How to optimize a slow Postgres query

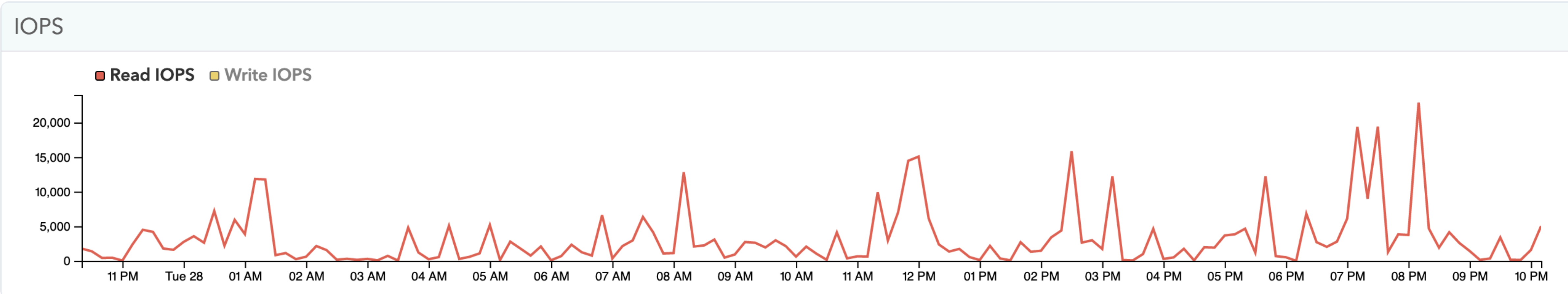
pganalyze

@LukasFittl
hachyderm.io/@lukas

1. **Let’s pick a slow query!**
2. **Debugging why a query is slow**
3. **Benchmarking with EXPLAIN (ANALYZE, BUFFERS)**
4. **Planner costing, and why it can never be perfect**
5. **JOIN order and parameterized index scans**
6. **Guiding the planner to the right plan**
7. **Query Tuning with pganalyze**

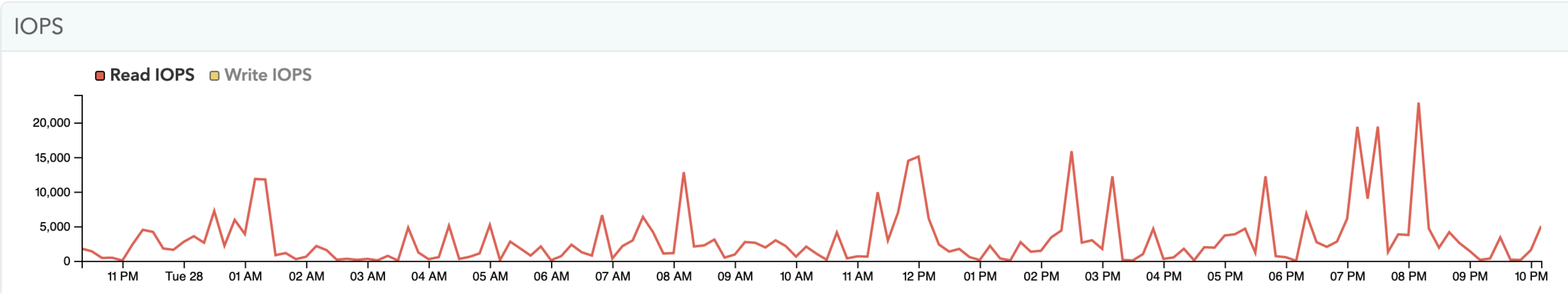
pganalyze

Let's pick a slow query!



Why is our database spending so much **[I/O Time | CPU Time | ...]?**

pganalyze



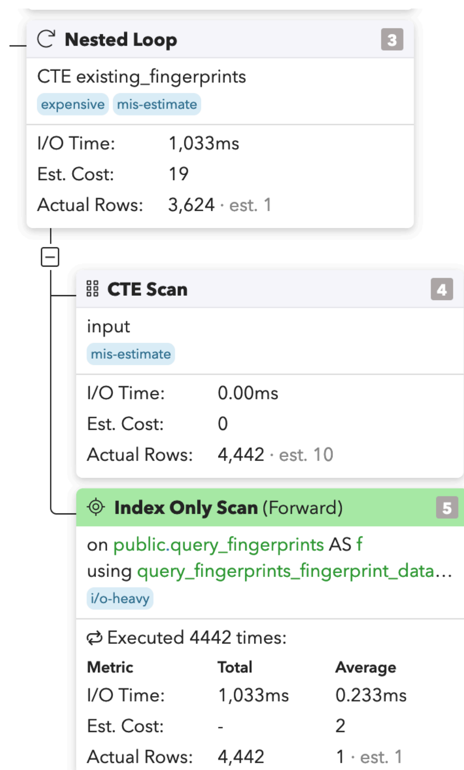
SELECT INSERT, UPDATE, DELETE DDL & other Compare to 7 days ago Search...

QUERY	ROLE	AVG TIME (MS)	CALLS / MIN	% OF ALL I/O	% OF ALL RUNTIME
WITH input AS (...), existing_fingerprints AS (...), update_queries AS (...)…	pgaweb_workers	3.78ms	14422.48	26.28%	17.28%
INSERT INTO query_stats_35d_20230328 (...) SELECT ... FROM unnest($1::int[],…	pgaweb_workers	145.54ms	246.50	8.02%	11.37%
INSERT INTO schema_index_stats_35d_20230329 (...) SELECT ... FROM unnest($1:…	pgaweb_workers	49.93ms	477.44	5.05%	7.55%
WITH total_times AS (...), table_queries AS (...), fingerprints AS (...), ra…	pgaweb_workers	123.95ms	181.59	9.20%	7.13%
WITH data AS (...), existing_rows AS (...), update_rows AS (...), insert_row…	pgaweb_workers	3.60ms	5229.28	9.15%	5.96%
WITH data(server_id, query_id, schema_table_scan_id, scan_node_type, scan_ta…	pgaweb_workers	18.60ms	760.78	4.95%	4.48%

pganalyze

```
WITH input AS (...)
SELECT *
  FROM query_fingerprints AS f
  JOIN input USING (database_id, fingerprint, postgres_role_id)
```

auto_explain + pganalyze



pganalyze

```
WITH input AS (...)
SELECT *
  FROM query_fingerprints AS f
  JOIN input USING (database_id, fingerprint, postgres_role_id)
```

EXPLAIN

```
->  Nested Loop  (cost=0.57..19.30 rows=1 width=45) (actual rows=3624 loops=1)
      Buffers: shared hit=19534 read=4214 dirtied=145
      I/O Timings: read=1033.376
      ->  CTE Scan on input_1  (cost=0.00..0.20 rows=10 width=60) (actual rows=4442 loops=1)
            CTE Name: input
      ->  Index Only Scan using … on public.query_fingerprints f (cost=0.57..1.91 rows=1 width=37) (…)
            Index Cond: ((…))
            Heap Fetches: 2603
            Buffers: shared hit=19534 read=4214 dirtied=145
            I/O Timings: read=1033.376
```

pganalyze

pganalyze

Debugging why a query is slow

Is the query always slow, or just sometimes?

`WITH indexes AS (...), index_sizes AS (...) SELECT ... FROM unpack_schema_table_stats(database_id...`

fingerprint ab2ba35b3f9acddf | role pgaweb_workers | line /app/services/dataload/schema/stats_series_for_tab... | job Issues::CheckUpSingleWorker

sentry_trace_id 2ec4aeebee694bbd8696d47dcb806944 and 118 more | View all query tags

Avg Time 1,449.80ms | **Calls Per Minute** 9.32 / min

Compare to 7 days ago

Overview | Index Advisor ? | Query Samples 5+ | EXPLAIN Plans 5+ | Query Tags 5+ | Log Entries 100+

Check-Up

1 new issue

Info Query **#43899555** takes 1287 ms on average (88397 ms max, 3.35 MB read from disk per call, 13390 calls in last 24h)

EXPLAINs

EXECUTED AT	PLAN	EST. COST	RUNTIME	I/O READ TIME		READ FROM DISK	PLAN NODES
2024-10-01 08:14:23pm PDT	a332ead	348	14,688.59ms	12,796.35ms	87%	42.6 MB	Sort · Nested Loop · CTE Scan +4 more
2024-10-01 08:03:23pm PDT	a332ead	348	12,812.28ms	10,883.24ms	85%	51.9 MB	Sort · Nested Loop · CTE Scan +4 more
2024-10-01 08:13:14pm PDT	a332ead	348	11,881.92ms	7,873.20ms	66%	476 MB	Sort · Nested Loop · CTE Scan +4 more
2024-10-01 07:52:43pm PDT	a332ead	348	9,564.42ms	7,342.84ms	77%	57.7 MB	Sort · Nested Loop · CTE Scan +4 more
2024-10-01 08:02:40pm PDT	a332ead	348	9,120.33ms	7,772.78ms	85%	45.8 MB	Sort · Nested Loop · CTE Scan +4 more

1.4s average vs **14.6 s** outlier execution

pganalyze

I/O Time is often the issue!

Plan Comparison

Plan A: 2024-10-01 08:14:23pm PDT - a332ead - runtime: 14,688.59ms - I/O read time: 12,796.35ms

Plan B: 2024-10-01 08:00:26pm PDT - a332ead - runtime: 1,684.27ms - I/O read time: 1,113.03ms

Cost Metric: ○ **Est. Total Cost (Self)** ○ **Runtime (Self)** ◉ **I/O Read Time (Self)** ○ **Rows**

Plan A/B	Plan A: I/O Time	Plan B: I/O Time
-> Sort	0.00ms	0.00ms
-> Aggregate	0.00ms	0.00ms
-> Index Scan	0.00ms	0.00ms
-> Function Scan	5,833.54ms	312.83ms
-> Nested Loop	0.00ms	0.00ms
-> Function Scan	6,962.81ms	800.20ms
-> CTE Scan	5,833.54ms	312.83ms

pganalyze

Cloud Database Provider I/O Latency can be bad (local NVMe disks = much much better)

I/O & Buffers

	Shared	Local	Temp
Hit	152.7 MB	0 B	-
Read	25.8 MB	0 B	0 B
Dirtied	0 B	0 B	-
Written	0 B	0 B	0 B

I/O Read Time	I/O Write Time
5,833.54ms	0.00ms

pganalyze

Is the plan the same, or does it change?

`WITH indexes AS (...), index_sizes AS (...) SELECT ... FROM unpack_schema_table_stats(database_id...`

fingerprint ab2ba35b3f9acddf | role pgaweb_workers | line /app/services/dataload/schema/stats_series_for_tab... | job Issues::CheckUpSingleWorker

sentry_trace_id 2ec4aeebee694bbd8696d47dcb806944 and 118 more | View all query tags

Avg Time 1,449.80ms | Calls Per Minute 9.32 / min

Compare to 7 days ago

Overview | Index Advisor ? | Query Samples 5+ | EXPLAIN Plans 5+ | Query Tags 5+ | Log Entries 100+

Check-Up

1 new issue

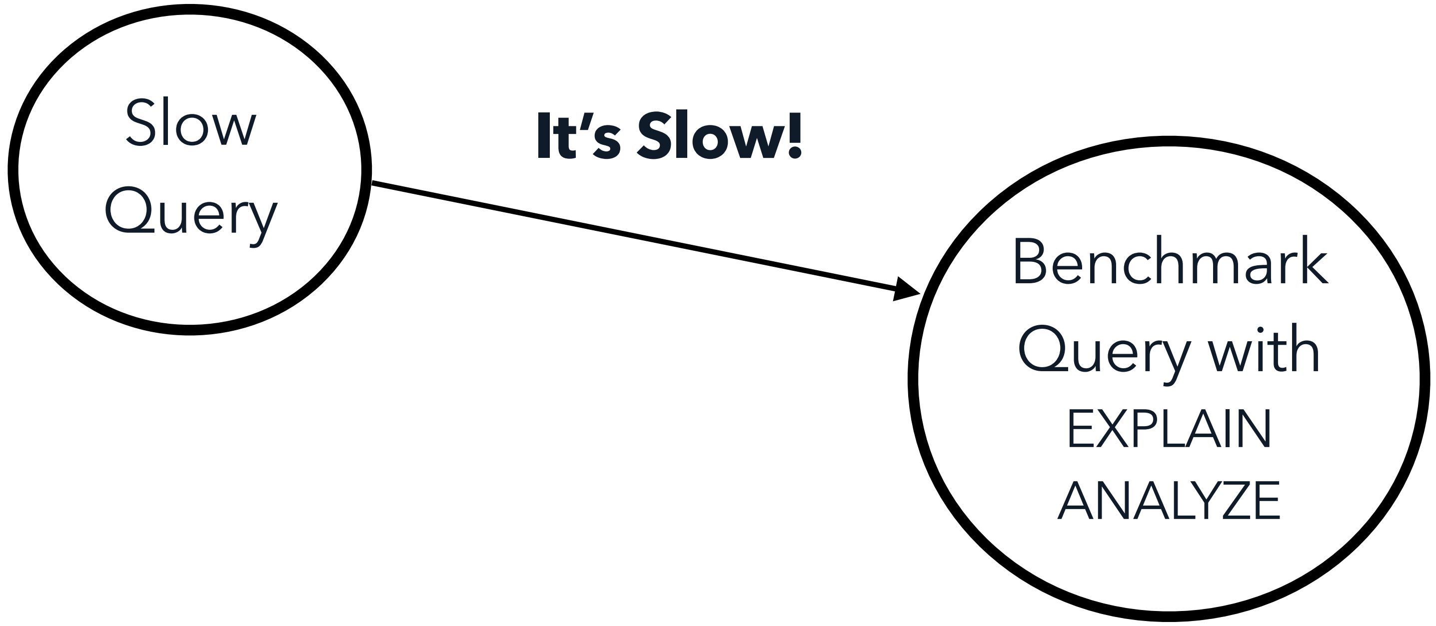
Info Query **#43899555** takes 1287 ms on average (88397 ms max, 3.35 MB read from disk per call, 13390 calls in last 24h)

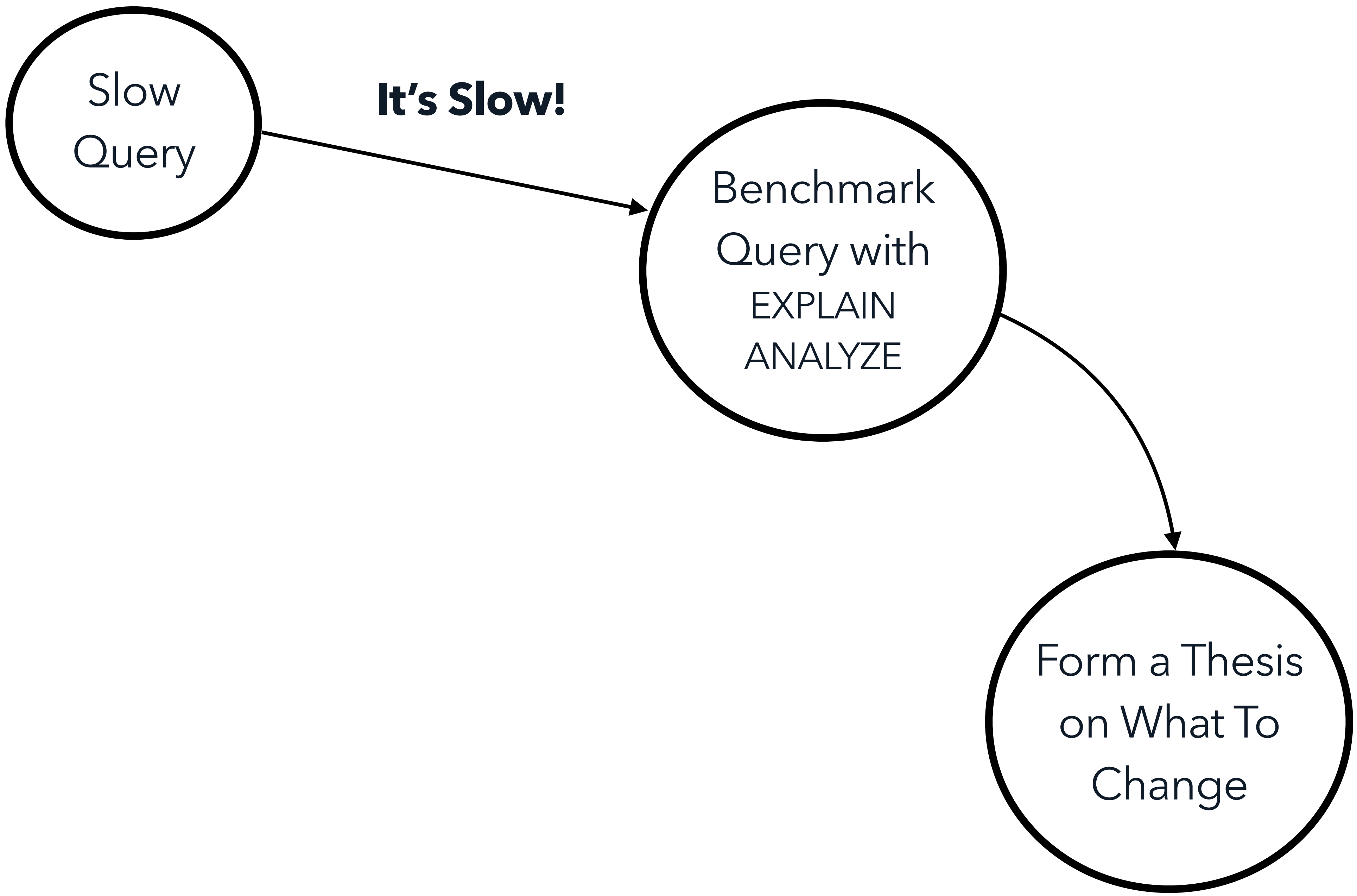
EXPLAINs

EXECUTED AT	PLAN	EST. COST	RUNTIME	I/O READ TIME		READ FROM DISK	PLAN NODES
2024-10-01 08:14:23pm PDT	a332ead	348	14,688.59ms	12,796.35ms	87%	42.6 MB	Sort · Nested Loop · CTE Scan +4 more
2024-10-01 08:03:23pm PDT	a332ead	348	12,812.28ms	10,883.24ms	85%	51.9 MB	Sort · Nested Loop · CTE Scan +4 more
2024-10-01 08:13:14pm PDT	a332ead	348	11,881.92ms	7,873.20ms	66%	476 MB	Sort · Nested Loop · CTE Scan +4 more
2024-10-01 07:52:43pm PDT	a332ead	348	9,564.42ms	7,342.84ms	77%	57.7 MB	Sort · Nested Loop · CTE Scan +4 more
2024-10-01 08:02:40pm PDT	a332ead	348	9,120.33ms	7,772.78ms	85%	45.8 MB	Sort · Nested Loop · CTE Scan +4 more

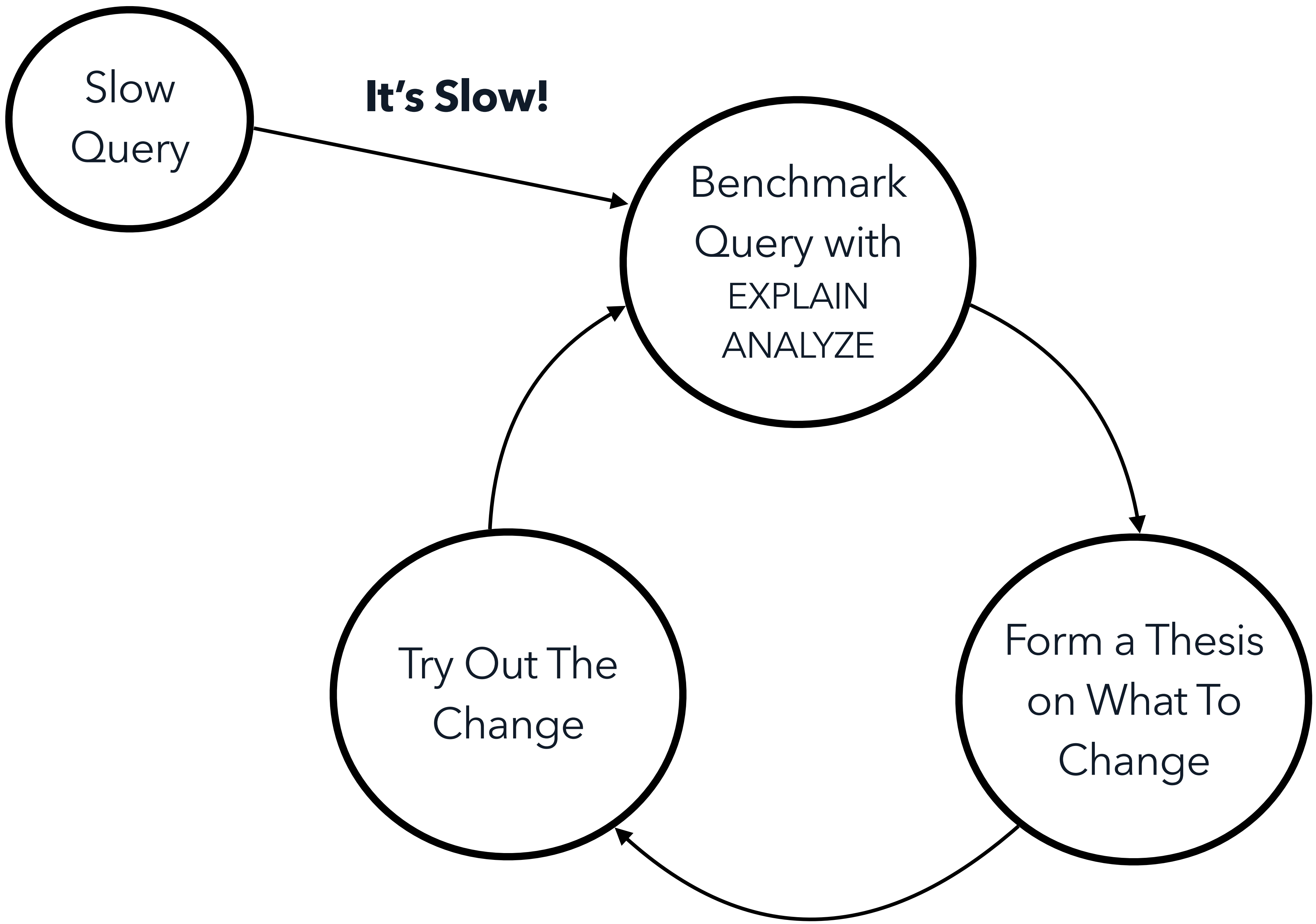
Plan Fingerprints show changes in plan structure

pganalyze

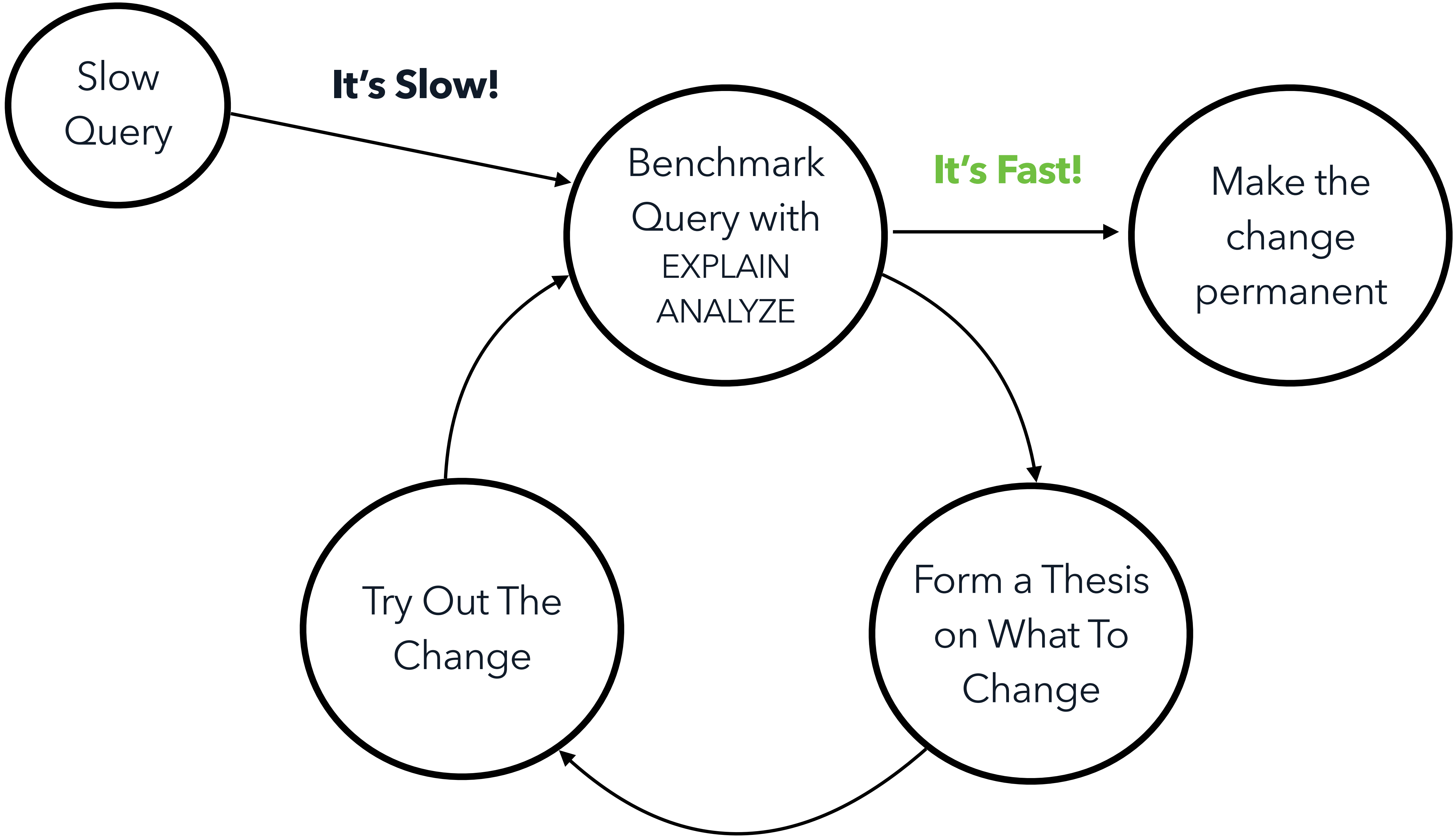

Slow Query
It's Slow!
Benchmark Query with EXPLAIN ANALYZE


Slow Query
It's Slow!
Benchmark Query with EXPLAIN ANALYZE
Form a Thesis on What To Change

pganalyze


Slow Query
It's Slow!
Benchmark Query with EXPLAIN ANALYZE
Form a Thesis on What To Change
Try Out The Change

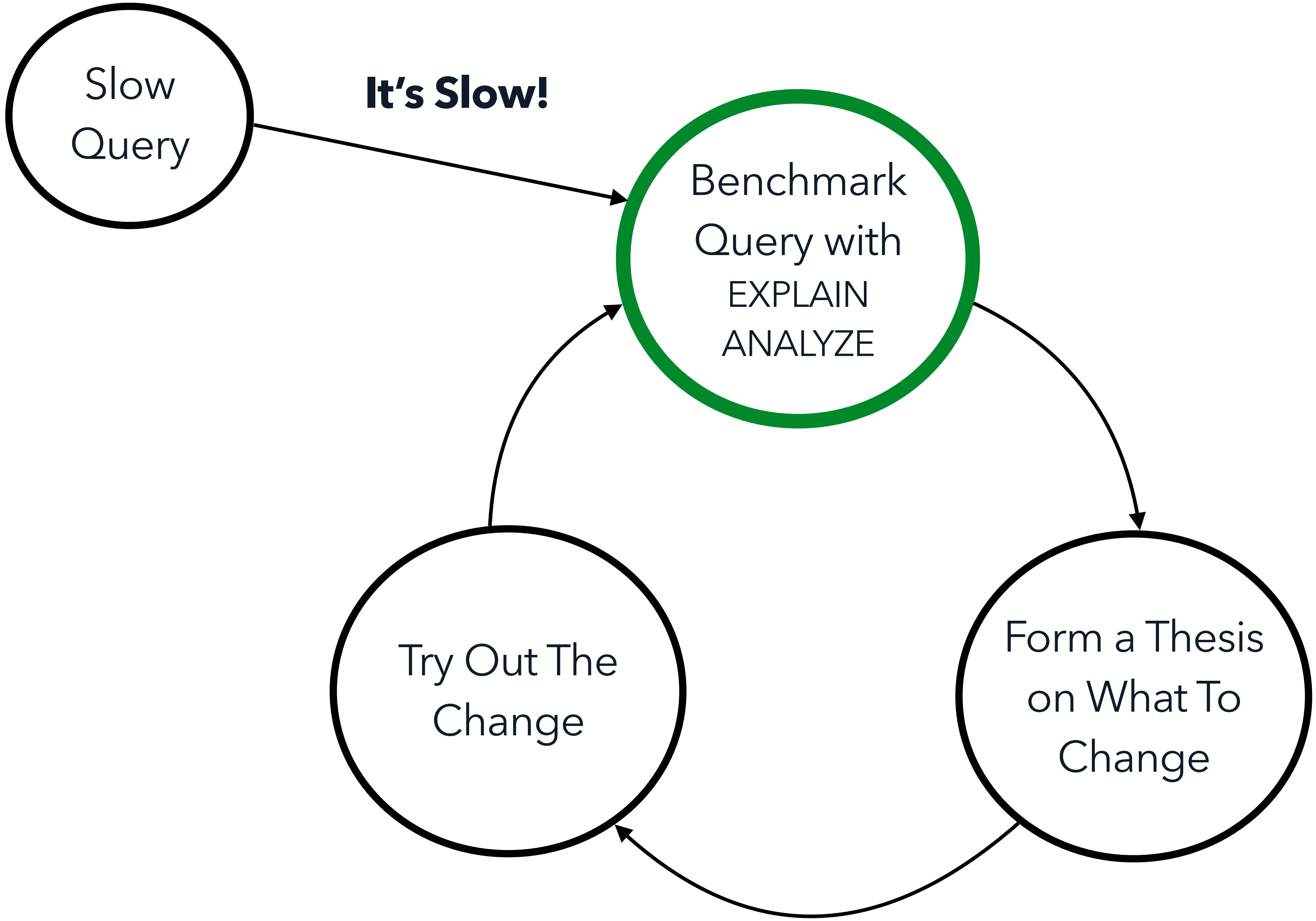
pganalyze


Slow Query
It's Slow!
Benchmark Query with EXPLAIN ANALYZE
It's Fast!
Make the change permanent
Form a Thesis on What To Change
Try Out The Change

pganalyze

pganalyze

Benchmarking with EXPLAIN (ANALYZE, BUFFERS)


Slow Query
It's Slow!
Benchmark Query with EXPLAIN ANALYZE
Form a Thesis on What To Change
Try Out The Change

pganalyze

EXPLAIN without ANALYZE

= The plan the planner chose (but no actual statistics)

EXPLAIN (ANALYZE)

= The plan chosen + runtime statistics

EXPLAIN(ANALYZE, BUFFERS)

= The plan chosen + runtime statistics + I/O statistics

```
postgres=# EXPLAIN SELECT * FROM test WHERE c = 123;
                              QUERY PLAN
------------------------------------------------------------------------
 Gather  (cost=1000.00..97366.28 rows=1 width=8)
   Workers Planned: 2
   ->  Parallel Seq Scan on test  (cost=0.00..96366.18 rows=1 width=8)
         Filter: (c = 123)
(4 rows)
```

pganalyze

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE c = 123;
                                                     QUERY PLAN
-----------------------------------------------------------------------------------
---------------------------------
 Gather  (cost=1000.00..97366.28 rows=1 width=8) (actual time=307.117..307.328
rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test  (cost=0.00..96366.18 rows=1 width=8) (actual
time=250.789..283.322 rows=0 loops=3)
         Filter: (c = 123)
         Rows Removed by Filter: 3333333
 Planning Time: 0.189 ms
 Execution Time: 307.371 ms
(8 rows)
```

pganalyze

```
postgres=# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM test WHERE c = 456;
                                                       QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..97366.28 rows=1 width=8) (actual time=303.560..304.600
rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=2757 read=41531
   I/O Timings: shared read=95.324
   ->  Parallel Seq Scan on test  (cost=0.00..96366.18 rows=1 width=8) (actual
time=256.848..286.938 rows=0 loops=3)
         Filter: (c = 456)
         Rows Removed by Filter: 3333333
         Buffers: shared hit=2757 read=41531
         I/O Timings: shared read=95.324
 Planning Time: 0.231 ms
 Execution Time: 304.649 ms
(12 rows)
```

pganalyze

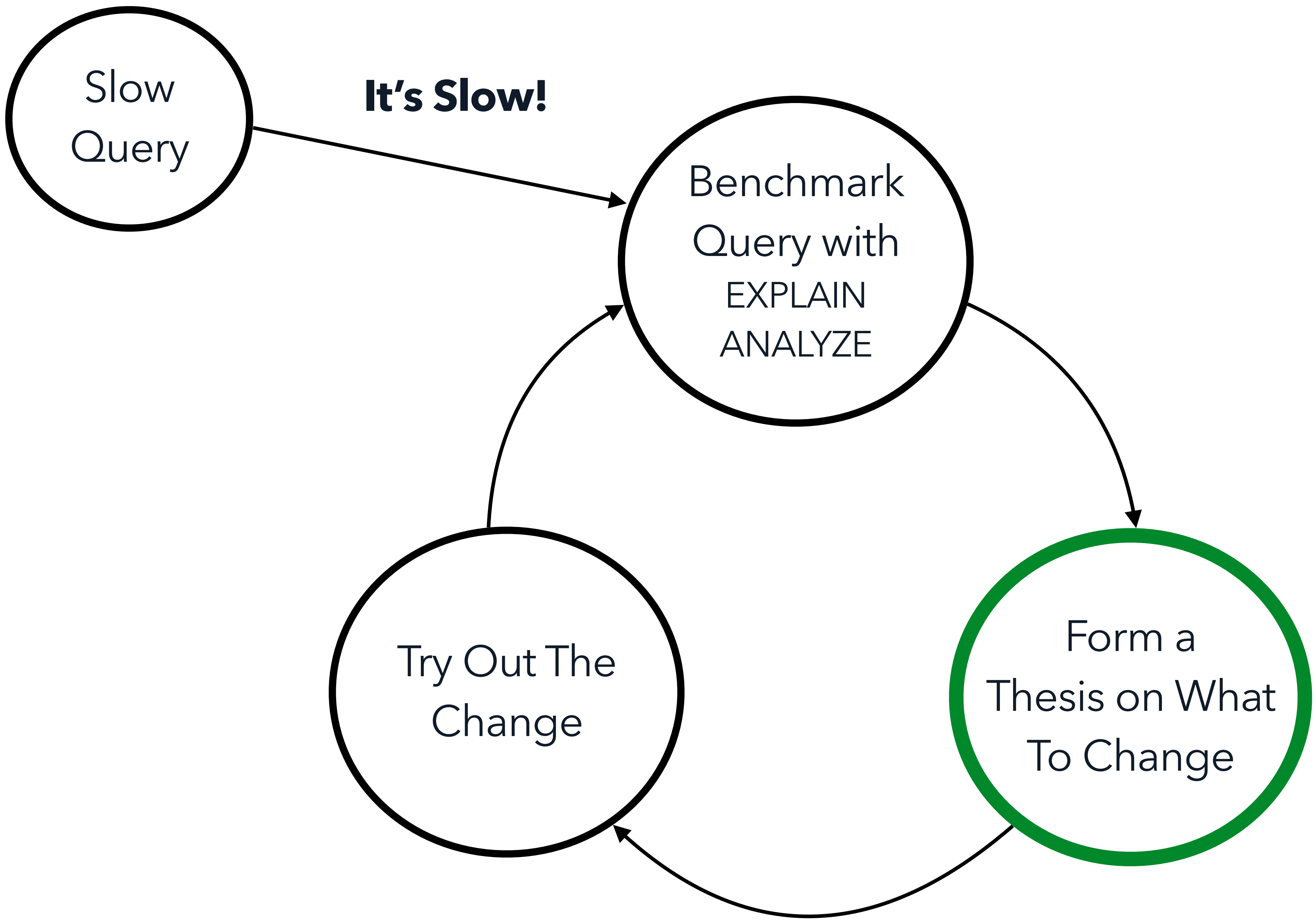
BUFFERS shows you the impact of the physical contents of the table (i.e. dead rows, empty space)

1 buffer = 8 kB buffer page

(on most Postgres installs)

pganalyze

Planner costing, and why it can never be perfect


Slow Query
It's Slow!
Benchmark Query with EXPLAIN ANALYZE
Form a Thesis on What To Change
Try Out The Change

pganalyze

“The planner's task is fuzzy, there can be many valid plans for the same query, and its not always clear which one is best.”

- Tom Lane in “Hacking the Query Planner” at PGCon ’11

Postgres planner responsibilities:

1. Find a good query plan.
2. Don't spend too much time (or memory) finding it.
3. Support the extensible aspects of Postgres.

What the planner doesn't do:

- Find all possible query plans
  (it discards seemingly worse plans quickly)
- Change a plan when its expectations don't hold true
  (e.g. a lot more rows match than expected)
- Keep track of execution performance
  (it will happily keep producing slow queries)

Cost estimation is what
really drives the planner's behavior. [...]

If it generates and rejects the plan you want,
you need to fix the cost estimation. [...]

"Garbage in, garbage out" applies here!
- Tom Lane

pganalyze

```
->  Index Scan using myindex on mytable
    (cost=0.56..11859.55 rows=10608 width=53)
```

Startup cost:
Effort to get the first row from the node
(matters a lot for LIMIT queries)

pganalyze

```
->  Index Scan using myindex on mytable
    (cost=0.56..11859.55 rows=10608 width=53)
```

Total cost:

What the planner aims to minimize

pganalyze

```
->  Index Scan using myindex on mytable
    (cost=0.56..11859.55 rows=10608 width=53)
```

Output row count:
Needed to estimate sizes of upper joins

pganalyze

```
->  Index Scan using myindex on mytable
    (cost=0.56..11859.55 rows=10608 width=53)
```

Average row width:
Estimate workspace for sorts, hashes
that store the node's output

pganalyze

What Is “Cost”?

pganalyze

Not a specific unit,
think of it as the “currency” that the planner operates in when it does **cost-based search**

pganalyze

What is the cost of a Sequential Scan?

pganalyze

src/backend/optimizer/path/costsize.c

```
/*
 * cost_seqscan
 *   Determines and returns the cost of scanning a relation sequentially.
 */
void
cost_seqscan(Path *path, PlannerInfo *root,
          RelOptInfo *baserel, ParamPathInfo *param_info)
{
   …
   /*
    * disk costs
    */
   disk_run_cost = spc_seq_page_cost * baserel->pages;

   /* CPU costs */
   …

   /* Adjust costing for parallelism, if used. */
   …

   path->startup_cost = startup_cost;
   path->total_cost = startup_cost + cpu_run_cost + disk_run_cost;
}
```

pganalyze

What is the cost of an Index Scan?

pganalyze

```
/*
* cost_index
*    Determines and returns the cost of scanning a relation using an index.
…
* In addition to rows, startup_cost and total_cost, cost_index() sets the
* path's indextotalcost and indexselectivity fields.  These values will be
* needed if the IndexPath is used in a BitmapIndexScan.
*/
void
cost_index(IndexPath *path, PlannerInfo *root, double loop_count,
           bool partial_path)
{
…
   /*
    * Call index-access-method-specific code to estimate the processing cost
    * for scanning the index, as well as the selectivity of the index (ie,
    * the fraction of main-table tuples we will have to retrieve) and its
    * correlation to the main-table tuple order.
    */
   amcostestimate(root, path, loop_count,
                  &indexStartupCost, &indexTotalCost,
                  &indexSelectivity, &indexCorrelation,
                  &index_pages);
```

src/backend/utils/adt/selfuncs.c

```
void btcostestimate(…)
{
   /*
    * For a btree scan, only leading '=' quals plus inequality quals for the
    * immediately next attribute contribute to index selectivity (these are
    * the "boundary quals" that determine the starting and stopping points of
    * the index scan).
    */
   indexBoundQuals = …

   /*
    * If the index is partial, AND the index predicate with the
    * index-bound quals to produce a more accurate idea of the number of
    * rows covered by the bound conditions.
    */
   selectivityQuals = add_predicate_to_index_quals(index, indexBoundQuals);

   btreeSelectivity = clauselist_selectivity(root, selectivityQuals,
                                             index->rel->relid,
                                             JOIN_INNER,
                                             NULL);
   numIndexTuples = btreeSelectivity * index->rel->tuples;
…
   costs.numIndexTuples = numIndexTuples;
   genericcostestimate(root, path, loop_count, &costs);
```

pganalyze

Selectivity is the hard part

- Tom Lane

pganalyze

src/backend/optimizer/path/clausesel.c

```
/*
 * clauselist_selectivity -
 * Compute the selectivity of an implicitly-ANDed list of boolean
 * expression clauses. The list can be empty, in which case 1.0
 * must be returned. List elements may be either RestrictInfos
 * or bare expression clauses --- the former is preferred since
 * it allows caching of results.
 *
 * The basic approach is to apply extended statistics first, on as many
 * clauses as possible, in order to capture cross-column dependencies etc.
 * The remaining clauses are then estimated by taking the product of their
 * selectivities, but that's only right if they have independent
 * probabilities, and in reality they are often NOT independent even if they
 * only refer to a single column. So, we want to be smarter where we can.
 * …
 */
Selectivity
clauselist_selectivity(PlannerInfo *root, List *clauses, int varRelid, JoinType
jointype, SpecialJoinInfo *sjinfo)
{
…
}
```

pganalyze

Selectivity also determines

how many rows are estimated to be returned from a plan node

(not just how expensive that node's cost is)

pganalyze

```
Seq Scan on mytable (… rows=1500, width=32)
  Filter: (mytable.user_id = 123)
```

↑

rows = total_rows * **selectivity**

pganalyze

The most typical bad row estimate on a scan is due to **clauses not actually being independent.**

pganalyze

a = 1 **AND** b = 1 **AND** c = 1 **AND** d = 1 **AND** e = 1

But what if all **"a=1"** also have **"b=1"**?

Or there are no **"c=1"** that have **"d=1"**?

pganalyze

To improve simple scan selectivity,
use **CREATE STATISTICS**
(extended statistics)

```
Nested Loop (… rows=1, width=24)
  Seq Scan on mytable (rows=1500 width=32)
  Seq Scan on othertable (rows=100 width=16)

join_selectivity = eqjoinselinner(…)
```

Join Estimates Are Complicated

(and often wrong)

pganalyze

src/backend/optimizer/path/clausesel.c

```
/*
 * eqjoinsel_inner --- eqjoinsel for normal inner join
 *
 * We also use this for LEFT/FULL outer joins; it's not presently clear
 * that it's worth trying to distinguish them here.
 */
static double
eqjoinsel_inner(…)
{
    double      selec;

    if (have_mcvs1 && have_mcvs2)
    {
        /*
         * We have most-common-value lists for both relations.  Run through
         * the lists to see which MCVs actually join to each other with the
         * given operator.  This allows us to determine the exact join
         * selectivity for the portion of the relations represented by the MCV
         * lists.  We still have to estimate for the remaining population, but
         * in a skewed distribution this gives us a big leg up in accuracy.
         * …
         */
```

pganalyze

To improve join selectivity (in some cases),
increase the both table column's statistics targets,
to collect more **MCVs**

pganalyze

Nested Loop `inefficient nested loop` 5

Actual Time:	3,375ms
I/O Time:	2,354ms
Est. Cost:	182
Actual Rows:	1,007 · est. 1

Function Scan `expensive` `i/o-heavy` `mis-estimate` 6

Actual Time:	1,592ms
I/O Time:	1,209ms
Est. Cost:	175
Actual Rows:	1,007 · est. 1

CTE Scan 7

index_sizes

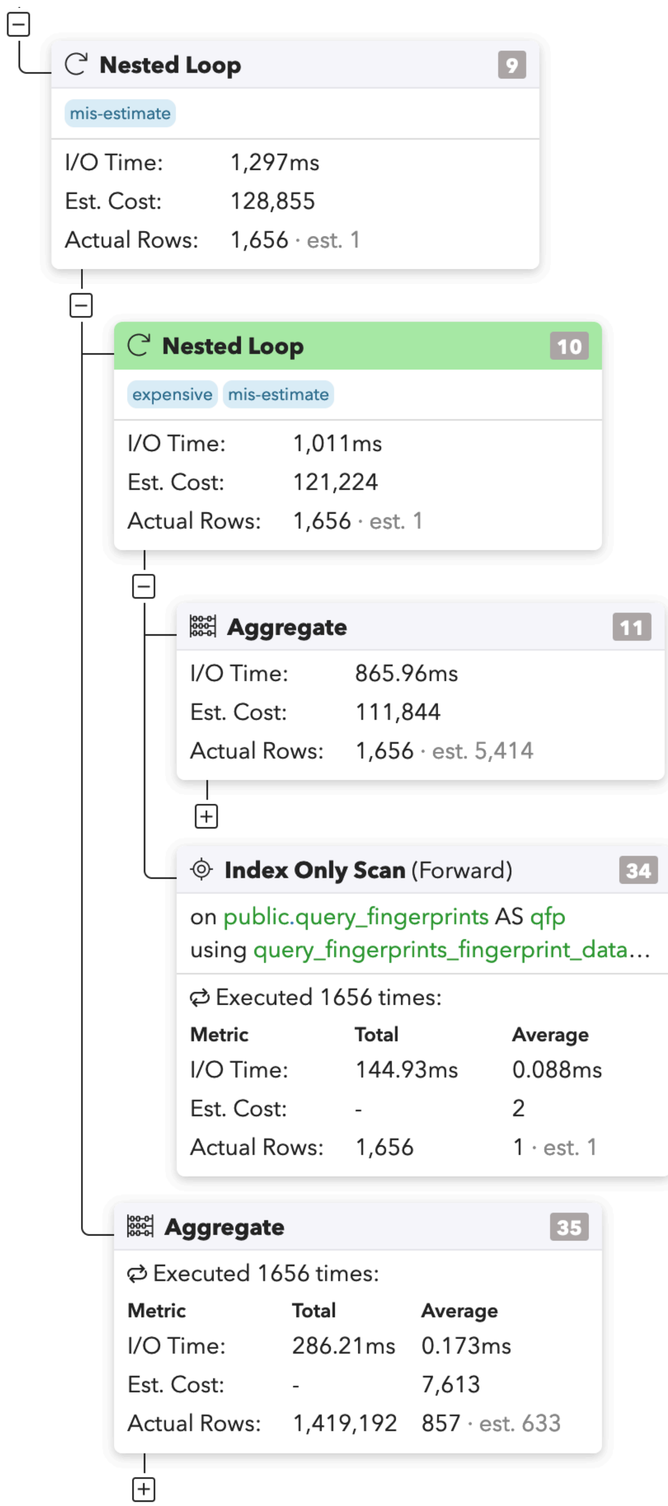
Executed 1007 times:

Metric	Total	Average
Actual Time:	1,705ms	1.69ms
I/O Time:	1,145ms	1.14ms
Est. Cost:	-	4
Actual Rows:	1,014,049	1,007 · est. 200

New pganalyze EXPLAIN Insight: Inefficient Nested Loop

```
->  Nested Loop  (cost=0.25..181.76 rows=1 width=152)
                 (actual rows=1007)
```

pganalyze



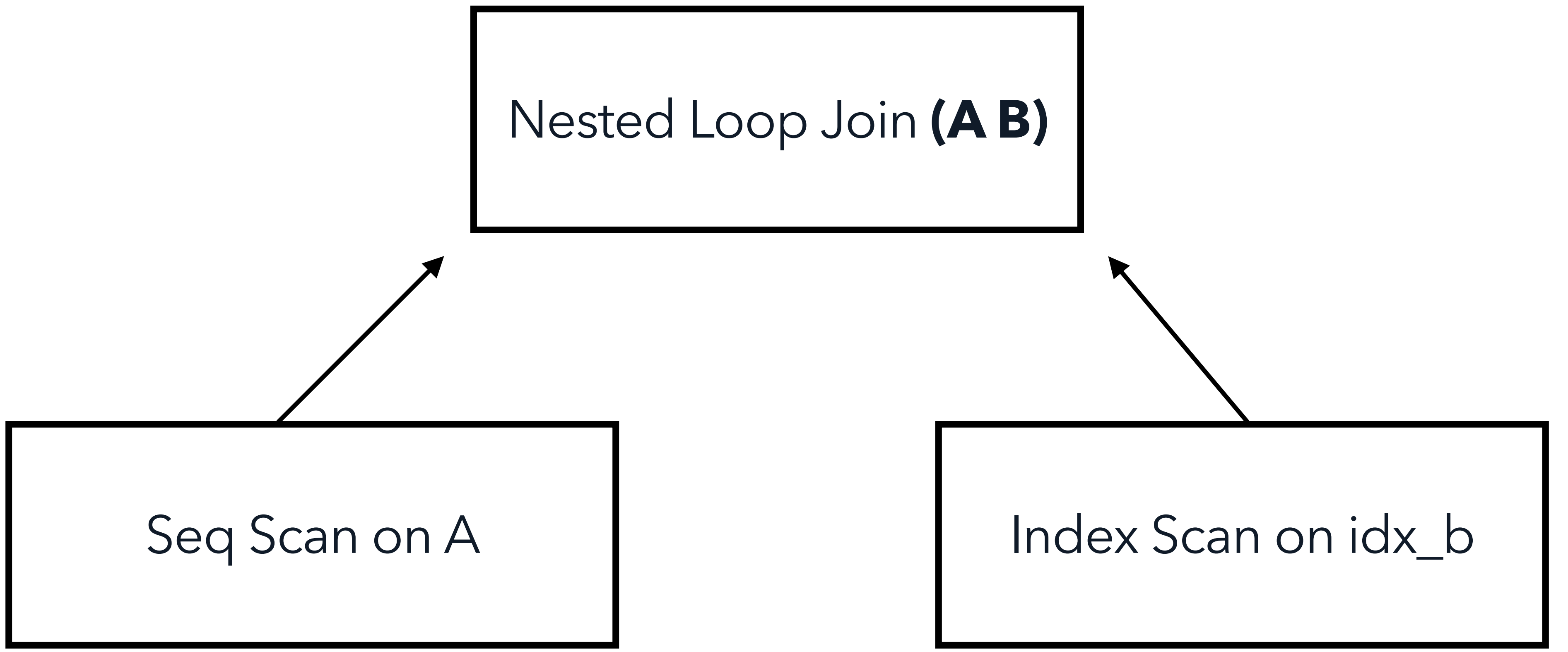
Both the lower Aggregate and the Index Only Scan had somewhat accurate row estimates.

But yet the Nested Loop estimate is wildly off, causing the upper Aggregate to run 1656 times, instead of the expected 1 time.

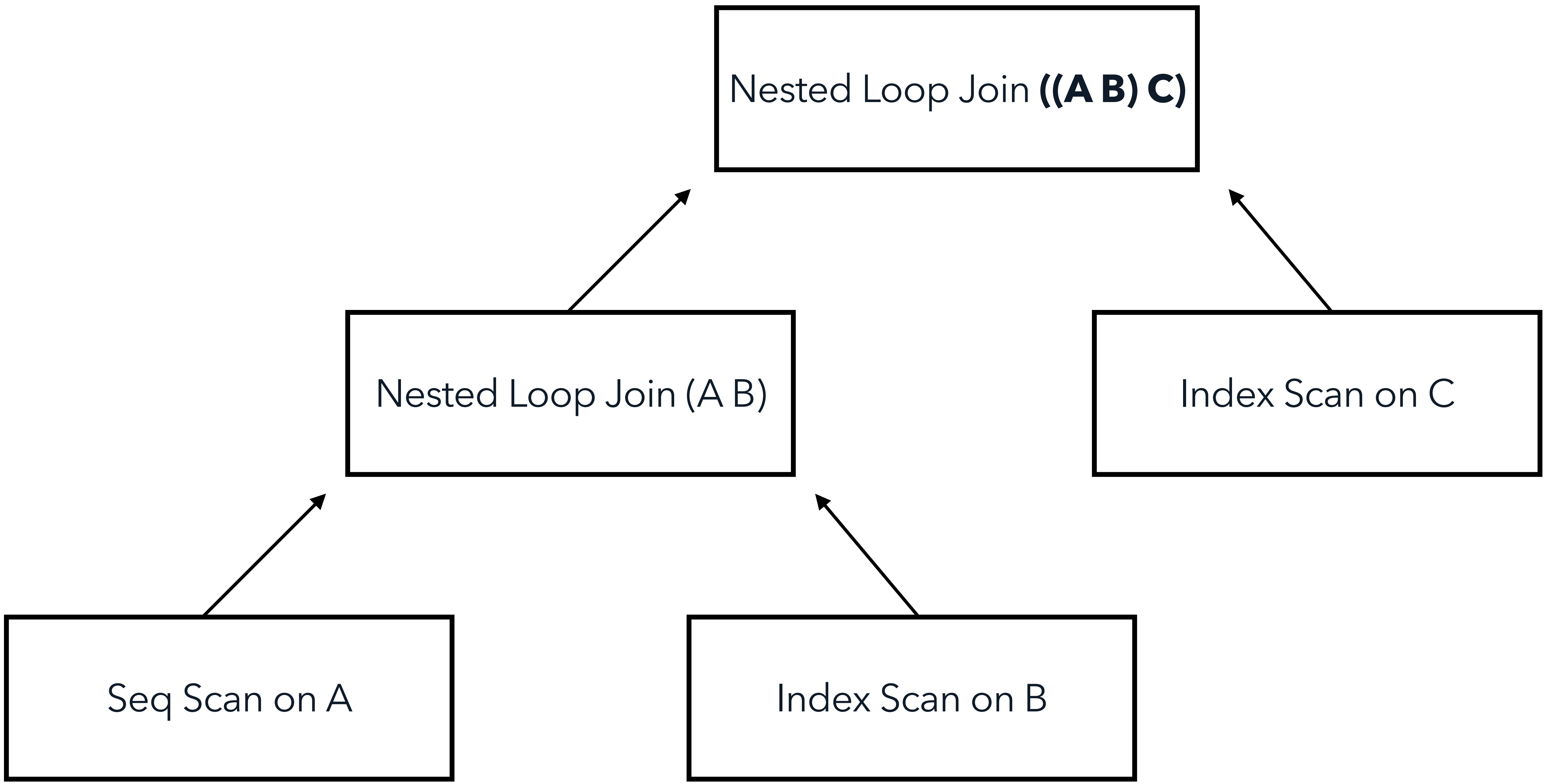
pganalyze

pganalyze

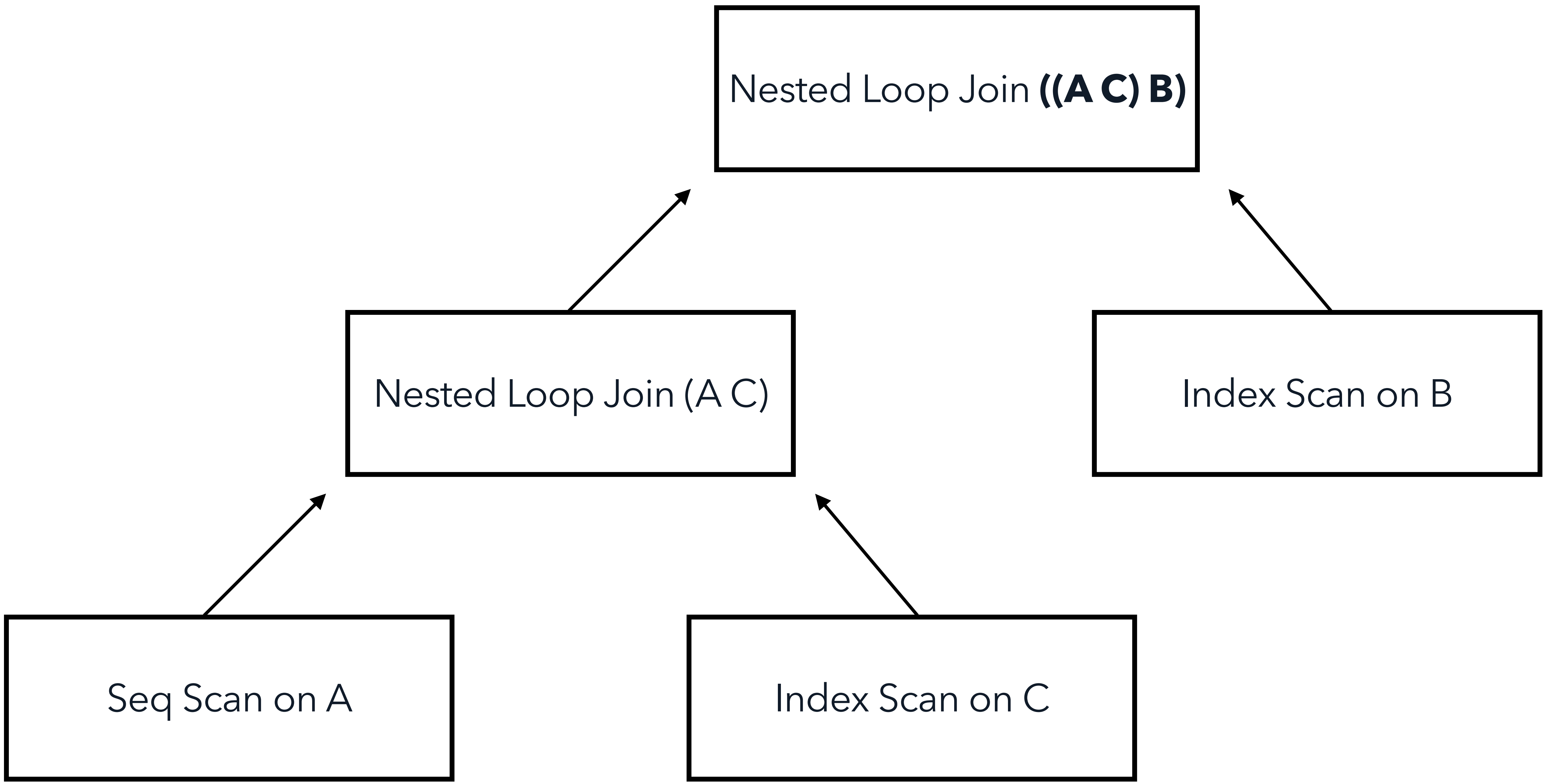
JOIN order and parameterized Index Scans



pganalyze


Nested Loop Join **((A B) C)**
Nested Loop Join (A B)
Index Scan on C
Seq Scan on A
Index Scan on B

pganalyze


Nested Loop Join **((A C) B)**
Nested Loop Join (A C)
Index Scan on B
Seq Scan on A
Index Scan on C

((A B) C)

= Join Order

First join A with B, then join the result of that with C

pganalyze

or, with join type and conditions:

(A leftjoin B on (Pab)) leftjoin C on (Pbc)

“Pab” = **P**redicate (aka JOIN condition) that references only columns from **A** and **B**

Joining lots of tables becomes expensive to analyze, fast.

n-way join could potentially have n! (n factorial) different join orders

If you join 12 or more tables, the genetic query optimizer (GEQO) is used by default

3 Essential Choices that cause
“Good” vs “Bad” plans for the same query:

1. **Scan Methods**
2. **Join Order**
3. **Join Methods**

You can detect Join Order in captured EXPLAINs:

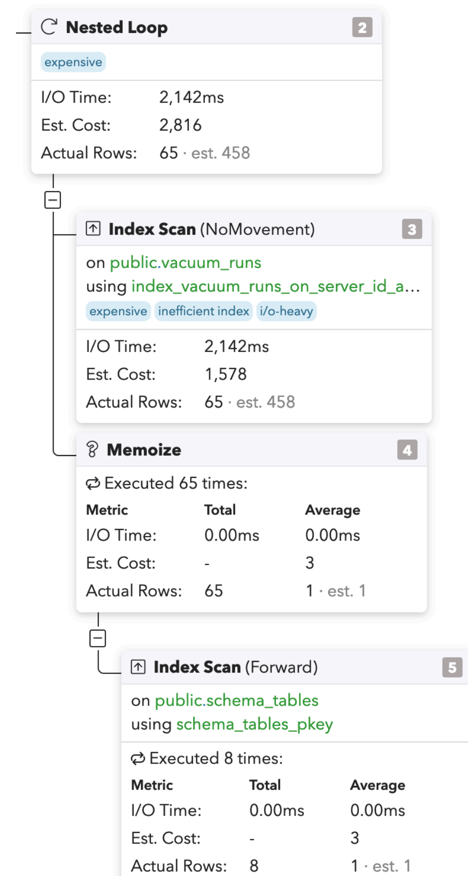
EXPLAINs

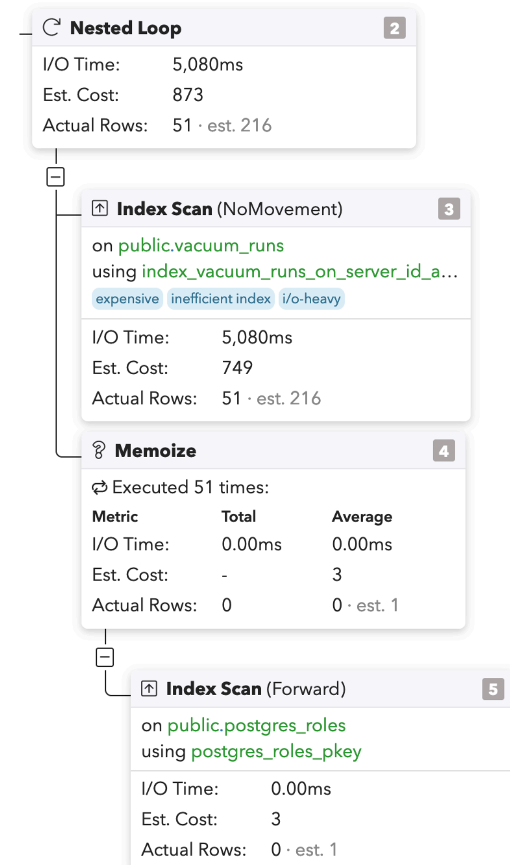
Join Orders:

- ((A B) C): ((vacuum_runs schema_tables) postgres_roles)
- ((A C) B): ((vacuum_runs postgres_roles) schema_tables)

EXECUTED AT	JOIN ORDER	EST. COST	RUNTIME
2023-03-28 04:12:13pm PDT	((A B) C)	59,195	14,532.70ms
2023-03-28 04:03:00pm PDT	((A B) C)	2,952	2,194.93ms
2023-03-28 04:02:18pm PDT	((A C) B)	1,469	5,281.25ms
2023-03-28 02:45:49pm PDT	((A B) C)	44,881	7,448.36ms
2023-03-28 01:36:25pm PDT	((A B) C)	90,977	9,588.22ms
2023-03-28 01:36:00pm PDT	((A B) C)	53,381	14,168.26ms
2023-03-28 12:52:07pm PDT	((A B) C)	29,286	4,211.10ms
2023-03-28 12:51:31pm PDT	((A B) C)	4,424	698.68ms
2023-03-28 12:32:39pm PDT	((A B) C)	11,460	1,578.15ms
2023-03-28 12:32:24pm PDT	((A B) C)	4,508	551.11ms
2023-03-28 11:57:40am PDT	((A B) C)	53,783	6,327.05ms

((A B) C) vs ((A C) B)





pganalyze

```
EXPLAIN SELECT *
   FROM t1
   JOIN t2 ON (t1.id = t2.t1_id)
  WHERE t1.field = '123';

                                         QUERY PLAN
-------------------------------------------------------------------
 Hash Join  (cost=13.74..37.26 rows=5 width=88)
   Hash Cond: (t2.t1_id = t1.id)
   ->  Seq Scan on t2  (cost=0.00..20.70 rows=1070 width=48)
   ->  Hash  (cost=13.67..13.67 rows=6 width=40)
         ->  Bitmap Heap Scan on t1  (…)
               Recheck Cond: (field = '123'::text)
               ->  Bitmap Index Scan on t1_field_idx  (…)
                     Index Cond: (field = '123'::text)
```

pganalyze

How can we **restrict (or filter)** a scan to a portion of the table's data?

1. Have an expression that uses fixed constant values (e.g. "WHERE NOT deleted_at")
2. Have a parameter value (or constant) passed from the client (e.g. "WHERE user_id = $1")
3. Filter based on another table's output, as part of a JOIN (e.g. "JOIN orgs ON (orgs.id = user.org_id)")

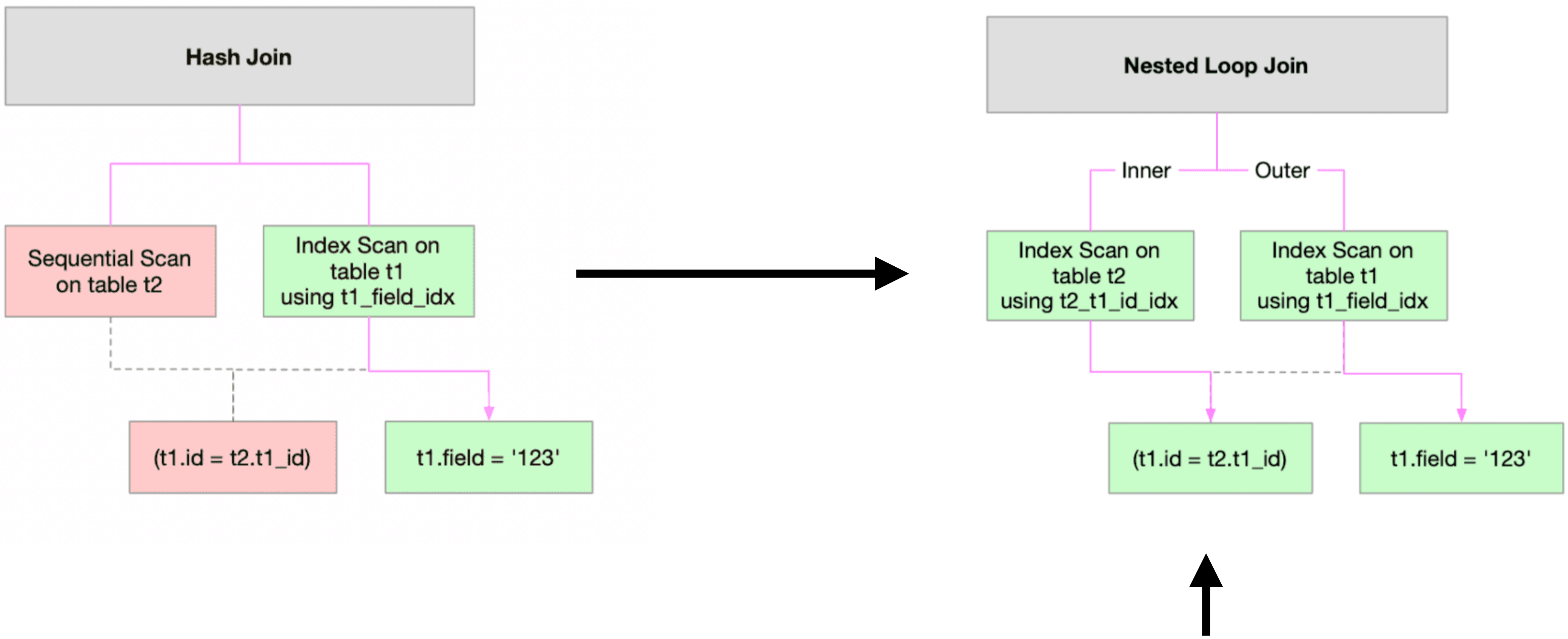
=> (1) and (2) are always eligible for an Index Scan.

=> (3) is only eligible when the Index Scan can be a **Parameterized Index Scan** (Inner Side of a Nested Loop)

```
EXPLAIN SELECT *
   FROM t1
   JOIN t2 ON (t1.id = t2.t1_id)
  WHERE t1.field = '123';

                                         QUERY PLAN
-------------------------------------------------------------------
 Nested Loop  (cost=0.55..16.60 rows=1 width=30)
   ->  Index Scan using t1_field_idx on t1  (…)
         Index Cond: (field = '123'::text)
   ->  Index Scan using t2_t1_id_idx on t2  (…)
         Index Cond: (t1_id = t1.id)
```

pganalyze



Parameterized Index Scan

Parameterized Index Scans must be on the inner side of a Nested Loop.

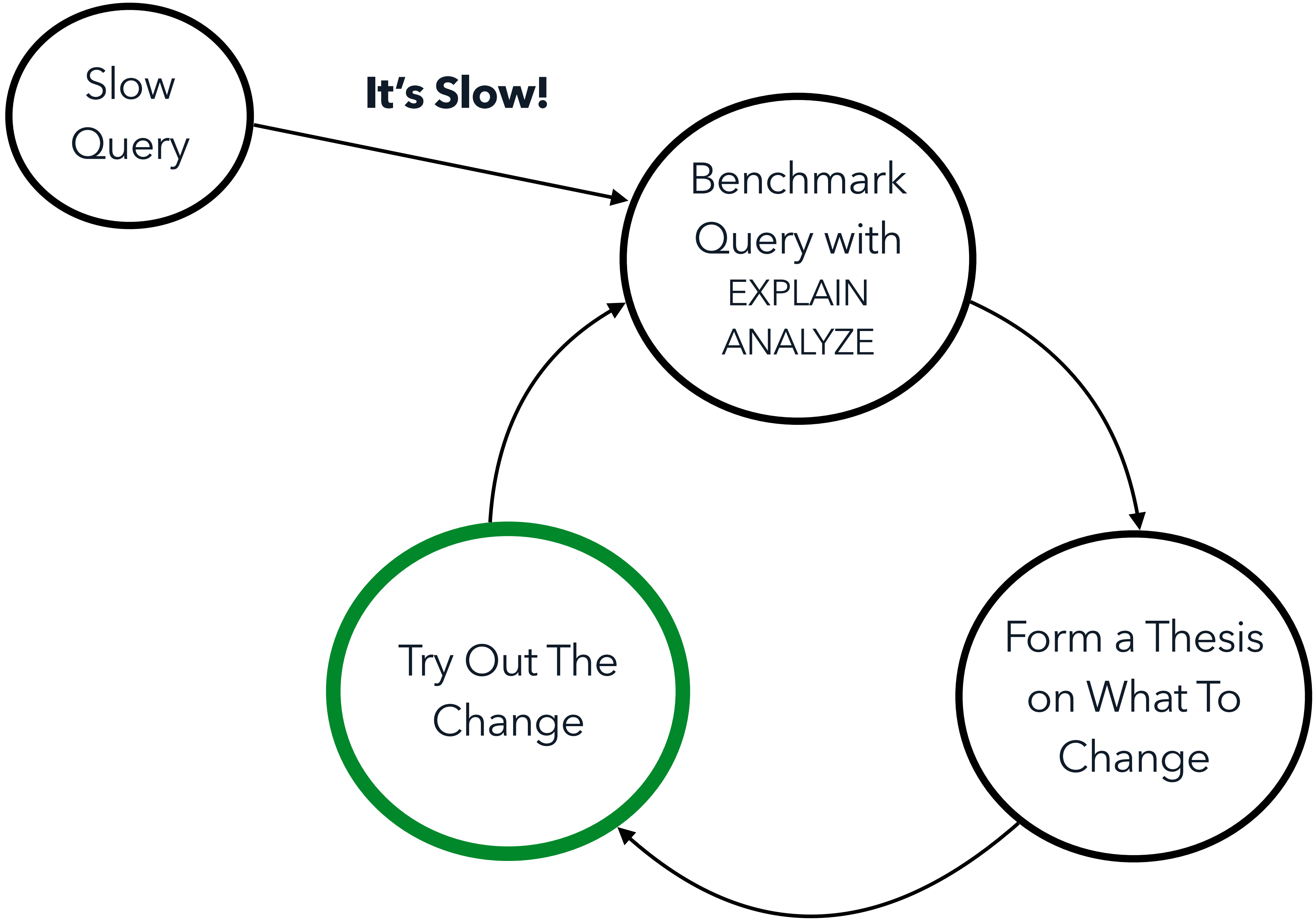
(Join order matters!)

pganalyze

pganalyze

Guiding the planner

to the right plan


Slow Query
It's Slow!
Benchmark Query with EXPLAIN ANALYZE
Form a Thesis on What To Change
Try Out The Change

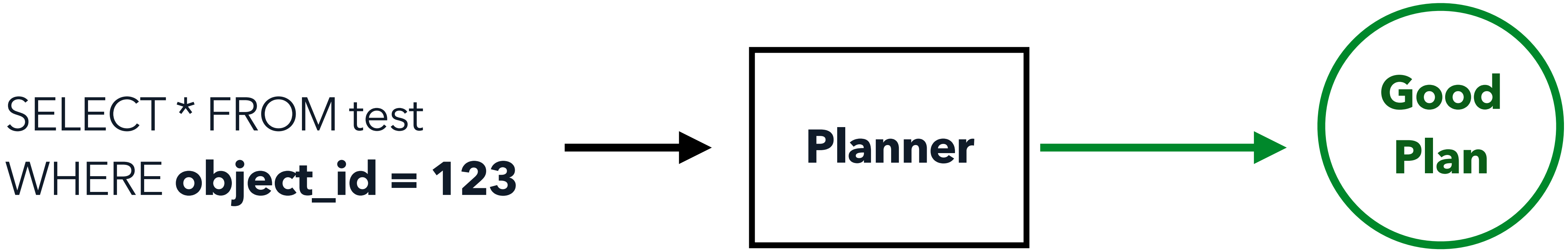
pganalyze

To Understand

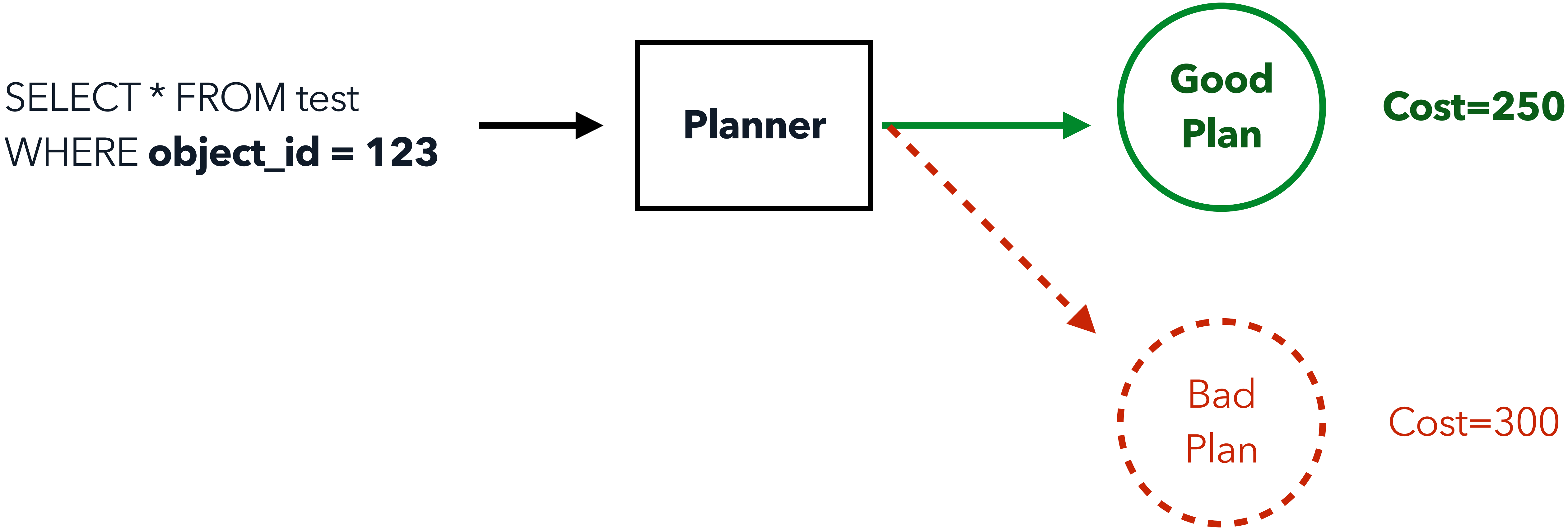
Why A "Bad" Plan Was Chosen

Start By Forcing The Good Plan

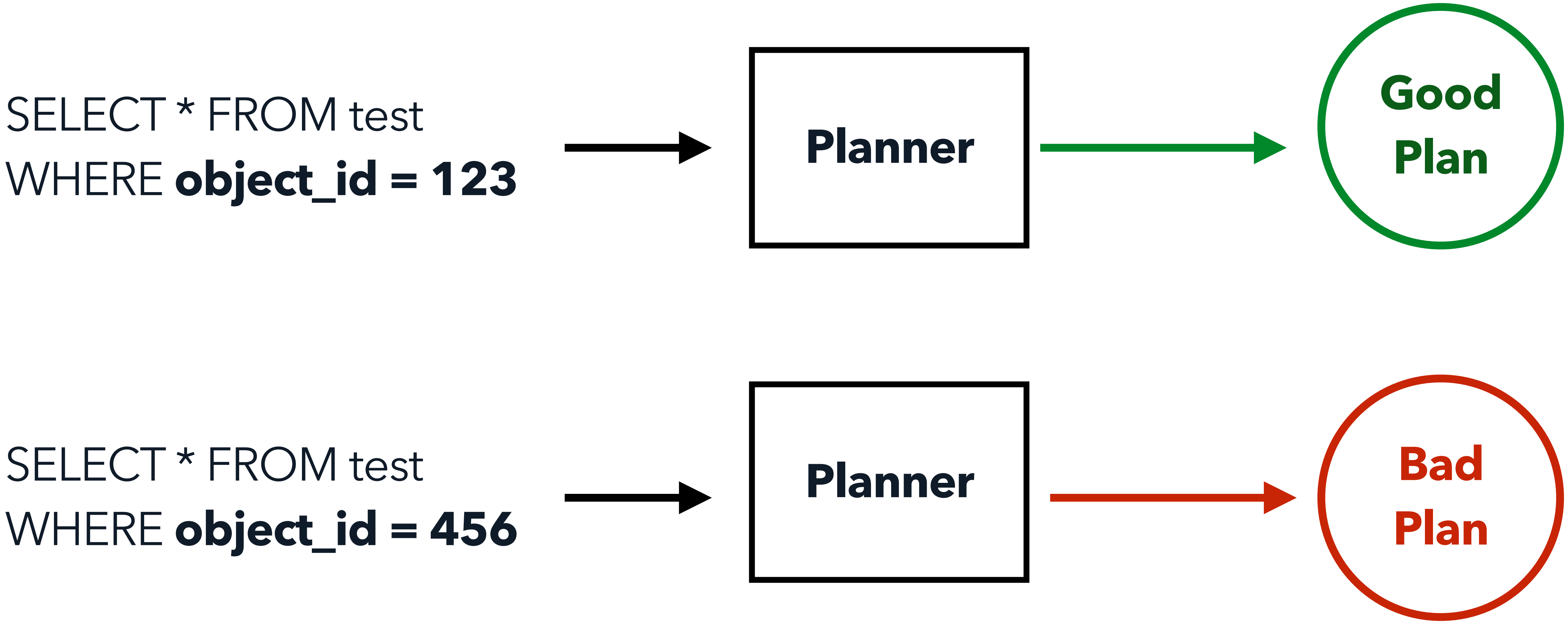
pganalyze



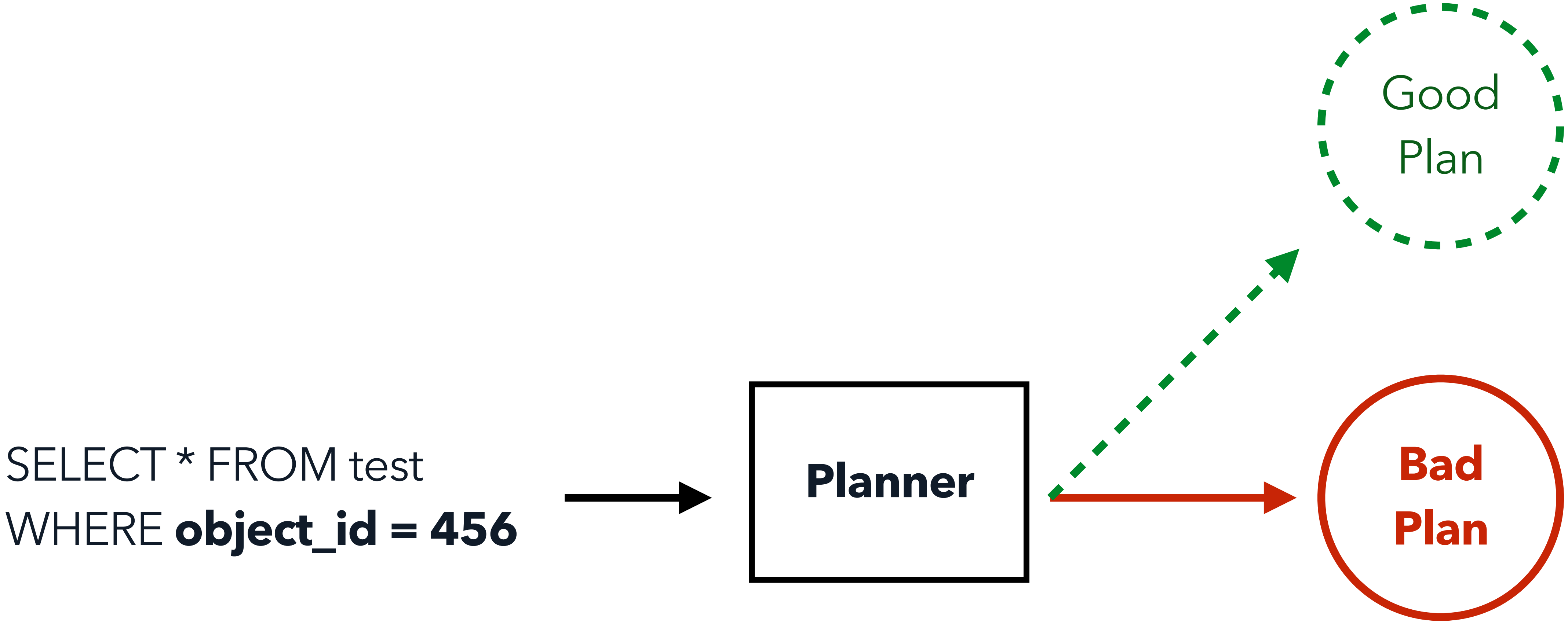
pganalyze


SELECT * FROM test
WHERE **object_id = 123**
Planner
Good Plan
Cost=250
Bad Plan
Cost=300

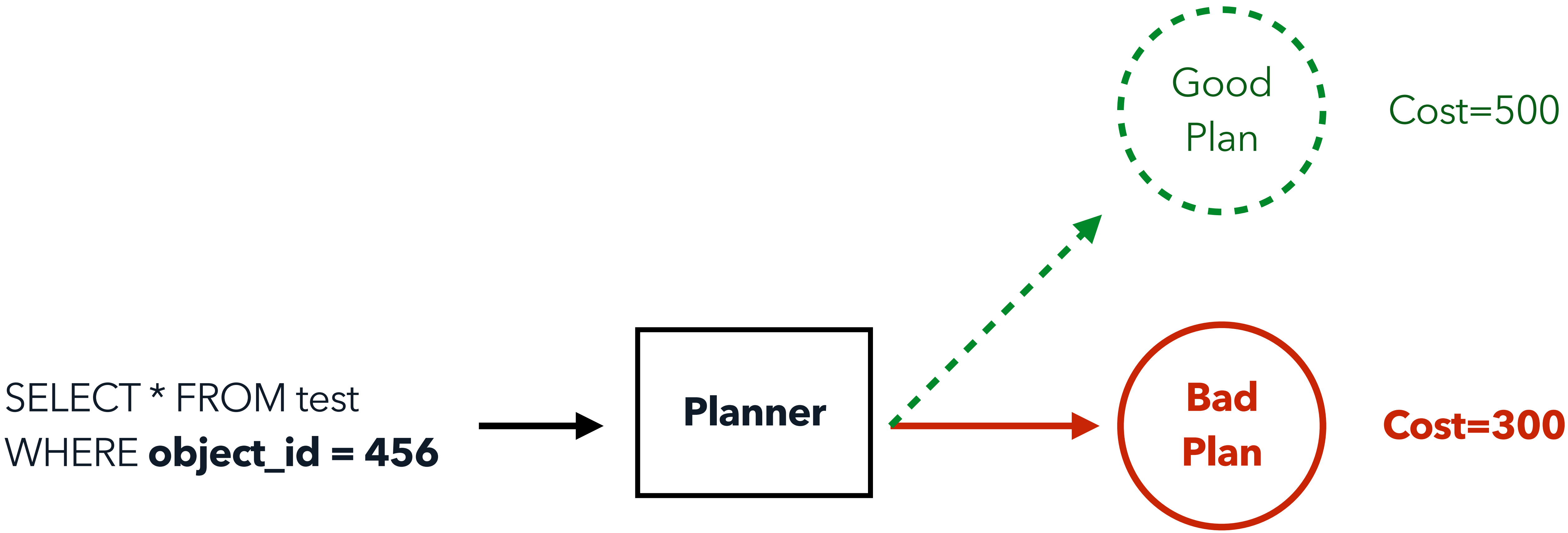
pganalyze


SELECT * FROM test
WHERE **object_id = 123**
Planner
Good
Plan
SELECT * FROM test
WHERE **object_id = 456**
Planner
Bad
Plan

pganalyze


SELECT * FROM test
WHERE **object_id = 456**
Planner
Good
Plan
Bad
Plan

pganalyze

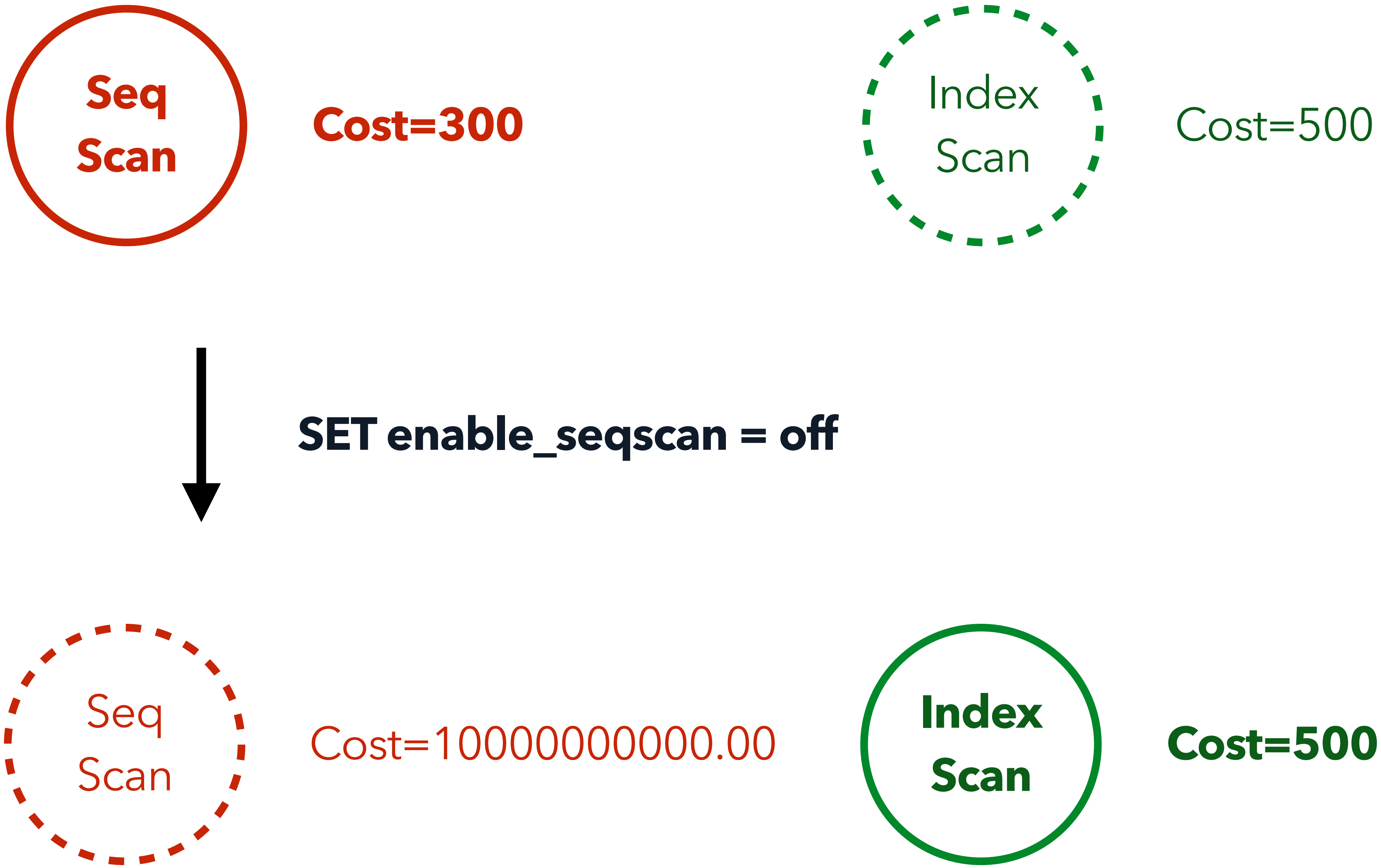

SELECT * FROM test
WHERE **object_id = 456**
Planner
Good
Plan
Cost=500
Bad
Plan
Cost=300

pganalyze

The easiest test:

If your bad plan
involves a **planner feature**,
turn it off.

pganalyze


Seq Scan
Cost=300
Index Scan
Cost=500
SET enable_seqscan = off
Seq Scan
Cost=10000000000.00
Index Scan
Cost=500

pganalyze

Once you have the right plan,
look at the individual plan nodes
and find out where the
cost mis-estimate originates

If you see a **Hash** or **Merge Join** being used instead of a **Nested Loop** + **Parameterized Index Scan**, try:

```
SET enable_mergejoin = off;
SET enable_hashjoin = off;
```

pganalyze

For more complicated cases,

Utilize pg_hint_plan to force the good plan

(to find the root cause of the cost mis-estimate)

```
EXPLAIN SELECT EXISTS (
  SELECT 1 FROM schema_column_stats scs WHERE scs.invalidated_at_snapshot_id IS NULL AND scs.table_id IN (
    SELECT id FROM schema_tables WHERE invalidated_at_snapshot_id IS NULL AND database_id = 12345));

                                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------
Result  (cost=9.13..9.14 rows=1 width=1)
   InitPlan 1 (returns $1)
     ->  Nested Loop  (cost=1.00..971672.56 rows=119623 width=0)
           ->  Index Only Scan using index_schema_column_stats_on_table_id on schema_column_stats scs
               (cost=0.43..372676.50 rows=23553966 width=8)
           ->  Memoize  (cost=0.57..0.61 rows=1 width=8)
                 Cache Key: scs.table_id
                 Cache Mode: logical
                 ->  Index Scan using schema_tables_pkey on schema_tables  (cost=0.56..0.60 rows=1 width=8)
                       Index Cond: (id = scs.table_id)
                       Filter: ((invalidated_at_snapshot_id IS NULL) AND (database_id = 12345))
```

Bad plan, with join order = (schema_column_stats schema_tables)

```
SET enable_memoize = off;

EXPLAIN SELECT EXISTS (
  SELECT 1 FROM schema_column_stats scs WHERE scs.invalidated_at_snapshot_id IS NULL AND scs.table_id IN (
    SELECT id FROM schema_tables WHERE invalidated_at_snapshot_id IS NULL AND database_id = 12345));

                                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------
Result  (cost=13.13..13.14 rows=1 width=1)
   InitPlan 1 (returns $1)
     ->  Nested Loop  (cost=0.99..1451807.35 rows=119623 width=0)
           ->  Index Scan using schema_tables_database_id_schema_name_table_name_idx on schema_tables
               (cost=0.56..37778.03 rows=34753 width=8)
                 Index Cond: (database_id = 12345)
           ->  Index Only Scan using index_schema_column_stats_on_table_id on schema_column_stats scs
               (cost=0.43..26.68 rows=1401 width=8)
                 Index Cond: (table_id = schema_tables.id)
```

Good plan, with join order = (schema_tables schema_column_stats)

pganalyze

```
/*+ Leading((scs schema_tables)) IndexOnlyScan(scs index_schema_column_stats_on_table_id) IndexScan(schema_tables
schema_tables_pkey) Set(enable_memoize off) */
EXPLAIN SELECT EXISTS (
  SELECT 1 FROM schema_column_stats scs WHERE scs.invalidated_at_snapshot_id IS NULL AND scs.table_id IN (
    SELECT id FROM schema_tables WHERE invalidated_at_snapshot_id IS NULL AND database_id = 12345));

                                                                      QUERY PLAN
--------------------------------------------------------------------------------------------------------------
Result  (cost=122.90..122.91 rows=1 width=1)
   InitPlan 1 (returns $1)
     ->  Nested Loop  (cost=0.99..14582869.23 rows=119623 width=0)
           ->  Index Only Scan using index_schema_column_stats_on_table_id on schema_column_stats scs
               (cost=0.43..372676.50 rows=23553966 width=8)
           ->  Index Scan using schema_tables_pkey on schema_tables  (cost=0.56..0.60 rows=1 width=8)
                 Index Cond: (id = scs.table_id)
                 Filter: ((invalidated_at_snapshot_id IS NULL) AND (database_id = 12345))
```

Bad plan, with join order = (schema_tables schema_column_stats)

pganalyze

Good plan:
1,451,807 cost

```
->  Nested Loop  (cost=0.99..1451807.35 rows=119623 wic
      ->  Index Scan using schema_tables_database_id_sc
          (cost=0.56..37778.03 rows=34753 width=8)
```

Bad plan without Memoize:
14,582,869 cost

```
->  Nested Loop  (cost=0.99..14582869.23 rows=119623 wi
      ->  Index Only Scan using index_schema_column_sta
          (cost=0.43..372676.50 rows=23553966 width=8)
```

Bad plan with Memoize:
971,672 cost

```
->  Nested Loop  (cost=1.00..971672.56 rows=119623 widt
      ->  Index Only Scan using index_schema_column_sta
          (cost=0.43..372676.50 rows=23553966 width=8)
```

pganalyze

6 ways to guide the planner:

1. For simple scan selectivity, look into CREATE STATISTICS
2. For join selectivity, try increasing statistics target
3. Review cost settings (e.g. random_page_cost)
4. Create multi-column indexes that
   align with the planner's biases (e.g. for bounded sorts)
5. For complex queries with surprising join order,
   try forcing materialization (WITH x AS MATERIALIZED…)
6. For multi-tenant apps, consider adding more explicit
   clauses like "WHERE customer_id = 123"

pganalyze

DB column stats check: Add filter on server_id to improve performance #2693

Edit | Code

Merged **lfittl** merged 1 commit into `main` from `improve-get-column-stats-helper-check` 3 weeks ago

Conversation 0 | Commits 1 | Checks 3 | Files changed 3 | +19 −5

lfittl commented last month • edited

The previous query was producing one of two plans in practice:

(1)

```
NestedLoop(schema_column_stats schema_tables)
- IndexScan(schema_tables_database_id_schema_name_table_name_idx)
  Index Cond: (database_id = $1)
- IndexOnlyScan(index_schema_column_stats_on_table_id)
   Index Cond: (table_id = schema_tables.id)
```

(2)

```
NestedLoop(schema_column_stats schema_tables)
- IndexOnlyScan(index_schema_column_stats_on_table_id)
  Index Cond: -
- Memoize
-- IndexScan(schema_tables_pkey)
   Index Cond: (id = schema_column_stats.table_id)
   Filter: (database_id = $1)
```

Plan (1) is the right choice here, however in the pathological case this is not chosen, due to an overestimate on the number of matching rows in schema_tables (~40k instead of 100).

Plan (2) appears to happen because the Memoize costing calculates a cache rate of ~95%, and thus makes the many iterations over schema_tables very cheap.

After multiple fruitless attempts at fixing the estimation for (1), instead make the plan with Memoize behave less bad, by introducing a filter on server_id resulting in one of these two plan choices:

Reviewers

msakrejda ✓

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Successfully merging this pull request may close these issues.

Index Advisor overview: Investigate issue wit...

Notifications Customize

Unsubscribe

You're receiving notifications because you're watching this repository.

pganalyze

If you can, choose

Better Statistics

over

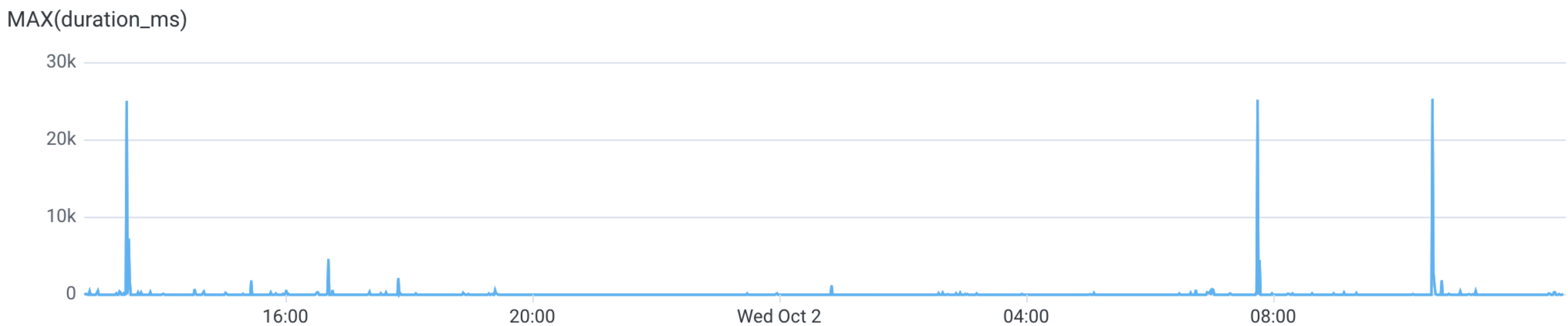
Planner Hints

pganalyze

pganalyze

Query Tuning with pganalyze

Let’s start with a trace of a slow web request

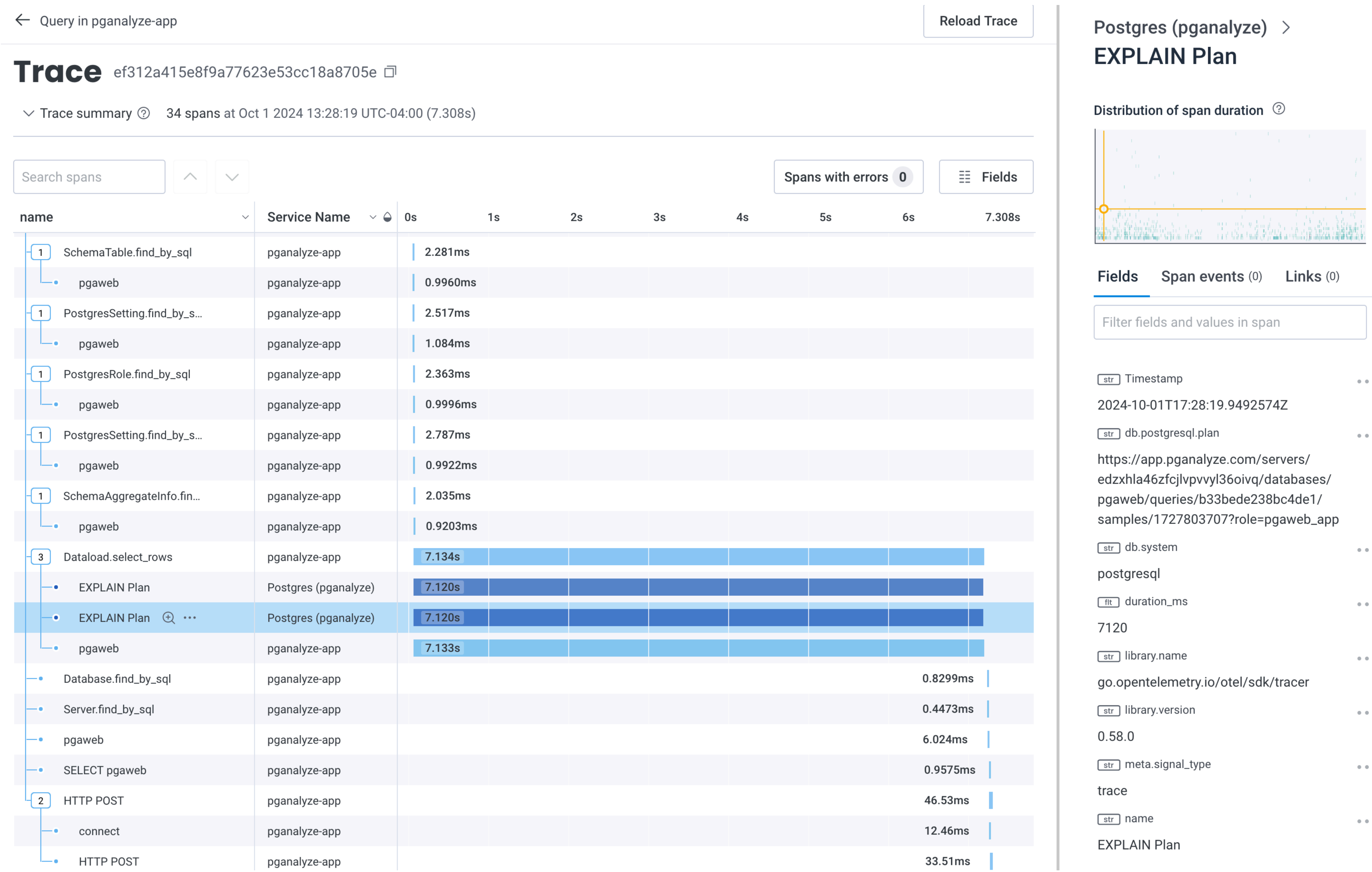


Overview BubbleUp Correlations **Traces** Explore Data

Shows up to 10 traces with the slowest spans from the selected time range. Learn more.

	Root Service Name	Root Name	Root Duration ms	Number of Spans	Span Summary	Trace ID
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	2,772.77562	26		8d59171091ac7ee7f4f5382d2754027c
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	2,133.37864	26		c0c4d95a6dd4647637b248a0a6161a29
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	25,356.02089	30		ec2decbb788ce9eaaae1d9d3b6bf1625
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	2,710.38595	28		a3278f71c6837a281da62551e7c9645a
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	4,484.71493	34		87856ed9c2651500187ef9bdd690d387
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	25,220.97727	29		60a1ce3242e9aebf397d32f03d2620dd
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	2,132.79932	30		f6dee27f224f8c0f0daabf3313c9bde4
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	4,626.37905	30		28a4c4ff9bd368d6cc42814d760b1391
	pganalyze-app	Api::GraphqlController#graphql (SchemaTableQueries)	7,257.8716	34		ef312a415e8f9a77623e53cc18a8705e

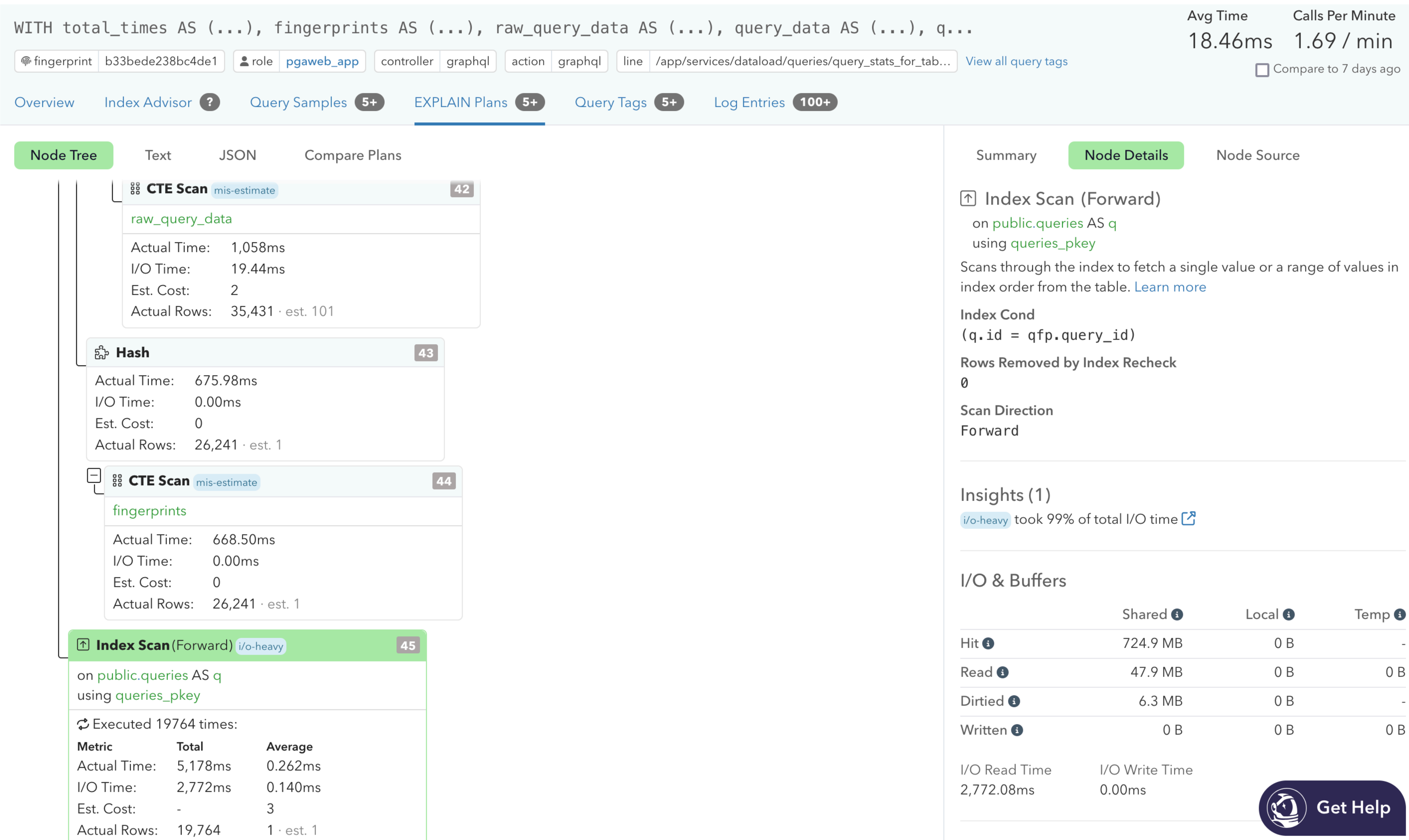
Let’s start with a trace of a slow web request



Multiple Mis-Estimates of Nested Loops

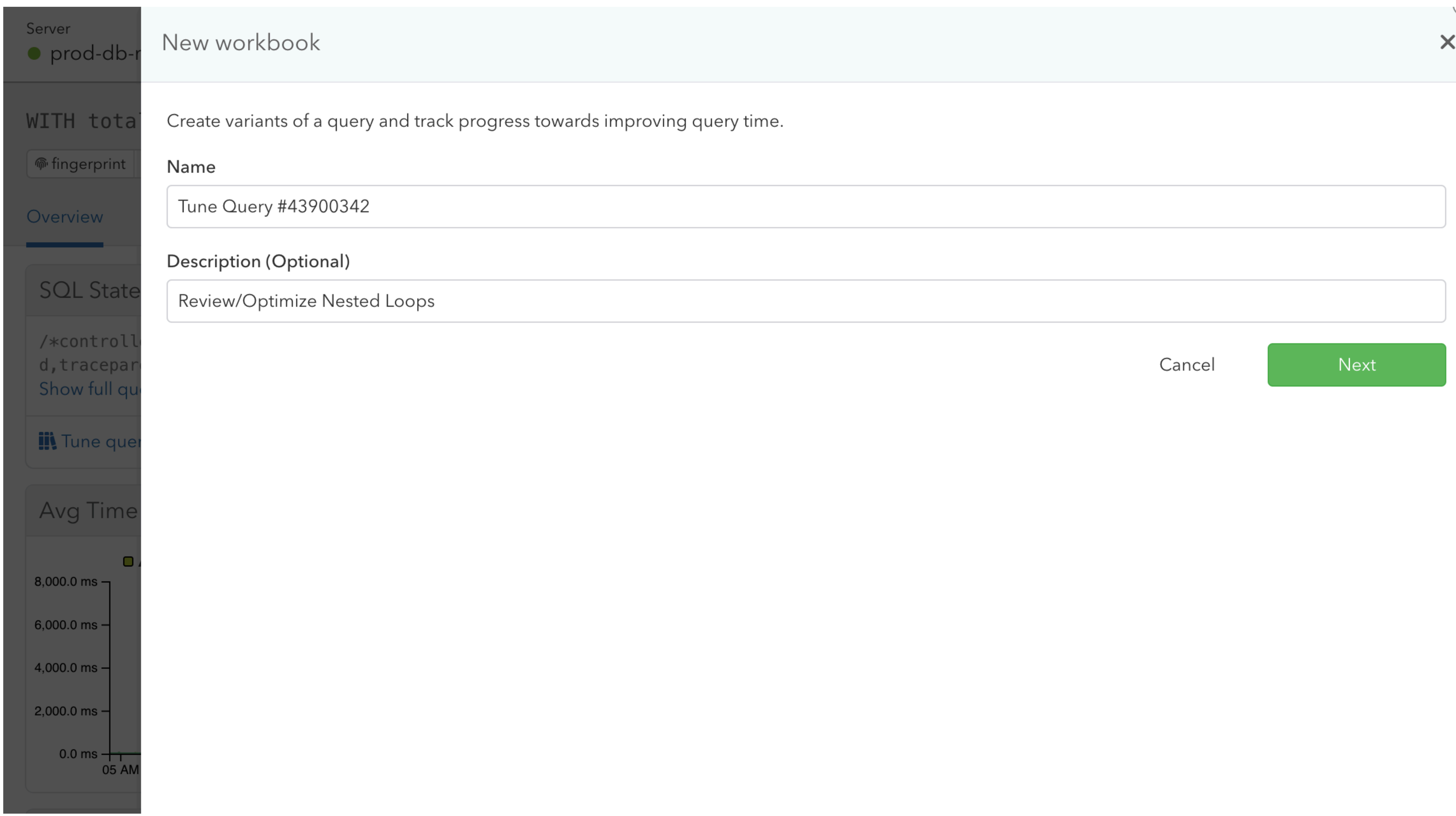
pganalyze

Index Scan in a Loop takes 99% of I/O Time



Let's Tune The Query!

Let’s Tune The Query!



Automatic Naming of Parameters

Tune Query #43900342

```
/*controller:graphql,action:graphql,line:/app/services/dataload/queries/query_stats_for_table.r
b:181:in `query_stats_for_table',sentry_trace_id:11c0590f5359469fbc1dd94a99fbe18d,traceparent:00-
c196797ad1cd8128c0baf25162809ad4-c3722293bedf5213-01,tracestate:pganalyze=t:1727812015.8315823*/
WITH total_times AS (
SELECT SUM(query_stats_blk_read_time_sum + query_stats_blk_write_time_sum) AS total_iotime,
SUM(query_stats_total_time_sum) AS total_runtime
FROM query_overview_stats_35d qos
WHERE qos.database_id = $database_id AND qos.collected_at BETWEEN $collected_at_3 AND $collected_
at_4
),
fingerprints AS (
SELECT qf.*
FROM query_table_associations qta
JOIN query_occurrences o ON o.query_id = qta.query_id AND o.database_id = $database_id_6 AND o.la
st >= $param_10::date
JOIN query_fingerprints qf ON qf.query_id = qta.query_id
WHERE qta.database_id = $database_id_2
AND table_name IN ($table_name, $param_11 || $param_12 || $param_9)
),
```

pganalyze

Paste a query sample to extract parameters

Custom query

```
WITH total_times AS (
SELECT SUM(query_stats_blk_read_time_sum + query_stats_blk_write_time_sum) AS total_iotime,
SUM(query_stats_total_time_sum) AS total_runtime
FROM query_overview_stats_35d qos
WHERE qos.database_id = [redacted] AND qos.collected_at BETWEEN '2024-09-28 04:30:00' AND '2024-09-28
14:30:00'
),
```

Add parameters from query

Add parameters manually

pganalyze

Benchmark the same query, with different parameters

Run EXPLAIN ANALYZE

Switch to Collector workflow

EXPLAIN for Param Set 1

Command

copy

```
EXPLAIN (ANALYZE, VERBOSE, BUFFERS, FORMAT JSON)
/*controller:graphql,action:graphql,line:/app/services/dataload/queries/query_stats_for_tabl
e.rb:181:in `query_stats_for_table',sentry_trace_id:11c0590f5359469fbc1dd94a99fbe18d,tracepa
rent:00-c196797ad1cd8128c0baf25162809ad4-c3722293bedf5213-01,tracestate:pganalyze=t:17278120
15.8315823*/ WITH total_times AS (
SELECT SUM(query_stats_blk_read_time_sum + query_stats_blk_write_time_sum) AS total_iotime,
SUM(query_stats_total_time_sum) AS total_runtime
FROM query_overview_stats_35d qos
WHERE qos.database_id = [redacted] AND qos.collected_at BETWEEN '2024-09-30 17:28:19' AND '2
024-10-01 17:28:19'
```

EXPLAIN output

Text or JSON format supported

Paste EXPLAIN output here...

We've recorded the Baseline

Tune Query #43900342

Overview | Compare Plans | Parameter Sets

All Query Plans

Baseline

Add Query Variant

Query

#43900342

Query tags

fingerprint b33bede238bc4de1

role pgaweb_app

Baseline

With parameter aliases

```
/*controller:graphql,action:graphql,line:/app/services/dataload/queries/query_stats_for_table.rb:181:in `query_stats_for_table',sentry_trace_id:11c0590f5359469fbc1dd94a99fbe18d,traceparent:00-c196797a...
```

Show full query text

Query Plans

VARIANT	PLAN	PARAMETER SET	EST. COST	RUNTIME
Baseline	a392842	Param Set 1	3,017	1,738.88ms
Baseline	a3ed913	Param Set 2	986	382.88ms
Baseline	a3aa4a4	Param Set 3	1,874	49.00ms

pganalyze

Why are the plans different?

Comparison

Plan A: Baseline - Parameter Set 23 - a392842 - runtime: 1,738.88ms - I/O read time: 0.00ms

Plan B: Baseline - Parameter Set 27 - a3aa4a4 - runtime: 49.00ms - I/O read time: 0.00ms

Cost Metric: ○ Est. Total Cost (Self) ○ Runtime (Self) ○ I/O Read Time (Self) ● Rows

Plan A	Plan B	Plan A: Rows	Plan B: Rows
-> Limit	-> Limit	100	23
-> Aggregate	-> Aggregate	1	1
-> Append	-> Append	1,440	1,440
-> Index Scan	-> Index Scan	391	36
-> Index Scan	-> Index Scan	1,049	1,404
-> Nested Loop	-> Nested Loop	31,973	56
-> Nested Loop	-> Nested Loop	31,973	244
-> Index Scan	-> Index Scan	128,992	244
-> Index Scan		0	
-> Index Scan	-> Index Scan	1	1
	-> Index Scan		0
-> Append	-> Append	35,597	32
-> Subquery Scan	-> Subquery Scan	3,551	2
-> Aggregate	-> Aggregate	3,551	2
-> CTE Scan	-> CTE Scan	31,973	56
-> Function Scan	-> Function Scan	30,499	66
-> Subquery Scan	-> Subquery Scan	9,897	0
-> Aggregate	-> Aggregate	9,897	0
-> Sort		26,504	
-> Nested Loop		26,504	
-> CTE Scan		31,973	
-> Index Scan		1	
	-> Result		0
-> Subquery Scan	-> Subquery Scan	0	0
-> Aggregate	-> Aggregate	0	0
-> Result	-> Result	0	0
-> Subquery Scan	-> Subquery Scan	15,976	19

Different Join Order

```
CTE fingerprints
->  Nested Loop  (cost=1.84..1140.04 rows=1 width=45) (actual time=0.166..428.961 rows=31973 loops=1)
      ->  Nested Loop  (cost=1.27..1137.25 rows=1 width=16) (actual time=0.157..349.766 rows=31973 loops=1)
            ->  Index Scan using index_query_table_associations_on_database_id_and_table_name on public.query_table_associations qta  (cost=0.70..327.43 rows=290 width=8) (actual time=0.022..64.070 rows=128992 loops=1)
            ->  Index Scan using index_query_occurrences_on_query_id on public.query_occurrences o  (cost=0.57..2.79 rows=1 width=8) (actual time=0.002..0.002 rows=0 loops=128992)
      ->  Index Scan using query_fingerprints_query_id_idx on public.query_fingerprints qf  (cost=0.57..2.77 rows=1 width=45) (actual time=0.002..0.002 rows=1 loops=31973)
```

```
CTE fingerprints
->  Nested Loop  (cost=1.84..8.14 rows=1 width=45) (actual time=0.058..2.619 rows=56 loops=1)
      ->  Nested Loop  (cost=1.27..7.52 rows=1 width=53) (actual time=0.032..1.473 rows=244 loops=1)
            ->  Index Scan using index_query_table_associations_on_database_id_and_table_name on public.query_table_associations qta  (cost=0.70..4.72 rows=1 width=8) (actual time=0.021..0.288 rows=244 loops=1)
            ->  Index Scan using query_fingerprints_query_id_idx on public.query_fingerprints qf  (cost=0.57..2.79 rows=1 width=45) (actual time=0.004..0.004 rows=1 loops=244)
      ->  Index Scan using index_query_occurrences_on_query_id on public.query_occurrences o  (cost=0.57..0.61 rows=1 width=8) (actual time=0.004..0.004 rows=0 loops=244)
```

pganalyze

Use query variants to test hypothesis

Name (Optional)

Try different join order

Baseline Query

```
/*controller:graphql,action:graphql,line:/app/services/dataload/queries/query_stats_for_table.rb:181:in `query_stats_for_tabl
e',sentry_trace_id:11c0590f5359469fbc1dd94a99fbe18d,traceparent:00-c196797a...
```

Show full query text

Variant Query

```
/*+ Leading((query_table_associations query_occurrences) query_fingerprints) */

WITH total_times AS (
  SELECT SUM(query_stats_blk_read_time_sum + query_stats_blk_write_time_sum) AS total_iotime,
         SUM(query_stats_total_time_sum) AS total_runtime
    FROM query_overview_stats_35d qos
   WHERE qos.database_id = $database_id AND qos.collected_at BETWEEN $collected_at_3 AND $collected_at_4
),
fingerprints AS (
  SELECT qf.*
```

Cancel   Check Query

pganalyze

Use query variants to test hypothesis

Query Plans

Filter by Parameter Set...

VARIANT	PLAN	PARAMETER SET	EST. COST	RUNTIME
Baseline	a339f88	Param Set 1	20,436	35.57ms
Baseline	a359a9c	Param Set 2	21,918	⚠1,126.47ms
Baseline	a3909ef	Param Set 3	21,892	⚠3,512.68ms
Re-run with warm cache	a339f88	Param Set 1	20,436	28.63ms
Re-run with warm cache	a359a9c	Param Set 2	21,918	⚡5.29ms
Re-run with warm cache	a3909ef	Param Set 3	21,892	36.75ms
Always use min_occurred_at …	a339f88	Param Set 1	20,436	29.80ms
Always use min_occurred_at …	a3da688	Param Set 2	40,355	376.17ms
Always use min_occurred_at …	a3d2c88	Param Set 3	40,413	122.17ms

pganalyze

thanks!

Get a free trial of pganalyze

PGANALYZE.COM

Get free pganalyze eBooks and Postgres blog posts

PGANALYZE.COM/RESOURCES

PGANALYZE.COM/BLOG

PGANALYZE.COM/NEWSLETTER

pganalyze