- `Message_hasCreator_Person` — union of the `Comment` and `Post` creator relations
- `Message_hasTag_Tag` — union of the `Comment` and `Post` tag relations
- `Person_likes_Message` — union of `Person_likes_Comment` and `Person_likes_Post`

The standard says nothing about how such relations should be supplied. A property graph is declared over whatever relations already exist, so the implementer must first decide what those relations are to be — a decision that is unavoidable and taken before any query runs. Three constructions are available in PostgreSQL, and all three were measured.

Plain tables (PT). Each derived relation becomes an ordinary table, filled once at load. The union is computed then and never again, and at query time the planner sees a plain table with statistics over it. The rows are stored twice, once in `Comment` and `Post` and once in the derived relation.

Non-materialised views (RV). Each becomes a view and nothing is stored. All five are defined as `UNION ALL` over two sources, so every reference expands into a union of two scans. A query touching four of the five, as Q4 does, is rewritten from four scans into eight scans and four union nodes before the planner sees it. This is the only construction in which a derived relation cannot fall out of step with the tables beneath it, since there is nothing to keep in step.

Materialised views (MV). Each becomes a materialised view, computed once at load. The rows are PT's; what differs is the object holding them, and the obligation to refresh after a change to the base data.

Table 5. The three strategies as their definitions imply, independent of measurement.

Strategy	Rows stored	Union computed	What the planner sees	Can fall out of step
PT	twice	once, at load	an ordinary table with statistics	no
RV	once	on every query that touches the relation	the expanded view definition, inside each query	no
MV	twice	once, at load	a materialised view	yes, until refreshed

Two expectations follow, both worth stating before the results contradict either. Plain tables should be the easiest case for the planner and the most expensive to store. Non-materialised views should be the cheapest to store and the most expensive to execute.

The six DDL scripts, one per configuration, are published with the harness [5]. `Message` under PT is filled by `INSERT ... SELECT ... UNION ALL` over `Comment` and `Post`, the four edge relations by two `COPY` statements each, and every one of the five carries a surrogate `id BIGSERIAL PRIMARY KEY` and its btree index. Under RV each relation is a `CREATE VIEW` and nothing is populated beyond the source tables. Under MV each is a `CREATE MATERIALIZED VIEW` populated at load, with no key and no index.

Four differences between the three schemas are not reducible to storage, and the interpretation of Sections 5.1 and 3.2 depends on them.

1. **Keys and indexes.** The `id BIGSERIAL PRIMARY KEY` on the plain tables has no counterpart in the materialised views. Part of the disk saving attributed to MV in Section 4.3 is therefore the absence of that column and its index rather than a property of materialisation, and PT gives the planner an index MV does not.
2. **Element keys.** RV and MV declare `Message KEY(id)` and `KEY (...)` clauses on five edge tables, because a view exposes no primary key from which they could be inferred; PT declares none. The difference extends to `Person_knows_Person`, a keyed base table in all three modes, declared `KEY (Person1Id, Person2Id)` under RV and MV and without a `KEY` clause under PT.
3. **Which source relations survive.** PT does not create `Comment_replyOf_Comment`: it folds that relation into `Comment_replyOf_Message` and loads `Comment_replyOf_Post` into two tables, while RV and MV keep it as a base table [5]. PT therefore defines 30 tables against 26. The disk differential between PT and RV in Section 4.3 is the net effect of all of this, not the cost of the five derived relations alone.
4. **A missing label.** `City_isPartOf_Country` is the only edge table declared without a `LABEL` clause, in all three modes [5], so it is reachable only by the label PostgreSQL derives from the table name. Q1 and Q3 name it accordingly, while every other edge in those queries uses a short label such as `hasMember`.

`Person_knows_Person` is not among the five in any mode. It is a base table throughout, loaded from a single CSV file with one row per undirected edge, as the `pgq` queries require (Section 6.3, item 4). The DDL issues one `COPY` where the relational DDL issues two, one per column order [5].

3.5 Metrics collected and the measurement protocol

Testing was performed on a cloud container platform. Instance size was varied with scale factor (Appendix A). **Campaign 2 held the instance flavour, the timeout, the repetition count and the query order constant across every scale factor it measured, so the only thing varying across that series is the data.** The SF 100 rows come from campaign 1, on a larger instance and a longer timeout, and are not part of that series (Section 3.4). Comparisons between storage strategies at a fixed scale factor are controlled too, and those carry the conclusions of Section 6.2.

The following metrics were collected:

- **Load time** — wall-clock duration of schema population from CSV sources, including materialisation of PT tables or MV views where applicable.
- **Disk footprint** — total on-disk size of the database cluster after loading.
- **Query latency** — end-to-end time from statement submission to full result consumption.
- **Peak RSS** — the most RAM the serving backend held at any moment during query execution, taken from the kernel's high-water mark rather than sampled.
- **Failure rate** — the proportion of query attempts (9 queries × repetitions) that did not return within the timeout.

Two measurement campaigns: initial vs controlled protocol

The measurements were made in two campaigns, under different protocols. This report draws on the second throughout, except where it says otherwise. Both are published in full [5], each with the harness that produced it and a machine-readable statement of its conditions.

	Campaign 1, initial	Campaign 2, controlled
Repetitions per query	3	5
Individual repetition times	not recorded	recorded, timeouts marked
`ANALYZE` before measuring	never run	run once per configuration
Query order	filesystem order, varies between configurations	sorted, identical everywhere
Query plans	captured by a separate invocation, in a different statistics state	captured per query, in the state its repetitions ran in
Statistics state verified	no	fingerprinted before and after the run
Coverage	SF 0.1–100, three pre-releases for SQL/PGQ, beta 3 for relational	SF 0.1–30, same coverage otherwise

Everything in this report rests on campaign 2, with one exception and one comparison. The exception is SF 100, measured only in campaign 1 and reported separately in Section 4.4. The comparison is the effect of collecting statistics, which can only be seen by

setting the two campaigns against each other, since campaign 2 collects them everywhere. Sections 5.2 and 5.4 do that, and treat coverage as the comparable quantity rather than latency.

That restriction is deliberate. The two campaigns differ in three conditions at once, so a latency difference between them cannot be attributed to any one of the three. Whether a query returns within the timeout is robust to repetition count and query order; how long it takes is not, since query order determines what has warmed the cache. Section 6.1 quantifies that: under campaign 1 two configurations differing in nothing relevant diverged by a factor of 18, and under campaign 2 the same comparison stays within 0.61–1.14.

What a repetition represents: warm-buffer, cold-backend latency

Each repetition opens its own connection, so backend-local state is cold every time: the plan cache, the catalogue cache and `VmHWM` all begin afresh. Shared state does not behave that way, since `shared_buffers` and the operating-system page cache survive backend exit, so the first repetition warms them and those that follow benefit. A median of five therefore approximates a warm-buffer, cold-backend latency. The query timeout was 10 minutes at SF 0.1–30 and 60 minutes at SF 100, and all latencies quoted below are medians over the repetitions that completed, unless stated otherwise.

4. Results

4.1 Coverage: which queries complete, and where

Query coverage by approach and scale factor

Table 6 gives the number of storage strategies, out of three, in which each query returned a result. Bold type marks a query that one approach completes and the other does not.

Partial failures are counted as follows. A configuration is deemed to have completed a query if it returned the correct result in at least one of its five repetitions. They are excluded from every latency aggregate (Sections 4.2, B.1, B.2), because a median taken over a set containing a driver-supplied zero is not a latency measurement. They are not excluded from coverage, which addresses only whether the query is feasible at all. Exactly one cell is affected under the controlled protocol: Q6 under relational RV at SF 10, where one of five repetitions timed out and the other four ran in 594.1 to 598.4 s against a 600 s limit (Section 6.3). It is the only cell marked `*` in this report, in Table 25. The same rule governs Section 4.2.

Table 6. Query coverage by approach and scale factor.

Query	SF 0.1	SF 0.3	SF 1	SF 3	SF 10	SF 30
Q1	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3
Q2	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	SQL 3 · PGQ 3	SQL 3 · PGQ 3	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**
Q3	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**
Q4	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 2
Q5	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3
Q6	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	SQL 0 · PGQ 0
Q7	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3
Q8	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 3	SQL 3 · PGQ 2	SQL 3 · PGQ 2
Q9	SQL 0 · PGQ 0	SQL 0 · PGQ 0	**SQL 3 · PGQ 0**	**SQL 3 · PGQ 0**	SQL 0 · PGQ 0	SQL 0 · PGQ 0

Q3, Q6 and Q9 never complete under SQL/PGQ at any of these six scale factors, nor indeed in any of the 21 SQL/PGQ configurations. The relational approach completes Q3 at all six, in 0.082–362 s, and Q6 at five of six, in 3.0–587 s. Q2 completes relationally at every scale factor and under SQL/PGQ only at SF 1 and SF 3. There is no query that the relational approach fails and SQL/PGQ completes: where the relational approach fails, on Q9 at SF 0.1, SF 0.3 and SF 10 and on Q6 and Q9 at SF 30, SQL/PGQ fails also.

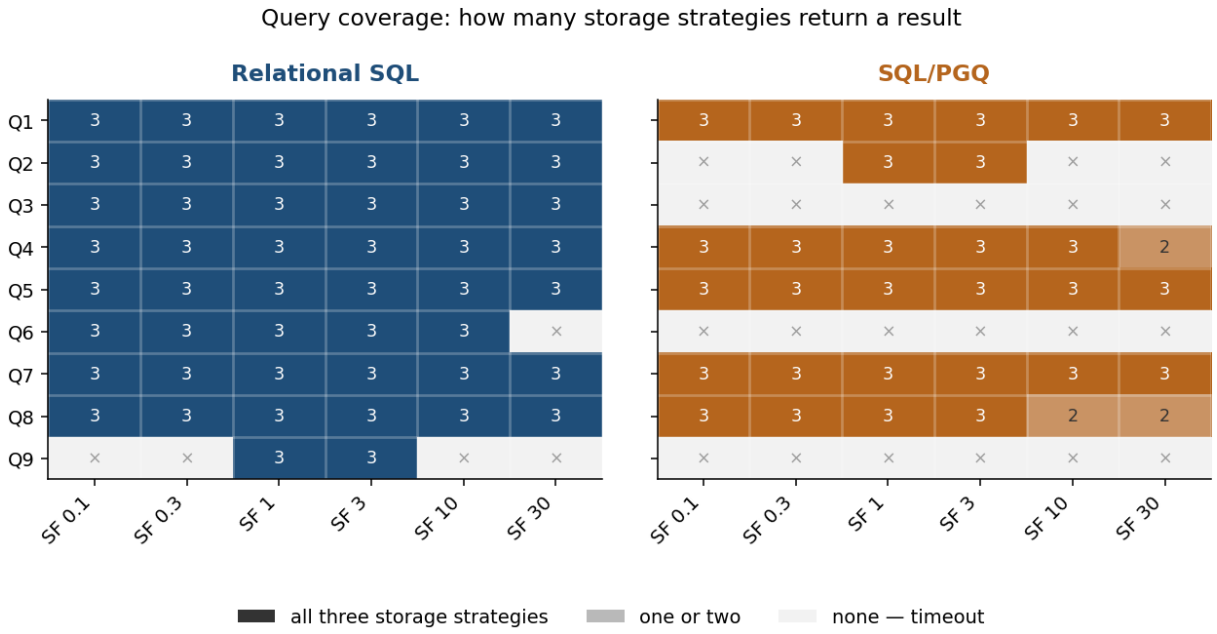

Figure 1. Query coverage of the two approaches. Each cell counts the storage strategies that returned a result. The relational approach is complete but for two queries: Q9 returns only at SF 1 and SF 3, and Q6 fails at SF 30. The SQL/PGQ approach has three empty rows throughout, and loses Q2 outside SF 1 and SF 3.

Failure rate

Failure rate — the share of the nine queries that did not return within the timeout — is reported per configuration in the columns of Tables 24 and 25, and is not tabulated separately here. Because coverage does not vary between repetitions of a query, it is with one exception a restatement of coverage in percentage points: every SQL/PGQ value is a multiple of 11.11 pp, so no SQL/PGQ configuration lost an individual repetition. The exception is relational RV at SF 10, at 13.33 %, where one of five repetitions of Q6 timed out.

The floor SQL/PGQ reaches is more informative than its never falling below it. At SF 1 and SF 3 all three strategies attain 33.33 %. That figure is exactly Q3, Q6 and Q9 — the three queries the benchmark designed to require worst-case optimal joins or factorisation, which PostgreSQL implements for no query, graph-shaped or not. At those two sizes SQL/PGQ answered everything else. Under PT or MV, Q1, Q4, Q5, Q7 and Q8 complete at every scale factor measured, the largest being SF 100 at 2.1 billion edges and 252 GB of database, though the SF 100 completions take 18 to 38 minutes.

The gap to the relational approach, which never exceeds 22.22 % and reaches 0 % in six of its eighteen configurations, is therefore Q2 outside SF 1 and SF 3, plus Q3 and Q6. The relational form answers those in 0.082–362 s and 3.0–588 s. Comparing the best configuration of each approach, the gap is 33.3 pp at every scale factor except SF 30, where it narrows to 22.2 pp. Both approaches are at their worst at their largest scale factor, and the two queries the relational approach loses at SF 30, Q6 and Q9, were never available under SQL/PGQ at any scale factor.

4.2 Latency: SQL/PGQ vs relational, across storage strategies

Aggregate latency: SQL/PGQ vs the relational implementation

Table 7. Aggregate latency of the two approaches: the geometric mean over the queries completed without partial failure by all six configurations at each scale factor, three storage strategies in each approach. The common set is dictated by SQL/PGQ coverage in every row. Q6 is additionally excluded at SF 10, where one of the five repetitions under relational RV timed out (Section 6.3). The geometric mean is used because it weights each query equally, irrespective of absolute runtime; an arithmetic mean over a set spanning four orders of magnitude would be decided by the slowest member.

SF	Common set	SQL PT	SQL RV	SQL MV	PGQ PT	PGQ RV	PGQ MV
0.1	Q1, Q4, Q5, Q7, Q8	0.278	**0.190**	0.384	**0.385**	0.777	0.556
0.3	Q1, Q4, Q5, Q7, Q8	**0.748**	0.910	0.761	**1.004**	3.293	1.247
1	Q1, Q2, Q4, Q5, Q7, Q8	2.278	4.907	**1.861**	4.910	9.670	**4.747**
3	Q1, Q2, Q4, Q5, Q7, Q8	7.060	18.200	**7.046**	**12.780**	32.847	13.189
10	Q1, Q4, Q5, Q7	**33.190**	54.366	37.960	**44.852**	126.468	53.786
30	Q1, Q5, Q7	149.893	217.308	**134.550**	172.859	326.083	**168.779**

The relational approach is faster in aggregate at every one of the six shared scale factors, by a factor of 1.25–2.55, comparing the best strategy of each approach. SF 100 has no relational counterpart and is excluded from every cross-approach figure in this report. The ratio does not vary monotonically with scale: it is 2.03, 1.34, 2.55, 1.81, 1.35 and 1.25 from SF 0.1 to SF 30.

The composition of the common set changes across those rows, contracting from six queries at SF 1 and SF 3 to three at SF 30. The set common to all six rows is Q1, Q5 and Q7 — and Q1 is the single query SQL/PGQ wins, so an aggregate over that intersection still carries SQL/PGQ's only advantage. Removing it leaves the fixed pair Q5 and Q7, which is the harsher comparison and the one worth stating alongside the per-row figures. Recomputing every row over that pair gives 2.55, 1.85, 2.90, 2.35, 1.98 and 2.03, uniformly above the per-row figures, since excluding Q1 removes SQL/PGQ's only advantage. The per-row series is therefore the more conservative of the two, and neither establishes a trend on six points. The defensible statement is that the margin lies between 1.25 and 2.55 on the per-row sets and between 1.85 and 2.90 on the fixed pair, with any dependence on scale unresolved.

The magnitude of the margin depends on granting each approach its best strategy: fixing either approach to a single strategy biases the comparison in whichever direction that strategy happens to lean.

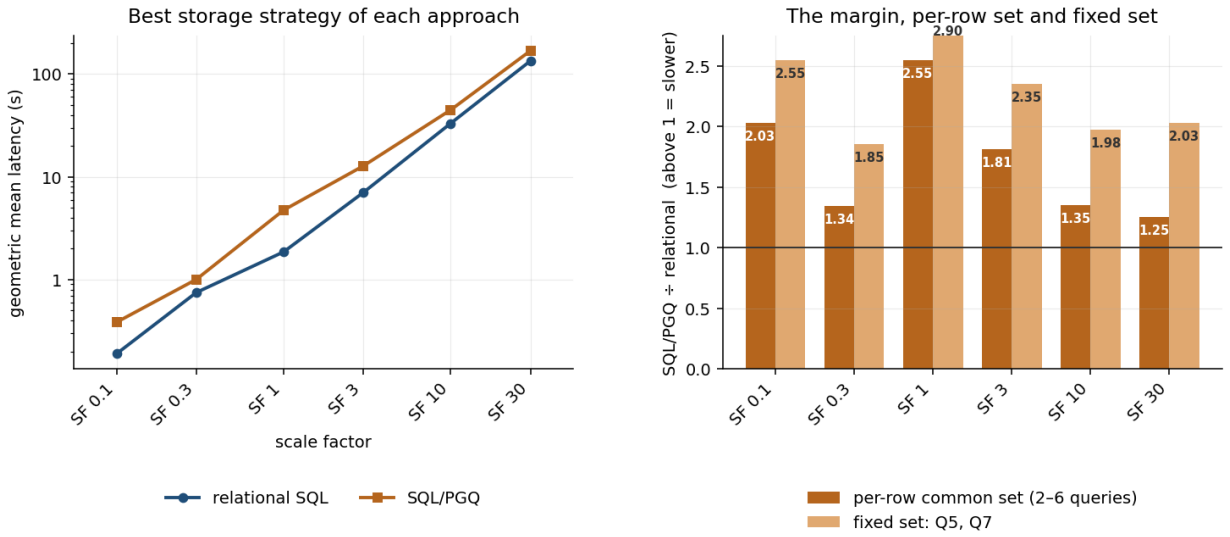

Figure 2. Aggregate latency, best strategy of each approach (left, both axes logarithmic), and the ratio between them on the per-row common set, in dark, against the fixed pair Q5 and Q7, in light (right). Both series are computed from Table 7. Neither is monotone in scale; the fixed pair runs higher at every scale factor because it excludes Q1, the only query SQL/PGQ wins.

Latency, best strategy of each approach

Table 8. Median latency in seconds, taking the fastest of PT, RV and MV within each approach. Partial failures count as completions under the rule stated in Section 4.1; no cell in this table is affected.

SF	Query	SQL best	SQL/PGQ best	Faster	Margin
0.1	Q1	0.668	0.418	**PGQ**	1.60×
0.1	Q2	0.055	timeout	SQL	—
0.1	Q3	0.082	timeout	SQL	—
0.1	Q4	0.116	0.309	SQL	2.67×
0.1	Q5	0.093	0.416	SQL	4.45×
0.1	Q6	2.938	timeout	SQL	—
0.1	Q7	0.218	0.317	SQL	1.46×
0.1	Q8	0.156	0.495	SQL	3.17×
0.1	Q9	timeout	timeout	—	—
0.3	Q1	1.877	1.046	**PGQ**	1.79×
0.3	Q2	0.127	timeout	SQL	—
0.3	Q3	0.428	timeout	SQL	—
0.3	Q4	0.492	0.889	SQL	1.81×
0.3	Q5	0.383	1.057	SQL	2.76×
0.3	Q6	13.122	timeout	SQL	—
0.3	Q7	0.690	0.871	SQL	1.26×
0.3	Q8	0.780	1.190	SQL	1.53×
0.3	Q9	timeout	timeout	—	—

SF	Query	SQL best	SQL/PGQ best	Faster	Margin
1	Q1	9.071	5.472	**PGQ**	1.66×
1	Q2	0.516	3.292	SQL	6.38×
1	Q3	1.689	timeout	SQL	—
1	Q4	1.734	4.478	SQL	2.58×
1	Q5	1.111	4.333	SQL	3.90×
1	Q6	44.164	timeout	SQL	—
1	Q7	2.051	4.051	SQL	1.98×
1	Q8	2.245	6.157	SQL	2.74×
1	Q9	103.397	timeout	SQL	—
3	Q1	39.814	19.286	**PGQ**	2.06×
3	Q2	1.961	6.892	SQL	3.51×
3	Q3	7.784	timeout	SQL	—
3	Q4	5.891	11.254	SQL	1.91×
3	Q5	4.018	10.405	SQL	2.59×
3	Q6	153.654	timeout	SQL	—
3	Q7	7.723	16.465	SQL	2.13×
3	Q8	8.192	16.522	SQL	2.02×
3	Q9	343.445	timeout	SQL	—
10	Q1	150.542	68.390	**PGQ**	2.20×
10	Q2	6.688	timeout	SQL	—
10	Q3	48.627	timeout	SQL	—
10	Q4	21.260	39.953	SQL	1.88×
10	Q5	13.862	35.465	SQL	2.56×
10	Q6	587.906	timeout	SQL	—
10	Q7	27.352	41.765	SQL	1.53×
10	Q8	29.113	59.173	SQL	2.03×
10	Q9	timeout	timeout	—	—
30	Q1	507.957	244.473	**PGQ**	2.08×
30	Q2	20.864	timeout	SQL	—
30	Q3	298.109	timeout	SQL	—
30	Q4	75.434	148.785	SQL	1.97×
30	Q5	49.655	130.722	SQL	2.63×
30	Q6	timeout	timeout	—	—
30	Q7	96.102	149.551	SQL	1.56×
30	Q8	111.201	209.040	SQL	1.88×
30	Q9	timeout	timeout	—	—

Across the 54 (scale factor, query) cells, the relational approach is faster in 26 and SQL/PGQ in 6. Only the relational approach completes in 17. Both fail in 5: Q9 at SF 0.1, SF 0.3 and SF 10, and Q6 and Q9 at SF 30. There is no cell in which only SQL/PGQ completes. All six SQL/PGQ advantages involve Q1, one at each scale factor, by a factor of 1.60–2.20. In every other cell the relational form is either faster, by a factor of 1.26–6.38, or is the only approach that completes.

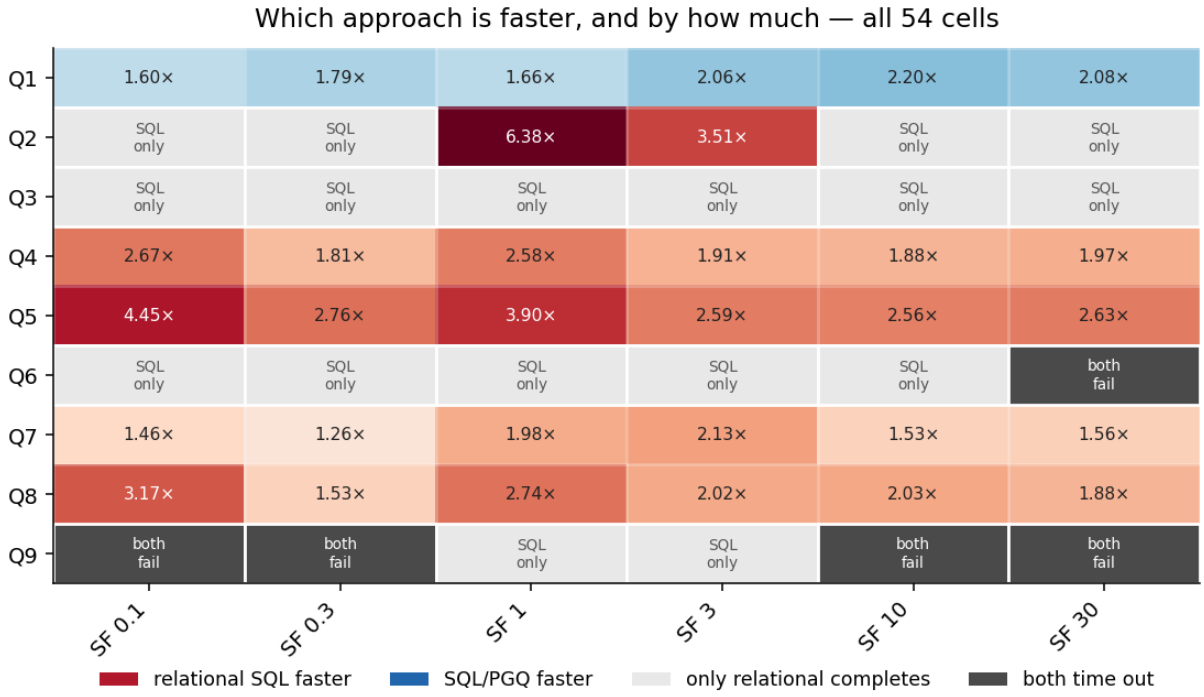

Figure 3. Relative speed of the two approaches across all 54 cells. Red denotes a relational advantage and blue an SQL/PGQ advantage; the figure in each cell is the ratio between the best strategies of the two approaches. Q1 is the only blue row, and Q3, Q6 and Q9 are grey or black throughout.

Between one third and two thirds of the queries time out in every configuration. A mean taken over all queries is therefore not meaningful. The timeouts are right-censored, and a strategy that fails more queries would appear artificially fast. Table 9 therefore reports the geometric mean over the set of queries completed by all three strategies, with that set stated explicitly in each row.

Aggregate latency across storage strategies (SQL/PGQ only)

Table 9. Aggregate SQL/PGQ latency, as the geometric mean over the common query set. At SF 100 that set is Q1 and Q7, the two queries RV completes, and the timeout there is 60 minutes rather than 10, so the last row is not directly comparable with the others (Section 6.3).

SF	Common query set	PT	RV	MV	RV/PT	MV/PT
SF 0.1	Q1, Q4, Q5, Q7, Q8	**0.385 s**	0.777 s	0.556 s	2.02×	1.44×
SF 0.3	Q1, Q4, Q5, Q7, Q8	**1.004 s**	3.293 s	1.247 s	3.28×	1.24×
SF 1	Q1, Q2, Q4, Q5, Q7, Q8	4.910 s	9.670 s	**4.747 s**	1.97×	0.97×
SF 3	Q1, Q2, Q4, Q5, Q7, Q8	**12.780 s**	32.847 s	13.189 s	2.57×	1.03×
SF 10	Q1, Q4, Q5, Q7	**44.852 s**	126.468 s	53.786 s	2.82×	1.20×
SF 30	Q1, Q5, Q7	172.859 s	326.083 s	**168.779 s**	1.89×	0.98×
SF 100	Q1, Q7	**1 434.040 s**	1 806.134 s	1 490.968 s	1.26×	1.04×

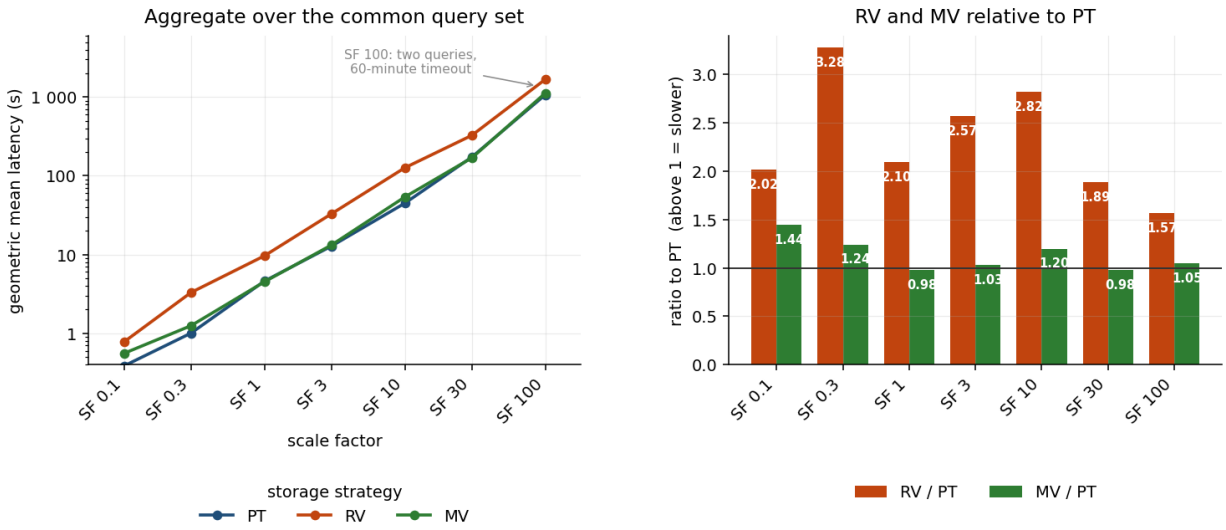

Figure 4. Aggregate latency and the ratios to PT. PT and MV are indistinguishable at this resolution. RV lies above both at every scale factor, including SF 100, where the common set is two queries and the timeout is 60 minutes rather than 10.

The common set contracts as scale increases, because RV fails progressively more queries. It completes five at SF 0.1 and SF 0.3, six at SF 1 and SF 3, four at SF 10, three at SF 30 and two at SF 100.

RV is dominated at every scale factor, by a factor of 1.89–3.28. There is no scale at which deferring materialisation is advantageous in aggregate.

PT and MV exhibit no stable ordering. The MV/PT ratio ranges over 0.97–1.44 with no trend in scale, as shown in Table 10.

Table 10. Ratio of MV to PT aggregate latency by scale factor.

Scale factor	MV/PT	Leader
SF 0.1	1.44×	PT
SF 0.3	1.24×	PT
SF 1	**0.97×**	**MV**
SF 3	1.03×	PT
SF 10	1.20×	PT
SF 30	**0.98×**	**MV**
SF 100	1.04×	PT

PT has the lower geometric mean at five of the seven scale factors and MV at SF 1 and SF 30. At several of them the margin falls inside run-to-run variation, so neither can be recommended as the default. At those same two scale factors, SF 1 and SF 30, the two lie within 3 % of each other, which is within run-to-run noise (Section 6.3). The two modes hold the same rows in the same five `Message`-derived relations, and `knows` is a base table in both. They are not otherwise identical. The PT copies carry a primary-key index that the materialised views lack, and the two declare different element keys in the property graph (Section 3.4). Either asymmetry is visible to the planner, so the absence of a stable ordering cannot be attributed to plan choice alone. The corresponding RV/PT ratios are the column of Table 9.

4.3 Disk footprint and load time by storage strategy

Rows loaded per scale factor

The figures in this subsection are ours, not LDBC's. They are counted from `pg_stat_user_tables` in each configuration before `ANALYZE` ran, where the row counter is the exact number of rows inserted rather than a sample estimate, and the snapshots are published with the results [5]. Table 11 gives them for the `projected` layout under plain tables; the other two storage strategies hold the same rows in different objects (Section 3.4).

Table 11. Rows loaded, by scale factor, in the SQL/PGQ schema under plain tables. Counted from the statistics snapshots [5]. The last column compares stored edge rows with the graph edge count of Table 3.

Scale factor	Vertex rows	Edge rows	All rows	Edge rows ÷ graph edges
SF 0.1	432 235	2 584 791	3 404 952	1.24×
SF 0.3	1 179 535	8 025 717	10 316 594	1.30×
SF 1	3 955 790	29 451 504	37 216 901	1.33×
SF 3	11 257 797	89 237 679	111 430 223	1.35×
SF 10	35 495 427	297 140 582	367 308 385	1.37×
SF 30	103 176 721	902 057 423	1 106 370 278	1.39×
SF 100	326 040 938	2 985 068 429	3 631 643 094	1.40×

Two things are worth reading off this table. The vertex counts agree with LDBC's to within 0.004 % at every scale factor, and exactly at the three smallest, which is a check on the load rather than a coincidence. And stored edge rows run 1.24 to 1.40 times the graph's edge count, rising steadily with scale. The reason is given in Section 3.3: the layout duplicates some edges and omits others, so rows in the database and edges in the graph are related but not equal. The row counts are the figure to read the disk sizes below against.

On-disk footprint by storage strategy

Table 12. On-disk database size by SQL/PGQ storage strategy and scale factor, in binary units. The relational approach is excluded for the reason given in Section 3.3.

Scale factor	PT	RV	MV	RV vs PT	MV vs PT
SF 0.1	253 MB	157 MB	218 MB	−37.9 %	−13.8 %
SF 0.3	738 MB	442 MB	628 MB	−40.1 %	−14.9 %
SF 1	2630 MB	1560 MB	2236 MB	−40.7 %	−15.0 %
SF 3	7884 MB	4663 MB	6692 MB	−40.9 %	−15.1 %
SF 10	25 GB	15 GB	22 GB	−40.0 %	−12.0 %
SF 30	77 GB	45 GB	65 GB	−41.6 %	−15.6 %
SF 100	252 GB	147 GB	213 GB	−41.7 %	−15.5 %

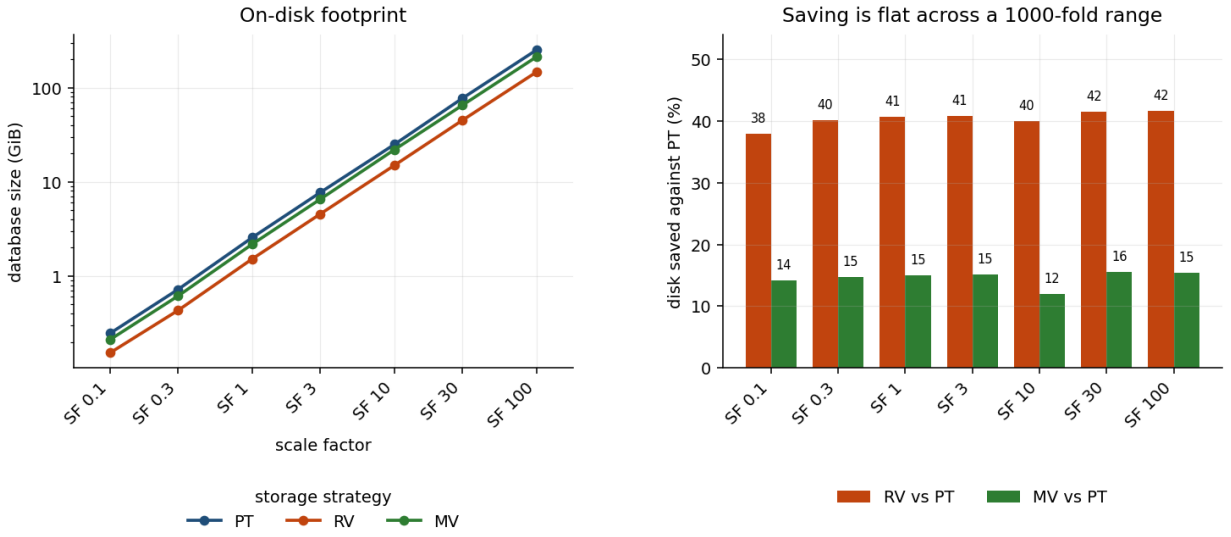

Figure 5. On-disk footprint by storage strategy and scale factor. Left: absolute size, both axes logarithmic. Right: saving relative to PT; both ratios remain flat across the whole range rather than converging.

PT is the largest representation at every scale, as expected, since it stores the five derived relations in full. RV saves 37.9–41.7 % and MV 12.0–15.6 %, at every scale factor. Both savings are flat across a thousandfold range of data volume: materialised views occupy roughly 84–88 % of the space required by equivalent base tables, and non-materialised views about 58–62 %.

The saving attributed to MV is not due to materialisation alone. The derived tables in PT mode carry a surrogate `id BIGSERIAL PRIMARY KEY` and its btree index. The materialised views carry neither (Section 3.4). An unquantified portion of the 12.0–15.6 % is therefore the cost of that column and index, rather than of storing the rows as tables. The RV saving is unaffected, since RV stores no rows for these relations at all.

Loading is where the choice of storage strategy shows most directly, and at the top of the range it stops scaling. Up to SF 30 load time grows roughly with the data: from SF 10 to SF 30, three times the data costs PT 3.37 times the load time, RV 3.55 and MV 3.09. From SF 30 to SF 100 the picture changes sharply. There 3.33 times the data costs 7.06, 7.24 and 7.80 times, which is about twice worse than linear. In absolute terms that is just over three hours to load plain tables at SF 100, against just under two for regular views.

RV loads fastest at every scale factor; MV and PT trade places. RV is ahead of PT by a factor of 1.49–1.95, while MV ranges from 0.89 to 1.40, taking longer than PT at some scale factors and less at others. Deferring the union is therefore worth roughly a third of the load time at every size measured, and materialising it once is worth rather less.

Load time by storage strategy

Table 13. Load times for the three SQL/PGQ storage strategies. The relational approach is excluded for the reason given in Section 3.3.

Scale factor	PT (s)	RV (s)	MV (s)	PT/RV	PT/MV
SF 0.1	4.31	**2.89**	3.93	1.49×	1.10×
SF 0.3	12.37	**8.16**	13.85	1.52×	0.89×
SF 1	54.33	**27.80**	41.38	1.95×	1.31×
SF 3	141.53	**90.32**	109.38	1.57×	1.29×
SF 10	482.17	**270.12**	374.98	1.79×	1.29×
SF 30	1 626.57	**959.78**	1 158.32	1.69×	1.40×
SF 100	11 486.32	**6 951.36**	9 036.08	1.65×	1.27×

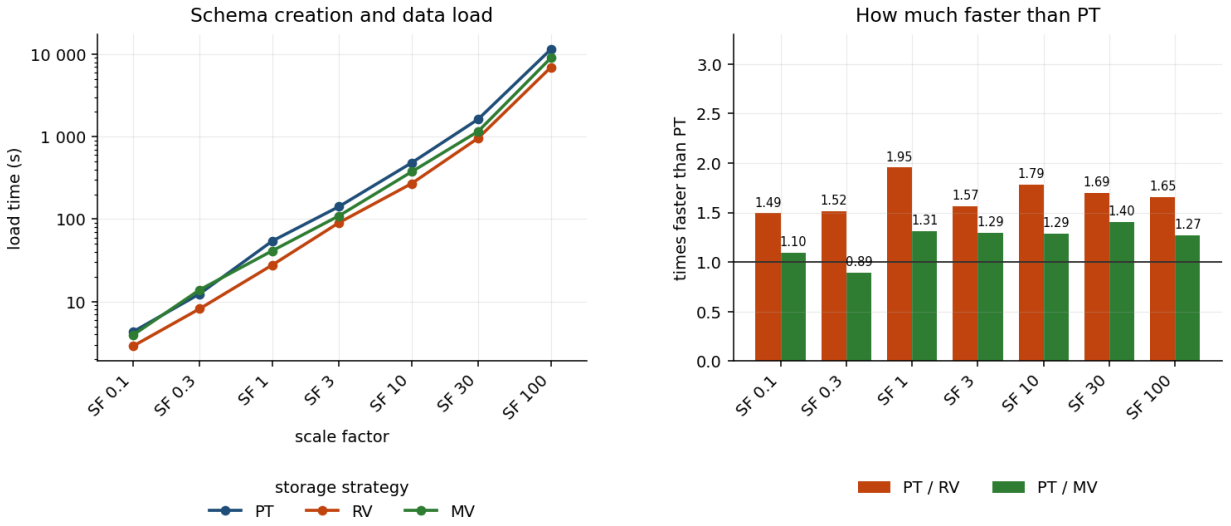

Figure 6. Load time by storage strategy and scale factor, from Table 13. RV is fastest at every scale factor; MV and PT trade places.

Amortizing load-time savings against extra query latency

Load is paid once; the query figures of Section 4.2 are paid on every execution. Dividing a load saving by the extra query time the strategy costs gives the number of passes over the query set after which the saving is gone. For RV that number is below one at every scale factor up to SF 10, 1.4 at SF 30 and 7.2 at SF 100. Its 1.49–1.95× load advantage is real and almost always irrelevant. For MV the arithmetic is mixed. At SF 1 and SF 30 there is nothing to repay, MV being the faster of the two on load and on queries alike. At SF 3 the saving is gone after about fifteen passes, and at SF 0.3 MV is simply worse on both axes. Load time therefore separates the three strategies most clearly and should weigh least in choosing between them.

Two qualifications on the figures. Each is a single measurement rather than a median, so the ratios carry more weight than the seconds. And the loss of scaling appears between SF 30 and SF 100, which is also where the hardware changes: SF 100 ran on a larger instance than SF 30, so the loss occurs despite more hardware rather than because of less.

Both expectations stated above survive in part and fail in part. The three strategies are compared dimension by dimension in Section 6.2, which is where the choice between them is made.

4.4 SF 100: the large-scale limit of SQL/PGQ

SF 100 is reported separately, and it is the one place in this report where the figures come from campaign 1: three repetitions, no `ANALYZE`, unsorted query order (Section 3.5). It does not belong to the comparison the rest of the report draws in any case. The relational approach was never measured there, so no cross-approach statement rests on

it, and its timeout is 60 minutes against the 10 applied elsewhere, so its coverage is not comparable with the other rows either. What it does establish is where the SQL/PGQ approach stops.

Table 14. SF 100: measured quantities by storage strategy. The timeout is 60 minutes.

Metric	PT	RV	MV	Best
Disk footprint	252 GB	**147 GB**	213 GB	**RV**
Load time	11 486 s	**6 951 s**	9 036 s	**RV**
Q1	**1 925.2 s**	1 945.8 s	1 985.3 s	**PT**
Q4	**1 346.4 s**	timeout	1 782.7 s	**PT**
Q5	1 306.4 s	timeout	**1 300.3 s**	**MV**
Q7	**1 068.2 s**	1 676.5 s	1 119.7 s	**PT**
Q8	2 303.4 s	timeout	**2 287.2 s**	**MV**
Geometric mean (Q1, Q7)	**1 434.0 s**	1 806.1 s	1 491.0 s	**PT**
Failure rate	**44.44 %**	77.78 %	**44.44 %**	**PT/MV**

Two things carry over from the smaller scale factors and one does not. PT and MV again complete the same set, and again cannot be separated: five queries each, failure rates equal, and per-query times within a few per cent except on Q4. RV again holds the smallest footprint and the fastest load, at 58 % of PT's disk and 1.65 times its load speed. What does not carry over is RV's viability: it completes two queries of the nine, against five for the other two, and Section 6.3 traces that to cardinality estimates that collapse under view expansion at this size.

Every success at SF 100 fell between 1 068 s and 2 303 s, which is 30 % to 64 % of the hour allowed. So no success sat close to the limit. How far beyond it the four failures lie is not known: a cancelled query reports no duration, and none was re-run under a longer limit.

4.5 Correctness: verification against the LSQB reference answers

LSQB publishes expected answers for every query at every scale factor, so that each returned count can be verified rather than merely timed. Every value reported in this document has been checked against `expected-output.csv` [4]. A second, independent check is available for one scale factor: the benchmark paper tabulates the number of matching subgraphs per query at SF 3, 10, 30 and 100 [1, Table 3]. Our SF 3 counts agree with all nine of its published figures to the precision printed there — 817.1 M for Q1, 3.3 M for Q2, 3.2 M for Q3, and so on down to 6.0 G for Q9.

The column headed "values checked" counts distinct (scale factor, query) results, that is, the number of independent comparisons against the reference. Because the three storage strategies agree everywhere, the same reference value is confirmed once for each strategy that completed the query; the rightmost column gives the total number of returned runs underlying those comparisons.

Table 15. Verification of returned aggregates against the LSQB reference answers.

Approach	Configurations	Values checked	Correct	Incorrect	Returned runs
Relational	18 (SF 0.1–30 × PT/RV/MV)	49	**49**	0	147
SQL/PGQ, SF 0.1–30	18	32	**32**	0	93
SQL/PGQ, SF 100	3	5	**5**	0	12

All 240 returned runs below SF 100 match the reference, as do the 12 at SF 100 in campaign 1. Within each approach the three storage strategies returned identical counts in every case, so that no observed defect can be attributed to PT, RV or MV. The agreement extends to the largest values encountered, Q1 at 38 292 918 372 and Q7 at 3 698 623 377, both at SF 100.

---

5. Where the implementation reaches its limits, and why

5.1 Unsupported pattern constructs, and the Q3 reproduction

Two of the nine patterns cannot be expressed in this implementation as the benchmark states them, because `GRAPH_TABLE` accepts a subset of the standard's pattern syntax. The analysis phase rejects eight constructions with `feature_not_supported`; Table 16 lists them, with the line numbers of the corresponding diagnostics in `src/backend/parser/parse_graphtable.c` at tag `REL_19_BETA3`.

Table 16. Pattern constructions rejected by the SQL/PGQ analysis phase.

Diagnostic	Line
`multiple path patterns in one GRAPH_TABLE clause not supported`	346
`element pattern quantifier is not supported`	246
`adjacent vertex patterns are not supported`	302
`path pattern cannot start with an edge pattern`	289
`path pattern cannot end with an edge pattern`	318
`edge pattern must be preceded by a vertex pattern`	294
`non-local element variable reference is not supported`	126
`unsupported element pattern kind`	281

The first entry is the consequential one. A comma-separated list of path patterns, with element variables shared between its members, is how the standard expresses a pattern that is not a single walk; the grammar accepts it and the analysis phase rejects it. The second entry, the absence of quantifiers, removes variable-length paths.

Q3 is the pattern most affected, and it was examined in four formulations at SF 0.1, the smallest database in the series at 253 MB, under the standard ten-minute timeout. Table 17 records the outcomes.

Table 17. Q3 at SF 0.1 under four `GRAPH_TABLE` formulations. The final row is the same computation as the fourth, expressed without `GRAPH_TABLE` and executed against the same database.

Formulation	Outcome
As the benchmark states it: five path patterns in one `GRAPH_TABLE`, sharing `country` and the three person variables	rejected during analysis, `multiple path patterns in one GRAPH_TABLE clause not supported`
Two `GRAPH_TABLE` expressions as common table expressions, joined five ways. This is the formulation measured throughout the report	no result within 600 s
The triangle as one closed pattern, with the country constraint applied outside `GRAPH_TABLE`	no result within 600 s
The triangle alone as one closed pattern, with no country constraint	no result within 600 s
The same unconstrained triangle as explicit `JOIN` operations over `Person_knows_Person`, on the same database	**200 280 rows in 1.459 s**

The final two rows are the same query, counted six times over because an undirected triangle is enumerated once per permutation of its vertices. So 200 280 is six times 33 380, and the 30 456 Q3 returns is six times the 5 076 triangles whose members share a country. The explicit form answers in 1.459 s. The `GRAPH_TABLE` form does not answer in 600 s, placing the gap at **more than a factor of 400**, with the true figure unknown because the measurement is censored.

Three conclusions follow. The failure of Q3 is not an artefact of the decomposition, since the undecomposed closed pattern fails as well. It is not a matter of data volume, since it fails on the smallest database in the series. And it is not a matter of the query being intrinsically expensive, since the same computation over the same data completes in under two seconds when the pattern is written as `JOIN` operations.

What remains must be located carefully, because there is no separate graph executor to which the behaviour could be attributed. PostgreSQL implements SQL/PGQ as a rewrite. The rewriter dispatches on the range-table entry kind `RTE_GRAPH_TABLE` [11]; `rewriteGraphTable()` then resolves each label expression to the concrete elements it matches and generates one ordinary `Query` per combination along the path [9], and `generate_union_from_pathqueries()` combines those into a union [10]. The graph itself is a catalogue entry of its own relkind, `RELKIND_PROPGRAPH` [12], rather than a view or a table: `DefineRelation` creates it with that relkind [14], and `RELKIND_HAS_STORAGE` does not list it [13], so it carries no storage of its own and the rows come from the base tables it is declared over. Cyclic patterns go through the same machinery. The difference between the two forms must therefore lie in the query the rewrite produces or in the plan chosen for it, and not in an execution path unique to graph patterns, because there is none.

That architecture is worth pausing on, because it sets the standard against which these limits should be read. There is no graph engine inside PostgreSQL 19 — a property graph is a declaration over existing tables, `GRAPH_TABLE` is a rewrite, and everything below it is the planner and executor that serve ordinary SQL. The patterns that stall are the ones the benchmark's authors single out as requiring worst-case optimal joins or factorised processing [1], techniques from the graph-processing literature that PostgreSQL implements for no query at all. Asking a cost-based relational optimiser to match those patterns is asking it to work outside the shape of problem it was designed for. On this

benchmark it does so for six of the nine queries — five of them at every scale factor of the controlled series, the sixth at two of six — returning the reference answer every time it returns at all. The three it loses are the three the benchmark's authors attribute to algorithms PostgreSQL does not implement, which points more strongly at a missing join strategy than at a defect spread across the feature.

The plans of Section 5.2 answer that in part. The rewrite produces a query scanning a median of 2.2 times as many relations as the explicit form, because each edge is joined back to the vertex relations its keys reference. For Q3 the ratio is 0.9, so the shape of the generated query does not account for the failure. That leaves the plan. Q3 reaches the timeout in both statistics states, and no executed plan exists for it, so which node runs away is not established (Section 6.4, item 2). Section 5.4 bounds the conclusion from one side: the same formulation did once complete, on beta 1 under the uncontrolled protocol, so the operator is capable of the pattern under some condition — though none measured here. The last row of Table 17 is the smallest self-contained reproducer in this report: two formulations of one computation on the smallest database in the series, one answering in 1.459 s and the other not at all.

5.2 Query plans: the effect of ANALYZE and the cost of the GRAPH_TABLE rewrite

A plan was captured for every query in every configuration, immediately before that query's repetitions and in the statistics state they ran in: 648 plans in all, 72 configurations of nine queries. They are `EXPLAIN` output without `ANALYZE`, so the query is not executed and a plan exists even where the query never finishes. Each set sits beside a snapshot of `pg_class` and `pg_stat_user_tables`, and the harness fingerprints that state before and after the run, so the claim that a plan and a timing share one state is checked rather than assumed. All of it is published [5], one JSON file per query under `runs/campaign-2-controlled/{version}/{approach}/analyze/{STRATEGY}-sf{N}/`; this report quotes aggregates and individual nodes rather than reproducing plan text, which for the larger patterns runs to hundreds of lines. Unlike the plans of campaign 1 (Section 3.5), these correspond to the timings reported in Section 4.2, so a plan and its latency can be read together.

Collecting statistics caused five previously completing cells to time out, while rescuing none. `ANALYZE` inspects a sample of each table and records row counts, most common values and a distribution histogram. The planner estimates from those records how many rows each step will produce, and chooses between candidate plans on the estimates. Running it is the standard remedy where a plan looks wrong. Measured against campaign 1, which ran without it, it moved no query from timeout to completion anywhere in the matrix. Five cells moved the other way. Four are Q9 in the relational approach. It completed at SF 0.1 in 4.0 to 5.5 s under all three storage strategies, and at SF 0.3 under MV in 21.7 s; it now exceeds 600 s in all four. The fifth is Q3 on beta 1, discussed in Section 5.4.

The plan chosen with statistics is the one Section 5.3 diagnoses: an anti-join estimated at one row, feeding a nested loop whose inner side is a sequential scan of an unindexed relation. Collecting statistics therefore does not uniformly improve plan quality for this workload, and any account of why these queries fail has to survive that.

The rewrite widens the query. The plans show what `GRAPH_TABLE` generates. A SQL/PGQ plan scans a median of 2.2 times as many relations as the equivalent `JOIN` plan for the same question, from 0.9 times on Q3 to 3.3 times on Q2. It contains a median of 31 nodes against 13.

The reason is visible in the scans themselves. Q6 relationally scans `Person_knows_Person` twice and `Person_hasInterest_Tag` once, three scans in all; the same question under SQL/PGQ scans `Person_knows_Person` twice, `Person` three times, `Person_hasInterest_Tag` once and `Tag` once, seven in all. The extra scans are the vertex tables. Each edge in the pattern is joined back to the relations its `SOURCE KEY` and `DESTINATION KEY` reference. The explicit form never needs that, because it joins edge relations to one another directly. Q2 shows the same at larger scale, ten scans against three.

Scan count is not what decides the latency, though, and Q1 is the internal control that shows it. Its SQL/PGQ plan scans fifteen relations against the relational plan's ten. It is nonetheless the faster of the two at every shared scale factor, by 1.60–2.20× (Section 6.1). What separates the two there is not how many relations are touched but how many rows pass through them: the executed plans of campaign 1 put 0.93 billion rows through the SQL/PGQ plan against 1.70 billion through the relational one. The median 2.2× in scans therefore describes the shape of the query the rewrite generates, not the cost of running it, and any account of the latency gap has to be made in rows rather than in scans. This report does not have executed row counts for the controlled campaign, so it states the constraint rather than settling the account (Section 6.4).

The three pre-releases plan alike. Of the 162 combinations of query, storage strategy and scale factor, 160 produce the same join methods over the same number of scans on beta 1, beta 2 and beta 3. The two that differ are Q1 at SF 3 under PT and under RV. That is independent support for the coverage result of Section 5.4: the versions are not merely returning the same answers, they are reaching them the same way.

One limit on what these plans can show. Taken without `ANALYZE`, they carry estimated row counts and no actual ones, so they cannot be used to measure how far an estimate misses. The earlier campaign captured executed plans for the queries that complete. It found the worst underestimate within a plan to have a median of 3.3×, with 32 of 249 plans containing a node underestimated by more than 100× and twelve by more than 1000×. Those figures come from a different statistics state and are quoted here only as an order of magnitude.

5.3 Cardinality misestimation and the Q9 plan-selection tie

Q9 is the query whose feasibility varies most across scale factors in Section B.2, completing at SF 1 and SF 3 and nowhere else, and the cause has been traced directly. The investigation was carried out against a PostgreSQL 19 beta 3 instance at SF 0.1, and the query text is the one published by the benchmark, `sql/q9.sql`, unmodified. The finding concerns relational planning and not SQL/PGQ.

Four variants of the same query were planned, all on data whose correct answer is 51 009 398. Table 18 records the inner side of the topmost join in each, which is where the cost is decided.

Table 18. Q9 at SF 0.1: the inner side of the topmost nested loop under four conditions.

Condition	Inner side of the top join	Estimated cost of one iteration	Rows	Outcome
Table statistics present	`Seq Scan on person_hasinterest_tag`	615.70	39 170	no result in 120 s
After an explicit `ANALYZE`	`Seq Scan`, plan otherwise unchanged	615.70	39 170	no result in 120 s
`enable_nestloop = off`	hash of the same table	615.70	39 170	51 009 398 returned
An index on `Person_hasInterest_Tag (PersonId)`	`Index Only Scan`	**0.72**	24	51 009 398 returned

Three observations follow, and together they account for the behaviour.

First, every one of the four plans estimates the anti-join that feeds the top join at **one row**, while its actual output is on the order of five million. Neither collecting statistics nor declaring extended statistics on the two columns of `Person_knows_Person` altered that estimate. Extended statistics did change how the anti-join is executed. They replaced a merge anti-join and its two sorts with a hash anti-join, and reduced the total estimated cost by a factor of 5.8. The row estimate was unchanged. That is consistent with the misestimate belonging to the join rather than to the correlation of two columns within one table.

Second, the outer input to the top join is estimated at one row. A nested loop over it therefore costs approximately one sequential scan of the inner table, and so appears no more expensive than a hash join. In the first session of that investigation the two plans were estimated at 353 669.59 and 353 669.93 respectively, a difference of 0.34 in a total of 353 670, or one part in a million. At that separation the choice between them is not a judgement the cost model is making; it is a tie broken by whatever rounding the estimates happen to produce.

Third, the two plans do not perform alike. The relational schema declares no indexes of any kind (Section 3.3). The inner side of the nested loop is therefore a full scan of 39 170 rows, repeated once per outer row. On the real cardinality that is on the order of 10^{11} tuple comparisons. Adding an index to the inner table replaces that scan with an index-only lookup, costed at 0.72 against 615.70, a factor of 855; the same nested-loop plan

then completes. The index was created purely as a diagnostic and is not part of the measured schema. Disabling nested loops instead also completes. The driver's own successful execution of Q9 at this scale factor took 4.06 s; its unsuccessful ones exceeded 600 s.

The three findings compose into a single account. A join-level misestimate of one row makes two plans indistinguishable to the cost model. The absence of indexes makes one of them far more expensive than the other. The outcome for any given execution is therefore settled arbitrarily. This is not a property of the query: the same query completes in seconds under either remedy. Nor is it a property of the data volume, since SF 0.1 is the smallest database in the series.

Two consequences bear on the rest of this report. The observation in Section 3.3 that the relational approach is the less physically supported of the two now carries a measured price: in this cell it is the difference between four seconds and not finishing.

Section 6.1 bounds the same problem from another direction, by comparing configurations that differ in nothing relevant to the query being run. Under this protocol they agree to within 0.61–1.14. Under the earlier, uncontrolled one they diverged by as much as a factor of 18, which is what a fixed query order removes.

And the limitation recorded in Section 6.3 is more serious than a matter of statistical power. Consider two plans that differ by one part in a million in estimated cost and by two orders of magnitude in execution time. A repetition is then not a sample from a distribution around a central value. It is a draw between two regimes, and a median describes neither. The recommendation to raise the repetition count and to capture `EXPLAIN` per repetition (Section 6.4) rests primarily on this.

Two caveats attach to the figures above. The cost comparison in the second observation is taken from a single session, in which both plans were obtained against one database state. Between sessions the row count of `Person_knows_Person` differed, at 36 270 against 30 652, so cost values are not comparable across them. And the index used in the fourth condition was created for diagnosis only; it is absent from the schema under which every measurement in this report was taken.

5.4 Beta 1, beta 2 and beta 3: a stable baseline, and what a beta series can and cannot show

The SQL/PGQ approach was measured on three PostgreSQL 19 pre-releases at **SF 0.1 through SF 30**, under conditions held constant throughout: same instances, dataset, schemas, query texts, `ANALYZE` before measuring, five repetitions, sorted query order and a 10-minute timeout. SF 100 was not measured for beta 1 or beta 2 and is excluded.

Coverage is identical across the three, query for query and configuration for configuration. Q3, Q6 and Q9 return nothing anywhere; Q2 completes in six configurations of eighteen, Q4 in seventeen and Q8 in sixteen; Q1, Q5 and Q7 complete in all eighteen. The best failure rate is 33.33 % in every version, and the per-scale-factor sequence is the same in all three. Latency and load time do differ, by the amounts in Table 19, but by less than the protocol can resolve.

That consistency is worth dwelling on, because it is not the null result it may look like. The three pre-releases agree query for query on what runs, and on the aggregate within 0.81–1.17 of one another. They produce the same join methods over the same number of scans in 160 of the 162 combinations of query, storage strategy and scale factor (Section 5.2). Nothing regressed between them. For a feature arriving in its first release that is the outcome to hope for. The behaviour measured here belongs to the implementation rather than to the particular build it was measured on, so a team evaluating SQL/PGQ can expect what it sees in one pre-release to hold in the next.

Plan choice is nonetheless not frozen across the series. Two cells changed between the pre-releases — Q1 at SF 3 under plain tables and under regular views — and Q1 is precisely the query where the pattern form outperforms the explicit joins. Two cells of 162 is not a trend and this report draws none from it, but the planner's choices over these generated queries were demonstrably still in motion while the series ran.

What a beta series cannot show is the feature's development trajectory, and the identity above should not be read as evidence about it in either direction. A pre-release window is for stabilisation: the optimiser work behind a release lands before its first beta, and work aimed at later releases is invisible to anyone measuring betas. Agreement between beta 1, beta 2 and beta 3 is therefore the expected outcome rather than a signal about how much is being done. What this report offers on that question is a baseline, not a verdict. It is a fixed, reproducible set of cases — the three patterns that never complete, the one that wins, and the storage comparison — against which the next release can be measured directly.

Table 19. SQL/PGQ across pre-release versions, SF 0.1 to SF 30.

Quantity	beta 1	beta 2	beta 3
Queries returning no result in any configuration	Q3, Q6, Q9	Q3, Q6, Q9	Q3, Q6, Q9
Configurations completing Q2 / Q4 / Q8, of 18	6 / 17 / 16	6 / 17 / 16	6 / 17 / 16
Best failure rate attained	33.33 %	33.33 %	33.33 %
Failure rate, best strategy, by scale factor	44.44, 44.44, 33.33, 33.33, 44.44, 44.44 %	identical	identical
Aggregate latency, best strategy, relative to beta 2	0.81–1.12	1.00	0.84–1.17
Median Q1 under PT, relative to beta 2	0.97–1.24	1.00	0.85–1.10
Load time under PT, relative to beta 2	0.96–1.11	1.00	0.85–1.07
On-disk footprint	identical to beta 2 in all 18 cells	baseline	differs in 1 of 18 cells, by 1 MB
Values disagreeing with the reference	none	none	none

Campaign 1 saw something the controlled protocol does not reproduce, and the difference is instructive. Measured without collecting statistics, beta 1 completed Q3 in one configuration of eighteen, under regular views at SF 0.3, returning the correct count in 1.504 s where beta 2 and beta 3 exceeded 600 s. At the time that looked like a regression between the first and second pre-release.

It is not one, or at least the measurements do not show it to be one. Under campaign 2 the same cell times out on beta 1 in all five repetitions, and coverage becomes identical across the three versions. The precise statement is therefore this: **Q3 was observed to complete on beta 1 under the initial protocol, and did not reproduce under the controlled protocol on beta 1, beta 2 or beta 3.** The two observations differ in protocol as well as in version, so they do not isolate a version change.

What they do establish is narrower and still useful. The pattern is executable by this implementation under at least one tested condition, in 1.5 s, although it does not complete under the controlled protocol on any of the three versions. The limitation reported in Section 5.1 is therefore not inherent to the operator. The likeliest candidate is the absence of statistics: the planner, working from defaults, chose a plan for that one configuration that the collected statistics subsequently argue against. Section 5.2 shows the same effect in the relational approach, where collecting statistics moved four cells from completion to timeout. The evidence available here does not identify the cause with certainty.

The aggregate of the best strategy places beta 1 within 0.81–1.12 of beta 2 and beta 3 within 0.84–1.17, moving in either direction with no ordering, so no claim of a latency change between versions is made.

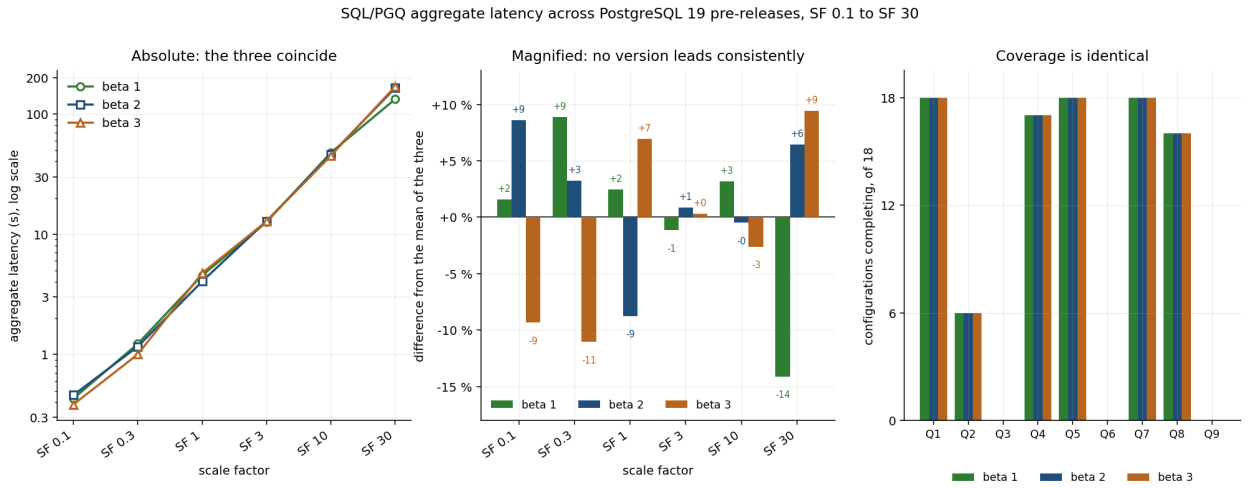

Figure 7. SQL/PGQ aggregate latency across PostgreSQL 19 pre-releases, SF 0.1 to SF 30.

How each panel is built. At every scale factor, one number is taken per version: the geometric mean of its median latencies over the query set all three versions complete at that scale factor, using whichever storage strategy is fastest for that version. That gives three numbers per scale factor, and no version is treated as a reference.

Left. Those three numbers plotted directly. The axis is logarithmic because they run from 0.38 s to 169 s across the range, which is a factor of 439. At that scale a difference of a few per cent between versions is narrower than the marker, so the three series lie on top of one another. That coincidence is the result: what the axis is too coarse to separate is what a reader would have to care about.

Centre. The same three numbers, magnified. For each scale factor the geometric mean of the three versions is computed, and each version is plotted as its percentage difference from that mean — so the bars at one scale factor sum to approximately zero by construction, and the panel says nothing about whether latency rose or fell with scale. What it does show is which version was fastest at each size and by how much. Two things are visible: the spread never exceeds −14 % to +9 %, and each version is above the mean at some scale factors and below at others, three or four sign changes each. A version that were genuinely faster would sit on one side throughout.

Right. The number of configurations, of the eighteen formed by three storage strategies and six scale factors, in which each query returned a result. The three bars are equal in every column, and Q3, Q6 and Q9 are absent in all three versions.

6. Discussion

6.1 SQL/PGQ against the relational implementation

Table 20. Summary of the cross-approach comparison.

Dimension	Verdict	Evidence
Coverage	Relational, decisively	Q3, Q6 and Q9 never complete under SQL/PGQ on this version, though Q3 did on beta 1 (Section 5.4); Q2 at two scale factors of six. The relational approach returns Q3 and Q2 at all six, answers every query at SF 1 and SF 3, and loses only Q9 elsewhere plus Q6 at SF 30
Aggregate latency	Relational, 1.25–2.55×	Best strategy of each approach over an identical query set, at all six shared scale factors (Section 4.2)
Per-query latency	Relational in 26 of 32 contested cells	SQL/PGQ wins six, all of them Q1, by 1.60–2.20× (Section 4.2)
Failure rate	Relational, by 22.2–33.3 pp	Relational never above 22.22 % and 0 % in six of eighteen configurations; SQL/PGQ never below 33.33 % (Section 4.1)
Best storage strategy	Undecided, and the ordering reverses between approaches	Under SQL/PGQ, PT leads five of seven scale factors (MV/PT 0.97–1.44×); relationally MV leads three of six (MV/PT 0.81–1.30×)
Worst storage strategy	RV in both approaches	1.89–3.28× above PT under SQL/PGQ, 0.79–1.92× relationally; the Q4 pathology appears in both, at 12.6–16.1× above PT under SQL/PGQ at SF 1–3 and 24.5× relationally at SF 3
Disk and load	Not compared across approaches	Different projections; within each approach RV is smallest and fastest to load (Sections 4.3 and 6.2)
Memory	Not compared across approaches	Metric does not separate shared-buffer fault-in from query-private allocation (Section B.3)

Two aspects of these results invite misreading. The first is that the latency gap is a factor, not a percentage. At 1.25–2.55, any claim that SQL/PGQ runs within a few tens of percent of an explicitly written plan requires the relational approach to be fixed to a single, non-optimal strategy. The second is that Q4 does not reverse with scale. The relational form is faster at all six shared scale factors, by a factor of 1.53–2.58, provided PT or MV is used. Only relational RV executes Q4 slowly, and a comparison drawn against that configuration alone would invert the result.

The SQL/PGQ implementation is therefore limited by coverage before it is limited by latency, and the queries it loses are the ones the benchmark was built to make hard. The LSQB authors identify Q3 as requiring worst-case optimal joins and Q6 and Q9 as calling for factorised processing [1], and PostgreSQL's planner and executor implement neither technique — for any query, graph-shaped or not.

The correspondence is exact, and it is the most encouraging thing in this comparison. The set SQL/PGQ fails is not an arbitrary subset of the benchmark, which would suggest something unsound in the operator; it is precisely the set whose difficulty the benchmark's authors attribute to two algorithms PostgreSQL has never had. Where those algorithms are not the obstacle, the pattern form runs — and runs correctly, and within a

small factor of joins written out explicitly, on an engine that resolves patterns entirely through a rewrite into relational plans (Section 5.1). Read as a first implementation of a paradigm the engine was not built around, that is a strong showing rather than a weak one. Read as a drop-in replacement for a graph database on cyclic workloads, it is not ready, and this report does not claim it is.

6.1.1 One column of Table 4 separates coverage exactly, on these nine queries

Which queries those are admits a sharper description than "the hard ones", and it is available from the query texts alone, before anything is run. Table 4 counts the `knows` edges each SQL/PGQ text matches. Set that column beside coverage and the separation is complete.

Table 21. `knows` edges matched against SQL/PGQ coverage, over the 18 configurations at SF 0.1–30. The final column is derived, not counted: it is the number of direction combinations the rewrite must enumerate, 2^n in the number of undirected `knows` edges the pattern matches. Whether the planner keeps all of them is a separate question, and the captured plans do not show the branches surviving as distinct nodes.

Queries	`knows` edges matched	Configurations completing, of 18	Union branches, 2^n
Q1, Q5, Q7	0	18	1
Q4	0	17	1
Q8	0	16	1
Q2	1	6	2
Q6, Q9	2	0	4
Q3	1, joined three times	0	8

Every query that matches no `knows` edge completes in at least 16 configurations of 18; every query that matches one or more completes in 6 or none. There is no overlap, and the ordering is monotone in the branch count: two branches still complete at two scale factors, four and eight never do.

Two qualifications before that column is used for anything. First, it is a separation over nine queries, not a complexity criterion: with n = 9 a perfect split is not a strong result, and `knows` is also the schema's largest many-to-many relation, so "matches `knows`" and "matches a large relation" are not distinguishable here. Second, the count itself is not quite the quantity that matters. Q3's row reads *1, joined three times*, because its text matches one `knows` edge in a CTE and then joins that expression to itself three ways; the combinatorial work is that of a triangle, not of a single edge. A plain count of matched edges therefore under-describes Q3, and would under-describe any pattern built by composition. What the column tracks well is the branch count in the final column, and that is the quantity the mechanism below concerns.

A mechanism is available for that, and the report has already established both halves of it separately. The rewrite resolves each label expression to the concrete elements it matches, generates one ordinary `Query` per combination along the path, and unions them (Section 5.1). And the `pgq` texts traverse friendship with the *undirected* pattern `-[k:knows]-`, which expands in both directions (Section 3.3). Each undirected `knows` edge

therefore doubles the number of generated branches, each branch being a multi-way join over the largest many-to-many relation in the schema. Q2 generates two, Q6 and Q9 four, and Q3's thrice-joined common table expression eight.

Two limits on how far this can be pushed. `knows` is not only the repeatedly traversed edge but also the largest many-to-many relation present, so "matches `knows`" is confounded here with "matches a large many-to-many edge", and nine queries cannot separate the two. And the relational implementation answers all four of the same queries — Q2 and Q3 at all six shared scale factors, Q6 at five of six and Q9 at two — so the difficulty is not created by `GRAPH_TABLE`. What the pattern form changes is that the same difficulty becomes the difference between running and never running. As a rule of thumb for a reader it is still the most useful one the data supports: **count the repeated many-to-many edges in the pattern before writing it.**

Q1 is the one systematic exception, faster under SQL/PGQ at every shared scale factor and the only query of the nine for which that holds. It is worth stating as a counterpoint rather than a curiosity: the advantage of the relational implementation is not universal, and the normalised graph projection can reduce work for patterns that span several separately stored edge relations. The plans show why. At SF 3 under PT the SQL/PGQ plan scans fifteen relations against ten, and runs in 19.6 s against 39.8 s. The extra scans are narrow two-column edge tables, while the relational form reaches the same edges through foreign-key columns of `Comment` and `Post`, the two largest relations in the schema. Executed plans from campaign 1, which recorded actual row counts, put 0.93 billion rows through the SQL/PGQ plan against 1.70 billion through the relational one.

That mechanism generalises into a rule a reader can apply before measuring anything. **SQL/PGQ is favoured where a pattern's edges live in narrow, separately stored edge relations and the relational alternative can reach the same edges only by scanning wide entity tables. It is disfavoured where the relational form reaches an edge through a foreign-key column of a table it was going to read anyway.** The normalised projection that SQL/PGQ requires is what makes the favourable case possible, so projection and query language are not independent choices. The evidence is one query. Q1 is also the longest pattern among those that complete, at seven edges against four or fewer for the rest, so the rule is a hypothesis worth testing against a reader's own workload rather than an established result.

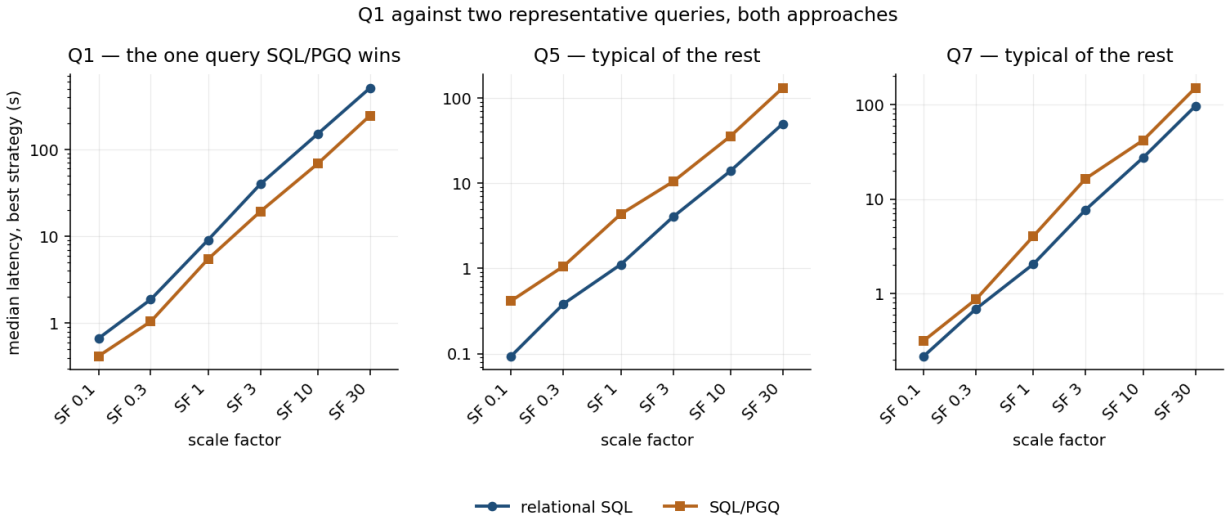

Figure 8. Q1 compared with two representative queries in both approaches. Q1 is the only query for which the SQL/PGQ line lies below the relational one, and it does so at all six shared scale factors; Q5 and Q7 exhibit the ordering found elsewhere.

6.1.2 A control group: separating view expansion from protocol noise

The DDL [5] admits a check on this comparison that the aggregate figures hide. Of the nine queries, four reference the derived relations and five do not. Q4 and Q7 reference four of the five, Q5 and Q8 reference two, and Q1, Q2, Q3, Q6 and Q9 reference none at all. For those last five the SQL text is identical under PT and RV, over base tables that are identical in both, so the two configurations should return the same number in the same time. They serve as a control on everything else in this section.

Mostly they do. Across the 25 cells in which a query referencing no derived relation completed under both PT and RV, the median ratio is 1.01 and fifteen sit within 6 % of parity. In the SQL/PGQ approach the same test gives a median of 1.04. That is the measurement working as intended.

The widest excursion is 1.14, and the narrowest 0.61. Under the earlier, uncontrolled protocol the same test produced ratios of 2.21, 3.92 and 18.07, reproducibly across three repetitions, on configurations that differ in nothing relevant to the query being run. Those excursions do not survive a fixed query order, which is the clearest evidence in this report that the earlier protocol, and not the planner, produced them.

What distinguishes the widest cells is which relations the queries read. Q1 and Q2 are the only two of the five that read `Comment` and `Post` directly, the largest base relations in the schema, and both excurse below parity at the small scale factors. The separation is not clean — Q3 and Q6 reach 0.82 at SF 0.3, as far from parity as Q1's 0.825 — so read this as a contributing effect rather than a sufficient explanation. Across the five queries the ratios span 0.82–1.14. The load explains the direction of the asymmetry: under PT and MV the derived relations are populated by `INSERT ... SELECT` and `CREATE MATERIALIZED VIEW ... AS SELECT` over `Comment` and `Post` [5], so those two relations are read during the load; under RV, `CREATE VIEW` reads nothing. Cache state at the first query is therefore not the same in the three modes, and Appendix A records that it was not controlled. Storage

strategy thus influences more than storage and plan choice: it also influences what the load leaves warm. The conclusions drawn about PT, RV and MV accordingly hold for the complete load-and-query protocol used here, and not for cold-cache query execution in isolation. The effect appears only at the two smallest scale factors, where the data still fits in cache.

Two conclusions follow for the rest of this report. Individual RV/PT ratios cannot be attributed to view expansion, since under the protocol of campaign 1 a pair of configurations differing in nothing relevant reached 18.07. And the attributable penalty is the one the control group isolates. Over the cells that do reference derived relations, the median ratio is 1.65 in the relational approach and 2.81 under SQL/PGQ, the SQL/PGQ figure being 2.88 if SF 100 is excluded. Q8 is the steadiest case, referencing two derived relations and costing 1.44–2.74 under SQL/PGQ at the four scale factors where RV completes it, and 0.52–1.76 relationally across all six.

6.2 Storage strategies compared: PT vs RV vs MV, and which to choose

Summary by dimension: disk, load time, latency, stability, memory

Dimension	Ordering	Detail
Disk	RV < MV < PT everywhere	RV saves 37.9–41.7 % over PT; MV saves 12.0–15.6 %, part of which is the absence of PT's primary-key index rather than materialisation (Section 3.4)
Load time	RV fastest everywhere; MV and PT trade places	PT/RV 1.49–1.95×, PT/MV 0.89–1.40×
Latency	PT ≈ MV < RV	PT leads five of seven scale factors and MV leads SF 0.3 and SF 1; MV/PT ranges 0.97–1.44× with no trend. RV is 1.89–3.28× above PT
Stability	PT = MV < RV	PT and MV have identical failure rates at every scale factor, including SF 100 at 44.44 %; RV is worse at SF 10 (55.56 %), SF 30 (66.67 %) and SF 100 (77.78 %)
Memory	MV ≤ PT ≪ RV	Every cell outside the RV/Q4 series stays within 88–401 MiB; RV on Q4 reaches 1 201 MiB

The latency deficit of RV is concentrated in two queries. On Q4 it is slower than PT by a factor of 3.96 at SF 0.1, rising to 6.73, 12.60 and 16.10 at SF 0.3, 1 and 3. It falls back to 9.22 at SF 10 and times out from SF 30. On Q5 the penalties from SF 0.1 through SF 30 are 2.89, 3.26, 2.66, 3.94, 3.95 and 3.91. A single outlying cell occurs on Q7 at SF 0.3, where RV is slower by a factor of 6.68, against 1.25–1.65 for Q7 at every other scale factor. On Q5 the penalty is broadly flat; on Q4 it peaks at SF 3, without a monotone trend.

RV is faster than PT in two cells, both at SF 1: Q1 by a factor of 1.11 and Q2 by 1.15. Deferring the union therefore helps almost nowhere, and where it does the margin is within what five repetitions can resolve.

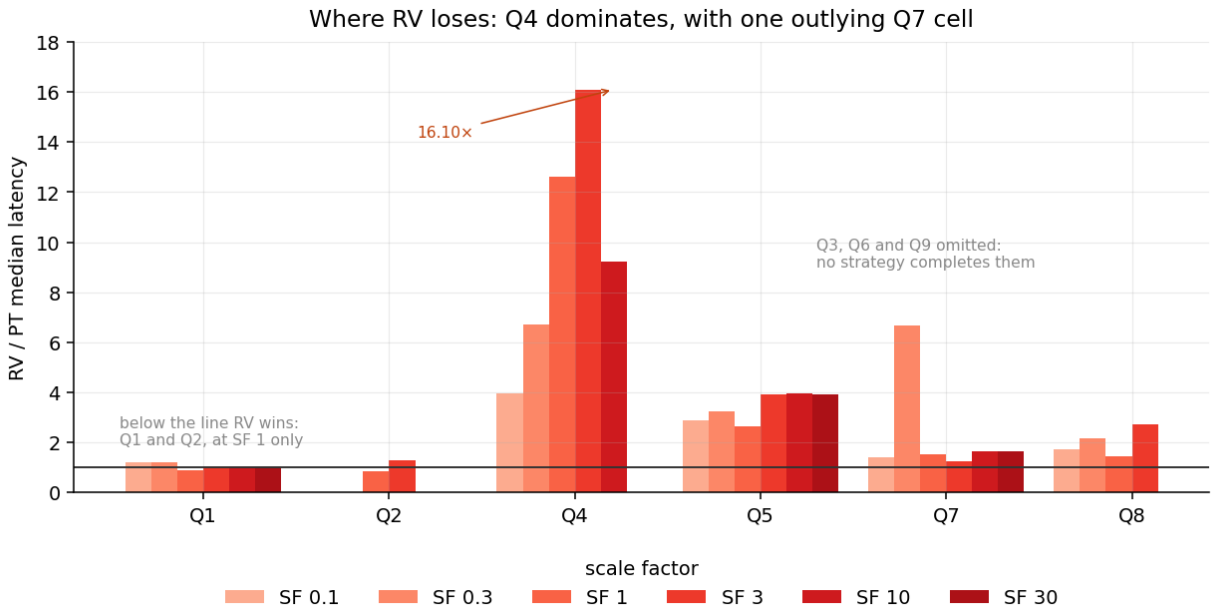

Figure 9. RV latency relative to PT, by query and scale factor. Bars above the line indicate that RV is slower. Q4 dominates and Q5 follows; the single tall Q7 bar corresponds to SF 0.3 and is an isolated cell. RV falls below the line in two cells only, both at SF 1: Q1 and Q2. Q3, Q6 and Q9 are omitted because no strategy completes them.

Within this approach the projection is held constant, so that disk footprint and load time are comparable across strategies. The relational schema supplies five derived relations: `Comment_replyOf_Message`, `Message_hasCreator_Person`, `Message_hasTag_Tag`, `Message_isLocatedIn_Country` and `Person_likes_Message`. It stores `Person_knows_Person` in both column orders. It declares no primary key, unique constraint or index anywhere, in any of the three modes. PT and MV are therefore both index-free here, in contrast to the SQL/PGQ approach (Section 3.4).

Relational approach: load, disk and latency by storage strategy

Table 22. Relational approach: load time and disk footprint by storage strategy.

SF	PT load (s)	RV load (s)	MV load (s)	PT disk	RV disk	MV disk	PT/RV load	PT/MV load
0.1	1.42	**0.50**	1.35	135 MB	**73 MB**	136 MB	2.84×	1.05×
0.3	5.03	**1.31**	4.25	392 MB	**198 MB**	393 MB	3.82×	1.18×
1	19.93	**6.81**	13.11	1400 MB	**695 MB**	1401 MB	2.92×	1.52×
3	49.71	**19.32**	44.42	4195 MB	**2083 MB**	4195 MB	2.57×	1.12×
10	169.94	**53.81**	162.74	14 GB	**6879 MB**	14 GB	3.16×	1.04×
30	608.55	**163.96**	445.14	41 GB	**20 GB**	41 GB	3.71×	1.37×

Table 23. Relational approach: aggregate latency, as the geometric mean over the queries completed without partial failure by all three strategies at each scale factor.

SF	Common set	PT	RV	MV	RV/PT	MV/PT
0.1	Q1–Q8	0.262 s	**0.208 s**	0.341 s	0.79×	1.30×
0.3	Q1–Q8	0.894 s	0.904 s	**0.878 s**	1.01×	0.98×
1	Q1–Q9	5.020 s	8.447 s	**4.090 s**	1.68×	0.81×
3	Q1–Q9	**15.494 s**	29.788 s	15.541 s	1.92×	1.00×
10	Q1, Q2, Q3, Q4, Q5, Q7, Q8	**27.364 s**	38.345 s	31.598 s	1.40×	1.15×
30	Q1, Q2, Q3, Q4, Q5, Q7, Q8	118.905 s	138.721 s	**104.073 s**	1.17×	0.88×

RV approximately halves the disk footprint, being smaller than PT by a factor of 1.85–2.08, and loads faster by a factor of 2.57–3.84. Both advantages exceed those observed in the SQL/PGQ approach, where the corresponding figures are 37.9–41.7 % and 1.49–1.95. The cost appears in latency, at a factor of 0.79–1.92 above PT, with no trend across the range.

MV and PT are indistinguishable on disk. They agree to within 1 MB wherever the figures are reported in megabytes, and exactly at the two scale factors reported in gigabytes. They load within a factor of 0.99–1.37 of each other. On latency the ordering here is the reverse of the SQL/PGQ approach: MV takes the aggregate at three of the six scale factors and PT at two, with the MV/PT ratio spanning 0.81–1.30. Taken together with Section 4.2, where PT leads five of seven, the conclusion available is the negative one, that neither constitutes a default, and that the ordering changes with the projection.

The guidance below applies to workloads resembling LSQB on PostgreSQL 19 beta 3 under the default configuration.

Choose the relational implementation where coverage or predictability is a hard requirement. It completed all nine queries under all three storage strategies at SF 1 and SF 3, eight of nine at SF 0.1, SF 0.3 and SF 10, and seven at SF 30. No SQL/PGQ configuration approached that: at its best, at SF 1 and SF 3, it answers six of nine, and five of nine everywhere else. The same holds where aggregate latency is the primary concern, or where the workload resembles Q2, Q3, Q6 or Q9. This is an observed result about the two implementations as configured, not an inherent advantage of relational SQL. Note the direction of the remaining asymmetry: the relational schema carries no indexes while every SQL/PGQ table carries one, and Section 5.3 shows a single index turning one of its timeouts into a completed query. An indexed relational baseline might change the comparison further, but it was not measured.

Choose SQL/PGQ where expressiveness and maintainability outweigh a latency penalty of 1.25–2.55× measured on these formulations and layouts, or where the workload is dominated by Q1-shaped patterns, the one case in which SQL/PGQ is faster.

Within SQL/PGQ, no stable latency advantage of PT over MV was observed for this workload and these physical definitions. PT has the lower geometric mean at five of seven scale factors and MV at SF 1 and SF 30, the MV/PT ratio spans 0.97–1.44 with no trend, and the failure rates are identical throughout. That is a null result on these definitions,

not a finding that the two storage modes are equivalent: they differ in indexing and in element keys as well as in materialisation, so the comparison isolates neither. Section 6.4, item 5, sets out the controlled version that would. MV's 12.0–15.6 % disk saving is the durable argument in its favour, and it also loads faster than PT at six of the seven scale factors, SF 0.3 being the exception. Set against that are a materialisation step and a refresh obligation the benchmark never exercises. Part of that saving is the absence of an index PT carries, not a property of materialisation (Section 3.4). RV is not recommended: 1.89–3.28× slower, the worst failure rate, and 1.17 GiB of peak memory on Q4.

Within the relational implementation the ranking differs. RV is a clearer trade-off, offering roughly half the disk and a 2.57–3.84× faster load for 0.79–1.92× the latency. Between PT and MV neither leads consistently: MV takes the aggregate at three of six scale factors and PT at two, with RV taking the smallest one, and the MV/PT ratio spans 0.81–1.30 with no trend. The two are indistinguishable on disk. The PT-versus-MV ordering therefore reverses with the projection: PT leads five of seven under SQL/PGQ's normalised layout, MV three of six under the merged one. Neither is a default that transfers across schemas, and Section 6.4, item 5 names the confound — indexing and element keys — that would have to be removed to settle it.

6.3 Threats to validity

1. **Formulation bias.** The relational queries are the benchmark's own reference implementations, used unchanged. The SQL/PGQ queries were written for this study. The results therefore evaluate the tested formulations, not the optimal SQL/PGQ expression of each pattern, and a better formulation may exist that this report did not find. This is the strongest threat to the latency comparison of Section 6.1, and its magnitude is not bounded by these measurements: nothing here establishes how much of the 1.25–2.55× gap it accounts for.

2. **SF 100 differs from the rest of the series in three conditions at once.** Its timeout is 60 minutes against 10; its instance is `20cpu-320ram` against the `8cpu-32ram` every other scale factor used; and its repetition count is three against five, because it belongs to campaign 1 throughout (Section 3.5). Coverage at SF 100 is therefore not comparable with the other rows, and the aggregate in Section 4.2 rests on the two queries RV completes. Sections 4.1 to 4.4 are unaffected: within campaign 2 the instance, timeout, repetition count and query order are the same at every scale factor measured.

3. **RV loses coverage at SF 100 because its cardinality estimates collapse.** PT and MV each return five queries there; RV returns two. Nothing in what is stored explains it, since the five derived relations hold the same rows in all three modes. The plans do. On Q5 the PT plan is six hash joins, with 165 million rows estimated at the topmost of them and a total cost of 3.5×10^{7}. The RV plan for the same question expands each derived relation into an `Append` of two scans, then chooses a merge join feeding a nested loop, estimates 4.2×10^{20} rows there, and costs the whole at 1.1×10^{18}. Q8 gives the same shape and order of magnitude, Q7 reaches 9.6×10^{32} rows, and Q4 reaches 2.5×10^{45} against 1.9×10^{8} under PT. Estimates of that size are not near

misses: the union introduced by view expansion defeats the estimate, and the join method follows it — hash joins throughout under PT, a nested loop over a supposed 10^{20} rows under RV. PT and MV, by contrast, produce the same join methods over the same relations on Q1, Q5, Q7 and Q8, with costs within 2 % of one another. These plans come from campaign 1, the only campaign that measured SF 100 (Section 3.5).

4. **Q3, Q6 and Q9 never complete under SQL/PGQ at any scale factor on this version.** The relational approach answers Q3 in 0.082–362 s throughout, and Q6, which has 55 607 896 matches at SF 0.1, in 2.9–3.0 s. A gap of that size belongs to the query the `GRAPH_TABLE` rewrite produces, or to the plan chosen for it, since the rewrite hands its output to the same executor (Section 5.1).

The loading convention deserves a note, since getting it wrong would invalidate the counts. The LSQB `pgq` queries traverse friendship with an *undirected* pattern, `-[k:knows]-`, so `Person_knows_Person` must hold one row per undirected edge. A symmetric load would inflate results by 2^n in the number of `knows` edges in the pattern: by a factor of two for Q2, four for Q6 and Q9, and eight for Q3. Q1, Q4, Q5, Q7 and Q8 are unaffected, because they do not traverse `knows`.

Inspection of the query texts confirms the accounting. Q2 has one `knows` edge in its pattern, Q6 and Q9 have two each, and the remaining five have none. Q3 reaches the same factor of eight by a different route: it matches a single `knows` edge in a common table expression and joins that expression three times (Table 4). Two further pieces of evidence support the convention. The formulation of Q9 constructs the symmetric closure itself, in a `UNION ALL` subquery joined to the pattern result, which would be redundant if the base table were already symmetric. And every returned count matches the reference (Section 4.5). The relational approach stores the closure instead, as its own formulation requires; its queries join `Person_knows_Person` directionally (Section 3.3).

1. **One cell is a partial failure.** Under relational RV at SF 10, one of five repetitions of Q6 timed out; the other four ran in 594.1 to 598.4 s against a 600 s limit. The cell is marginal rather than anomalous, and it is excluded from every aggregate.

2. **Repetitions agree closely, which was not true of campaign 1.** Across the 426 cells that completed at least twice, no cell spans more than a factor of 1.80 between its fastest and slowest run. The relative standard deviation has a median of 1.5 % and a ninetieth percentile of 5.7 %; two cells exceed 20 %, both at SF 0.1 where the absolute times are tens of milliseconds.

This matters beyond reassurance. Campaign 1 ran at three repetitions, in an uncontrolled query order and without collecting statistics. It produced five cells whose median and mean implied interior spreads of 6-fold to 241-fold, read at the time as evidence of two incompatible execution regimes. Nothing of that kind appears here. Whatever produced them was a property of that protocol, most plausibly the cache state left by whichever queries happened to run first, and not of the planner. The figures in this report are

correspondingly firmer than the earlier ones. The control group of Section 6.1 shows the same improvement: configurations that differ in nothing now agree to within 0.61–1.14, where before they diverged by as much as a factor of 18.

1. **Feasibility is non-monotonic in scale in both approaches.** Under SQL/PGQ, Q2 completes at SF 1 and SF 3 and at no other scale factor, returning the correct count in 3.29–3.78 s and 6.89–8.79 s respectively. In the relational approach, Q9 completes at SF 1 and SF 3 and nowhere else, in 103.4–130.9 s and 343.4–365.0 s. Success on multi-gigabyte databases coexists with failure on a few hundred megabytes, and neither pattern is resource-bound.

Q2 has a precedent: the benchmark's authors saw non-monotone times for it on one of their two systems, and traced them to the optimiser switching to a multi-way join where binary joins were cheaper [1]. Q2 is cyclic but has one many-to-many edge label only, so binary joins suffice and a planner that decides otherwise pays for it. The behaviour appears to be a property of the query rather than of this implementation. Q9's pattern has a second explanation available: Section 5.2 shows that collecting statistics is what removes it from SF 0.1 and SF 0.3, where it completed in four seconds without them.

1. **The feasibility of Q3 does not track the amount of work it has to do.** Section 5.1 shows that at SF 0.1 no `GRAPH_TABLE` formulation completes within 600 s. The identical computation written as `JOIN` operations completes in 1.459 s. The relational approach answers Q3 at every scale factor, from 0.082 s at SF 0.1 to 361.9 s at SF 30, so the computation is not intrinsically beyond reach at any size measured here.

Capturing plans per repetition rather than per query would strengthen the third and fourth of these. The plans in this report are one per query, taken in the statistics state the repetitions ran in, which establishes what the planner chose but not whether it chose the same thing five times.

6.4 Follow-up work

The following are listed in the order in which we would run them: the cheap tests of the reasoning above first, then the re-runs most likely to extend coverage, then the schema work needed for full parity. Expected value is not the same as position — item 4 is the one most likely to add a query, and item 1 the least.

1. Re-run with tuned `work_mem`, `shared_buffers` and parallelism settings, and capture the plans in both configurations. This is a test rather than an expected remedy. Section 3.2 sets out why the failing queries are not expected to be short of memory, and why `work_mem` is unlikely to move a plan whose decisive row estimates are one. What the re-run establishes is whether that reasoning holds outside the single cell in which the plans were examined.
2. Capture executed plans, with actual row counts, for every query that completes under the controlled protocol. Section 5.2 shows that scan count is the wrong currency for the latency gap: Q1 scans more relations than its relational counterpart and is twice as fast. The account has to be made in rows, and this report has actual row counts only from campaign 1. `EXPLAIN (ANALYZE)` on the completing cells would settle how much of

the 1.25–2.55× is rows processed and how much is per-row overhead. It costs one extra execution per cell and changes no other condition.

3. Randomise the query order and report cold and warm latency separately. Campaign 2 already holds the instance flavour, the timeout and the repetition count constant across scale factors, and runs five repetitions; what it does not do is randomise the order, which is the protocol the benchmark's authors used for their own experiments, under a five-minute timeout [1]. Adding that, and separating cold from warm, would let the PT-versus-MV comparison be settled rather than left as a null result (Sections 4.2 and 6.2). This subsumes SF 100, whose timeout of 60 minutes differs from the 10 minutes applied elsewhere. Every success there fell between 1 068 s and 2 303 s, or 30–64 % of that budget. The queries that fail at SF 100 therefore exceed it by a wide margin rather than sitting near the threshold, and a further extension would convert little.
4. Re-measure Q6 and Q9 at SF 30 under a raised timeout. Both fail there in all three strategies. Q6 consumed 588–597 s of the 600 s budget at SF 10, so extrapolating its growth from SF 10 to SF 30 puts it in the tens of minutes. That is an extrapolation, not a measurement, and a raised timeout would settle it. This is the case where a longer limit looks most likely to extend coverage.
5. Equalise the three storage modes on everything except storage: index the materialised views to match PT's primary keys, and declare the same element keys in all three property graphs. Until then the PT-versus-MV comparison under SQL/PGQ mixes storage with indexing and key declaration (Section 3.4). The relational approach needs no such correction, since none of its three modes is indexed.
6. Re-run the relational comparison over a single shared physical schema, so that disk footprint and load time become comparable across the two approaches as well. Parity would also require equalising what the two approaches differ on beyond the projection. The relational schema stores `knows` in both column orders and declares no indexes at all, while every SQL/PGQ table carries a primary-key index (Section 3.3).

Two questions are left open rather than recommended, because neither is needed to read the results above. The plans of Section 5.2 were taken after `ANALYZE`, so a second capture without it would correspond exactly to the runs of Section 3. And a capture per repetition, rather than one per cell, would show whether the planner chooses different plans across identical runs, which is what the dispersion of Section 6.3 and the signals of Section 6.3 leave undecided. Both are cheap; neither changes any figure reported here.

6.5 What survives the revert

SQL/PGQ was withdrawn from PostgreSQL after beta 3 (see the note at the head of this report). The measurements above were taken on that code and remain valid as measurements of it; what changes is which of them still say anything about the next implementation.

The defects that motivated the revert are largely outside the scope of what we measured here. The discussion that led to it [15] — the revert is verifiable in the source tree, where the two files exist at tag `REL_19_BETA3` and not on `master` — turned on property graphs carrying needless attribute rows, `pg_dump` missing dependencies,

cascade drops leaving orphaned labels and properties that then blocked `ALTER PROPERTY GRAPH`, rewrite paths reading element-table schemas without taking locks, `AlterPropGraph` locking too weakly to block a concurrent rewrite, out-of-bounds reads on unknown types, whole-row `GRAPH_TABLE` references expanding to zero columns, missing privilege checks, `DROP OWNED BY` failures, and unresolved design questions about whether labels and properties are scoped globally or per label.

None of these findings concerns the query-evaluation performance measured by this benchmark. Read as a list, they are primarily catalogue, dependency, DDL, privilege and concurrency problems. Two of them do touch query semantics — the unknown-type read and the whole-row reference — and one, the unlocked schema read, sits in the rewrite path itself, so the separation is one of scope rather than of kind: this report measured what the feature computed, not how it behaved under DDL, concurrency or dump and restore.

That matters for how this report should be read, because the benchmark exercises almost none of it. Each configuration declares its property graph once, loads, and then runs read-only queries on a single connection: no concurrent DDL, no dump and restore, no `ALTER PROPERTY GRAPH`, no privilege variation, no `DROP OWNED BY`. The two query-semantics defects are demonstrably not reached: every `COLUMNS` clause in the eighteen query texts names its output columns explicitly, so no whole-row reference is ever formed, and every column projected is an integer identifier, so the unknown-type path is not entered. The element keys are the thirty primary keys of the schema, so the non-unique-key defect does not arise either. The unlocked schema read in the rewrite path is a hazard only against concurrent DDL, and no configuration here issues any.

One reported defect deserves separating from the rest, because its name invites a confusion. The discussion records silent loss of graphs whose edge keys came from foreign keys [15]. That is a loss of the graph *definition* under DDL, not a corruption of query results, and the two would show up differently: a lost definition makes the next query fail to compile, a corrupted result does not announce itself at all. The schema used here declares no foreign key constraints — every table carries a primary key, and the property graph's `SOURCE KEY` and `DESTINATION KEY` clauses reference those primary keys directly — so the reported path is not the one this benchmark takes.

The only reported defect with a plausible direct path to the measured results would have been one that silently altered or lost the graph definition or the data beneath it. Our correctness check gives evidence against such an effect in the runs that completed: all 105 SQL/PGQ runs that returned a value returned the published answer, at every scale factor and under every storage strategy.

The denominator matters and is worth stating plainly. Those 105 are of 189 attempted — 84 cells reached the timeout and returned nothing to check. Correctness and coverage are separate axes here, and this evidence sits entirely on the first: it says the feature was right wherever it answered, and says nothing whatever about the 84 cells where it did not. Nor would it catch a defect that altered a timing without altering a count.

So the withdrawal does not invalidate the measurements. It does change what they predict, and the distinction below is not rhetorical: some findings are properties of the reverted code, and some are properties of the planner and executor underneath it, which the revert did not touch.

Tied to the code we measured. Everything cited in this report remains where it is cited: the tag `REL_19_BETA3` is immutable, and the files, line numbers and quotations of References [9] to [14] resolve exactly as they did. What the revert removed is that code's future, not its past. `parse_graphtable.c` is still there to be read, but it is no longer the parser PostgreSQL will ship, so the eight rejected pattern constructions of Section 5.1 describe one attempt rather than a standing limitation — a redesign is free to accept any of them, and the absence of quantifiers in particular is the kind of gap a second attempt would be expected to close. The same holds for the shape of the generated query, one `Query` per label combination combined with a union, and with it the scan counts of Section 5.2 and the view-expansion behaviour of Section 6.3. Every latency figure in Section 4.2 depends on that generated query, so the 1.25–2.55× aggregate gap, the 1.60–2.20× advantage on Q1, and the storage-strategy comparison of Section 6.2 would all have to be re-measured against a new implementation. They are a record of what one implementation cost, not a prediction of what the next one will.

Properties of the engine, not of the operator. The coverage limit is the substantive finding of this report for the relational execution model we tested, and it does not rest on SQL/PGQ-specific catalog or DDL behaviour. Q3, Q6 and Q9 are the three queries the benchmark's authors attribute to worst-case optimal joins and factorised processing [1], and the PostgreSQL 19 beta series implements neither, for any query, graph-shaped or not. That is a property of the planner and the executor of the version tested, not of the operator layered on top of them.

The transfer to a future release is therefore conditional, and the condition is worth stating rather than assuming. A new SQL/PGQ front end that rewrites patterns into ordinary relational plans — the design the withdrawn one used, and the one a redesign would most plausibly keep — would meet the same wall in the same three places, however much cleaner its syntax and catalog handling. An implementation that instead brought its own join strategies, or that arrived alongside worst-case optimal joins in the planner, would not, and this report predicts nothing about it. The negative result of Section 5.2 transfers on the same condition: collecting statistics rescued no query and cost five, because the estimate that fails is formed at the join rather than at the table, and that too belongs to the planner rather than to the operator.

Independent of SQL/PGQ altogether. Section 3.5 and Section 6.3 record what the two measurement campaigns established about protocol: three repetitions in filesystem order produced apparent bimodality that five repetitions in a fixed order did not reproduce, and a control group of queries touching no derived relation is what settled it. Those conclusions are about benchmarking, not about graph queries.

What this means for the next release. If a redesigned SQL/PGQ arrives in PostgreSQL 20 or later, the useful baseline from this report is not a latency figure. It is the coverage boundary observed under the relational execution model we tested: the three queries that never complete are the three the benchmark associates with join strategies the

tested engine did not implement. Whether that boundary remains in a redesigned SQL/PGQ implementation is a question for the next benchmark run. That is a checkable question, because the harness, the schemas and both query formulations are published [5] and the benchmark has not changed. The coverage table of Section 4.1 is the baseline to re-run.

7. Conclusion

The conclusions separate what the benchmark returned from what that shows about either implementation. Only the first is a measurement.

What was measured. Q3, Q6 and Q9 returned no result in any of the eighteen SQL/PGQ configurations and Q2 in two scale factors of six, so the best SQL/PGQ failure rate was 33.33 %. The relational implementation reached 0 % in six configurations of eighteen and never exceeded 22.22 %. On the common set of queries that complete in every configuration, the relational implementation has the lower geometric-mean latency at all six shared scale factors, by 1.25–2.55×, with Q1 the sole exception at 1.60–2.20× the other way. The common set shrinks with scale factor, so these aggregates are not a fixed-workload scalability curve. Every returned value in both approaches matched the reference answers.

What that shows about SQL/PGQ. The implementation is correct everywhere it answers and, on the queries it runs, already within a small factor of joins written out explicitly. It achieves that with no graph executor at all, by rewriting each pattern into an ordinary relational plan (Section 5.1). Its incompleteness falls on the queries the benchmark was designed to make hard — the ones LSQB's authors identify as calling for worst-case optimal joins or factorisation [1], neither of which PostgreSQL implements for any query, graph-shaped or not. That the failures land exactly there, and nowhere else, is the useful part. It is consistent with a missing algorithm rather than a defect distributed across the feature, though the measurements establish the coincidence and not the cause: a cancelled query yields only an estimate-only plan, so the execution that would decide the question was never observed. The rewrite it generates scans a median of 2.2 times as many relations as the explicit form, which is enough to explain the direction of the latency gap but not its size (Section 5.2). Where that gap sits is the encouraging part. It points at the generated query and the estimates formed over it, not at a graph execution path, because there is no such path: PostgreSQL implements SQL/PGQ as a rewrite into ordinary relational plans (Section 5.1, and [9]). The limits found here are planner and estimator limits, which are the kind later releases have historically improved without a change to the operator.

What that shows about the three pre-releases. Beta 1, beta 2 and beta 3 behave alike on this workload. Coverage is identical query for query and configuration for configuration, the best failure rate is the same, aggregates sit within 0.81–1.17 of one another, and the join methods match over the same number of scans in 160 of 162 combinations. Two cells, both on Q1, are where the chosen plan did change. Nothing regressed across the series. For a feature this new that is the reassuring outcome, and it is also the limit of what a beta series can reveal. A pre-release window is for stabilisation, so the optimiser work behind a release lands before its first beta, and work aimed at later releases is invisible from here. These figures are a baseline for measuring the next release, not a reading on how much is being done.

What that shows about the relational baseline. Its coverage advantage was measured on a schema with no indexes at all, while every SQL/PGQ table carries a primary key and its btree index (Section 3.3). On this dimension it is the relational side that is handicapped. Section 5.3 shows what the handicap costs: a single index on the inner relation of Q9's top join turned a timeout into a completed query, cutting the estimated cost of one iteration from 615.70 to 0.72. A team that indexes its relational schema, as most do, should expect the measured margin to widen rather than narrow, so the figures here are conservative in that direction. The advantage remains a property of these two implementations as configured and should not be generalised to relational SQL as such.

What that shows about statistics. Collecting them is the standard first response to a plan that looks wrong; on this workload it rescued no query and cost five cells. That is a useful negative result rather than merely a disappointing one. It rules out the simplest explanation for these failures and points instead at join-level cardinality estimation, which Section 5.3 shows missing by six orders of magnitude in the one cell diagnosed in full. Table-level statistics cannot fix an estimate formed over a join, which is a more precise place to look than "collect statistics".

Open items. SF 100 was measured only in campaign 1, under a different timeout and protocol, and the relational implementation was never measured there. Two non-monotone coverage patterns remain unexplained. The two implementations differ in projection, indexing, `knows` representation and query provenance (Section 3.3). No run was made under tuned server configuration. None of this bears on the coverage result, which rests on queries that returned nothing to measure or mis-measure.

Outlook. Read together, these measurements describe a first implementation that is correct wherever it answers, stable across three pre-releases, and within 1.25–2.55× of LSQB's reference joins, on these formulations and layouts, for the queries it runs. One query, the longest pattern among those that complete, is where the pattern form wins outright, by 1.60–2.20×. The remaining gap is concentrated in cardinality estimation and access-path choice over the query the rewrite produces. Section 5.4 records a condition under which even Q3, which times out throughout the controlled campaign, returned the correct count in 1.504 s. That the operator executed the pattern at all is weak evidence that the current ceiling is plan selection rather than the operator; the observation did not reproduce under the controlled protocol and should not be leaned on. None of this forecasts a later release and this report measures none. What it offers is a baseline for SQL/PGQ in PostgreSQL, and a set of reproducible cases the next release can be measured against.

The register we would end on is this. Graph pattern matching is not the shape of problem PostgreSQL was built around, and the version 19 betas took it on without adding a graph engine. A property graph is a declaration over existing tables, a pattern is a rewrite, and the relational planner does the rest. Measured on a benchmark written specifically to separate graph workloads from relational ones, that arrangement executes six of the nine patterns, five of them at every scale factor of the controlled series, returns the correct count for every one it executes, and pays a factor rather than an order of magnitude for the privilege. The patterns it cannot yet run are the patterns whose difficulty the

benchmark's own authors trace to algorithms PostgreSQL has never implemented, and pattern syntax is a subset rather than the whole standard. Both are real constraints and neither is a surprise at this stage. For a first release of a paradigm this foreign to the engine, the result is a good deal better than the null hypothesis, and it is a foundation worth building on.

Appendix A. Reproducibility

The benchmark was executed on a cloud-based container platform, with one instance per configuration and nothing else running on it. Every configuration in campaign 2 ran on the same instance flavour, as the table below shows; only the SF 100 rows, which come from campaign 1, were measured on different hardware. The client is a single-threaded Python harness on the same instance, so the figures reported are single-client latencies with no contention; no throughput or concurrency metric was collected.

Table A1. The test bench. Campaign 2, on which this report rests, held the instance flavour constant across every scale factor it measured. SF 100 was measured only in campaign 1, on a larger flavour and a longer timeout, and is not comparable with the rows above it (Section 3.5).

Item	Value
SF 0.1 – SF 30 (campaign 2)	`8cpu-32ram`, 8 vCPU, 32 GiB, 10 min query timeout
SF 100 (campaign 1 only)	`20cpu-320ram`, 20 vCPU, 320 GiB, 60 min
Campaign 1, for reference	`2cpu-16ram` at SF 0.1, `8cpu-32ram` at SF 0.3 through SF 30
Harness	Python 3.13.5, single-threaded, one query in flight at any time
Database driver	`psycopg2-binary` 2.9.12
Other dependencies	`pydantic` 2.12.5
Connections	One per repetition, closed immediately afterwards
Memory instrumentation	`VmRSS` and `VmHWM` read once each from `/proc/[pid]/status`

Holding the flavour constant matters for how the per-scale figures may be read. In campaign 2 the instance, the timeout, the repetition count and the query order are all the same at every scale factor, so the only thing that changes across the series is the data. Campaign 1 used the same flavour from SF 0.3 upwards and a smaller one at SF 0.1, so its cross-scale series is controlled above SF 0.1 as well; what makes its figures the weaker of the two is the protocol of Table A3, not the hardware.

One consequence of 32 GiB is worth stating, because it bears on the cache. Every database measured on that flavour fits in memory except at SF 30, where the largest is 77 GB against 32 GiB available; the operating system cannot hold that one, and the SF 30 latencies are the only ones in either campaign's series that include reading from disk in steady state.

Table A2. Server settings as captured. These ten are the ones the harness read from `pg_settings` in every configuration and stored beside the results [5]; the values below are from SF 1 under plain tables and are identical in every other configuration.

Setting	Value	Source reported by `pg_settings`
`server_version`	19beta3 (Debian `19~beta3-1.pgdg13+1`)	default
`shared_buffers`	16 384 × 8 kB = **128 MB**	configuration file
`work_mem`	4 096 kB = **4 MB**	default
`effective_cache_size`	524 288 × 8 kB = **4 GB**	default
`max_parallel_workers`	8	default
`max_parallel_workers_per_gather`	2	default
`jit`	**off**	default
`random_page_cost`	4	default
`autovacuum_naptime`	60 s	default
`statement_timeout`	600 000 ms	client — the driver sets it per connection

Two of these carry most of the weight. `work_mem` at 4 MB is what a hash or sort node may use before spilling. `effective_cache_size` at 4 GB is what the planner assumes the operating system will cache, on an instance with 32 GiB of memory — where every database except the largest at SF 30 is smaller than the memory available to it (Table A1, Table 4). Both are packaged defaults, and Section 6.3 records that no run was made under tuned configuration.

Settings not in the list above were not captured, so this report says nothing about them beyond the fact that the image alters `listen_addresses` and the two values marked above; everything else was left as the package shipped it.

Because each repetition runs on its own backend process, `VmHWM`, the kernel's high-water mark for resident memory, is scoped to that repetition and needs no sampling. The run parameters are below, and Section B.3 records what the metric does and does not capture.

Run parameters: campaign 1 vs campaign 2

Table A3. Run parameters. The left column is the protocol of campaign 2, on which this report rests; the right records campaign 1, which supplies the SF 100 rows of Section 4.4 and the without-statistics condition of Section 5.2.

Parameter	Campaign 2, controlled	Campaign 1, initial
Query timeout	10 min (SF 0.1–30)	10 min (SF 0.1–30); 60 min (SF 100)
Repetitions	5 per query per configuration	3 per query per configuration
Reported statistic	Median over the repetitions that completed	Median of three (mean also recorded)
Per-repetition times	Recorded, with timeouts marked	Not recorded
Table statistics	`ANALYZE` once per configuration, before any query runs	Never collected; neither driver nor DDL runs `ANALYZE`
Query order	Sorted, identical in every configuration	Filesystem enumeration order, not fixed across configurations
Plan capture	Per query, immediately before its repetitions, in the same statistics state	Separate invocation, after the timings, in a different state
Statistics state	Fingerprinted before and after the run; drift reported	Not verified
Timing window	`perf_counter` around `execute` + `fetchall` only; connection setup, backend-PID lookup and `/proc` reads excluded	as campaign 2
Memory statistic	`VmHWM − VmRSS_before`, maximum over repetitions	as campaign 2
Cold/warm cache	Not controlled; no flush of the OS page cache or `shared_buffers`	as campaign 2

Table A4. What a reader needs to reproduce the measurements.

Item	Value
Server	`postgres:19beta1`, `postgres:19beta2`, `postgres:19beta3` Docker images; `19beta3 (Debian 19~beta3-1.pgdg13+1)`
Server configuration	Packaged defaults throughout; the image alters `listen_addresses` only. Table A2 lists the settings that bear on the results
Instances	Table A1; one flavour for every scale factor in campaign 2, SF 100 excepted
Dataset	LSQB pre-generated datasets from the SURF repository [3], used unmodified
Schema and load	The six DDL scripts of the harness repository [5], `ddl/{pgq,sql}/data-{plain-tables,regular-views,materialized-views}.sql`
Query texts	`queries/sql` from the benchmark unchanged; `queries/pgq` written for this study (Section 3.3)
Client	Python 3.13, `psycopg2`, one connection per repetition
Raw output	Every result file, plan and statistics snapshot [5], each campaign with the harness that produced it and a machine-readable statement of its conditions
Result files	`runs/campaign-{1-initial,2-controlled}/{version}/{approach}/{STRATEGY}-sf{N}.txt`
Query plans	`runs/campaign-{1-initial,2-controlled}/{version}/{approach}/analyze/{STRATEGY}-sf{N}/q{1..9}.json`, as `EXPLAIN (FORMAT JSON)` output, with `_state.json` beside each set recording the statistics the planner had

A configuration is run as a single invocation, which loads the schema, populates it from the CSV files and then executes the nine queries:

```
python3 benchmark.py \
    --queries-path queries/pgq \
    --result-path /path/to/output/dir/0.1.result.txt \
    --timeout-millis 600000 \
    --times 5 \
    --path-to-data /path/to/data/csv \
    --path-ddl-query ddl/pgq/data-materialized-views.sql
```

The harness default timeout is 60 000 ms, so the 10-minute limit used here was supplied explicitly, as above. A reader who omits the flag will measure against a one-minute limit and obtain a quite different pattern of failures.

The per-scale series is controlled in campaign 2 and not in campaign 1, for the reason given in Section 3.5.

Appendix B. Full tables

The tables below give every measured cell. The body of the report quotes summaries of them.

B.1 SQL/PGQ query latency

Table 24. Median latency in seconds for all nine queries, seven scale factors and three storage strategies. The entry `timeout` denotes failure to return within the applicable threshold: 10 minutes for SF 0.1–30 and 60 minutes for SF 100.

Failure rate is the number of failed attempts over nine queries times five repetitions, so it is quantised in steps of 2.22 percentage points. A value that is not a multiple of 11.11 pp marks a partial failure, where some repetitions of one query succeeded. Differences of a single query should not be over-interpreted.

SF	Mode	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Failure rate
0.1	PT	0.418	timeout	timeout	0.309	0.416	timeout	0.317	0.495	timeout	44.44 %
0.1	RV	0.504	timeout	timeout	1.225	1.199	timeout	0.449	0.848	timeout	44.44 %
0.1	MV	0.519	timeout	timeout	0.583	0.479	timeout	0.659	0.554	timeout	44.44 %
0.3	PT	1.046	timeout	timeout	0.889	1.057	timeout	0.871	1.190	timeout	44.44 %
0.3	RV	1.259	timeout	timeout	5.983	3.440	timeout	5.818	2.569	timeout	44.44 %
0.3	MV	1.222	timeout	timeout	1.285	1.261	timeout	1.075	1.419	timeout	44.44 %
1	PT	6.099	3.778	timeout	4.478	4.896	timeout	4.051	6.844	timeout	33.33 %
1	RV	5.472	3.292	timeout	56.416	13.046	timeout	6.246	9.871	timeout	33.33 %
1	MV	5.814	3.607	timeout	4.626	4.333	timeout	4.419	6.157	timeout	33.33 %
3	PT	19.580	6.892	timeout	11.254	10.548	timeout	16.465	16.522	timeout	33.33 %
3	RV	20.383	8.791	timeout	181.223	41.574	timeout	20.577	45.206	timeout	33.33 %
3	MV	19.286	7.072	timeout	13.408	10.405	timeout	16.529	16.739	timeout	33.33 %
10	PT	68.390	timeout	timeout	39.953	35.465	timeout	41.765	59.173	timeout	44.44 %
10	RV	72.761	timeout	timeout	368.353	139.935	timeout	68.208	timeout	timeout	55.56 %
10	MV	77.745	timeout	timeout	52.159	37.690	timeout	54.758	63.305	timeout	44.44 %
30	PT	253.068	timeout	timeout	148.785	130.722	timeout	156.130	209.040	timeout	44.44 %
30	RV	263.192	timeout	timeout	timeout	511.493	timeout	257.557	timeout	timeout	66.67 %
30	MV	244.473	timeout	timeout	183.307	131.503	timeout	149.551	223.106	timeout	44.44 %
100	PT	1 925.159	timeout	timeout	1 346.425	1 306.447	timeout	1 068.208	2 303.377	timeout	44.44 %
100	RV	1 945.842	timeout	timeout	timeout	timeout	timeout	1 676.457	timeout	timeout	77.78 %
100	MV	1 985.260	timeout	timeout	1 782.691	1 300.257	timeout	1 119.745	2 287.208	timeout	44.44 %

No cell in this table is a partial failure: every SQL/PGQ cell either completed in all five repetitions or in none of them.

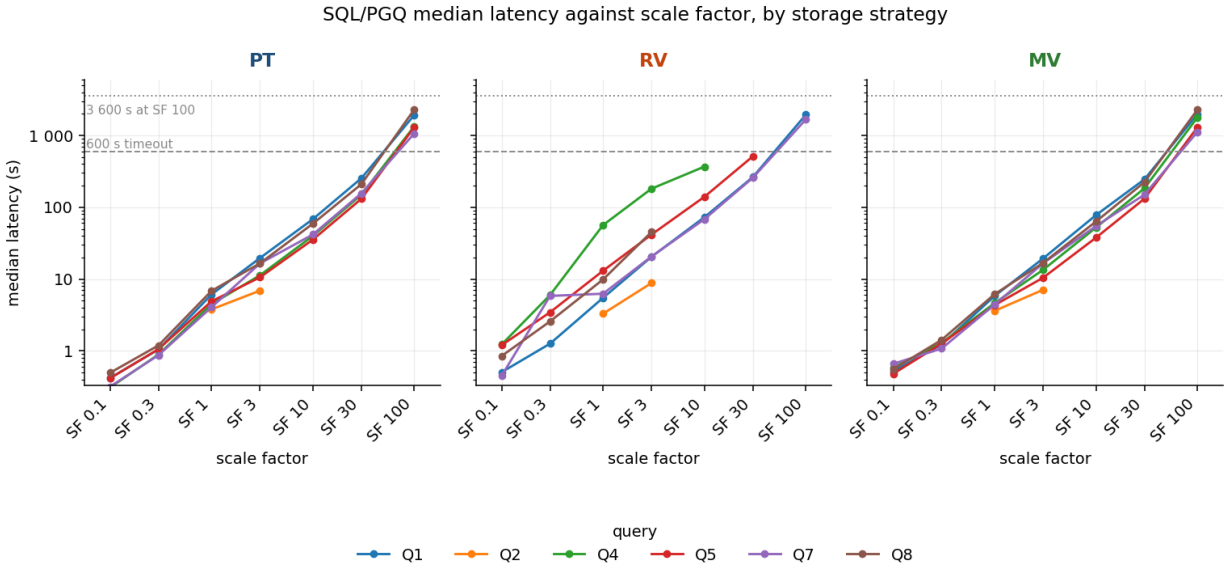

Figure 10. Median latency against scale factor by storage strategy. Only the six queries that complete in at least one configuration are drawn. The dashed line marks the 600 s timeout applied at SF 0.1–30 and the dotted line the 3 600 s timeout at SF 100.

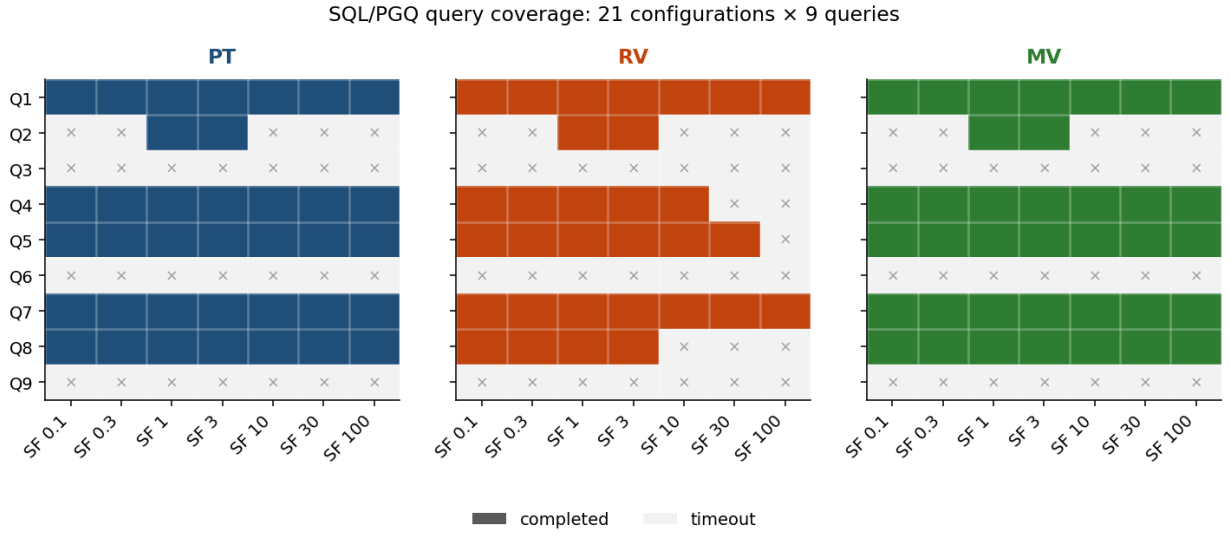

Figure 11. Query coverage across the 21 SQL/PGQ configurations. Q3, Q6 and Q9 form empty rows under every strategy. At SF 100, PT and MV complete the same five queries while RV completes two of them.

Q1 and Q7 are the only queries that complete in every configuration. Q5 completes everywhere except under RV at SF 100, Q4 everywhere except under RV at SF 30 and SF 100, and Q8 everywhere except under RV at SF 10, SF 30 and SF 100. Q3, Q6 and Q9 form empty rows throughout, including at SF 0.1 — a database of approximately 253 MiB under a ten-minute timeout, at which the relational approach answers Q3 in 0.082 s and Q6 in 3.0 s. The boundary is therefore categorical rather than a limit of scalability. Q2 completes at SF 1 and SF 3 only, in 3.3–8.8 s.

Sections 4.1 and 6.3 treat that coverage pattern and its non-monotonicity; RV's disproportionate degradation on Q4 and Q5 is quantified in Section 6.2, and the separation of the three strategies at SF 100 in Section 4.4. The lowest failure rate observed anywhere is 33.33 %, attained by all three strategies at SF 1 and SF 3.

B.2 Relational query latency

Table 25. Relational approach: median latency in seconds over five repetitions, `merged` projection, ten-minute timeout. The relational approach covers SF 0.1, 0.3, 1, 3, 10 and 30; only SF 100 lacks a relational counterpart, so the cross-approach statements below cover six of the seven scale factors treated in this report.

SF	Strategy	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Failure rate
0.1	PT	0.676	0.055	0.083	0.202	0.137	2.981	0.296	0.299	timeout	11.11 %
0.1	RV	0.668	0.059	0.082	0.116	0.093	2.938	0.218	0.156	timeout	11.11 %
0.1	MV	1.123	0.084	0.088	0.352	0.186	2.941	0.255	0.448	timeout	11.11 %
0.3	PT	2.274	0.209	0.523	0.492	0.388	15.998	0.690	0.782	timeout	11.11 %
0.3	RV	1.877	0.127	0.428	0.813	0.638	13.122	0.804	0.799	timeout	11.11 %
0.3	MV	2.310	0.164	0.526	0.499	0.383	16.065	0.740	0.780	timeout	11.11 %
1	PT	11.297	0.649	2.033	2.000	1.378	55.574	2.505	2.759	128.266	0.00 %
1	RV	11.489	0.686	2.139	25.197	3.389	55.968	5.049	4.109	130.930	0.00 %
1	MV	9.071	0.516	1.689	1.734	1.111	44.164	2.051	2.245	103.397	0.00 %
3	PT	39.814	1.961	7.784	5.891	4.021	155.474	7.889	8.483	343.445	0.00 %
3	RV	40.686	2.191	8.868	144.574	10.970	156.992	17.185	14.956	364.985	0.00 %
3	MV	40.120	1.990	8.134	6.030	4.018	153.654	7.723	8.192	345.786	0.00 %
10	PT	150.542	6.688	48.627	21.260	13.862	587.906	27.352	29.113	timeout	11.11 %
10	RV	152.243	6.826	48.672	32.220	33.023	596.996 *	53.929	41.997	timeout	13.33 %
10	MV	152.615	7.337	56.722	24.929	17.559	595.663	31.082	36.394	timeout	11.11 %
30	PT	529.912	24.330	361.909	87.996	58.508	timeout	108.623	128.788	timeout	22.22 %
30	RV	507.957	20.864	298.109	107.812	107.708	timeout	187.565	143.657	timeout	22.22 %
30	MV	510.452	21.556	300.227	75.434	49.655	timeout	96.102	111.201	timeout	22.22 %

* Partial failure: at least one of the five repetitions timed out. The median shown is taken over the repetitions that completed. The one such cell is excluded from the aggregates below.

Coverage is complete at SF 1 and SF 3 and eases off on either side, at eight of nine below and eight then seven above. Of the 54 (scale factor, query) combinations, 49 returned a result, and every one of those returned under all three strategies. The failures are Q9 at SF 0.1, SF 0.3 and SF 10, and Q6 together with Q9 at SF 30.

Q6 and Q9 are the two queries that bound this approach. Q6 rises from 2.9–3.0 s at SF 0.1 to 588–597 s at SF 10, which is 98–99 % of the 600 s budget, and fails at SF 30. Q9 reaches 103–365 s at SF 1 and SF 3, and fails at every other scale factor. Both are therefore lost to super-linear growth rather than to an abrupt discontinuity.

The failures of Q9 are non-monotonic. It completes under all three strategies at SF 1 and SF 3, and under none at any other scale factor. In campaign 1, before statistics were collected, it had completed at SF 0.1 under all three and at SF 0.3 under MV (Section 5.2). Section 5.3 diagnoses that instability directly and finds it to be a matter of plan selection rather than of resources.

RV is the weakest strategy on Q4 at the middle scale factors. It takes 25.20 s against 2.00 s for PT and 1.73 s for MV at SF 1, and 144.57 s against 5.89 s and 6.03 s at SF 3: factors of 12.6 and 24.5. The penalty is small at the extremes, 0.57 at SF 0.1 and 1.23 at SF 30. This reproduces the pathology documented for the SQL/PGQ approach in Section B.1. Q4 touches four of the five derived relations, and each expands into a union of two scans (Section 3.4), so under RV the query the planner receives has eight scans and four union nodes where PT has four. The same rewriting applies in both approaches, which is why the pathology appears in both and belongs to the construction rather than to either query language.

Repetitions within a cell agree closely throughout this table: no cell spans more than a factor of 1.80 between its fastest and slowest run, and the median relative standard deviation is 1.5 % (Section 6.3).

B.3 Peak memory

Table 26. Peak memory, computed as `VmHWM − VmRSS_before` in MiB and taken as the maximum over the five repetitions. Appendix A describes how the two values are read. Rows are omitted where no configuration completed, and queries that timed out are shown as `–`. The metric does not separate query-private allocation from shared-buffer pages faulted into the backend, so these values bound the private component from above rather than measuring it. Memory is not compared across approaches for that reason (Section 3.3), and no conclusion in this report rests on it.

Query	SF	PT	RV	MV
Q1	0.1	130.0	117.6	116.1
Q1	0.3	213.5	205.9	204.9
Q1	1	171.1	175.8	149.1
Q1	3	156.0	175.2	119.0

Query	SF	PT	RV	MV
Q1	10	195.3	194.5	162.5
Q1	30	210.6	211.9	209.4
Q1	100	246.6	238.3	230.8
Q2	1	121.3	158.5	110.9
Q2	3	134.9	112.7	134.9
Q4	0.1	152.0	165.8	136.1
Q4	0.3	160.2	358.9	166.8
Q4	1	173.6	1002.0	128.5
Q4	3	177.2	1197.0	135.2
Q4	10	200.6	1201.5	156.9
Q4	30	224.7	—	179.6
Q4	100	292.8	—	265.3
Q5	0.1	134.2	97.1	121.5
Q5	0.3	130.6	160.7	150.6
Q5	1	88.5	166.7	122.4
Q5	3	108.5	171.4	124.2
Q5	10	131.1	168.6	142.1
Q5	30	154.5	191.5	164.6
Q5	100	250.0	—	182.6
Q7	0.1	122.2	115.9	121.8
Q7	0.3	137.6	401.2	140.5
Q7	1	109.9	165.3	121.9
Q7	3	131.1	173.7	113.5
Q7	10	183.8	183.0	177.0
Q7	30	235.6	192.6	214.3
Q7	100	237.1	250.4	220.3
Q8	0.1	140.6	108.5	127.8
Q8	0.3	158.8	165.5	163.8
Q8	1	127.4	169.1	145.8
Q8	3	152.8	178.6	142.9
Q8	10	189.3	—	169.7
Q8	30	216.8	—	194.3
Q8	100	306.0	—	262.6

RV on Q4 is the only configuration with a working set of gigabyte scale, reaching 1 201 MiB. The RV/Q4 series is the sole outlier of that magnitude: no other cell in the matrix exceeds 401 MiB, the largest being RV on Q7 at SF 0.3.

Memory is otherwise largely decoupled from data volume, ranging over 88–401 MiB across the remaining matrix for databases spanning 253 MB to 252 GB. PT and MV are comparable, with MV the lower of the two in 26 of the 37 cells where both completed.

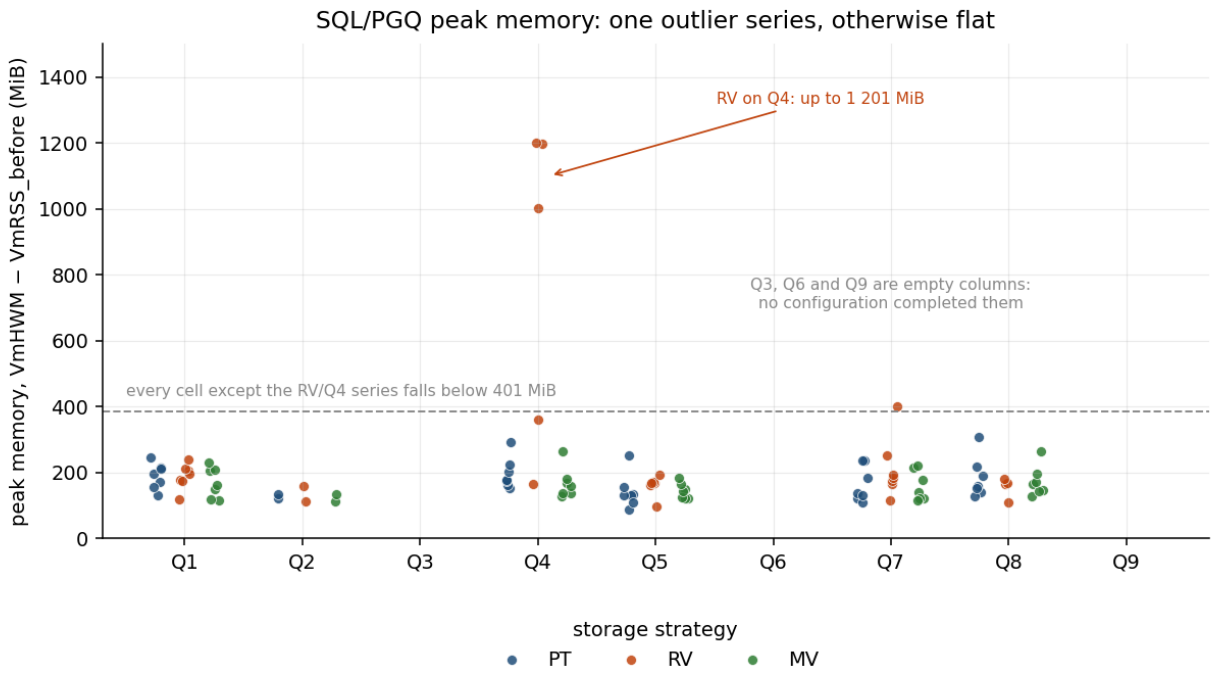

Figure 12. Peak memory by query and storage strategy. Each point represents one completed cell. Q3, Q6 and Q9 are empty because no configuration completed them.

References

[1] A. Mhedhbi, M. Lissandrini, L. Kuiper, J. Waudby and G. Szárnyas. LSQB: a large-scale subgraph query benchmark. *GRADES-NDA '21*, article 8, pages 1–11. DOI 10.1145/3461837.3464516.

[2] LDBC. Labelled Subgraph Query Benchmark, source repository: github.com/ldbc/lsqb, `main` branch.

[3] LDBC. LSQB datasets, SURF data repository: repository.surfsara.nl/datasets/cwi/lsqb.

[4] LDBC. Expected output for the LSQB queries: `expected-output/expected-output.csv`, `main` branch.

[5] Harness, schema definitions, query texts, raw results and query plans for this report: github.com/atberium/postgres-lsqb-benchmark, commit `c997f99`.

[6] ISO/IEC 9075-16:2023. *Information technology — Database languages SQL — Part 16: Property Graph Queries (SQL/PGQ)*.

[7] DuckPGQ, a DuckDB extension implementing SQL/PGQ: github.com/cwida/duckpgq-extension.

[8] PostgreSQL documentation, `ANALYZE`: doc/src/sgml/ref/analyze.sgml#L201-L203, tag `REL_19_BETA3`.

[9] PostgreSQL source, `rewriteGraphTable()`: rewriteGraphTable.c#L110-L124, tag `REL_19_BETA3`.

[10] PostgreSQL source, `generate_union_from_pathqueries()`: rewriteGraphTable.c#L608-L620, tag `REL_19_BETA3`.

[11] PostgreSQL source, the rewriter call on `RTE_GRAPH_TABLE`: rewriteHandler.c#L2083-L2091, tag `REL_19_BETA3`.

[12] PostgreSQL source, `RELKIND_PROPGRAPH`: pg_class.h#L181, tag `REL_19_BETA3`.

[13] PostgreSQL source, `RELKIND_HAS_STORAGE`: pg_class.h#L205-L210, tag `REL_19_BETA3`.

[14] PostgreSQL source, `DefineRelation` creating a property graph: propgraphcmds.c#L266, tag `REL_19_BETA3`.

[15] "PostgreSQL to pull property graphs from v19 after design flaws", freenode.net, September 2026: freenode.net/article/postgresql-to-pull-property-graphs-from-v19-after-design-flaws.

End of report