Building a queue table's index around its live set

A partial index on a Postgres 17 queue table: 41 times smaller, a cliff on the generic plan, bloat under 15 minutes of churn, and an autovacuum that never came.

01 POSTGRESQL · MEASUREMENT

Building a queue table's index around its live set

Single machine, Postgres 17, synthetic bench: a 30-second measurement, the planner, and 15 minutes of churn.

Muhammet Şafak · October 3, 2026



41×

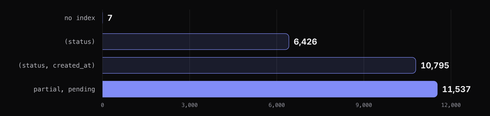
02 AT 10 MILLION DEAD ROWS

smaller index

Partial is 7.6 MB, 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.

03 10M DEAD ROWS, 5,000 LIVE ROWS

A queue table without an index vanishes



partial: (created_at) WHERE status = 'pending' · 8 clients, 30 seconds, median of three runs · bench/run.sh

04 SAME TABLE, SAME INDEX, SAME QUERY

The partial index on a generic plan

11,752
tps, custom plan
Index Scan, 0.68 ms

7
tps, generic plan
Sort → Seq Scan; index never scanned

1,113
ms, generic latency
Roughly 1,635 times; forced with force_generic_plan

11,416
Composite, generic
No predicate to prove, unaffected

05 DEFAULT AUTO MODE, SINGLE SESSION

Forty runs, forty custom plans

#0	First five runs: all three on custom plan	done
#1	Sixth: composite switches to generic	done
#2	Partial: 40 of 40 custom	done
#3	(status): 40 of 40 custom	done

06 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.

07 15 MIN · 2,000 JOBS/S · PG_RELATION_SIZE, EVERY 15 S

The live set is constant, the indexes grow

TIME	PARTIAL	COMPOSITE
0 s	0.125 MB	301 MB
286 s	12.4 MB	344 MB
587 s	25.3 MB	388 MB
888 s	38.2 MB	427 MB
Growth	~305×	42%, +126 MB

08 TARGET 2,000 JOBS/S · SECOND 900

Two indexes that were nearly equal at thirty seconds split apart at fifteen minutes

	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; 427 MB does not fit in memory	✓ Met the target

09 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

10 WHAT TO DO

Build the queue table around its live set

- [x] Index the queue: partial (created_at) WHERE status = 'pending'
- [x] Do not reach a predicate index through a parameter
- [x] Decouple the autovacuum threshold from the table; the effect was not measured, 50,000 is a starting point
- [x] First look at the pending/done ratio, the index size, n_dead_tup and the threshold
- [ ] Drop the pattern: if you delete completed rows right away or move them to an archive, if your ORM binds status as a parameter and a generic plan is forced, if you partition the table by time



Table size is noise

github.com/muhammetsafak/pg-queue-bench

Build the index and vacuum around the set that does the work. The numbers come from a single machine, Postgres 17 and a single access pattern; they do not represent months of a real queue.

