

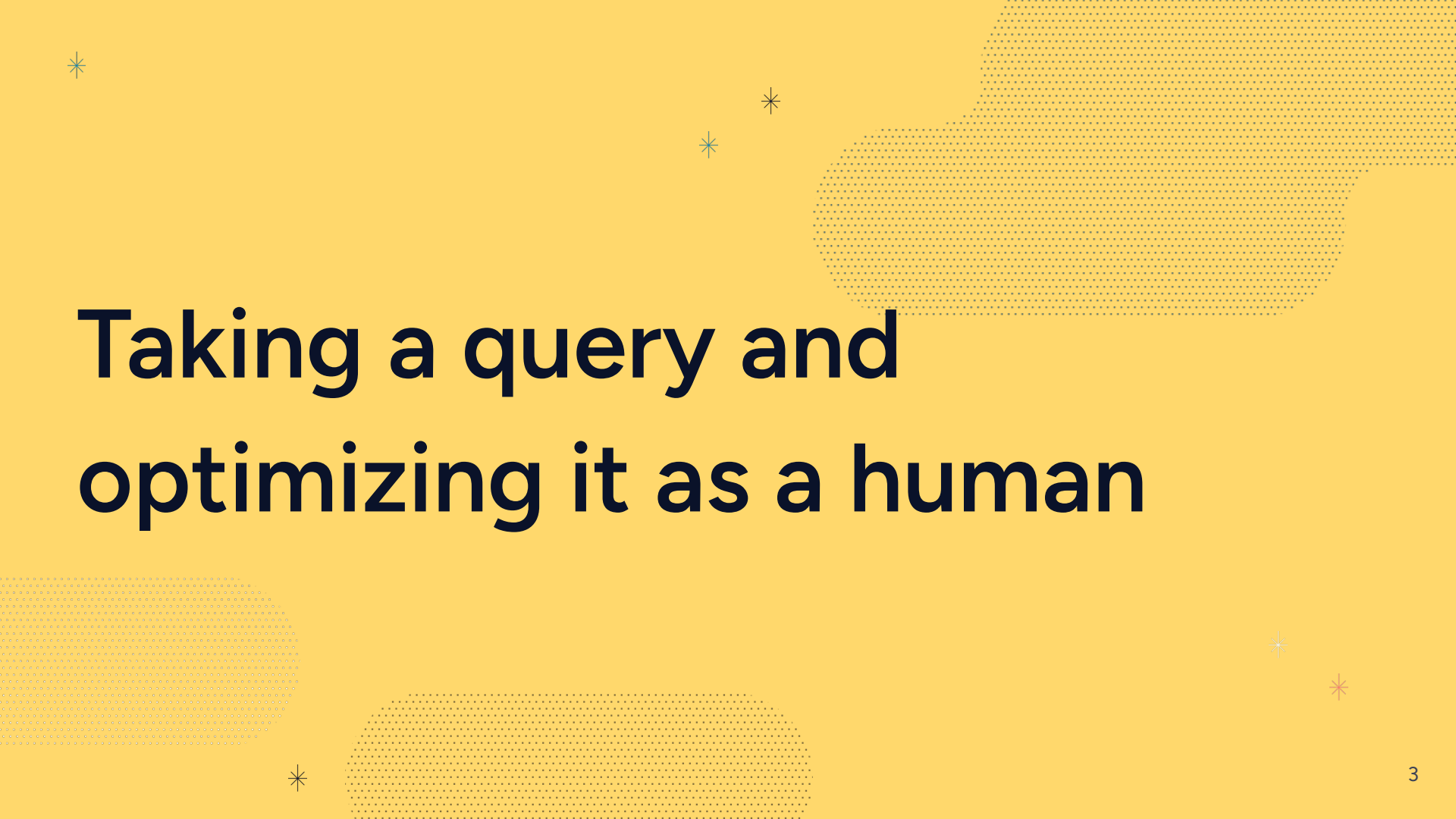


Optimizing Postgres Queries with Visualizations and LLMs

Lukas Fittl & Maciek Sakrejda - Postgres Summit US 2026

Agenda for today

- Taking a query and optimizing it (as a human), and how visualizations help
- Working with LLMs to optimize queries
- Enabling LLMs to test database changes



Taking a query and optimizing it as a human



Let's find a slow query

Let's find a slow query

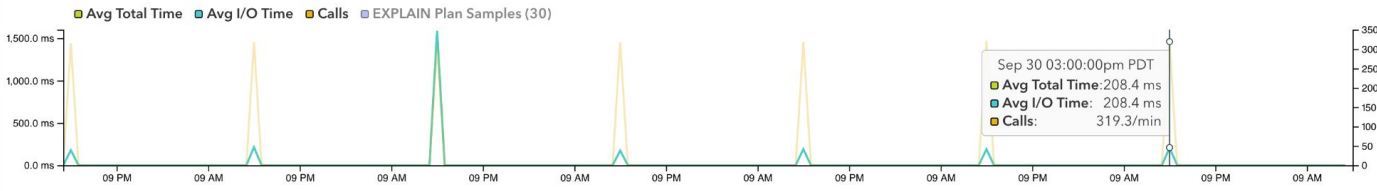
SQL Statement

```
(  
  SELECT query_occurrences.query_id  
  FROM query_occurrences JOIN query_analyses USING (query_id)  
  WHERE query_occurrences.database_id = $1 AND query_occurrences.last >= current_date.  
)
```

Show full query text

Create Workbook

Avg Time & Calls



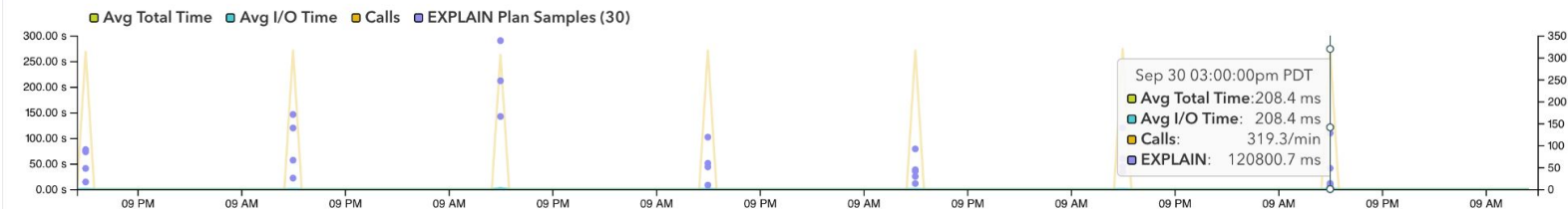
The average is less than 1 second - but we've seen issues with it taking much longer.

Let's find a slow query

```
(
  (
    SELECT query_occurrences.query_id
    FROM
      query_occurrences
    JOIN query_analyses USING (query_id)
    WHERE
      query_occurrences.database_id = $1
      AND query_occurrences.last >= (current_date - $2)
      AND query_analyses.database_id = $3
      AND (query_analyses.updated_at IS NULL OR query_analyses.updated_at < (now() - $4::interval))
    ORDER BY query_analyses.updated_at ASC NULLS FIRST
    LIMIT $5
  )
  UNION (
    SELECT query_id
    FROM query_analyses
    WHERE
      database_id = $6
      AND output_json IS NULL
      AND (summary -> $7 IS NULL OR (summary ->> $8)::int < $9 OR summary ->> $10 IS NULL OR (summary ->> $11)::int < $12)
      AND error !~ ALL(ARRAY[$13, $14, $15, $16, $17, $18, $19, $20, $21, $22])
    ORDER BY updated_at ASC
    LIMIT $23
  )
)
UNION (
  WITH updated_tables AS (
    SELECT id
    FROM schema_tables
    WHERE
      database_id = $26
      AND updated_at > $27
      AND removed_at IS NULL
  )
  SELECT query_id
  FROM query_analyses
  WHERE
    database_id = $24
    AND table_ids && (
      SELECT array_agg(id)
      FROM updated_tables
    )
    AND output_json IS NOT NULL
  ORDER BY updated_at ASC
  LIMIT $25
)
```

Let's find a slow query

Avg Time & Calls



Outliers for certain customers can be 1000x the average.

Let's find a slow query

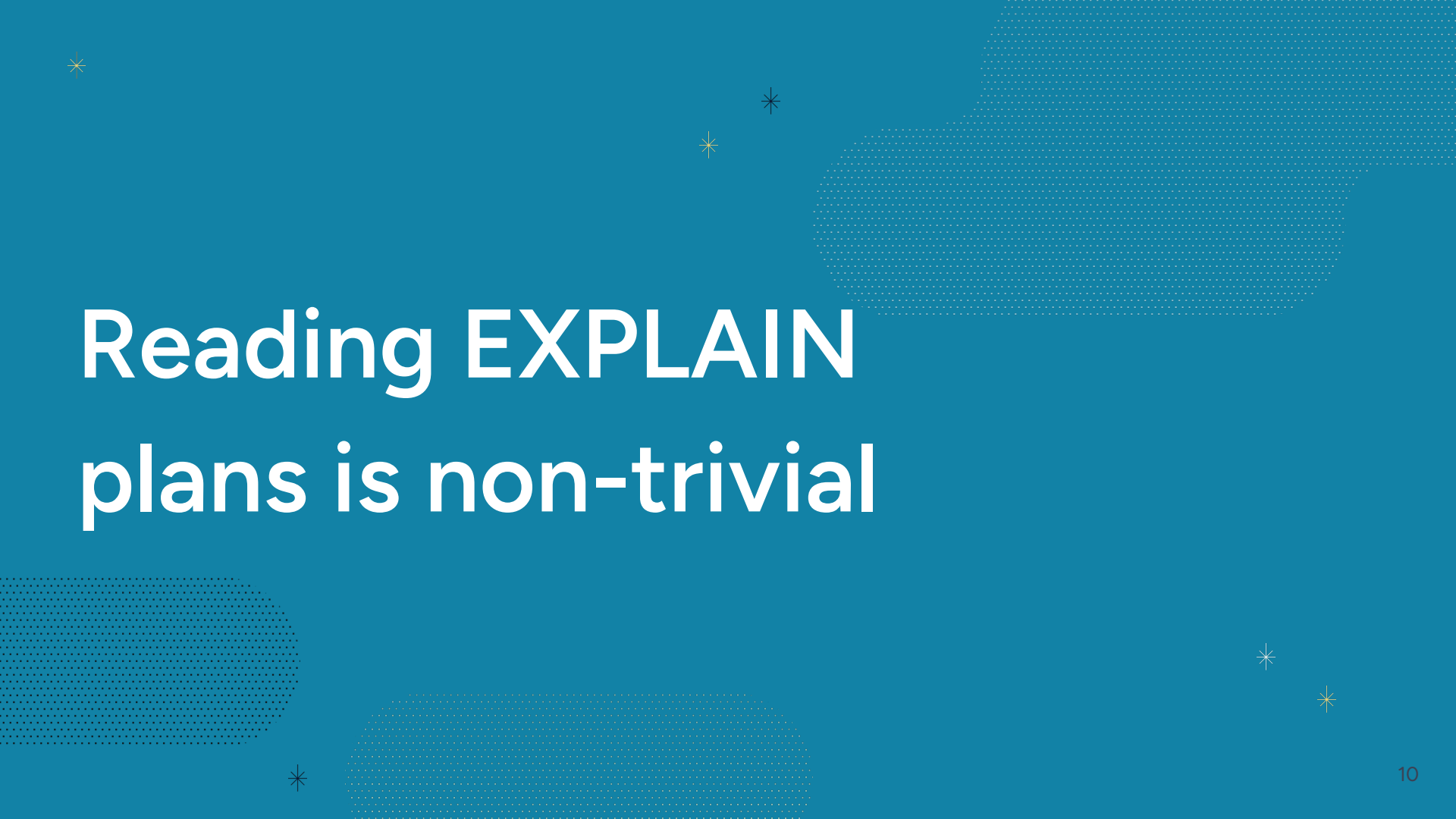
This outlier recorded with **auto_explain** took 46,793ms - why?

```
HashAggregate (cost=1892701.98..1892719.92 rows=1794 width=8) (actual time=46787.033..46792.976 rows=1000 loops=1)
  Group Key: "SELECT* 1".query_id
  Batches: 1 Memory Usage: 129kB
  Buffers: shared hit=526224 read=86714 dirtied=22020
  I/O Timings: shared/local read=118575.841
-> Append (cost=638322.59..1892697.49 rows=1794 width=8) (actual time=18128.808..46792.669 rows=1000 loops=1)
  Buffers: shared hit=526224 read=86714 dirtied=22020
  I/O Timings: shared/local read=118575.841
-> Subquery Scan on "SELECT* 1" (cost=638322.59..638406.59 rows=720 width=8) (actual time=18128.807..18130.661 rows=1000 loops=1)
  Buffers: shared hit=413919 read=33664 dirtied=11129
  I/O Timings: shared/local read=52027.476
-> Limit (cost=638322.59..638406.59 rows=720 width=16) (actual time=18128.806..18130.570 rows=1000 loops=1)
  Buffers: shared hit=413919 read=33664 dirtied=11129
  I/O Timings: shared/local read=52027.476
-> Gather Merge (cost=638322.59..638406.59 rows=720 width=16) (actual time=18128.805..18130.488 rows=1000 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  Buffers: shared hit=413919 read=33664 dirtied=11129
  I/O Timings: shared/local read=52027.476
-> Sort (cost=637322.56..637323.46 rows=360 width=16) (actual time=18123.883..18123.942 rows=1000 loops=3)
  Sort Key: query_analyses.updated_at NULLS FIRST
  Sort Method: top-N heapsort Memory: 113kB
  Buffers: shared hit=413919 read=33664 dirtied=11129
  I/O Timings: shared/local read=52027.476
  Worker 0: actual time=18124.745..18124.795 rows=1000 loops=1
  Sort Method: top-N heapsort Memory: 116kB
  Buffers: shared hit=137847 read=11177 dirtied=3785
  I/O Timings: shared/local read=17259.475
  Worker 1: actual time=18121.731..18121.782 rows=1000 loops=1
  Sort Method: top-N heapsort Memory: 115kB
  Buffers: shared hit=138899 read=11260 dirtied=3725
  I/O Timings: shared/local read=17316.994
-> Nested Loop (cost=4264.11..637307.28 rows=360 width=16) (actual time=191.335..18118.139 rows=15264 loops=3)
  ...
-> Subquery Scan on "SELECT* 2" (cost=1215744.10..1215744.10 rows=1 width=8) (actual time=28657.560..28661.876 rows=0 loops=1)
...
```

Let's find a slow query

EXPLAIN (COSTS OFF) can help us see the **plan shape** more clearly:

```
HashAggregate
-> Append
  -> Subquery Scan on "SELECT* 1"
    -> Limit
      -> Gather Merge
        Workers Planned: 2
        -> Sort
          Sort Key: query_analyses.updated_at NULLS FIRST
          -> Nested Loop
            -> Parallel Bitmap Heap Scan on query_occurrences
              Recheck Cond: ((database_id = '...':integer) AND (last >= (CURRENT_DATE - 1)))
              -> Bitmap Index Scan on index_query_occurrences_on_database_id_and_last
                Index Cond: ((database_id = '...':integer) AND (last >= (CURRENT_DATE - 1)))
            -> Index Scan using index_query_analyses_on_query_id on query_analyses
              Index Cond: (query_id = query_occurrences.query_id)
              Filter: ((database_id = '...':integer) AND ((updated_at IS NULL) OR (updated_at < (now() - '7 days':interval))))
  -> Subquery Scan on "SELECT* 2"
    -> Limit
      -> Sort
        Sort Key: query_analyses_1.updated_at
        -> Gather
          Workers Planned: 2
          -> Parallel Bitmap Heap Scan on query_analyses query_analyses_1
            Recheck Cond: (database_id = '...':integer)
            Filter: ...
            -> Bitmap Index Scan on query_analyses_database_id_error_idx
              Index Cond: (database_id = '...':integer)
  -> Subquery Scan on "SELECT* 3"
    -> Limit
      InitPlan 1 (returns $0)
      -> Aggregate
        -> Index Scan using schema_tables_database_id_removed_at_idx on schema_tables
          Index Cond: ((database_id = '...':integer) AND (removed_at IS NULL))
          Filter: (updated_at > '2026-09-29 22:06:16.442424+00':timestamp with time zone)
      -> Sort
        Sort Key: query_analyses_2.updated_at
        -> Bitmap Heap Scan on query_analyses query_analyses_2
          Recheck Cond: ((table_ids && $0) AND (database_id = '...':integer))
          Filter: (output_json IS NOT NULL)
          -> BitmapAnd
            -> Bitmap Index Scan on index_query_analyses_on_table_ids
              Index Cond: (table_ids && $0)
            -> Bitmap Index Scan on query_analyses_database_id_error_idx
              Index Cond: (database_id = '...':integer)
```



Reading EXPLAIN plans is non-trivial

Reading EXPLAIN plans is non-trivial



Different way to look at a query plan

- overall plan shape vs. node details
- cumulative metrics vs. node's own metrics
- aggregate metrics vs. metrics per loop

Reading EXPLAIN plans is non-trivial



Different way to look at a query plan

Overall plan shape vs. node details:

```
-> Subquery Scan on "*SELECT* 1"  
  -> Limit  
    -> Gather Merge  
      Workers Planned: 2  
      -> Sort  
        Sort Key: query_analyses.updated_at NULLS FIRST  
        -> Nested Loop  
          -> Parallel Bitmap Heap Scan on ...  
            Recheck Cond: ...  
            -> Bitmap Index Scan on ..  
              Index Cond: ...  
          -> Index Scan using ..  
            Index Cond: (query_id =...)  
            Filter: ...
```

```
-> Gather Merge  
(cost=638322.59..638406.59 rows=720 width=16)  
(actual time=18128.805..18130.488 rows=1000 loops=1)  
  Workers Planned: 2  
  Workers Launched: 2  
  Buffers: shared hit=413919 read=33664 dirtied=11129  
  I/O Timings: shared/local read=52027.476
```

Reading EXPLAIN plans is non-trivial



Different way to look at a query plan

Cumulative metrics vs. node's own metrics

- > Parallel Bitmap Heap Scan on public.query_occurrences
(cost=4263.54..**310106.47** rows=120438 width=8)
(actual time=185.923..7344.525 rows=30794 loops=3)
 - > Bitmap Index Scan on index_query_occurrences_on_database_id_and_last
(cost=0.00..**4191.28** rows=289050 width=0)
(actual time=184.243..184.243 rows=102496 loops=1)

Own cost of Bitmap Heap Scan = 305915.19 (310106.47 - 4191.28)

Reading EXPLAIN plans is non-trivial



Different way to look at a query plan

Aggregate metrics vs. metrics per loop

```
-> Parallel Bitmap Heap Scan on public.query_occurrences  
    (cost=4263.54..310106.47 rows=120438 width=8)  
    (actual time=185.923..7344.525 rows=30794 loops=3)
```

Time per loop is 7,344.525 ms

Aggregate time is 22,033.58 ms

CTEs are kind of weird

```
...
-> Limit (cost=38535.14..38537.83 rows=1073 width=16) (actual time=0.042..0.046 rows=0 loops=1)
    Buffers: shared hit=13
    InitPlan 1 (returns $0)
        -> Aggregate (cost=741.26..741.27 rows=1 width=32) (actual time=0.031..0.032 rows=1 loops=1)
            Buffers: shared hit=13
            -> Index Scan using ... on public.schema_tables (...) (actual time=0.029..0.029 rows=0 loops=1)
                Filter: (schema_tables.updated_at > '2026-09-29 22:06:16.442424+00'::timestamp with time zone)
                Index Cond: ((schema_tables.database_id = '1234'::integer) AND (schema_tables.removed_at IS NULL))
                Rows Removed by Filter: 39
                Buffers: shared hit=13
        -> Sort (cost=37793.87..37796.56 rows=1073 width=16) (actual time=0.041..0.044 rows=0 loops=1)
            Sort Key: query_analyses_2.updated_at
            Sort Method: quicksort Memory: 25kB
            Buffers: shared hit=13
...
```

Reading EXPLAIN plans is non-trivial

This InitPlan corresponds to

```
...
UNION (
  WITH updated_tables AS (
    SELECT id
    FROM schema_tables
    WHERE database_id = 1234 AND updated_at > '2026-09-29 22:06:16.442424'
    AND removed_at IS NULL
  )
  SELECT query_id
  FROM query_analyses
  WHERE database_id = 1234
    AND table_ids && (SELECT array_agg(id) FROM updated_tables)
    AND output_json IS NOT NULL
  ORDER BY updated_at ASC
  LIMIT 5000
)
...
```

Parallelism is kind of weird

```
...
-> Gather Merge (cost=638322.59..638406.59 rows=720 width=16) (actual time=18128.805..18130.488 rows=1000 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    Buffers: shared hit=413919 read=33664 dirtied=11129
    I/O Timings: shared/local read=52027.476
-> Sort (cost=637322.56..637323.46 rows=360 width=16) (actual time=18123.883..18123.942 rows=1000 loops=3)
    Sort Key: query_analyses.updated_at NULLS FIRST
    Sort Method: top-N heapsort Memory: 113kB
    Buffers: shared hit=413919 read=33664 dirtied=11129
    I/O Timings: shared/local read=52027.476
    Worker 0: actual time=18124.745..18124.795 rows=1000 loops=1
        Sort Method: top-N heapsort Memory: 116kB
        Buffers: shared hit=137847 read=11177 dirtied=3785
        I/O Timings: shared/local read=17259.475
    Worker 1: actual time=18121.731..18121.782 rows=1000 loops=1
        Sort Method: top-N heapsort Memory: 115kB
        Buffers: shared hit=138899 read=11260 dirtied=3725
        I/O Timings: shared/local read=17316.994
...
```

Reading EXPLAIN plans is non-trivial



Parallelism is kind of weird

```
...
-> Gather Merge (cost=638322.59..638406.59 rows=720 width=16) (actual time=18128.805..18130.488 rows=1000 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    Buffers: shared hit=413919 read=33664 dirtied=11129
    I/O Timings: shared/local read=52027.476
-> Sort (cost=637322.56..637323.46 rows=360 width=16) (actual time=18123.883..18123.942 rows=1000 loops=3)
    Sort Key: query_analyses.updated_at NULLS FIRST
    Sort Method: top-N heapsort  Memory: 113kB
    Buffers: shared hit=4139 read=33664 dirtied=11129
    I/O Timings: shared/local read=52027.476
    Worker 0:  actual time=18124.745..18124.795 rows=1000 loops=1
        Sort Method: top-N heapsort  Memory: 116kB
        Buffers: shared hit=13784 read=11177 dirtied=3785
        I/O Timings: shared/local read=17259.475
    Worker 1:  actual time=18121.731..18121.782 rows=1000 loops=1
        Sort Method: top-N heapsort  Memory: 115kB
        Buffers: shared hit=13889 read=11260 dirtied=3725
        I/O Timings: shared/local read=17316.994
...
```

Reading EXPLAIN plans is non-trivial



Parallelism is kind of weird

```
...
-> Gather Merge (cost=638322.59..638406.59 rows=720 width=16) (actual time=18128.805..18130.488 rows=1000 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    Buffers: shared hit=413919 read=33664 dirtied=11129
    I/O Timings: shared/local read=52027.476
-> Sort (cost=637322.56..637323.46 rows=360 width=16) (actual time=18123.881..18123.942 rows=1000 loops=3)
    Sort Key: query_analyses.updated_at NULLS FIRST
    Sort Method: top-N heapsort Memory: 113kB
    Buffers: shared hit=413919 read=33664 dirtied=11129
    I/O Timings: shared/local read=52027.476
    Worker 0: actual time=18124.745..18124.795 rows=1000 loops=1
        Sort Method: top-N heapsort Memory: 116kB
        Buffers: shared hit=137847 read=11177 dirtied=3785
        I/O Timings: shared/local read=17259.475
    Worker 1: actual time=18121.731..18121.782 rows=1000 loops=1
        Sort Method: top-N heapsort Memory: 115kB
        Buffers: shared hit=138899 read=11260 dirtied=3725
        I/O Timings: shared/local read=17316.994
...
```

And other gotchas

- hard to correlate plan with original query
- buffer counts are accesses, not distinct pages touched
- workers planned vs. launched, disk spills, etc.
- many, many more

And the docs are pretty thin...

14.1. Using EXPLAIN

14.1.1. EXPLAIN Basics

14.1.2. EXPLAIN ANALYZE

14.1.3. Caveats



The pganalyze approach



Show ⓘ: ☐ Est. Cost ☒ Runtime ☐ Rows ☐ Buffers ☐ Reads ☐ Writes

Plan

		Runtime
1	Aggregate	0.31ms
2	Append	0.09ms
3	Subquery Scan	0.09ms
4	Limit	0.08ms
5	Gather Merge	0.00ms
6	Sort	17.41ms ↻ 3.0
7	Nested Loop	79.87ms ↻ 3.0
8	Parallel Bitmap Heap Scan on query_occurrences	21,849.33ms ↻ 3.0
9	Bitmap Index Scan on index_query_occurrences_on_database...	184.24ms
10	Index Scan (Forward) on query_analyses	32,240.97ms ↻ 92k
11	Subquery Scan	0.00ms
12	Limit	0.00ms
13	Sort	0.01ms
14	Gather	0.00ms
15	Parallel Bitmap Heap Scan on query_analyses	77,165.29ms ↻ 3.0
16	Bitmap Index Scan on index_query_analyses_on_database_id	8,794.51ms
17	Subquery Scan mis-estimate	0.00ms
18	Limit mis-estimate	0.00ms
21	Sort mis-estimate	0.01ms
22	Bitmap Heap Scan on query_analyses mis-estimate	0.00ms
23	BitmapAnd	0.00ms
24	Bitmap Index Scan on index_query_analyses_on_ta... mis-estimate	0.03ms
25	Bitmap Index Scan on index_que... (never executed) mis-estimate	0.00ms

InitPlan 1 (returns \$0)

19	Aggregate	0.00ms
20	Index Scan (Forward) on schema_tables	0.03ms

5	└─┬─ Gather Merge	0.00ms
6	└─┬─┬─ Sort	17.41ms 3.0
7	└─┬─┬─┬─ Nested Loop	79.87ms 3.0
8	└─┬─┬─┬─┬─ Parallel Bitmap Heap Scan on query_occurrences	21,849.33ms 3.0
9	└─┬─┬─┬─┬─┬─ Bitmap Index Scan on index_query_occurrences_on_database...	184.24ms
10	└─┬─┬─┬─┬─┬─ Index Scan (Forward) on query_analyses	32,240.97ms 92k
11	└─ Subquery Scan	0.00ms
12	└─┬─ Limit	0.00ms
13	└─┬─┬─ Sort	0.01ms
14	└─┬─┬─┬─ Gather	0.00ms
15	└─┬─┬─┬─┬─ Parallel Bitmap Heap Scan on query_analyses	77,165.29ms 3.0
	on public.query_analyses AS query_analyses_1	
	Exact Heap Blocks: 54063	
	Filter: ((query_analyses_1.output_json IS NULL) AND (query_analyses_1.error !~ ALL ('{... 3.0	
	Lossy Heap Blocks: 0	
	Recheck Cond: (query_analyses_1.database_id = ' '::integer)	
	Rows Removed by Filter: 387376	
	Rows Removed by Index Recheck: 0	
16	└─┬─┬─┬─┬─ Bitmap Index Scan on index_query_analyses_on_database_id	8,794.51ms
17	└─ Subquery Scan mis-estimate	0.00ms
18	└─┬─ Limit mis-estimate	0.00ms
21	└─┬─┬─ Sort mis-estimate	0.01ms
22	└─┬─┬─┬─ Bitmap Heap Scan on query_analyses mis-estimate	0.00ms
23	└─┬─┬─┬─┬─ BitmapAnd	0.00ms
24	└─┬─┬─┬─┬─┬─ Bitmap Index Scan on index_query_analyses_on_ta... mis-estimate	0.03ms
25	└─┬─┬─┬─┬─┬─ Bitmap Index Scan on index_que... (never executed) mis-estimate	0.00ms

InitPlan 1 (returns \$0)

19	└─ Aggregate	0.00ms
20	└─┬─ Index Scan (Forward) on schema_tables	0.03ms



How to compare plans

Plan Comparison

Select plans

Compare **1**: ☐ Est. Cost ☒ Runtime ☐ Rows ☐ Buffers ☐ Reads ☐ Writes

Plan A

2026-09-30 10:07:03pm UTC

```
-> Aggregate
-> Append
-> Subquery Scan
-> Limit
-> Gather Merge
-> Sort

-> Nested Loop
-> Bitmap Heap Scan on que...
-> Bitmap Index Scan4
-> Index Scan2 on query_an...
-> Subquery Scan
-> Limit
-> Sort
-> Gather
-> Bitmap Heap Scan on quer...
-> Bitmap Index Scan1
-> Subquery Scan
-> Limit
-> Sort
-> Bitmap Heap Scan on query...
-> BitmapAnd
-> Bitmap Index Scan3
-> Bitmap Index Scan1
```

InitPlan 1 (returns \$0)

```
-> Aggregate
-> Index Scan5 on schema_tables
```

Plan B

2026-10-01 10:12:44pm UTC

```
-> Aggregate
-> Append
-> Subquery Scan
-> Limit
-> Sort
-> Gather
-> Nested Loop
-> Bitmap Heap Scan on que...
-> Bitmap Index Scan4
-> Index Scan2 on query_an...
-> Subquery Scan
-> Limit
-> Sort
-> Gather
-> Bitmap Heap Scan on quer...
-> Bitmap Index Scan1
-> Subquery Scan
-> Limit
-> Sort
-> Bitmap Heap Scan on query...
-> BitmapAnd
-> Bitmap Index Scan1
-> Bitmap Index Scan3
```

```
-> Aggregate
-> Index Scan5 on schema_tables
```

Plan A

Runtime

```
0.31ms
0.09ms
0.09ms
0.08ms
5.24ms
5.89ms

42.90ms
7,385.55ms
61.41ms
10,934.46ms
0.00ms
0.00ms
0.01ms
7.06ms
25,723.30ms
2,931.50ms
0.00ms
0.00ms
0.01ms
0.00ms
0.00ms
0.03ms
0.00ms

0.00ms
0.03ms
```

Plan B

Runtime

```
0.77ms
0.21ms
0.10ms
0.08ms

4.23ms
3.86ms
11.61ms
849.97ms
5.58ms
2,677.13ms
0.00ms
0.00ms
0.01ms
1.57ms
416.93ms
68.24ms
0.15ms
0.12ms
0.36ms
1.86ms
1.13ms

4.94ms
5.91ms

0.01ms
0.36ms
```

From Plan B

Gather

Combines the results from several children producing data in parallel.

Single Copy: false
Workers Launched: 1
Workers Planned: 1

Insights (1)

mis-estimate had rows under-estimated by a factor of 18921

I/O & Buffers

☒ Include cumulative costs of node children

	Shared 1	Local 1	Temp 1
Hit 1	999.4 MB	0 B	-
Read 1	57.5 MB	0 B	0 B
Dirtyed 1	8.6 MB	0 B	-
Written 1	0 B	0 B	0 B

I/O Read Time 6,646.51ms
I/O Write Time 0.00ms

Output Columns

- query_occurrences.query_id
- query_analyses.updated_at



SQL Statement

```
( SELECT query_occurrences.query_id
FROM query_occurrences
JOIN query_analyses USING (query_id)
WHERE query_occurrences.database_id =
AND query_occurrences.last >= curr...
```

Show full query text

Plan Comparison

Select plans

Compare ⓘ: ☐ Est. Cost ☒ Runtime ☐ Rows ☐ Buffers ☐ Reads ☐ Writes

Plan A

2026-09-30 10:07:03pm UTC

```
-> Aggregate
-> Append
-> Subquery Scan
-> Limit
-> Gather Merge
-> Sort
-> Nested Loop
-> Bitmap Heap Scan on que...
-> Bitmap Index Scan4
-> Index Scan2 on query_an...
-> Subquery Scan
-> Limit
-> Sort
-> Gather
-> Bitmap Heap Scan on quer...
-> Bitmap Index Scan1
-> Subquery Scan
```

Plan B

2026-10-01 10:12:44pm UTC

```
-> Aggregate
-> Append
-> Subquery Scan
-> Limit
-> Sort
-> Gather
-> Nested Loop
-> Bitmap Heap Scan on que...
-> Bitmap Index Scan4
-> Index Scan2 on query_an...
-> Subquery Scan
-> Limit
-> Sort
-> Gather
-> Bitmap Heap Scan on quer...
-> Bitmap Index Scan1
-> Subquery Scan
```

Plan A

Runtime

```
0.31ms
0.09ms
0.09ms
0.08ms
5.24ms
5.89ms
42.90ms
7,385.55ms
61.41ms
10,934.46ms
0.00ms
0.00ms
0.01ms
7.06ms
25,723.30ms
2,931.50ms
0.00ms
```

Plan B

Runtime

```
0.77ms
0.21ms
0.10ms
0.08ms
4.23ms
3.86ms
11.61ms
849.97ms
5.58ms
2,677.13ms
0.00ms
0.00ms
0.01ms
1.57ms
416.93ms
68.24ms
0.15ms
```

From Plan A

Sort

Takes data from a child node and produces sorted output, using either memory if available (depending on the work_mem setting) or “spilling” to disk. This is obviously necessary if the output needs to be sorted, though sometimes the input can be scanned in an already-sorted manner instead—e.g., if scanning a btree index compatible with the desired sort order.

Sort Method: top-N heapsort

Sort Key: query_analyses.updated_at NULLS FIRST

Sort Space Type: Memory

Sort Space Used: 113

I/O & Buffers

Include cumulative costs of node children

	Shared ⓘ	Local ⓘ	Temp ⓘ
Hit ⓘ	3.2 GB	0 B	-
Read ⓘ	263 MB	0 B	0 B
Dirtied ⓘ	86.9 MB	0 B	-
Written ⓘ	0 B	0 B	0 B

I/O Read Time
52,027.48ms

I/O Write Time
0.00ms

Output Columns

- query_occurrences.query_id
- query_analyses.updated_at

SQL Statement

```
( SELECT query_occurrences.query_id
FROM query_occurrences
JOIN query_analyses USING (query_id)
WHERE query_occurrences.database_id =
AND query_occurrences.last >= curr...
```

Show full query text

Plan Comparison

Select plans

Compare **1**: ☐ Est. Cost ☒ Runtime ☐ Rows ☐ Buffers ☐ Reads ☐ Writes

Plan A

2026-09-30 10:07:03pm UTC

- > Aggregate
- > Append
- > Subquery Scan
- > Limit
- > Gather Merge
- > Sort
- > Nested Loop
- > Bitmap Heap Scan on que...
- > Bitmap Index Scan⁴
- > Index Scan² on query_an...
- > Subquery Scan
- > Limit
- > Sort
- > Gather
- > Bitmap Heap Scan on quer...
- > Bitmap Index Scan¹
- > Subquery Scan

Plan B

2026-10-01 10:12:44pm UTC

- > Aggregate
- > Append
- > Subquery Scan
- > Limit
- > Sort
- > Gather
- > Nested Loop
- > Bitmap Heap Scan on que...
- > Bitmap Index Scan⁴
- > Index Scan² on query_an...
- > Subquery Scan
- > Limit
- > Sort
- > Gather
- > Bitmap Heap Scan on quer...
- > Bitmap Index Scan¹
- > Subquery Scan

Plan A

Runtime

0.31ms
0.09ms
0.09ms
0.08ms
5.24ms
5.89ms
42.90ms
7,385.55ms
61.41ms
10,934.46ms
0.00ms
0.00ms
0.01ms
7.06ms
25,723.30ms
2,931.50ms
0.00ms

Plan B

Runtime

0.77ms
0.21ms
0.10ms
0.08ms
4.23ms
3.86ms
11.61ms
849.97ms
5.58ms
2,677.13ms
0.00ms
0.00ms
0.01ms
1.57ms
416.93ms
68.24ms
0.15ms

From Plan B

Sort

Takes data from a child node and produces sorted output, using either memory if available (depending on the work_mem setting) or "spilling" to disk. This is obviously necessary if the output needs to be sorted, though sometimes the input can be scanned in an already-sorted manner instead—e.g., if scanning a btree index compatible with the desired sort order.

Sort Method: top-N heapsort

Sort Key: query_analyses.updated_at NULLS FIRST

Sort Space Type: Memory

Sort Space Used: 114

Insights (1)

mis-estimate had rows under-estimated by a factor of 1000

I/O & Buffers

Include cumulative costs of node children

	Shared 1	Local 1	Temp 1
Hit 1	999.4 MB	0 B	-
Read 1	57.5 MB	0 B	0 B
Dirtyed 1	8.6 MB	0 B	-
Written 1	0 B	0 B	0 B
I/O Read Time	I/O Write Time		
6,646.51ms	0.00ms		

Output Columns



Where to test query changes

Staging?

Where to test query changes

Production?

Production Clone?

Where to test query changes

Add GitHub workflow to spin up copy database and test migrations #6181

Code ▾

 **Merged** [seanlinsley](#) merged 7 commits into `main` from `db_clone`  on Sep 2

 Conversation **46**  Commits **7**  Checks **7**  Files changed **20**

+1,228 -238 



seanlinsley commented on Aug 27 • edited ▾

...

Currently, testing new pco migrations on production data takes around 4 hours of manual work: I have to spin up a copy of the production database, run `database:preload_table` so the tables are fully hydrated, copy any updated Rust files onto a production console, then finally recompile and run the new migrations on the copy database.

This PR fully automates that process as a user-triggered GitHub workflow, allowing you to easily test a migration against a copy of the production data on any branch. As a final step after the migration is complete, the workflow prints out AWS commands that you can run locally to spin up an interactive console using the code from your branch, so you're able to do additional testing.

In passing, `infra/console.sh` and `infra/db-migrate.sh` were rewritten in Ruby so they can use the same helper functions that the new `infra/db_clone.rb` script uses.



Reviewers



 **msakrejda**



 **keiko713**



 **lfittl**



Assignees



No one—[assign yourself](#)

Labels



None yet

Projects



None yet

Where to test query changes

← DB clone and migration test

✓ DB clone and migration test #12

Summary

All jobs

✓ build_docker

✓ db_clone

✓ preload_tables

✓ migrate

Run details

Usage

Workflow file

Manually triggered yesterday

msakrejda → bfc68c1 main

Status

Success

Total duration

1h 22m 13s

Artifacts

1

db_clone.yml

on: workflow_dispatch

✓ build_docker / build

16m 11s

✓ db_clone

50m 20s

✓ preload_tables

28m 19s

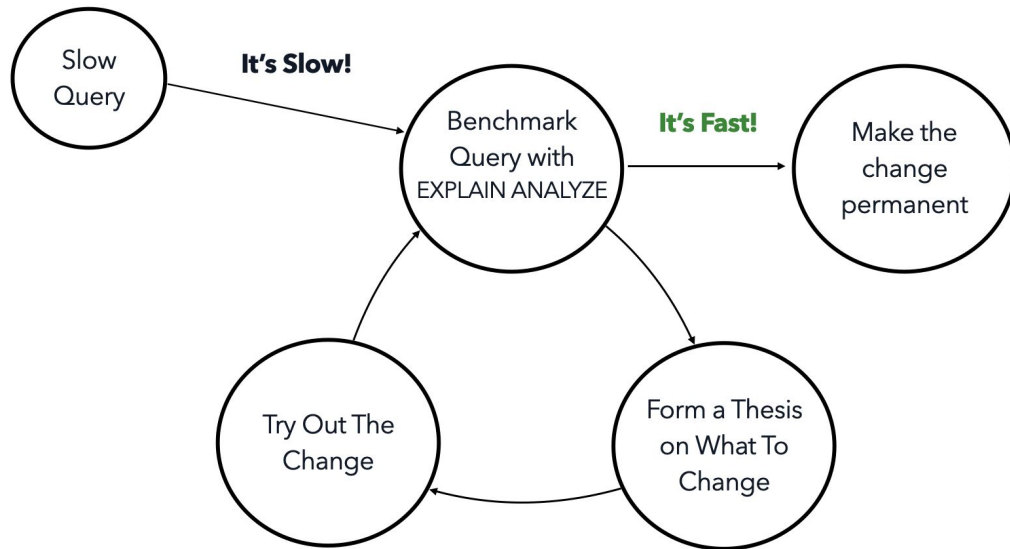
✓ migrate

3m 16s



How to come up with ideas on improving queries

How to come up with ideas on improving queries



How to come up with ideas on improving queries

Plan	Runtime
1 Aggregate	0.31ms
2 Append	0.09ms
3 Subquery Scan	0.09ms
4 Limit	0.08ms
5 Gather Merge	0.00ms
6 Sort	17.41ms (3.0)
7 Nested Loop	79.87ms (3.0)
8 Parallel Bitmap Heap Scan on query_occurrences	21,849.33ms (3.0)
9 Bitmap Index Scan on index_query_occurrences_on_database_id...	184.24ms
10 Index Scan (Forward) on query_analyses	32,240.97ms (92k)
11 Subquery Scan	0.00ms
12 Limit	0.00ms
13 Sort	0.01ms
14 Gather	0.00ms
15 Parallel Bitmap Heap Scan on query_analyses	77,165.29ms (3.0)
16 Bitmap Index Scan on index_query_analyses_on_database_id	8,794.51ms
17 Subquery Scan mis-estimate	0.00ms
18 Limit mis-estimate	0.00ms
21 Sort mis-estimate	0.01ms
22 Bitmap Heap Scan on query_analyses mis-estimate	0.00ms
23 BitmapAnd	0.00ms
24 Bitmap Index Scan on index_query_analyses_on_table... mis-estimate	0.03ms
25 Bitmap Index Scan on index_query_... (never executed) mis-estimate	0.00ms

InitPlan 1 (returns \$0)

19 Aggregate	0.00ms
20 Index Scan (Forward) on schema_tables	0.03ms

We have two distinct parts of the query that are slow.

How to come up with ideas on improving queries

Plan	Runtime
1 Aggregate	0.31ms
2 Append	0.09ms
3 Subquery Scan	0.09ms
4 Limit	0.08ms
5 Gather Merge	0.00ms
6 Sort	17.41ms (3.0)
7 Nested Loop	79.87ms (3.0)
8 Parallel Bitmap Heap Scan on query_occurrences	21,849.33ms (3.0)
9 Bitmap Index Scan on index_query_occurrences_on_database_id...	184.24ms
10 Index Scan (Forward) on query_analyses	32,240.97ms (92k)
11 Subquery Scan	0.00ms
12 Limit	0.00ms
13 Sort	0.01ms
14 Gather	0.00ms
15 Parallel Bitmap Heap Scan on query_analyses	77,165.29ms (3.0)
16 Bitmap Index Scan on index_query_analyses_on_database_id	8,794.51ms
17 Subquery Scan mis-estimate	0.00ms
18 Limit mis-estimate	0.00ms
21 Sort mis-estimate	0.01ms
22 Bitmap Heap Scan on query_analyses mis-estimate	0.00ms
23 BitmapAnd	0.00ms
24 Bitmap Index Scan on index_query_analyses_on_table... mis-estimate	0.03ms
25 Bitmap Index Scan on index_query_... (never executed) mis-estimate	0.00ms

InitPlan 1 (returns \$0)

19 Aggregate	0.00ms
20 Index Scan (Forward) on schema_tables	0.03ms

**They both have a
Get Data -> Sort -> Limit
pattern**

How to come up with ideas on improving queries

Plan	Runtime
1 Aggregate	0.31ms
2 Append	0.09ms
3 Subquery Scan	0.09ms
4 Limit	0.08ms
5 Gather Merge	0.00ms
6 Sort	17.41ms 3.0
7 Nested Loop	79.87ms 3.0
8 Parallel Bitmap Heap Scan on query_occurrences	21,849.33ms 3.0
9 Bitmap Index Scan on index_query_occurrences_on_database_id...	184.24ms
10 Index Scan (Forward) on query_analyses	32,240.97ms 92k
11 Subquery Scan	0.00ms
12 Limit	0.00ms
13 Sort	0.01ms
14 Gather	0.00ms
15 Parallel Bitmap Heap Scan on query_analyses	77,165.29ms 3.0
16 Bitmap Index Scan on index_query_analyses_on_database_id	8,794.51ms
17 Subquery Scan mis-estimate	0.00ms
18 Limit mis-estimate	0.00ms
21 Sort mis-estimate	0.01ms
22 Bitmap Heap Scan on query_analyses mis-estimate	0.00ms
23 BitmapAnd	0.00ms
24 Bitmap Index Scan on index_query_analyses_on_table... mis-estimate	0.03ms
25 Bitmap Index Scan on index_query_... (never executed) mis-estimate	0.00ms
InitPlan 1 (returns \$0)	
19 Aggregate	0.00ms
20 Index Scan (Forward) on schema_tables	0.03ms

How to come up with ideas on improving queries

11		Subquery Scan	0.00ms
12		└─ Limit	0.00ms
13		└─┬─ Sort	0.01ms
14		└─┬─┬─ Gather	0.00ms
15		└─┬─┬─┬─ Parallel Bitmap Heap Scan on query_analyses	77,165.29ms (3.0)
16		└─┬─┬─┬─┬─ Bitmap Index Scan on index query_analyses on database id	8,794.51ms

```
SELECT query_id
FROM query_analyses
WHERE
  database_id = $6
  AND output_json IS NULL
  AND (summary -> $7 IS NULL OR (summary ->> $8)::int < $9 OR summary ->> $10 IS NULL OR (summary ->> $11)::int < $12)
  AND error !~ ALL(ARRAY[$13, $14, $15, $16, $17, $18, $19, $20, $21, $22])
ORDER BY updated_at ASC
LIMIT 5000;
```

Because of the **ORDER BY**, we have to get **all** results matching the **WHERE**, before we can LIMIT.

How to come up with ideas on improving queries

11	Subquery Scan	0.00ms
12	└─ Limit	0.00ms
13	└─┬─ Sort	0.01ms
14	└─┬─ Gather	0.00ms
15	└─┬─ Parallel Bitmap Heap Scan on query_analyses	77,165.29ms 3.0
	on public.query_analyses AS query_analyses_1	
	Exact Heap Blocks: 54063	
	Filter: ((query_analyses_1.output_json IS NULL) AND (query_analyses_1.error !~ ALL ('{"rel...	
	Lossy Heap Blocks: 0	
	Recheck Cond: (query_analyses_1.database_id = ' '::integer)	
	Rows Removed by Filter: 387376	
	Rows Removed by Index Recheck: 0	
16	└─┬─ Bitmap Index Scan on index_query_analyses_on_database_id	8,794.51ms

And we're throwing away a lot of rows..

How to come up with ideas on improving queries

Show : ☐ Est. Cost ☐ Runtime ☒ Rows ☐ Buffers ☐ Reads ☐ Writes

Plan

	Actual Rows	Est. Rows
1 Aggregate	1,000	1,794
2 Append	1,000	1,794
3 Subquery Scan	1,000	720
4 Limit	1,000	720
5 Gather Merge	1,000	720
6 Sort	3,000	360
7 Nested Loop	45,792	360
8 Parallel Bitmap Heap Scan on query_occurrences	92,382	120,438
9 Bitmap Index Scan on index_query_occurrences_on...	102,496	289,050
10 Index Scan (Forward) on query_analyses	0	1
11 Subquery Scan	0	1
12 Limit	0	1
13 Sort	0	1
14 Gather	0	1
15 Parallel Bitmap Heap Scan on query_analyses on public.query_analyses AS query_analyses_1 Exact Heap Blocks: 54063 Filter: ((query_analyses_1.output_json IS NULL) AND (query_analyses_1.error !~ ALL ('{"rel... Lossy Heap Blocks: 0 Recheck Cond: (query_analyses_1.database_id = ' :':integer) Rows Removed by Filter: 387376 Rows Removed by Index Recheck: 0	0	1
16 Bitmap Index Scan on index_query_analyses_on_datab...	1,230,615	1,195,934

In fact, we're throwing away **all of them!** It just takes us a long time to do so...

How to come up with ideas on improving queries

What if...

We change the indexes so we don't need to filter as much?

How to come up with ideas on improving queries



The naive index didn't work:

```
CREATE INDEX ON query_analyses(database_id, error);

/*+ IndexScan(query_analyses query_analyses_database_id_error_idx1) */

...

Limit (cost=1258669.61..1258682.11 rows=5000 width=16) (actual time=310.990..311.786 rows=5000 loops=1)
  -> Sort (cost=1258669.61..1258739.41 rows=27918 width=16) (actual time=310.989..311.429 rows=5000 loops=1)
        Sort Key: updated_at
        Sort Method: top-N heapsort  Memory: 715kB
        -> Index Scan using query_analyses_database_id_error_idx1 on query_analyses (cost=0.57..1256814.78 rows=27918 width=16) (actual
time=0.024..0.305 rows=27468 loops=1)
            Index Cond: (database_id = '...'::integer)
            Filter: (((output_json IS NULL) AND (error !~ 'relation does not exist'::text) AND (((summary -> 'error_version'::text) IS
NULL) OR (((summary ->> 'error_version'::text))::integer < 29) OR ((summary ->> 'error_coll
ector_version'::text) IS NULL) OR (((summary ->> 'error_collector_version'::text))::integer < 7001)))
                Rows Removed by Filter: 1134358
        Planning Time: 0.159 ms
        Execution Time: 312.140 ms
```

But a sophisticated partial index would:

```
CREATE INDEX ON query_analyses(database_id) WHERE (output_json IS NULL) AND (error !~ ALL ( '{"relation \".+\" does not exist","text search configuration \".+\" does not exist","collation \".+\" for encoding \".+\" does not exist","type \".+\" does not exist","multiple queries detected as input","utility statements are not supported","cross-database references are not implemented","custom internal functions aren.t currently supported","Unexpected execution of","pg_plan timeout exceeded"}'::text[]));
```

```
/*+ IndexScan(query_analyses query_analyses_database_id_idx) */
```

```
...
Limit  (cost=2.68..2.69 rows=1 width=16) (actual time=0.061..0.061 rows=0 loops=1)
  -> Sort  (cost=2.68..2.69 rows=1 width=16) (actual time=0.060..0.060 rows=0 loops=1)
        Sort Key: updated_at
        Sort Method: quicksort  Memory: 25kB
        -> Index Scan using query_analyses_database_id_idx on query_analyses  (cost=0.43..2.67 rows=1 width=16) (actual
time=0.056..0.056 rows=0 loops=1)
            Index Cond: (database_id = '...::integer)
            Filter: (((summary -> 'error_version'::text) IS NULL) OR (((summary -> 'error_version'::text)::integer < 29) OR
((summary -> 'error_collector_version'::text) IS NULL) OR (((summary -> 'error_collector_version'
::text)::integer < 7001))
                Rows Removed by Filter: 15
        Planning Time: 8.052 ms
        Execution Time: 0.084 ms
(10 rows)
```

How to come up with ideas on improving queries

What if...

We change the data model?

Treating “retry this error” as a write-time decision:

```
CREATE TABLE query_analyses_modified (LIKE query_analyses INCLUDING INDEXES);

ALTER TABLE query_analyses_modified ADD COLUMN retryable_error BOOLEAN NOT NULL;

INSERT INTO query_analyses_modified SELECT *, output_json IS NULL AND error IS NOT NULL AND
(error !~ ALL ('{"relation \".+\" does not exist","text search configuration \".+\" does not
exist","collation \".+\" for encoding \".+\" does not exist","type \".+\" does not
exist","multiple queries detected as input","utility statements are not
supported","cross-database references are not implemented","custom internal functions aren.t
currently supported","Unexpected execution of","pg_plan timeout exceeded"}'::text[])) FROM
query_analyses;
```

(and update application to do that in the future as well)



Validating query and index changes

Validation query and index changes



Optimizing query_analyses refresh task

Overview Compare Plans Parameter Sets Activity

All Query Plans

Choose a query variant to see its query text and settings, or create a new variant. Variants can be used to test Postgres planner behavior or rewrite queries to improve performance. [Learn more.](#)

Query Plans

Filter by Parameter Set...

PLAN	VARIANT	PARAMETER SET	EST. COST	RUNTIME
☐ 60bd80c	Baseline	Parameter Set 1	1,881,811	335.72ms
☐ be1a787	Baseline	Parameter Set 2	5,610,956	1,095.12ms
☐ 017da67	With extra indexes on query_analyses	Parameter Set 1	1,477,550	319.89ms
☐ ef5eb34	With extra indexes on query_analyses	Parameter Set 2	4,746,827	1,046.99ms
☐ a8b357f	With partial index on query_analyses	Parameter Set 1	32,383	⚡ 0.16ms
☐ a8b357f	With partial index on query_analyses	Parameter Set 2	71,594	308.33ms

pganalyze Workbooks

Structured testing of
different query variants
and input parameters.

Validation query and index changes

Optimizing query_analyses refresh task

Overview **Compare Plans** Parameter Sets Activity

Plan Comparison			
Compare: ☐ Est. Cost ☒ Runtime ☐ Rows ☐ Buffers ☐ Reads ☐ Writes			
Plan A	Plan B	Plan A	Plan B
Baseline - Parameter Set 1	With partial index on query_analyses - Parameter Set 1	Runtime	Runtime
-> Aggregate	-> Aggregate	0.31ms	0.01ms
-> Append	-> Append	0.08ms	0.00ms
-> Subquery Scan	-> Subquery Scan	0.08ms	0.00ms
-> Limit	-> Limit	0.07ms	0.00ms
-> Gather Merge		1.13ms	
-> Sort	-> Sort	0.71ms	0.00ms
-> Nested Loop	-> Nested Loop	0.01ms	0.00ms
-> Bitmap Heap Scan on query_analyses		7.36ms	
-> Bitmap Index Scan ¹		0.54ms	
-> Index Scan ² on query_analyses	-> Index Scan ⁴ on query_analyses	24.44ms	0.01ms
-> Subquery Scan	-> Index Scan ² on query_analyses	0.00ms	0.00ms
-> Limit	-> Subquery Scan	0.00ms	0.00ms
-> Sort	-> Limit	0.00ms	0.00ms
-> Gather	-> Sort	6.23ms	
-> Bitmap Heap Scan on query_analyses		280.27ms	
-> Bitmap Index Scan ¹	-> Index Scan ⁶ on query_analyses	14.77ms	0.03ms
-> Subquery Scan	-> Subquery Scan	0.00ms	0.00ms
-> Limit	-> Limit	0.00ms	0.00ms
-> Sort	-> Sort	0.01ms	0.00ms
-> Bitmap Heap Scan on query_analyses	-> Bitmap Heap Scan on query_analyses	0.00ms	0.00ms
-> BitmapAnd	-> BitmapAnd	0.00ms	0.00ms
-> Bitmap Index Scan ³	-> Bitmap Index Scan ³	0.05ms	0.02ms
-> Bitmap Index Scan ¹	-> Bitmap Index Scan ²	0.00ms	0.00ms
InitPlan 1 (returns \$0)			
-> Aggregate	-> Aggregate	0.00ms	0.00ms
-> Index Scan ⁷ on schema_tables	-> Index Scan ⁷ on schema_tables	0.04ms	0.02ms

Summary

Node Details

Node Source

Plan A: Baseline - Parameter Set 1

Seen At	Total Est. Cost	Runtime
Oct 01 12:06pm	1,881,811	335.72ms
Plan Fingerprint	Read From Disk	I/O Read Time
60bd80c	0 B	0.00ms

Plan B: With partial index on query_analyses - Parameter Set 1

Seen At	Total Est. Cost	Runtime
Oct 02 05:45am	32,383	0.16ms
Plan Fingerprint	Read From Disk	I/O Read Time
a8b357f	0 B	0.00ms

Index usage

A B Index

- ✓ 1. index_query_analyses_on_database_id
- ✓✓ 2. index_query_analyses_on_query_id
- ✓✓ 3. index_query_analyses_on_table_ids
- ✓✓ 4. index_query_occurrences_on_database_id_and_last
- ✓ 5. query_analyses_database_id_error_idx1
- ✓ 6. query_analyses_database_id_idx
- ✓✓ 7. schema_tables_database_id_removed_at_idx

pganalyze Workbooks

Structured testing of different query variants and input parameters.



Working with LLMs to optimize queries

**What would Claude
(or any LLM really) do?**

Working with LLMs to optimize queries

 Welcome, Lukas

```
HashAggregate (cost=189
2701.98..1892719.92 rows=1
794 width=8) (actual time
=46787.033..46792.976 ro
ws=1000 loops=1) Group
Key: "SELECT* 1".query...
```

PASTED



I have a slow query plan in Postgres that I'd like to optimize. Here is the plan recorded via `auto_explain`:



Chat

Cowork

Table 5.1 High



Working with LLMs to optimize queries

Reworking the nested loop's indexing and vacuum strategy. >

This is an I/O-bound plan: ~118s of cumulative read time across 3 processes for 86k page reads (~1.4 ms per read), and 22k pages *dirtied* by a read-only query, which means it's setting hint bits on recently written, not-yet-vacuumed pages. The three UNION branches behave very differently:

Branch 2 is the biggest waste (28.7s, returns 0 rows). It bitmap-scans every `query_analyses` row for the database (1.23M rows via `index_query_analyses_on_database_id`), then throws all of them away with the `output_json IS NULL AND error !~ ALL(...) AND version...` filter. The index scan alone took 8.8s. A partial index would make this near-instant:

```
sql

CREATE INDEX CONCURRENTLY index_query_analyses_pending_retry
ON query_analyses (database_id, updated_at)
WHERE output_json IS NULL;
```

Check selectivity first (`count(*) FILTER (WHERE output_json IS NULL)` for that database). If a large share still have null output but fail the error-pattern test, you may want to narrow the predicate further (e.g. `AND error IS NOT NULL`, or whichever stable condition excludes the common cases). The `(database_id, updated_at)` key also serves the `ORDER BY updated_at LIMIT` without a sort.

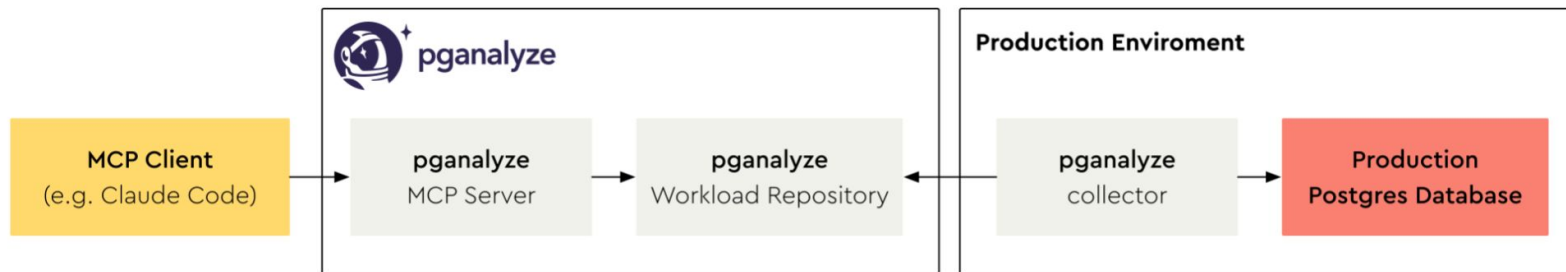
Branch 3 is fine (0.04 ms).

Two structural points:

- Branch 1 already returned its 1000 rows, but the UNION forces branches 2 and 3 to run anyway. If the application semantics allow it, run the branches as separate queries and only fall through to the next when the previous returns fewer than 1000 — that alone would have saved 28s here.
- The dirtied-page counts (11k in branch 1, 10.9k in branch 2) say autovacuum isn't keeping up with the write rate on `query_analyses` and `query_occurrences`. More aggressive per-table autovacuum settings (lower `autovacuum_vacuum_scale_factor`, or a scale factor of 0 with a fixed threshold) would set hint bits and the visibility map earlier, which is also what makes the index-only scans above actually work.

Since you have pganalyze connected, I can pull the current index definitions and column stats for `query_analyses`, run the Index Advisor on it, or create a workbook to EXPLAIN the rewritten/deduped variant against the baseline — just tell me which server this is on.

The pganalyze MCP Server



Working with LLMs to optimize queries



Yes, a few things stand out — and one of them changes an earlier suggestion of mine.

`query_occurrences` is 1:1 with `query_analyses`. It has a `UNIQUE` index on `query_id`, so there's exactly one occurrence row per query and the "dedupe with `DISTINCT query_id`" idea is moot. Branch 1 is a pure 1:1 lookup; the only lever left there is making the probe cheaper (covering index, better cache locality) or probing fewer rows.

Both tables look heavily bloated. `query_occurrences` has four columns totalling ~24 bytes of payload, yet it's 45 GB of heap plus 52 GB of indexes, with the unique `query_id` index alone at 40.6 GB. The production plan corroborates this: 92k occurrence rows were spread across 27k heap pages, about 3.4 rows per page where ~150 would fit. The likely cause is structural: `last` is a key in `index_query_occurrences_on_database_id_and_last`, so the daily update of `last` for every active query can't be HOT — each one writes a new heap tuple and new entries into both indexes (the write overhead figures of 0.39 and 0.31 reflect that). On `query_analyses`, `index_query_analyses_on_database_id` is 6.6 GB while the freshly built (`database_id`, `updated_at`) index covering the same rows is 1.4 GB; a single-column index over 4,011 distinct values should deduplicate to well below that. The prod plan read ~210 entries per index page, which is what a pre-deduplication or badly bloated btree looks like. `REINDEX CONCURRENTLY` (or `pg_repack` for the heaps) on both tables would shrink the random I/O in branch 1 directly, independent of any query change. Whether to stop indexing `last` so updates become HOT is a bigger trade, since branch 1 relies on that index.

Column stats confirm the shape of the data. On `query_analyses`, `updated_at` is null for 86% of rows, `output_json` for 91%, `error` for 95%. So across the whole table about 86% of analyses have never run, 9% succeeded and 5% failed.

<https://app.pganalyze.com/databases/.../tables/.../columns>

Table: public.query_analyses

Statistics Partitions Queries Columns Indexes Index Advisor Constraints VACUUM/ANL

Row Statistics

Average Row Size 176 B

Columns

NAME	TYPE	NULL % ⓘ	AVG. SIZE ⓘ	NUM. DISTINCT ⓘ
server_id	uuid	0.00%	16	1,497
database_id	bigint	0.00%	8	4,011
query_id	bigint	0.00%	8	0*
output_json	jsonb	90.81%	756	224,352
error	text	95.37%	96	2,556
summary	jsonb	86.18%	71	5,871,408
table_ids	bigint[]	0.00%	16	2,326
created_at	timestamp(6...	0.00%	8	30,972
updated_at	timestamp(6...	86.18%	8	0*
updated_columns	jsonb	90.81%	70	926

Working with LLMs to optimize queries

So the predicate `error IS NOT NULL` is both stable and matches the query's actual semantics:

sql

```
CREATE INDEX CONCURRENTLY index_query_analyses_failed_on_database_id_and_updated_at
ON query_analyses (database_id, updated_at)
WHERE output_json IS NULL AND error IS NOT NULL;
```

Then add `AND error IS NOT NULL` explicitly to branch 2's WHERE clause. The planner can sometimes infer that from a strict operator, but with the `!~ ALL(array)` form I wouldn't count on it — spelling it out guarantees the partial index is eligible. That branch drops from a 1.2M-row bitmap scan to ~60k index entries walked in `updated_at` order with the regex applied on the way, and the LIMIT should cut it off early.

Let's try out Claude's suggestion on a clone!

Working with LLMs to optimize queries

Query Plans

Filter by Parameter Set...

PLAN	VARIANT	PARAMETER SET	EST. COST	RUNTIME
☐  60bd80c	Baseline	Parameter Set 1	1,881,811	335.72ms
☐  be1a787	Baseline	Parameter Set 2	5,610,956	1,095.12ms
☐  017da67	With extra indexes on query_analyses	Parameter Set 1	1,477,550	319.89ms
☐  ef5eb34	With extra indexes on query_analyses	Parameter Set 2	4,746,827	1,046.99ms
☐  a8b357f	With partial index on query_analyses	Parameter Set 1	32,383	 0.16ms
☐  a8b357f	With partial index on query_analyses	Parameter Set 2	71,594	308.33ms
☐  7ae9af6	Trying out Claude's index idea	Parameter Set 1	89,942	317.60ms
☐  19843cf	Trying out Claude's index idea	Parameter Set 2	221,094	395.75ms

Working with LLMs to optimize queries

-> Subquery Scan	-> Subquery Scan	0.00ms	0.00ms
-> Limit	-> Limit	0.00ms	0.00ms
-> Sort		0.00ms	
-> Gather		6.23ms	
-> Bitmap Heap Scan on quer...		280.27ms	
-> Bitmap Index Scan ²		14.77ms	
	-> Index Scan ¹ on query_analy...		317.44ms

Working with LLMs to optimize queries

Look at the workbook
<https://staging.pganalyze.com/databases/10087482/workbooks/314/variants> via the MCP server

Needs your input >

Get Workbook Details



Claude wants to use Get Workbook
Details from pganalyze staging

• 4 Weighing index-scan tradeoffs against partial-index predicate limits 50s ▾

Searched available tools >

Get Workbook Details >

Weighing index-scan tradeoffs against partial-index predicate limits. >

Working with LLMs to optimize queries



The workbook tells a clear story, with one big caveat: every run here was fully cached (`shared read=0`), so the baseline is 335 ms / 1.1 s instead of the 46 s you saw in production. Buffer counts are the number to compare, since they're what turn into I/O when the cache is cold.

Branch 2 (the retry queue), buffers touched:

Variant	DB	DB
Baseline	165,307	377,880
551 "extra indexes" <code>(database_id, error), (database_id, updated_at)</code>	153,471	389,257
553 my <code>WHERE output_json IS NULL AND error IS NOT NULL</code>	60,205	39,603
552 partial index (regex baked into predicate)	18	2,648

Variant 551 is a wash — a `(database_id, error)` index can't be used for `!~ ALL`, so the planner still scans every row for the database.

My suggestion (553) was wrong in an instructive way. It cut the candidate set to ~60k rows, but the planner walks the index in `updated_at` order and must visit the heap for every entry to evaluate the regex, so it's one scattered heap page per row (60,227 rows removed, 60,205 buffers). Cached that's 317 ms; cold, 60k random reads could be as slow as the original. The old bitmap plan at least read the heap in physical order.

Agent backpressure

Automated feedback mechanisms — type systems, test suites, linters, and pre-commit hooks — that allow AI agents to detect and correct their own mistakes without human intervention.

latentpatterns.com

What if:

- We let an agent do EXPLAIN ANALYZE with restrictions (timeouts, recurrence, read-only)
- We let an agent test things on a database copy/clone (e.g. to verify indexing outcomes)
- We tracked what was suggested, acted on, & provide that as memory back to the agent



Enabling LLMs to test database changes

If possible, for database problems:
Provide **production context**
(stats, plan samples),
and the **application source code**
to the LLM context

Enabling LLMs to test database changes



```
lukasfittl@Mac pganalyze % claude
Claude Code v2.1.220
Opus 5 (1M context) · Claude Team
~/Code/pganalyze
```

```
> /mcp
```

Manage MCP servers

7 servers

```
Local MCPs (/Users/lukasfittl/.claude.json [project: /Users/lukasfittl/Code/pganalyze])
honeycomb · ✓ connected · 23 tools
pganalyze · ○ disabled
pganalyze-ee-test · ○ disabled
pganalyze-local-oauth · ○ disabled
> pganalyze-prod · ✓ connected · 29 tools
pganalyze-staging · ✓ connected · 29 tools
sentry · ✓ connected · 9 tools
```

Enabling LLMs to test database changes



› I have a specific trace that I found to be the one the customer reported. Can you check it: d8e364fd82bbc6576bda560c401c4367

Called honeycomb (ctrl+o to expand)

• This one is TableTreeMap – 15.9s, and 13.1s of it is a single SQL query. It even has EXPLAIN Plan spans attached. Let me print the plan.

Called honeycomb 2 times (ctrl+o to expand)

• Now I can fetch the actual EXPLAIN plan for that 13s query.

• Calling pganalyze-prod... (ctrl+o to expand)

Tool use

```
pganalyze-prod - get_query_explain_from_trace(serverId: "edzxhla46zfcjlvppvyl36oivq", datname: "pgaweb", fingerprint:
"171ad5ff5fcc6370", timestamp: 1784817206) (MCP)
  Resolve a "db.postgresql.plan" trace span attribute (structured as
"/servers/{serverId}/databases/{datname}/queries/{fingerprint}/samples/{timestamp}?role={role}") from another MCP server to the
EXPLAIN plan in pganalyze that has the specific slow execution
  details.
```

Enabling LLMs to test database changes



- I have the root cause. Let me confirm two supporting details in the schema and callers.

- ```
Bash(cd /Users/lukasfittl/Code/pganalyze
 echo "=== partitioning of schema_table_infos_35d ==="...)
└─ === partitioning of schema_table_infos_35d ===
1537:CREATE TABLE public.schema_table_infos_35d (
1538- schema_table_id bigint NOT NULL,
... +30 lines (ctrl+o to expand)
```

- That trace is TableTreeMap, and I have the root cause. Here's what's happening.

The customer's trace: 15.9s, and 13.1s is one SQL query

d8e364fd82bbc6576bda560c401c4367 breaks down as:

| Phase                                                      | Time         |
|------------------------------------------------------------|--------------|
| SchemaTable::WithStats SQL                                 | 13.12s (83%) |
| Uninstrumented Ruby after resolver                         | 2.38s        |
| Everything else (auth, Rust::TableStats, snapshot lookups) | ~90ms        |

The 13.1s query is the `find_by_sql` in `app/services/dataload/schema/table_list_for_database.rb:24-48`.

Root cause: a LATERAL join that can't prune partitions

# Enabling LLMs to test database changes



The LEFT JOIN LATERAL at table\_list\_for\_database.rb:31-42:

```
LEFT JOIN LATERAL (
 SELECT * ... FROM schema_table_infos_35d
 WHERE schema_table_id = t.id
 ORDER BY collected_at DESC
 LIMIT 1
) info ON (true)
```

schema\_table\_infos\_35d is PARTITION BY RANGE (collected\_at) (db/structure.sql:1555) with daily partitions – 36 live ones in this plan. The lateral filters on schema\_table\_id and orders by collected\_at, but has no collected\_at predicate, so Postgres cannot prune a single partition.

From the actual EXPLAIN (runtimeMs: 13023):

```
Nested Loop Left Join (actual time=311..13015 rows=2094 loops=1)
 I/O Timings: shared/local read=12780
 -> Index Scan on schema_tables t (actual time=0.5..1497 rows=2094)
 -> Limit (actual time=5.499..5.499 rows=0 loops=2094)
 -> Append (rows=0 loops=2094)
 -> Index Scan ...schema_table_infos_35d_20260724 (rows=0 loops=2094)
 -> Index Scan ...schema_table_infos_35d_20260723 (rows=0 loops=2094)
 ... 36 partitions total
```

2094 tables × 36 partitions ≈ 75,000 index scans, 223,726 buffer hits + 8,548 disk reads, and 11.3s of the 13.0s is I/O wait.

The damning part: every partition scan returns rows=0. This database has no schema\_table\_infos\_35d rows at all. All 13 seconds produce zero rows.

# Enabling LLMs to test database changes



› Lets focus on keep the data, but rewrite the query to perform better

- Good – keeping the data changes the calculus. Let me check one thing first, because it determines which rewrite actually helps.

Looking at the plan's I/O distribution: each of the 36 partitions shows shared hit~6050 read~232, and the reads total 8,170 with ~11.3s of I/O. These reads are one-time index-page loads per partition (every scan returns rows=0, so there are no heap fetches). That means a purely set-based rewrite – one DISTINCT ON pass instead of 2094 nested-loop iterations – would eliminate the 222k buffer hits (CPU) but leave most of the 11.3s of read I/O intact, because it would still probe 2094 scattered ids into all 36 partition indexes.

So the win has to come from partition pruning. Let me check the write cadence to find a safe collected\_at bound.

- ```
Bash(cd /Users/lukasfittl/Code/pganalyze
  echo "=== writes / model for schema_table_infos ===..."
  | === writes / model for schema_table_infos ===
  app/models/schema_table_info.rb
  app/models/schema_table.rb
  ... +14 lines (ctrl+o to expand)
```

- ```
Bash(cd /Users/lukasfittl/Code/pganalyze
 echo "=== SchemaTableInfo model ===..."
 | === SchemaTableInfo model ===
 class SchemaTableInfo < ApplicationRecord
 self.table_name = :schema_table_infos_35d
 ... +24 lines (ctrl+o to expand)
```

Let me make the change.

- Update(app/services/dataload/schema/table\_list\_for\_database.rb)

```
app/services/dataload/schema/table list for database.rb
```

1. Yes

```
...n (shift+tab)
```

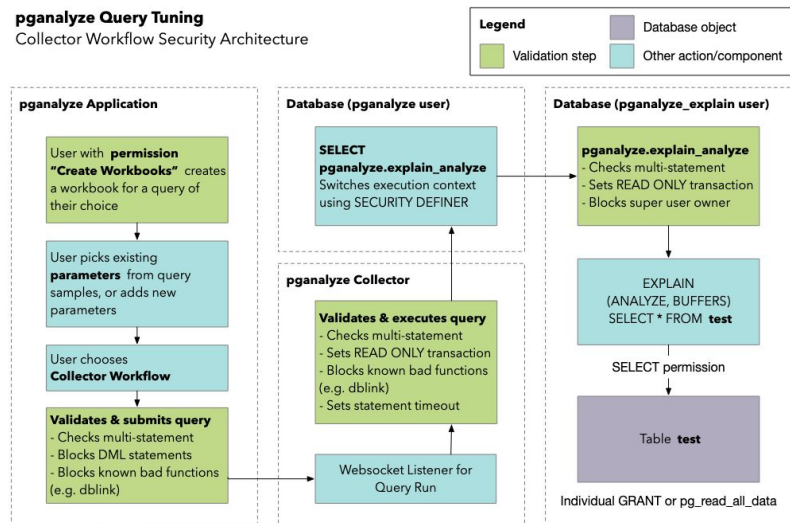
```
> 3. No, lets validate the change first.
```

# Enabling LLMs to test database changes

- Good call. The right way to validate is Workbooks – baseline vs. a variant with the bound added, real EXPLAIN on the actual data. Let me get the query text first.
- Calling `pganalyze-prod...` (ctrl+o to expand)

## pganalyze Query Tuning

Collector Workflow Security Architecture



pganalyze Collector Workflow enables **safe execution of EXPLAIN ANALYZE** (but not more) against production DBs or clones by humans and agents

# Enabling LLMs to test database changes



- 171ad5ff5fcc6370 is the URL fingerprint, not the numeric id – but I don't need it, since I have the exact SQL source. Let me reproduce the reported case in a workbook.

Note this runs real EXPLAIN ANALYZE against prod-db-main / pgaweb (read-only, ~13s for the baseline).

Called pganalyze-prod (ctrl+o to expand)

- The baseline is running. Let me poll for results.

Called pganalyze-prod (ctrl+o to expand)

- Excellent – this is a decisive result. Warm cache: 151ms. Same 232,274 buffer accesses, but read=0.

That confirms the diagnosis exactly: the work volume is identical, and the customer's 13.0s vs. this 151ms is pure cache state (8,548 disk reads × ~1.5ms on that storage = the 12.78s of I/O). The query burns ~111 buffer accesses returned row to fetch nothing.

# Enabling LLMs to test database changes



Now the variant with the pruning bound:  
Calling pganalyze-prod... (ctrl+o to expand)

## Tool use

```
pganalyze-prod - create_workbook_variant(databaseId: "1", workbookId: "3451", baselineVariantId: "5389", name: "Bounded collected_at (partition pruning)", desc: "Adds a collected_at lower bound to the LATERAL so Postgres can prune all but the most recent partitions. Bound is anchored on the server's last_snapshot_at minus 2 days.", queryText: "SELECT\n t.*, info.estimated_table_bytes, info.estimated_bloat_bytes,\n info.estimated_pct_bloat, info.toast_reltuples\nFROM\n schema_tables t\n LEFT JOIN LATERAL (\n SELECT\n *,\n GREATEST(estimated_table_bytes - ideal_table_bytes, $param_4) AS estimated_bloat_bytes,\n CASE estimated_table_bytes\n WHEN $param_5 THEN $param_6\n ELSE round((GREATEST(estimated_table_bytes - ideal_table_bytes,\n $param_8) * $param_7) / estimated_table_bytes)\n END AS estimated_pct_bloat\n FROM schema_table_infos_35d\n WHERE schema_table_id = t.id\n AND\n collected_at > '2026-07-28 00:00:00Z'::timestampz\n ORDER BY collected_at DESC\n LIMIT $param_2\n) info ON $param\nWHERE\n database_id = $database_id\n AND removed_at IS NULL\n AND relation_type = ANY($param_3::int[])\n AND parent_table_id IS NULL") (
 Create a workbook variant from a queryText (rewrite) and/or plannerSettings, then run EXPLAIN. The query must use $alias-style parameters from the baseline. The query is validated server-side against the baseline before the variant is created; invalid queries return early with no variant created. Source the queryText from your own reasoning, the user, or get_insight_rewrite. Response status is "complete", "error", or "awaiting_results" (EXPLAIN still running).
```

Do you want to proceed?

1. Yes
2. Yes, and don't ask again for pganalyze-prod - create\_workbook\_variant commands in /Users/lukasfittl/Code/pganalyze
3. No

# Enabling LLMs to test database changes



<https://app.pganalyze.com/databases/1/workbooks/3453>

## Results

|                     | Partitions | Buffers | Warm runtime    | Cost   | Rows  |
|---------------------|------------|---------|-----------------|--------|-------|
| Baseline @ 2d param | 36         | 232,274 | 147.8 / 143.0ms | 10,432 | 2,094 |
| Baseline @ 7d param | 36         | 232,274 | 150.9ms         | 10,432 | 2,094 |
| Variant @ 7d        | 9          | 56,378  | 28.0ms          | 3,562  | 2,094 |
| Variant @ 2d        | 4          | 24,968  | 13.7 / 13.4ms   | 2,346  | 2,094 |

Two useful controls fell out of running both parameter sets against both queries:

- The baseline is invariant to the cutoff – byte-identical buffers and cost at 2d and 7d. That proves the tautology contributes nothing and isolates the pruning as the sole cause of the variant's gain.
- Pruning is plan-time, not execution-time. The pruned partitions vanish from the plan entirely (41 nodes → 8), with no "Subplans Removed" line, and the bound folds into the existing index: Index Cond: (schema\_table\_id = t.id AND collected\_at > '2026-07-28...'). That matches how Rails will emit it, since `find_by_sql` interpolates named binds into the SQL string rather than sending protocol-level parameters.

Row count is 2,094 in all four runs.

I've written the findings into the workbook description so it stands on its own for whoever opens it next.

One caveat worth restating: all four runs are fully cached (`read=0`). The customer's 13.0s was the same 232,274 accesses with 8,548 of them cold at ~1.5ms each. Pruning to 4 partitions cuts reads ~9x, which should put that 13.1s near ~1.4s – at which point the 2.38s Ruby tail becomes the larger half of their request.

Ready to apply the code change – `SCHEMA_TABLE_INFO_LOOKBACK = 2.days` (13.7ms) or `7.days` (28.0ms)?

# Enabling LLMs to test database changes



## TableTreeMap: schema\_table\_infos lateral (baseline vs pruned)

[Overview](#) [Compare Plans](#) [Parameter Sets](#) [Activity](#)

### All Query Plans

Choose a query variant to see its query text and settings, or create a new variant. Variants can be used to test Postgres planner behavior or rewrite queries to improve performance. [Learn more](#).

### Query Plans

Filter by Parameter Set... ▼

| PLAN                                                                                                               | VARIANT                                | PARAMETER SET   | EST. COST | RUNTIME                                                                                     |
|--------------------------------------------------------------------------------------------------------------------|----------------------------------------|-----------------|-----------|---------------------------------------------------------------------------------------------|
| <input type="checkbox"/>  84f94a7 | Baseline                               | Parameter Set 1 | 10,432    | 142.95ms                                                                                    |
| <input type="checkbox"/>  84f94a7 | Baseline                               | Parameter Set 2 | 10,432    | 150.94ms                                                                                    |
| <input type="checkbox"/>  168d9fb | Bounded collected_at (partition pru... | Parameter Set 1 | 2,346     |  13.73ms |
| <input type="checkbox"/>  09aaa1d | Bounded collected_at (partition pru... | Parameter Set 2 | 3,562     | 27.97ms                                                                                     |

# Enabling LLMs to test database changes

## TableTreeMap: schema\_table\_infos lateral (baseline vs pruned)

Overview Compare Plans Parameter Sets Activity

### Plan Comparison

Select plans

Compare : ☐ Est. Cost ☒ Runtime ☐ Rows ☐ Buffers ☐ Reads ☐ Writes

#### Plan A

Baseline - Parameter Set 1

#### Plan B

Bounded collected\_at (partition pruning) -  
Parameter Set 1

#### Plan A

Runtime

#### Plan B

Runtime

-> Result

-> Nested Loop

-> Index Scan<sup>38</sup> on schema\_tables

-> Limit

-> Append

-> Index Scan<sup>34</sup> on schema\_tabl...

-> Index Scan<sup>5</sup> on schema\_tabl...

-> Index Scan<sup>16</sup> on schema\_tabl...

-> Index Scan<sup>33</sup> on schema\_tabl...

-> Index Scan<sup>15</sup> on schema\_tabl...

-> Index Scan<sup>32</sup> on schema\_tabl...

-> Index Scan<sup>31</sup> on schema\_tabl...

-> Index Scan<sup>30</sup> on schema\_tabl...

-> Index Scan<sup>29</sup> on schema\_tabl...

-> Index Scan<sup>28</sup> on schema\_tabl...

-> Index Scan<sup>27</sup> on schema\_tabl...

-> Index Scan<sup>4</sup> on schema\_tabl...

-> Index Scan<sup>26</sup> on schema\_tabl...

-> Index Scan<sup>25</sup> on schema\_tabl...

-> Index Scan<sup>24</sup> on schema\_tabl...

-> Nested Loop

-> Index Scan<sup>38</sup> on schema\_tables

-> Limit

-> Append

-> Index Scan<sup>34</sup> on schema\_table...

-> Index Scan<sup>5</sup> on schema\_table...

-> Index Scan<sup>16</sup> on schema\_table...

-> Index Scan<sup>33</sup> on schema\_table...

0.48ms

1.78ms

4.10ms

2.09ms

0.00ms

0.00ms

2.09ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

4.19ms

0.25ms

2.83ms

0.00ms

4.19ms

0.00ms

2.09ms

2.09ms

2.09ms

2.09ms

# Enabling LLMs to test database changes

## The Human Variant

Edit name and description

1 Rewrite Query and Edit Planner 2 Run EXPLAIN

Edit Unified Diff Split Diff

```
1 WITH server_schema_table_infos AS MATERIALIZED (
2 SELECT info.schema_table_id, info.estimated_table_bytes,
3 info.ideal_table_bytes, info.toast_reltuples, info.collected_at
4 FROM schema_table_infos_35d info
5 WHERE server_id = $server_id
6)
7 SELECT
8 t.*, info.estimated_table_bytes, info.estimated_bloat_bytes,
9 info.estimated_pct_bloat, info.toast_reltuples
10 FROM
11 schema_tables t
12 LEFT JOIN LATERAL (
13 SELECT
14 *,
15 GREATEST(estimated_table_bytes - ideal_table_bytes, $param_5) AS estimated_bloat_bytes,
16 CASE estimated_table_bytes
17 WHEN $param_6 THEN $param_7
18 ELSE round((GREATEST(estimated_table_bytes - ideal_table_bytes, $param_9) * $param_8) / estimated_table_bytes)
19 END AS estimated_pct_bloat
20 FROM server_schema_table_infos
21 WHERE schema_table_id = t.id
22 ORDER BY collected_at DESC
23 LIMIT $param_2
24) info ON $param
25 WHERE
26 database_id = $database_id
27 AND removed_at IS NULL
```

# Enabling LLMs to test database changes



## The Human Variant

Edit name and description

Cancel

1 Rewrite Query and Edit Planner 2 Run EXPLAIN

Manual Workflow

Collector Workflow

```
WITH server_schema_table_infos AS MATERIALIZED (
 SELECT
 info.schema_table_id, info.estimated_table_bytes, info.ideal_table_bytes,
 info.toast_reltuples, info.collected_at
 FROM schema_table_infos_35d info
 WHERE server_id = sserver_id
```

RUN

EXECUTED

RUN

☐ Parameter Set 1

-

☐ Parameter Set 2

-

Collect Plans

# Enabling LLMs to test database changes



## TableTreeMap: schema\_table\_infos lateral (baseline vs pruned)

Overview Compare Plans Parameter Sets Activity

### All Query Plans

Choose a query variant to see its query text and settings, or create a new variant. Variants can be used to test Postgres planner behavior or rewrite queries to improve performance. [Learn more](#).

### Query Plans

Filter by Parameter Set... ▼

| PLAN                             | VARIANT                                | PARAMETER SET   | EST. COST | RUNTIME  |
|----------------------------------|----------------------------------------|-----------------|-----------|----------|
| <input type="checkbox"/> 84f94a7 | Baseline                               | Parameter Set 1 | 10,432    | 142.95ms |
| <input type="checkbox"/> 84f94a7 | Baseline                               | Parameter Set 2 | 10,432    | 150.94ms |
| <input type="checkbox"/> 168d9fb | Bounded collected_at (partition pru... | Parameter Set 1 | 2,346     | 13.73ms  |
| <input type="checkbox"/> 09aaa1d | Bounded collected_at (partition pru... | Parameter Set 2 | 3,562     | 27.97ms  |
| <input type="checkbox"/> 966e3b0 | The Human Variant                      | Parameter Set 1 | 488,380   | ⚡ 4.70ms |
| <input type="checkbox"/> 966e3b0 | The Human Variant                      | Parameter Set 2 | 488,380   | 4.99ms   |

# Enabling LLMs to test database changes



## TableTreeMap: schema\_table\_infos lateral (baseline vs pruned)

Overview

Compare Plans

Parameter Sets

Activity

### Activity

-  **2026-07-30 08:57:18am PDT:** new variant "The Human Variant" created by  Lukas Fittl
-  **2026-07-30 08:53:52am PDT:** workbook description updated by  Lukas Fittl MCP
-  **2026-07-30 08:50:51am PDT:** new variant "Bounded collected\_at (partition pruning)" created by  Lukas Fittl MCP
-  **2026-07-30 08:50:16am PDT:** new variant "Baseline" created by  Lukas Fittl MCP
-  **2026-07-30 08:50:16am PDT:** workbook created by  Lukas Fittl MCP

# Enabling LLMs to test database changes



› I added another variant on that workbook. Can you take a look?

Called pganalyze-prod (ctrl+o to expand)

• Output was too large. Let me extract just the variant summaries.

• Bash(cd /Users/lukasfittl/.claude/projects/-Users-lukasfittl-Code-pganalyze/3a66346c-0a2b-48c2-9981-f1f4d50c5504/tool-results  
f=mcp-pganalyze-prod-get\_workbook\_detai...)  
└─ 2: "id": "3453",  
3: "name": "TableTreeMap: schema\_table\_infos lateral (baseline vs pruned)",  
26: "id": "3416",  
... +59 lines (ctrl+o to expand)  
└─ Shell cwd was reset to /Users/lukasfittl/Code/pganalyze

• "The Human Variant" – 4.7ms and only 2,044 buffers, but a planner cost of 488,379. Let me see what you did.

• Bash(cd /Users/lukasfittl/.claude/projects/-Users-lukasfittl-Code-pganalyze/3a66346c-0a2b-48c2-9981-f1f4d50c5504/tool-results  
f=mcp-pganalyze-prod-get\_workbook\_detai...)  
└─ === queryTextWithAlias ===  
WITH server\_schema\_table\_infos AS MATERIALIZED (  
SELECT  
... +173 lines (ctrl+o to expand)  
└─ Shell cwd was reset to /Users/lukasfittl/Code/pganalyze

# Enabling LLMs to test database changes



• Looked at it – your instinct is better than mine, but the measurement is hiding a trap.

What your variant does right

Pulling the server's info rows once via the per-partition `server_id` indexes changes the algorithm, not just the constant factor. The Append does 36 index descents total (110 buffers) in  $2,094 \times 36$ . And because there's no `collected_at` predicate, it's semantically identical to today's query – no cutoff, no data-loss risk, which is strictly better than the 2-day bound I proposed.

But the 4.7ms isn't real

Every run in this workbook is against `database_id = ...`, which has zero `schema_table_infos` rows. Look at the CTE:

CTE `server_schema_table_infos`

-> Append (cost=0.43..38736.90 rows=34721 width=36) (actual ... rows=0 loops=1)

Estimated 34,721 rows, actual 0. So the inner side never does any work, and the comparison can't distinguish the approaches. The plan shape is where the problem is:

```
-> Limit (actual time=0.001..0.001 rows=0 loops=2094)
 -> Sort Sort Key: server_schema_table_infos.collected_at DESC
 -> CTE Scan on server_schema_table_infos
 Filter: (server_schema_table_infos.schema_table_id = t.id)
```

That's a full scan + sort of the CTE per outer row, 2,094 times – a CTE can't be indexed. With  $M$  rows in the CTE and  $N$  tables it's  $O(N \times M)$ : at the estimated  $M$  that's ~73M row comparison, 2,094 sorts. The planner is telling you this loudly – cost 488,380 vs 2,346 for the bounded lateral, 208x higher. It measures at 4.7ms only because  $M$  happens to be 0.

The fix – variant 5396, added to the workbook

Same CTE, but deduped inside it with `DISTINCT ON (schema_table_id) ... ORDER BY schema_table_id, collected_at DESC`, then a plain `LEFT JOIN`:

# Enabling LLMs to test database changes

```
-> Hash Right Join (cost=826.92..1612.57)
 Hash Cond: (info.schema_table_id = t.id)
 Inner Unique: true
 -> CTE Scan on latest_info (loops=1) <- once, not 2094x
 -> Hash on schema_tables (rows=2094)
```

CTE scanned once.  $O(N + M)$  instead of  $O(N \times M)$ .

|                      | Runtime (empty data) | Cost    | Buffers | Semantics   |
|----------------------|----------------------|---------|---------|-------------|
| Baseline             | 143ms                | 10,432  | 232,274 | —           |
| Bounded lateral (2d) | 13.7ms               | 2,346   | 24,968  | cutoff risk |
| The Human Variant    | 4.7ms                | 488,380 | 2,044   | unchanged   |
| CTE + DISTINCT ON    | 4.1ms                | 43,159  | 2,044   | unchanged   |

Runtimes tie because  $M=0$ ; the cost estimates are the only signal that differentiates them, and they're computed from real statistics.

# Enabling LLMs to test database changes



> can you make the code change?

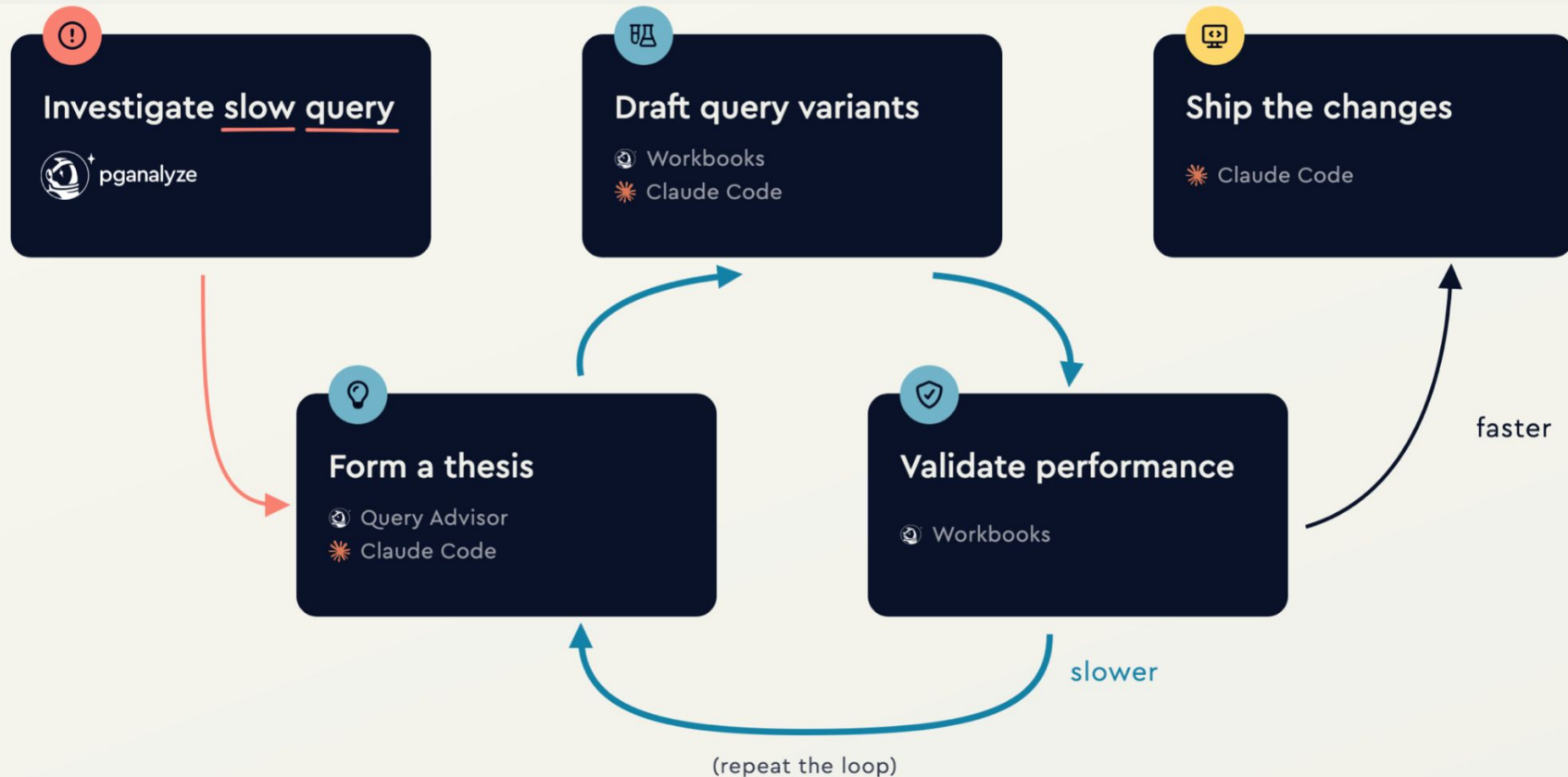
• Making the change. Given your earlier "keep the data" constraint, I'm going with the CTE + DISTINCT ON form – it's the only option with zero semantic change. Let me verify one thing first:

• Bash(`cd /Users/lukasfittl/Code/pganalyze`  
  `echo "=== databases.server_id column ===..."`)  
  `=== databases.server_id column ===`  
  426:CREATE TABLE public.databases (  
  427-     id BIGSERIAL PRIMARY KEY,  
  ... +14 lines (ctrl+o to expand)

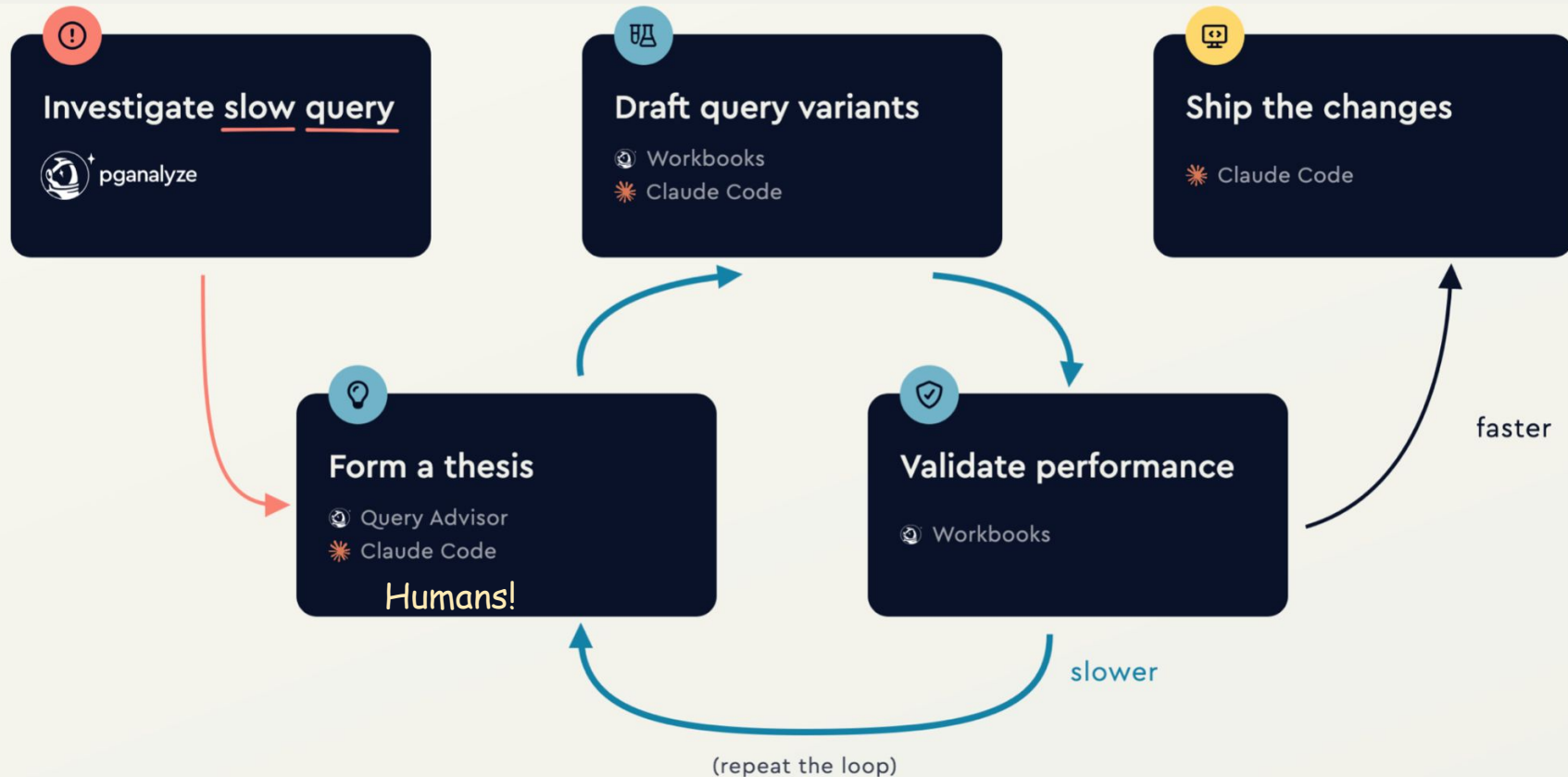
• Update(`app/services/dataload/schema/table_list_for_database.rb`)  
  Added 1 line, removed 1 line  
  4     `extend self`  
  5  
  6     `def table_list_for_database(database, views: false, limit: nil, offset: nil, filter: nil, sort_by: nil, sort_direction: nil,`  
`include_partitions: false, parent_table_id: nil) # rubocop:disable Metrics/ParameterLists`  
  7 -     `binds = { database_id: database.id }`  
  7 +     `binds = { database_id: database.id, server_id: database.server_id }`  
  8     `relation_type = if views`  
  9         `format("ANY('{%d, %d}':int[])", SchemaTable.relation_types[:view], SchemaTable.relation_types[:materialized_`  
  10         `else`

• Update(`app/services/dataload/schema/table_list_for_database.rb`)

# pganalyze + AI enabled Tuning Loop

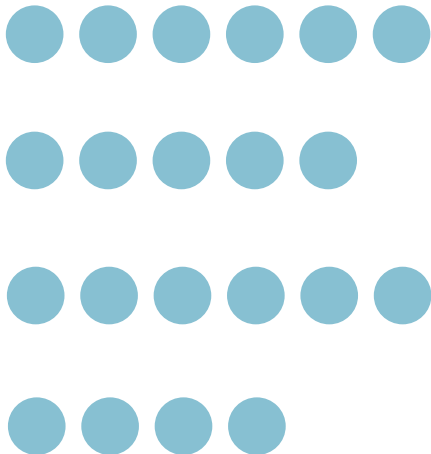


# pganalyze + AI enabled Tuning Loop



# A Typical Organization Using Postgres

## The Application Teams



Database A

Database B

Database C

Database D

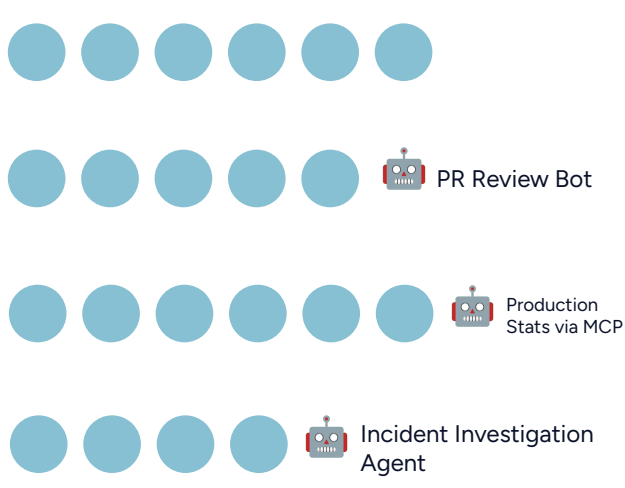
## The Data Platform Team / DBAs



# An Organization with the pganalyze MCP



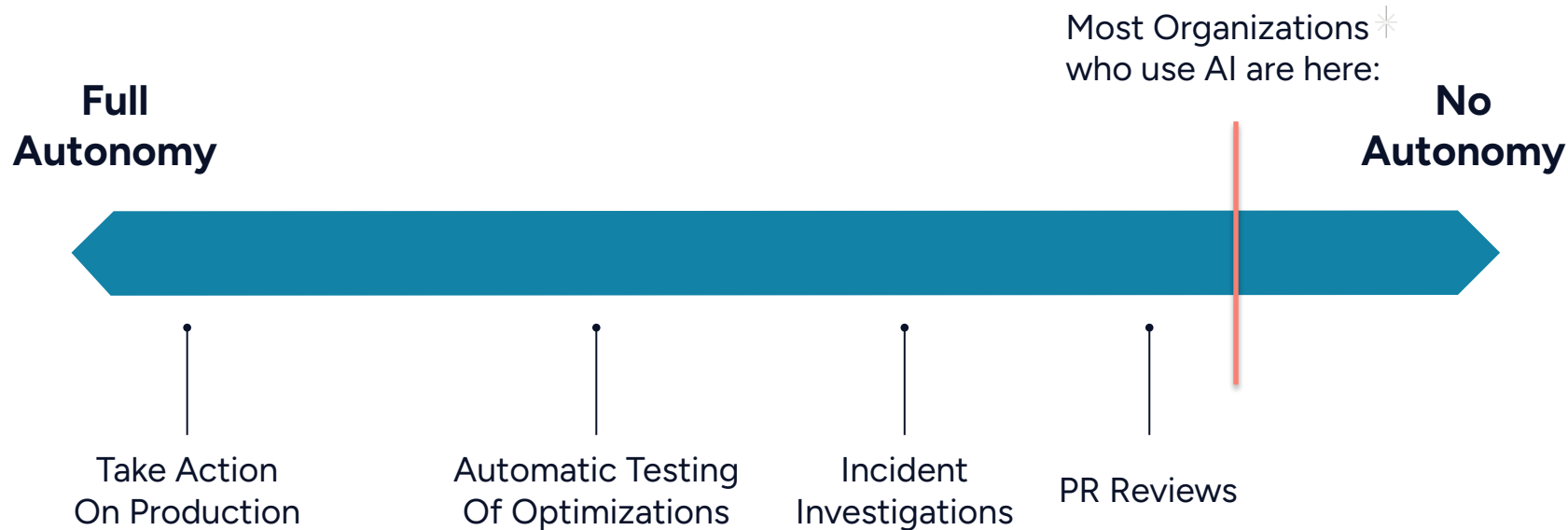
## The Application Teams



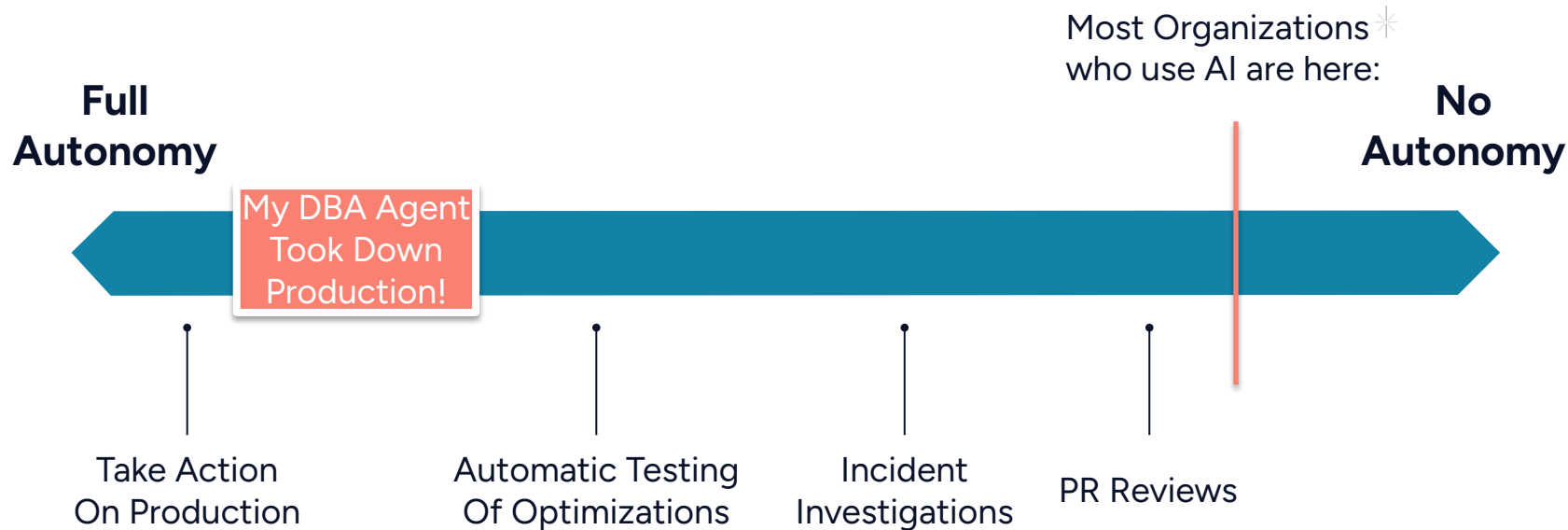
## The Data Platform Team / DBAs



# The Agent Autonomy Problem



# The Agent Autonomy Problem



# The Agent Autonomy Problem

**Full  
Autonomy**

My DBA Agent  
Took Down  
Production!

Take Action  
On Production

Automatic Testing  
Of Optimizations

I did what it said,  
but its still slow.

Incident  
Investigations

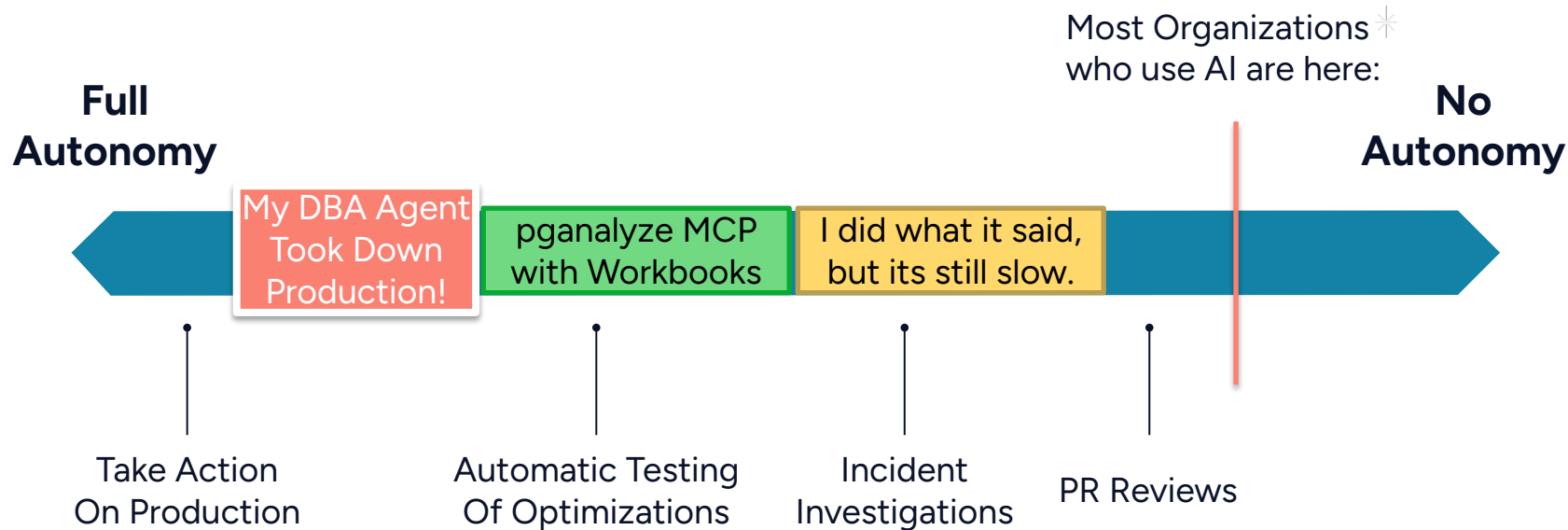
PR Reviews

Most Organizations  
who use AI are here:

**No  
Autonomy**



# The Agent Autonomy Problem





Postgres-specific monitoring, analysis and recommendation logic.

Advisors running in the background. Purpose-built API endpoints for AI tools.



Postgres-specific monitoring, analysis and recommendation logic.

Advisors running in the background. Purpose-built API endpoints for AI tools.

**pganalyze**  
MCP Server with Workbooks



Postgres-specific monitoring, analysis and recommendation logic.

Advisors running in the background. Purpose-built API endpoints for AI tools.

**pganalyze**  
MCP Server with Workbooks



Third-party Agent



Postgres-specific monitoring, analysis and recommendation logic.

Advisors running in the background. Purpose-built API endpoints for AI tools.

**pganalyze**  
MCP Server with Workbooks



Third-party Agent

**Early Access in Q4:**  
AI Investigations and  
Optimizations in  
pganalyze (Opt-In)



Postgres-specific monitoring, analysis and recommendation logic.

Advisors running in the background. Purpose-built API endpoints for AI tools.

**pganalyze**  
MCP Server with Workbooks



Third-party Agent

**Early Access in Q4:**  
AI Investigations and  
Optimizations in  
pganalyze (Opt-In)

**Future:**  
Automations

\*  

# Thank you!



VISIT US AT OUR BOOTH TO LEARN MORE:

- pganalyze Workbooks are now available with the MCP Server
- Your feedback — Thoughts on webinar, topics, Postgres challenges
- Request Meeting — Discuss tuning your Postgres with pganalyze



Learn more at [pganalyze.com](https://pganalyze.com)

Contact us: [lukas@pganalyze.com](mailto:lukas@pganalyze.com)  
[maciek@pganalyze.com](mailto:maciek@pganalyze.com)