POSTGRESQL · MEASUREMENT

Queue table: the index that wins at 30 s, loses at 15 min



Same table, Postgres 17, three measurements.

Muhammet Şafak • October 3, 2026

SINGLE MACHINE, POSTGRES 17, SYNTHETIC BENCH

Nearly equal at 30 seconds. Apart at 15 minutes.

Partial and composite index: 11,537 vs 10,795 tps at thirty seconds. Under fifteen minutes of sustained load, composite missed the target.

POSTGRES DOES NOT KNOW THIS DISTINCTION

Two sets in one table

	Archive · done	Live set · pending
WHO LOOKS	Nobody queries it	✓ Workers search it, claims lock it
SIZE	Grows forever unless deleted	✓ Stays roughly constant
MEASURED	n_live_tup 10,005,726	✓ 5,000 rows
POSTGRES	Live tuples: the index, statistics and vacuum threshold all count them	✓ Same treatment; the two are not told apart

SECTION 01

Thirty seconds

Four strategies, three dead-row tiers. The live set is 5,000 rows at every tier.

SKIP LOCKED · 8 CLIENTS · 30 S · MEDIAN OF THREE RUNS

Three tiers, four strategies

STRATEGY	100K DEAD ROWS	1M	10M	INDEX (10M)
no index	2,001 tps	247 tps	7 tps	—
(status)	6,499 tps	6,417 tps	6,426 tps	66.1 MB
(status, created_at)	12,414 tps	11,202 tps	10,795 tps	310.4 MB
partial (created_at) WHERE pending	13,041 tps	11,707 tps	11,537 tps	7.6 MB

NO INDEX

A queue without an index vanishes.

It does not just degrade with dead rows: 2,001 down to 7 tps. A sequential scan can find the 5,000 rows among 10 million only seven times a second.

41×

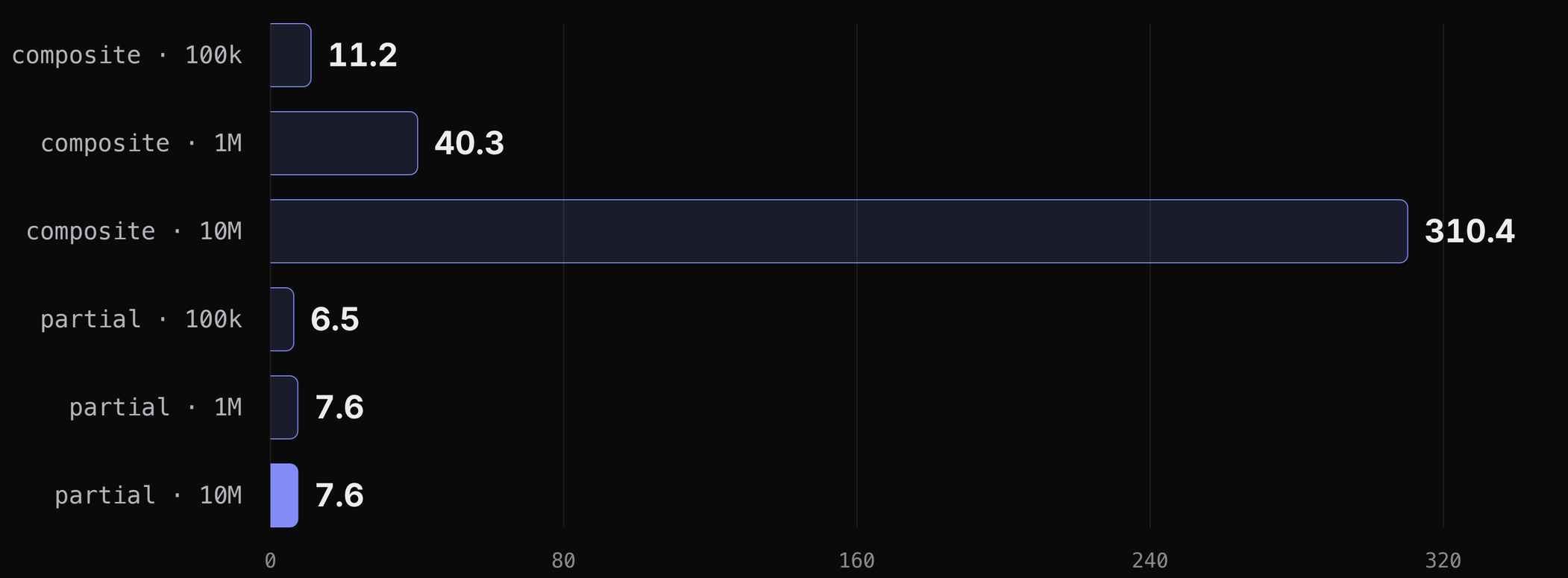
THE REAL DIFFERENCE IS SIZE

smaller index

At 10 million dead rows, partial is 7.6 MB and composite is 310.4 MB. The throughput gap is only 6.9%: the reason to pick it is not speed but that the index does not grow with the table.

AS DEAD ROWS GROW

The partial index does not grow with the table



pg_relation_size, bench/run.sh

SECTION 02

Planner

Same table, same index, same query. The only thing that changes is the plan type.

10M DEAD ROWS · SAME QUERY, PARAMETERIZED

The generic plan cliff

INDEX · PLAN	TPS	LATENCY	PLAN
partial · custom	11,752	0.68 ms	`Index Scan`
partial · generic	7	1,113 ms	`Sort → Seq Scan`
composite · custom	11,105	0.72 ms	`Index Scan`
composite · generic	11,416	0.70 ms	`Index Scan`

WHY

Roughly 1,635 times

NOTE A generic plan is built without knowing what $1 is. It cannot prove that status = $1 means 'pending', so it rules out the partial index.

TIP Composite is unaffected: it has no predicate to prove, and status is just a column inside the index.

WARNING Falling off the cliff took force_generic_plan. That is not a production setting.

SINGLE SESSION, DEFAULT AUTO MODE

Forty runs, forty custom plans

00

First five runs: all three on custom plan

done

01

Sixth: composite switches to generic

done

02

Partial: 40 of 40 custom

done

03

(status): 40 of 40 custom

done

CORRECTION

It did not switch under these conditions

CORRECTION In my earlier post I said it 'degrades as it warms up'. When measured, that was wrong: all forty of forty runs stayed on the custom plan.

WHY Most likely explanation: the generic plan, which cannot use the partial index, comes out with a high estimated cost. Plan costs were not recorded.

COST Partial is re-planned on every run. At this scale the difference could not be measured: 0.20 ms vs 0.21 ms.

SECTION 03

Now, 15 minutes

Sustained churn: 2,000 jobs per second, default autovacuum settings.

03

PARTIAL · ~305×

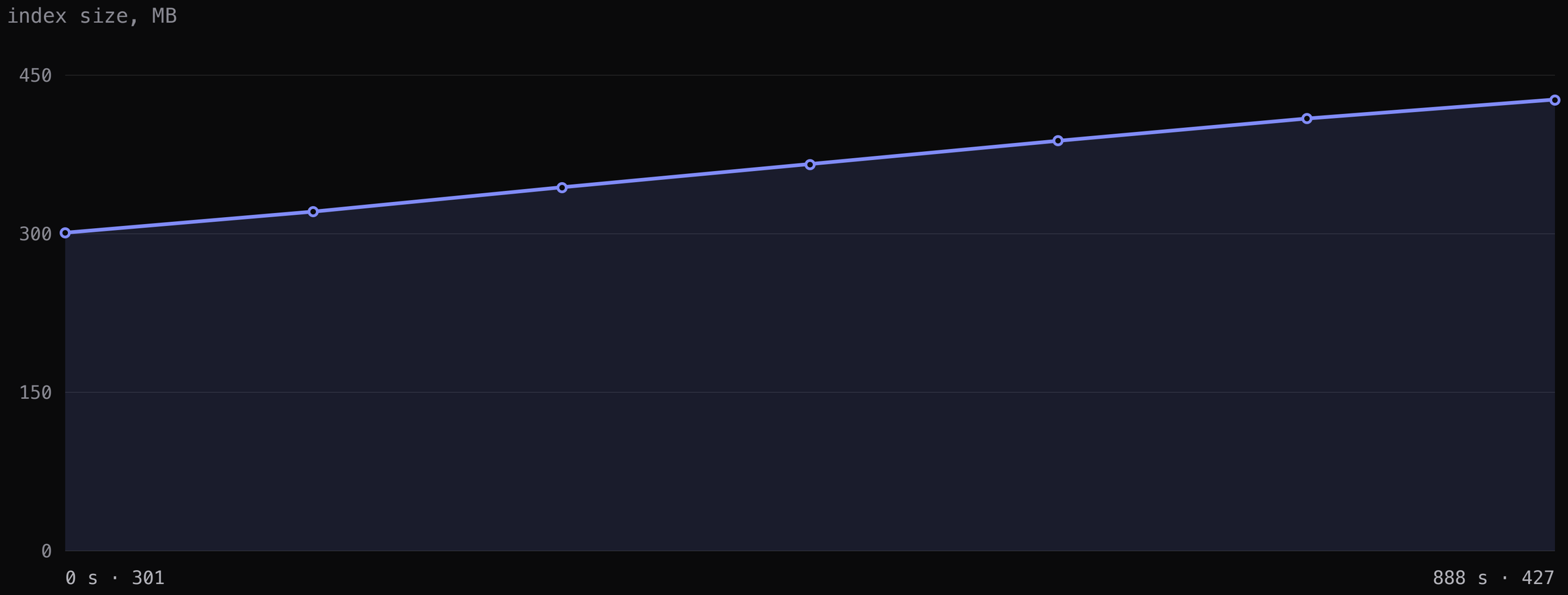
The live set is constant, the index grows



pg_relation_size, sampled every 15 seconds, bench/endurance.sh. The pending→done transition leaves dead entries until vacuum.

COMPOSITE · 42%, +126 MB

Composite bloated by a smaller ratio, but more MB



pg_relation_size, sampled every 15 seconds, bench/endurance.sh

TARGET 2,000 JOBS/S · SECOND 900

A bloated index cannot carry the sustained load

	composite	partial
START	2,000 tps · 0.52 ms	✓ 2,024 tps · 1.7 ms
SECOND 900	1,204 tps · 61,533 ms	✓ 2,016 tps · 4,906 ms
PENDING JOBS	5,000 → 126,024	✓ 5,003 → 14,364
RESULT	Missed the target	✓ Met the target

WHILE READING

Two notes

NOTE

Latency includes pgbench's schedule lag: the debt of a client that cannot keep up with the target rate is booked here.

NOTE

The gap opens under sustained load: an index bloated to 427 MB no longer fits in memory, and every scan goes to disk.

THE THRESHOLD GROWS WITH THE TABLE

Autovacuum never came

1.75

Million dead rows

1,753,949 in 15 minutes

0

Autovacuum

Did not run even once

≈2

Threshold, million rows

50 + 0.2 × 10,005,000 ≈ 2,001,050

17

Minutes, estimated interval

Specific to this workload; vacuum cost was not measured

RECOMMENDATION

Decouple the threshold from the table

I did not measure this setting. 50,000 is a starting point; tune it to your own throughput.

jobs.sql

```sql
ALTER TABLE jobs SET (
  autovacuum_vacuum_scale_factor = 0,
  autovacuum_vacuum_threshold    = 50000
);

CREATE INDEX CONCURRENTLY
  jobs_pending_created_at
  ON jobs (created_at)
WHERE status = 'pending';
```

RESULT

The claims in the Sor Bakalım answer, once measured

- A partial index stays small: 7.6 MB at 10M
- Beats the (status) index: 11,537 vs 6,426 tps
- The ORDER BY column should go into the index
- With a parameter, the planner may not pick the partial: true, but narrow
- Churn and bloat: not measured in the first post, measured afterwards



Find the condition

github.com/muhammetsafak/pg-queue-bench

The value of measuring a piece of advice is not confirming it. It is finding the condition under which a correct sentence flips.

