# Best Practices for Tuning Slow Postgres Queries

pganalyze

lukas@pganalyze.com

1. Debugging why a query is slow
2. Benchmarking with EXPLAIN (ANALYZE, BUFFERS)
3. Planner costing, and why it can never be perfect
4. JOIN order and parameterized index scans
5. Guiding the planner to the right plan

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
5,833.54ms

I/O Write Time
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

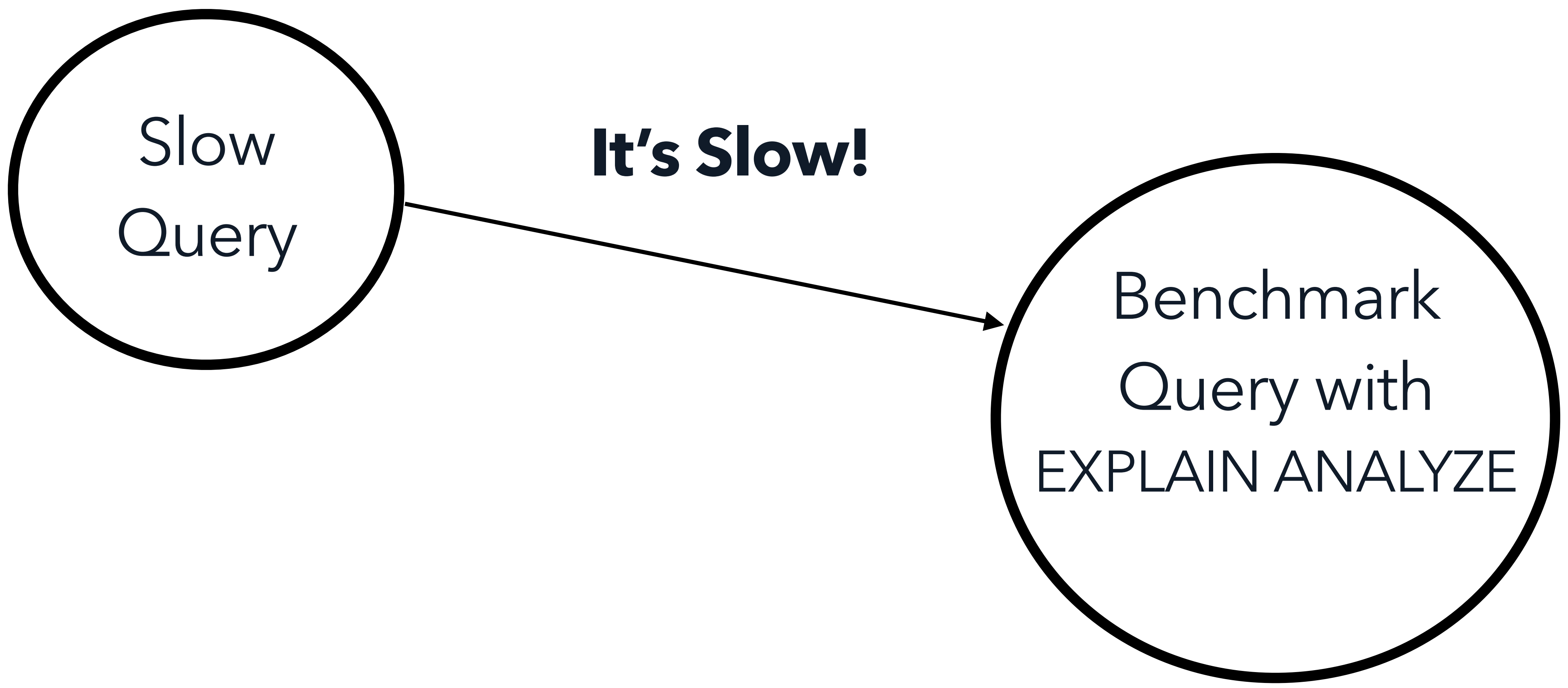
Info Query **#43899555** takes 1287 ms on average (88397 ms max, 3.35 MB read from disk per call, 13390 calls in last 24h)

EXPLAINs

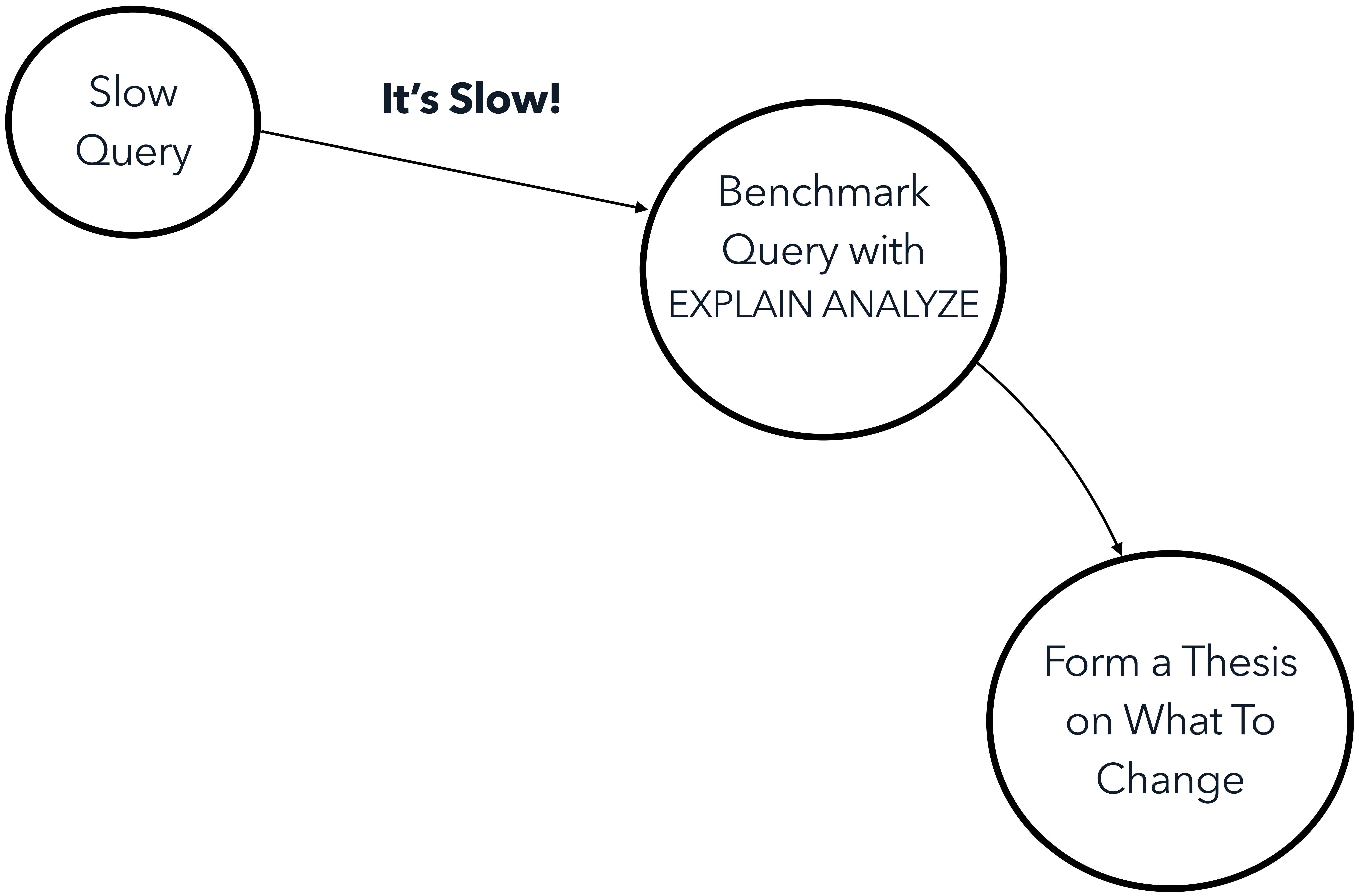
| EXECUTED AT | PLAN | EST. COST | RUNTIME | I/O READ TIME | | READ FROM DISK | PLAN NODES |
|---|---|---|---|---|---|---|---|
| 2024-10-01 08:14:23pm PDT | a332ead | 348 | 14,688.59ms | 12,796.35ms | 87% | 42.6 MB | Sort · Nested Loop · CTE Scan +4 more |
| 2024-10-01 08:03:23pm PDT | a332ead | 348 | 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 |

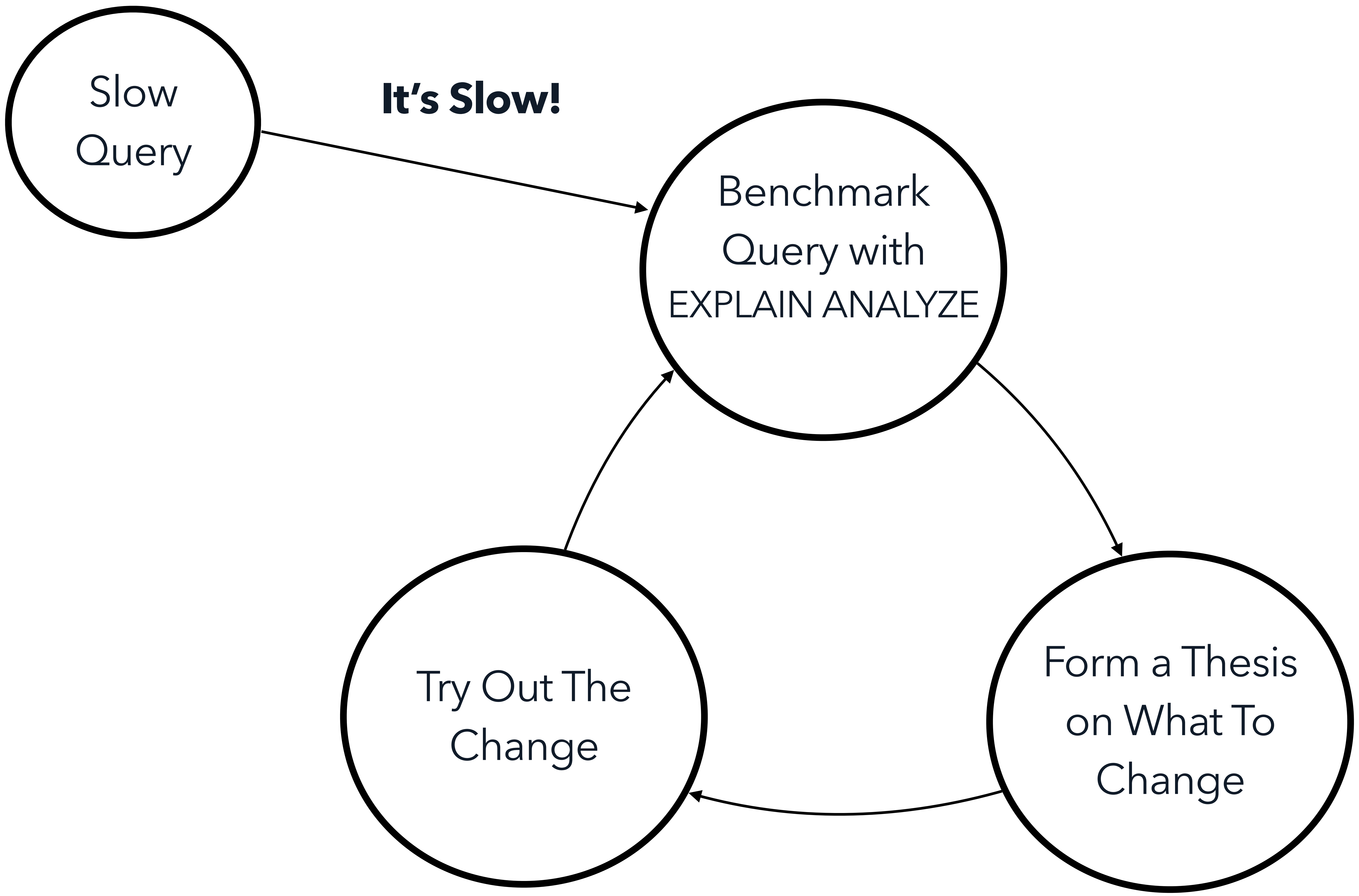
# Plan Fingerprints show changes in plan structure

pganalyze

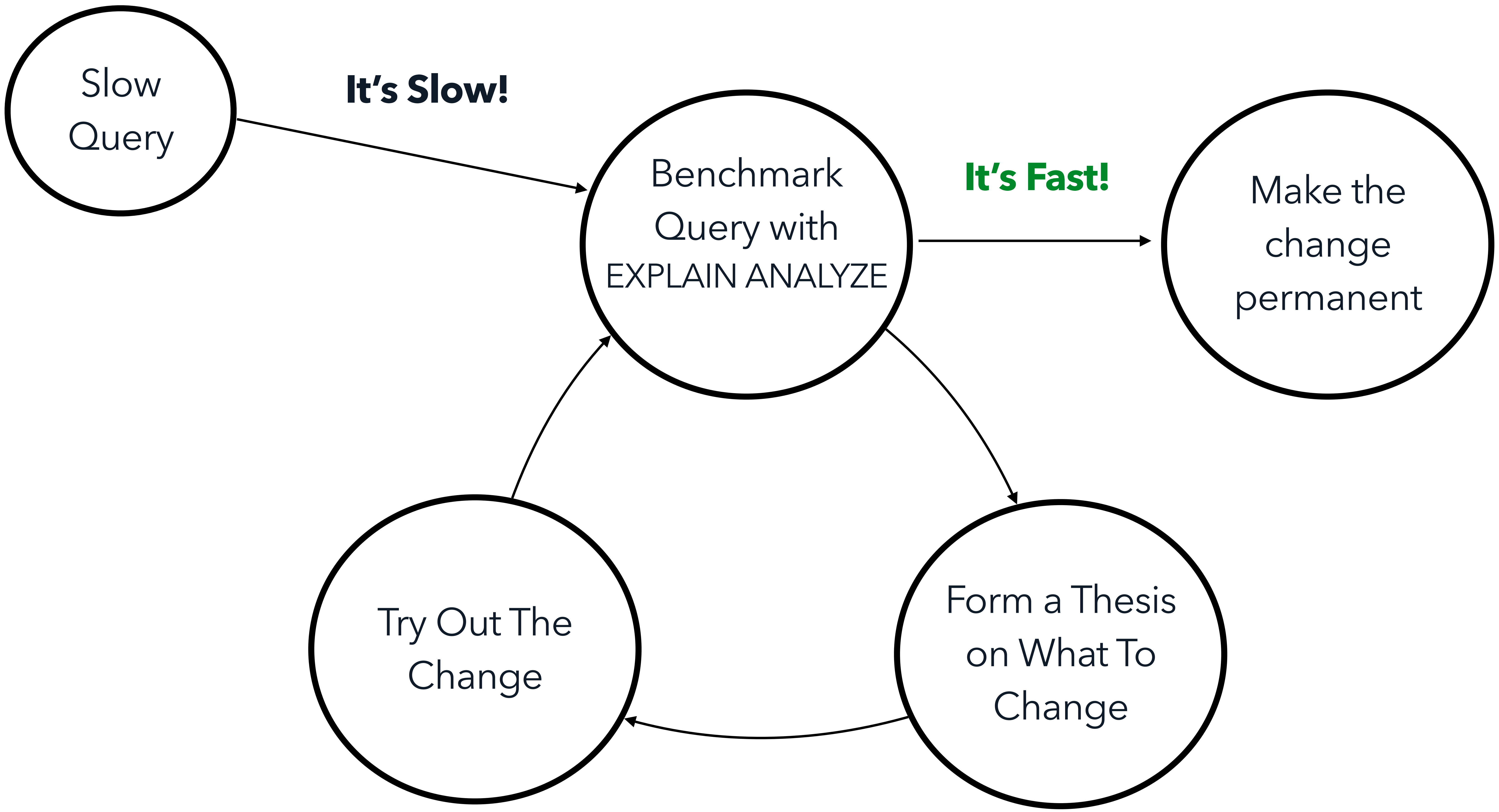



pganalyze


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


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

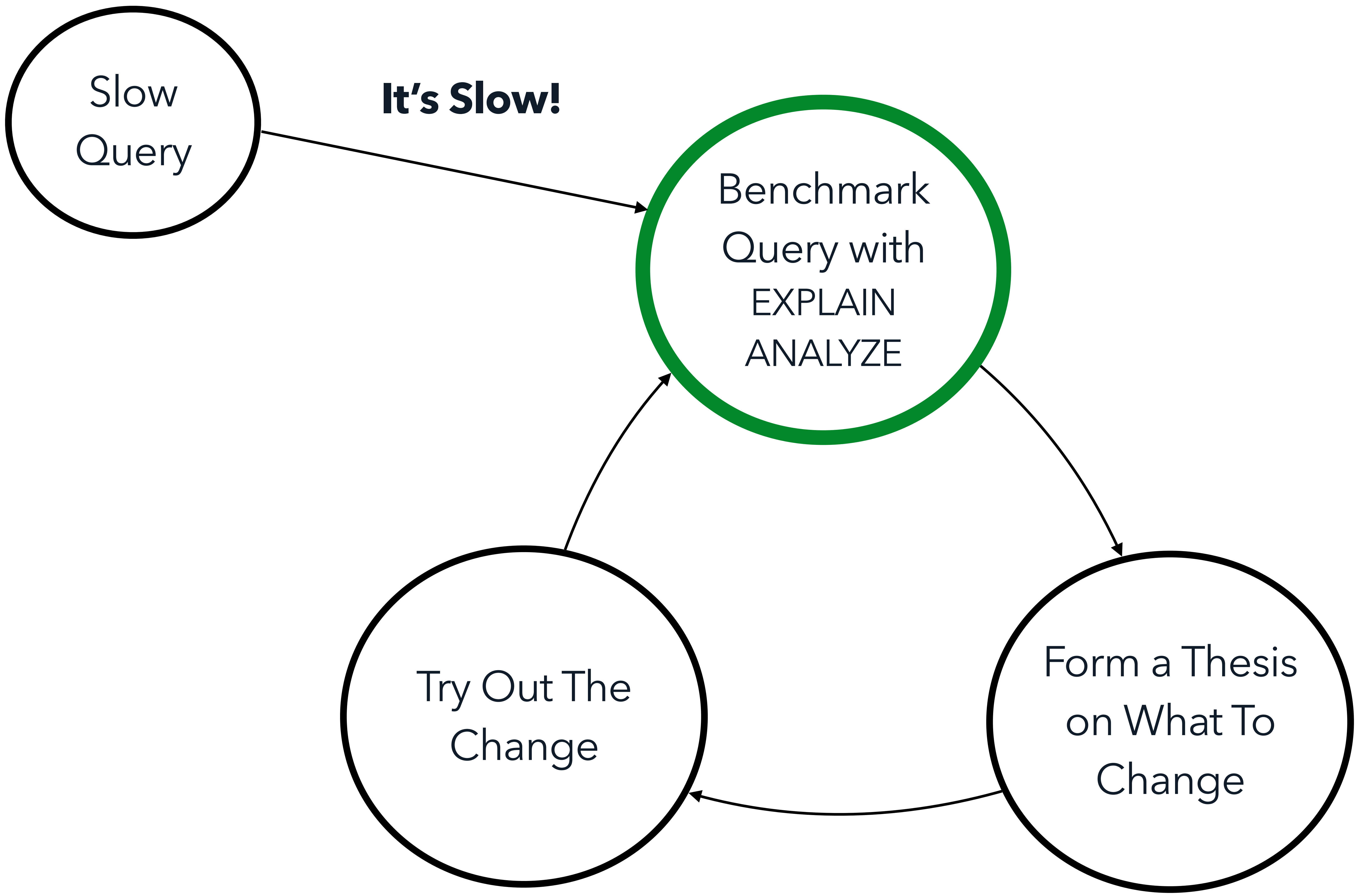
pganalyze


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

pganalyze

```
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=
   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

# Dead rows and bloat greatly influence buffer counts

```
test=# explain (analyze, buffers) select * from t2 where num > 10000 order by num limit 1000;
                                                            QUERY PLAN
-------------------------------------------------------------------------------------------------
Limit  (cost=0.43..24.79 rows=1000 width=16) (actual time=0.071..0.395 rows=1000 loops=1)
   Buffers: shared hit=11
   ->  Index Scan using i_t2_num on t2  (cost=0.43..219998.90 rows=9034771 width=16)
                                        (actual time=0.068..0.273 rows=1000 loops=1)
         Index Cond: (num > 10000)
         Buffers: shared hit=11
 Planning Time: 0.183 ms
 Execution Time: 0.491 ms
(7 rows)

…

test=# explain (analyze, buffers) select * from t2 where num > 10000 order by num limit 1000;
                                                             QUERY PLAN
-------------------------------------------------------------------------------------------------
 Limit  (cost=0.43..52.28 rows=1000 width=16) (actual time=345.347..345.431 rows=1000 loops=1)
   Buffers: shared hit=50155
   ->  Index Scan using i_t2_num on t2  (cost=0.43..93372.27 rows=1800808 width=16)
                                        (actual time=345.345..345.393 rows=1000 loops=1)
         Index Cond: (num > 10000)
         Buffers: shared hit=50155
 Planning Time: 0.222 ms
 Execution Time: 345.481 ms
(7 rows)
```

from [Nikolay Samokhvalov - EXPLAIN (ANALYZE) needs BUFFERS to improve the Postgres query optimization process]

pganalyze

# Be careful with hit counters in loops!

```
Nested Loop
    Sequential Scan
        shared hit: 12
    Index Scan (loops=100)
        shared hit: 1200
```

**This does not mean 1200 buffers were loaded.**

It could have been as little as 12 buffers, if each loop iteration looke same part of the index.

pganalyze

# EXPLAIN ANALYZE has overhead

**It's low when you have few rows:**

```
\timing
SELECT * FROM test LIMIT 1;
Time: 2.401 ms

EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM test LIMIT 1;
Execution Time: 0.087 ms

EXPLAIN (ANALYZE, BUFFERS, TIMING OFF) SELECT * FROM test LIMIT 1;
Execution Time: 0.117 ms

EXPLAIN (ANALYZE, BUFFERS, TIMING ON) SELECT * FROM test LIMIT 1;
Execution Time: 0.069 ms
```

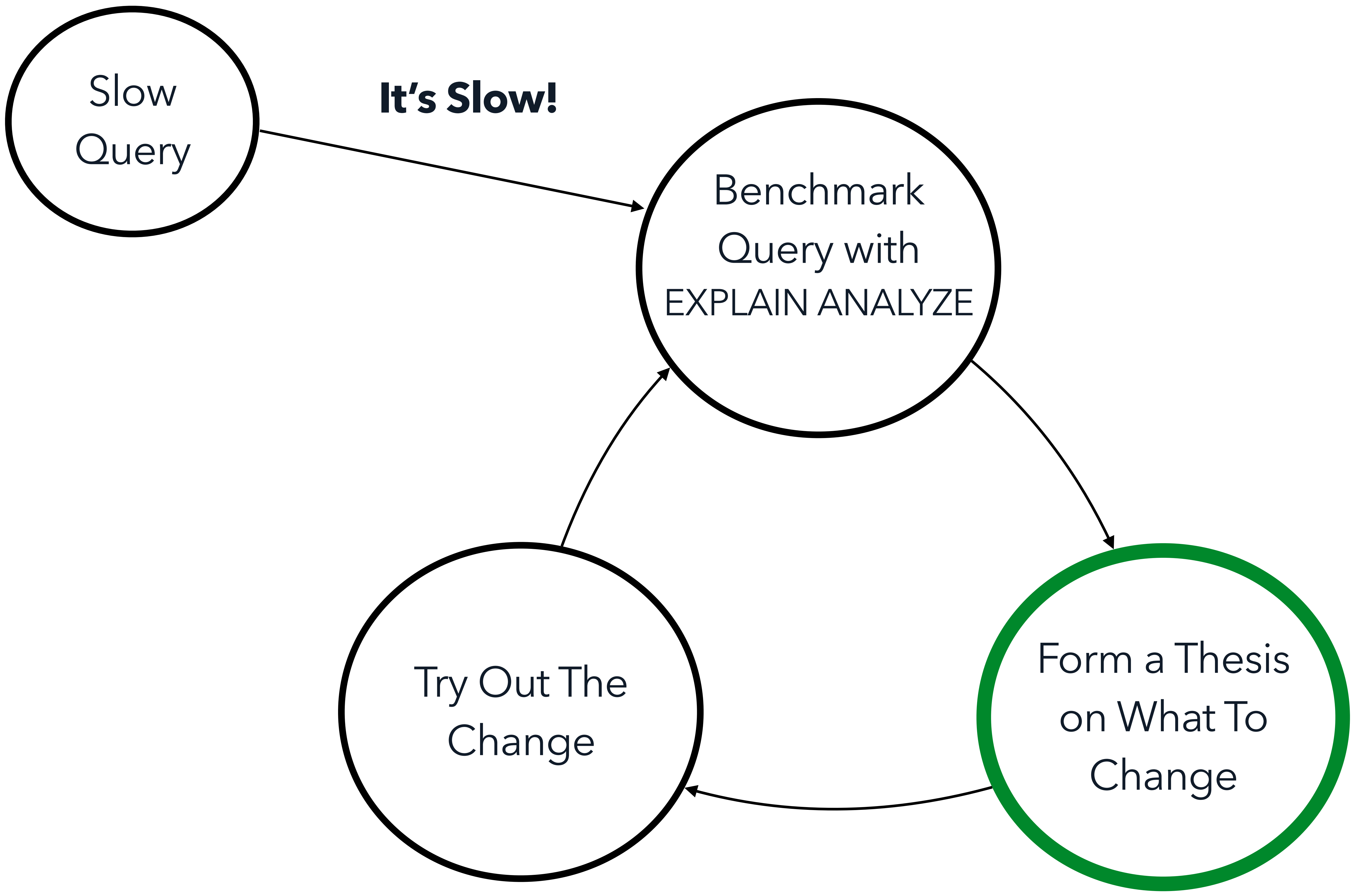
**And high when you have many rows:**

```
\timing
SELECT COUNT(*) FROM test;
Time: 342.119 ms

EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM test;
Execution Time: 404.309 ms

EXPLAIN (ANALYZE, BUFFERS, TIMING OFF) SELECT COUNT(*) FROM test;
Execution Time: 441.049 ms

EXPLAIN (ANALYZE, BUFFERS, TIMING ON) SELECT COUNT(*) FROM test;
Execution Time: 972.943 ms
```

```
SELECT COUNT(*) FROM test;
  count
----------
 10000001
(1 row)
```

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*

pganalyze

**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

```
->  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

# 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;
}
```

# What is the cost of an Index Scan?

pganalyze

**src/backend/optimizer/path/costsize.c**

```
/*
* 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);
```

pganalyze

**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

## 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, Specia
*sjinfo)
{
…
}
```

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

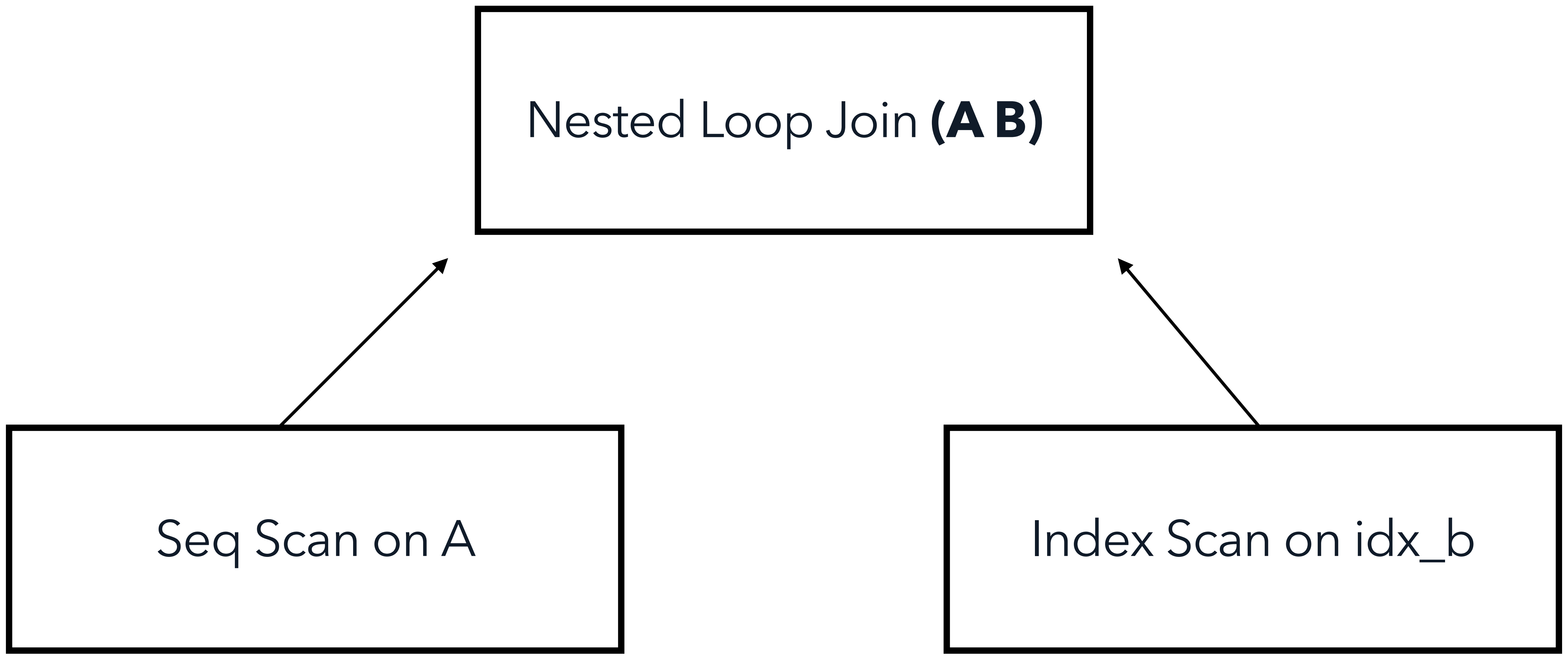
```
/*
 * eqjoinsel_inner --- eqjoinsel for normal inner join
 *
 * We also use this for LEFT/FULL outer joins; it's not presently clear
 * that it's worth trying to distinguish them here.
 */
static double
eqjoinsel_inner(…)
{
    double      selec;

    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.
         * …
         */
```

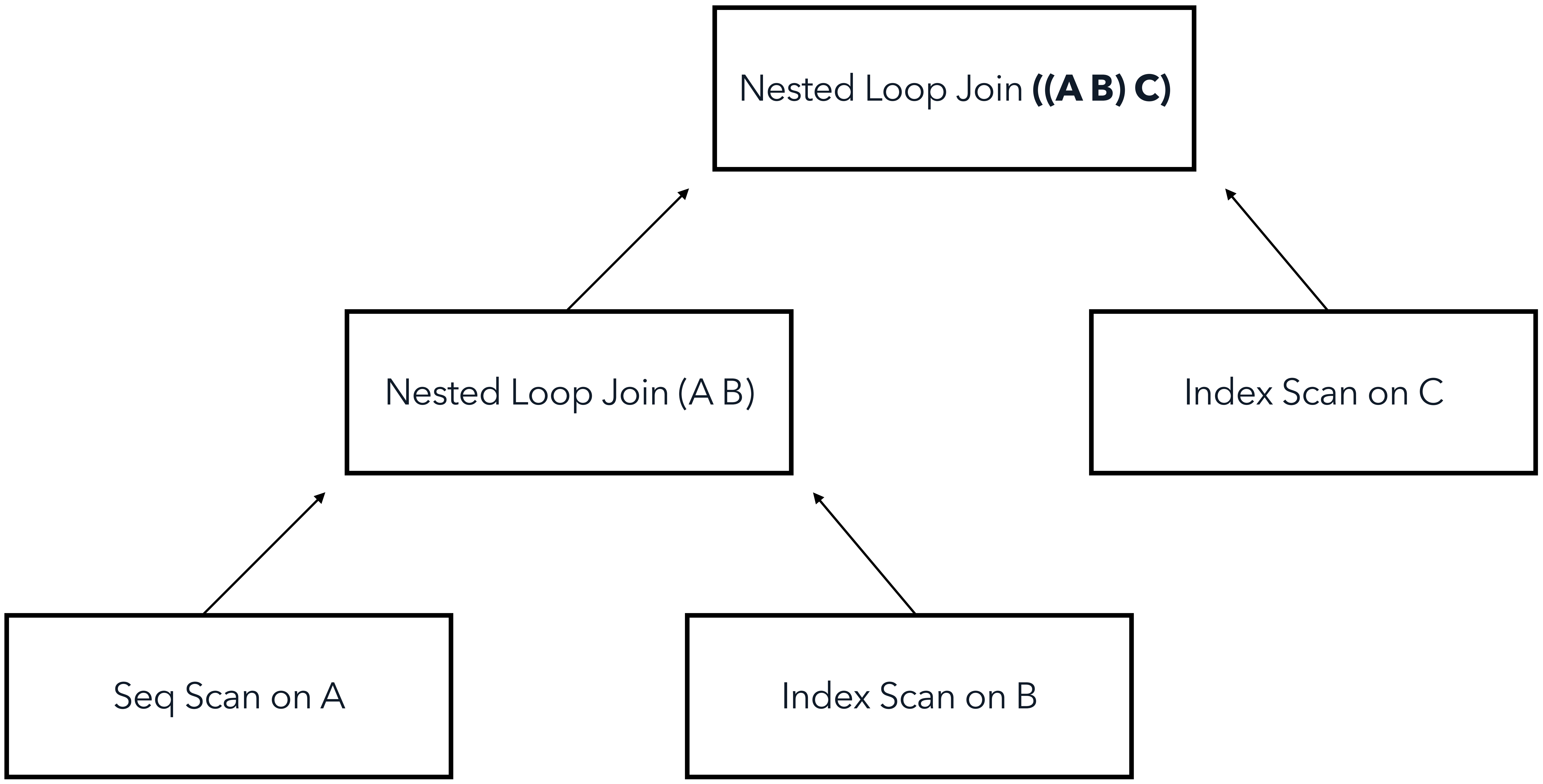
To improve join selectivity (in some cases),
**increase the both table column’s statistics targets**,
to collect more **MCVs**

pganalyze

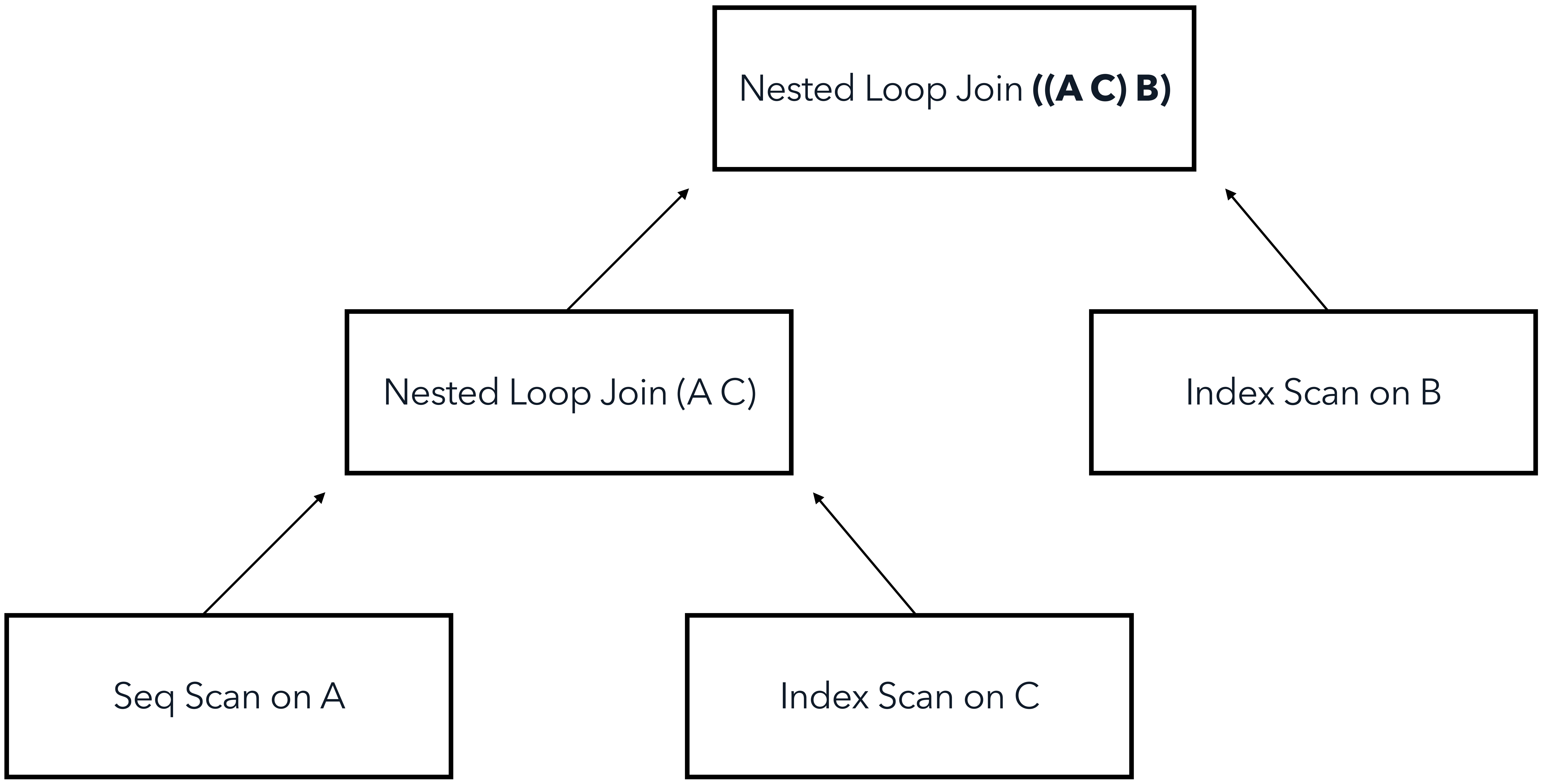
pganalyze

# JOIN order and parameterized Index Scans


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


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

pganalyze

((A B) C)

**= Join Order**

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

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

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

pganalyze

# You can detect Join Order in captured EXPLAINs:

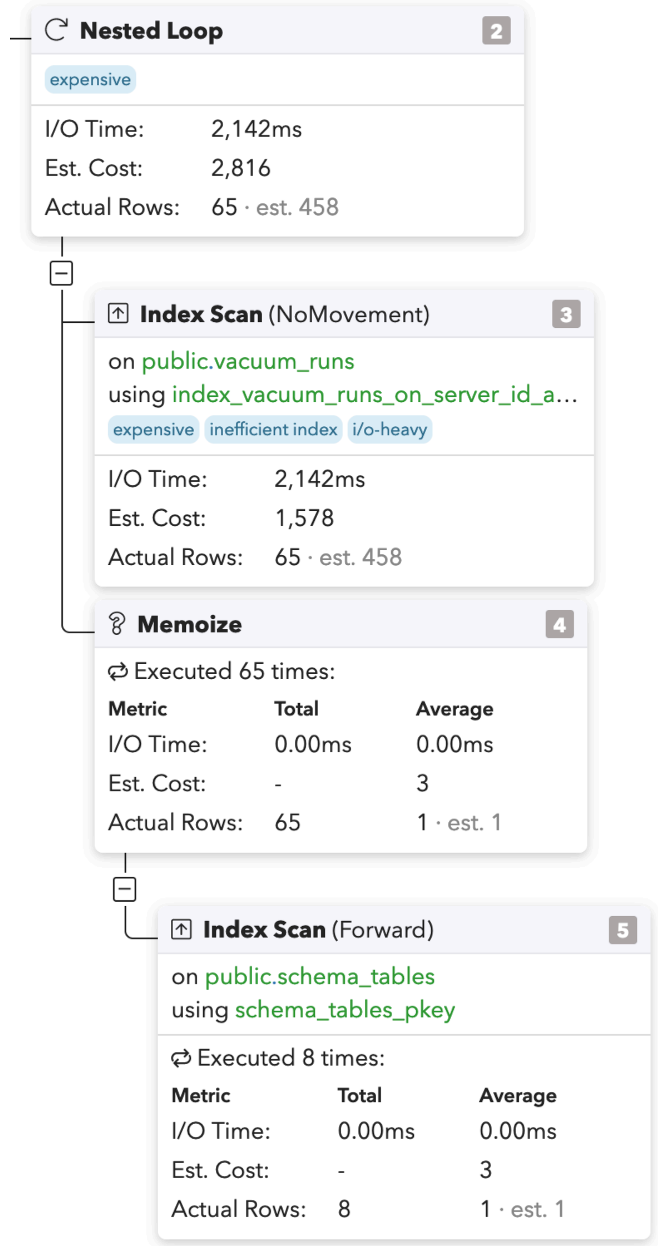
## EXPLAINs

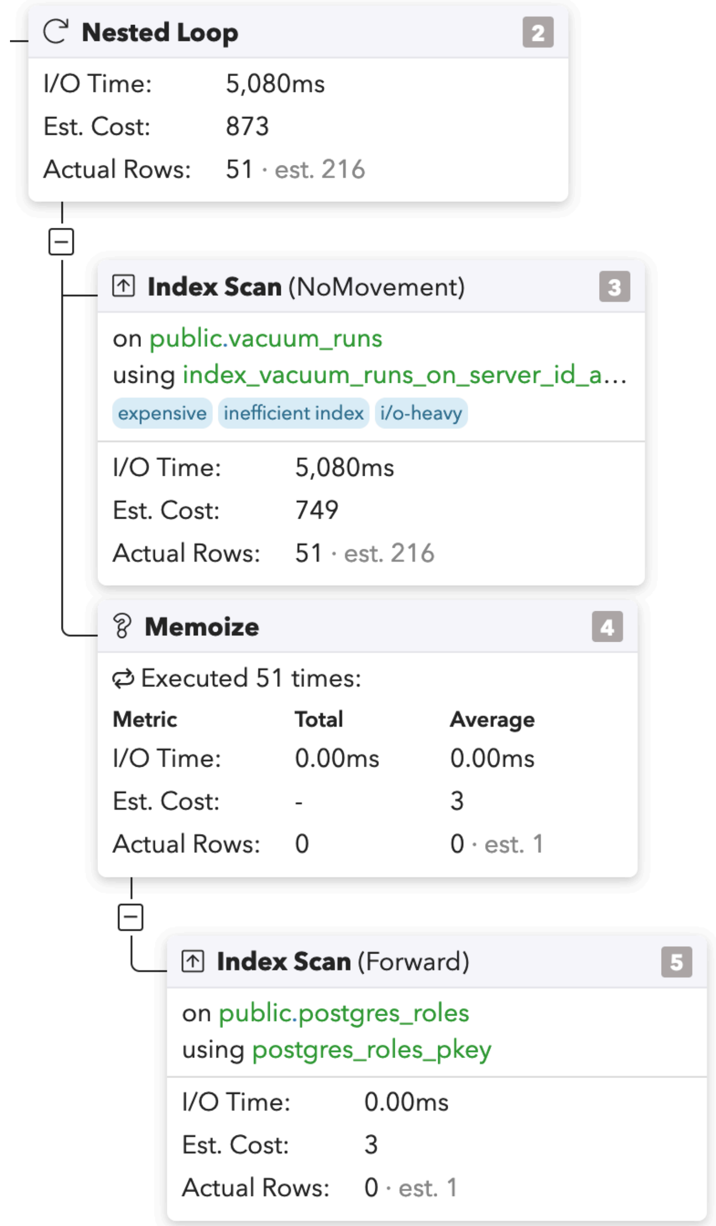
**Join Orders:**

- ((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)")

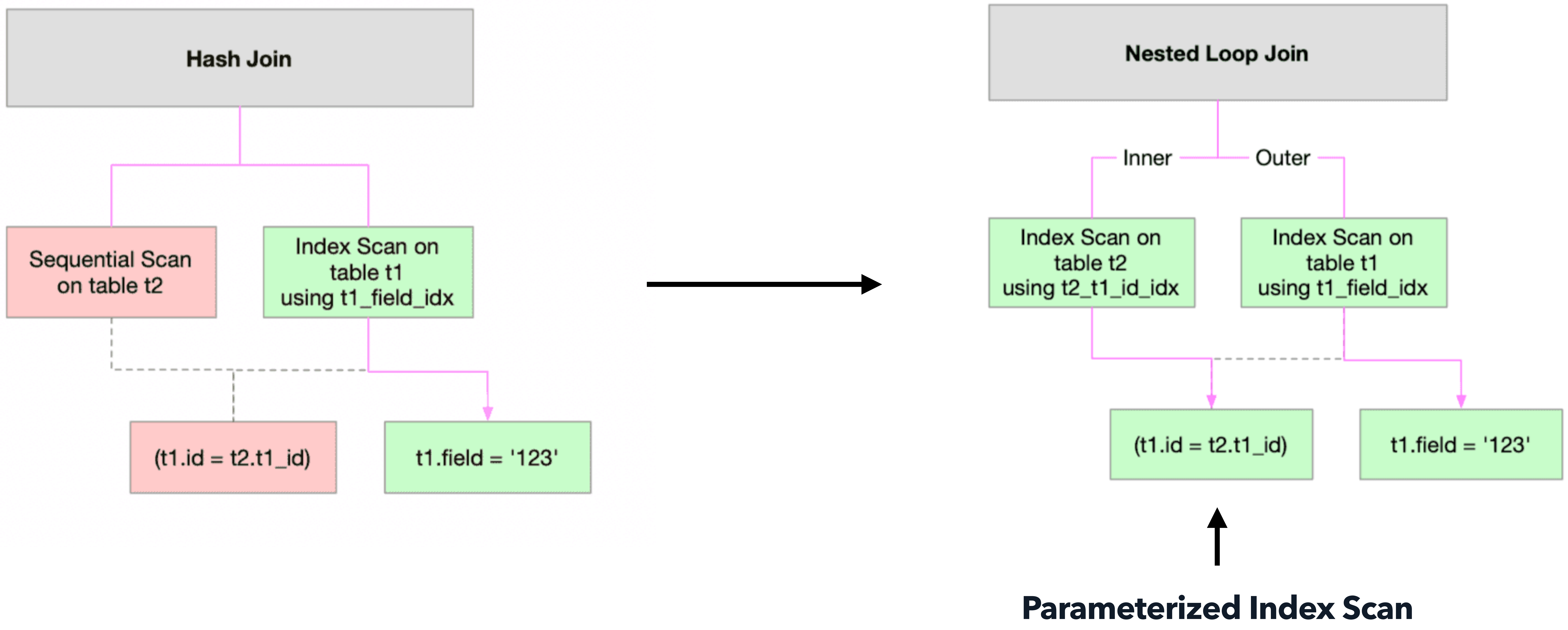
=> (1) and (2) are always eligible for an Index Scan.

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

pganalyze

```
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


Hash Join
Sequential Scan on table t2
Index Scan on table t1 using t1_field_idx
(t1.id = t2.t1_id)
t1.field = '123'
Nested Loop Join
Inner
Outer
Index Scan on table t2 using t2_t1_id_idx
Index Scan on table t1 using t1_field_idx
(t1.id = t2.t1_id)
t1.field = '123'
Parameterized Index Scan

pganalyze

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

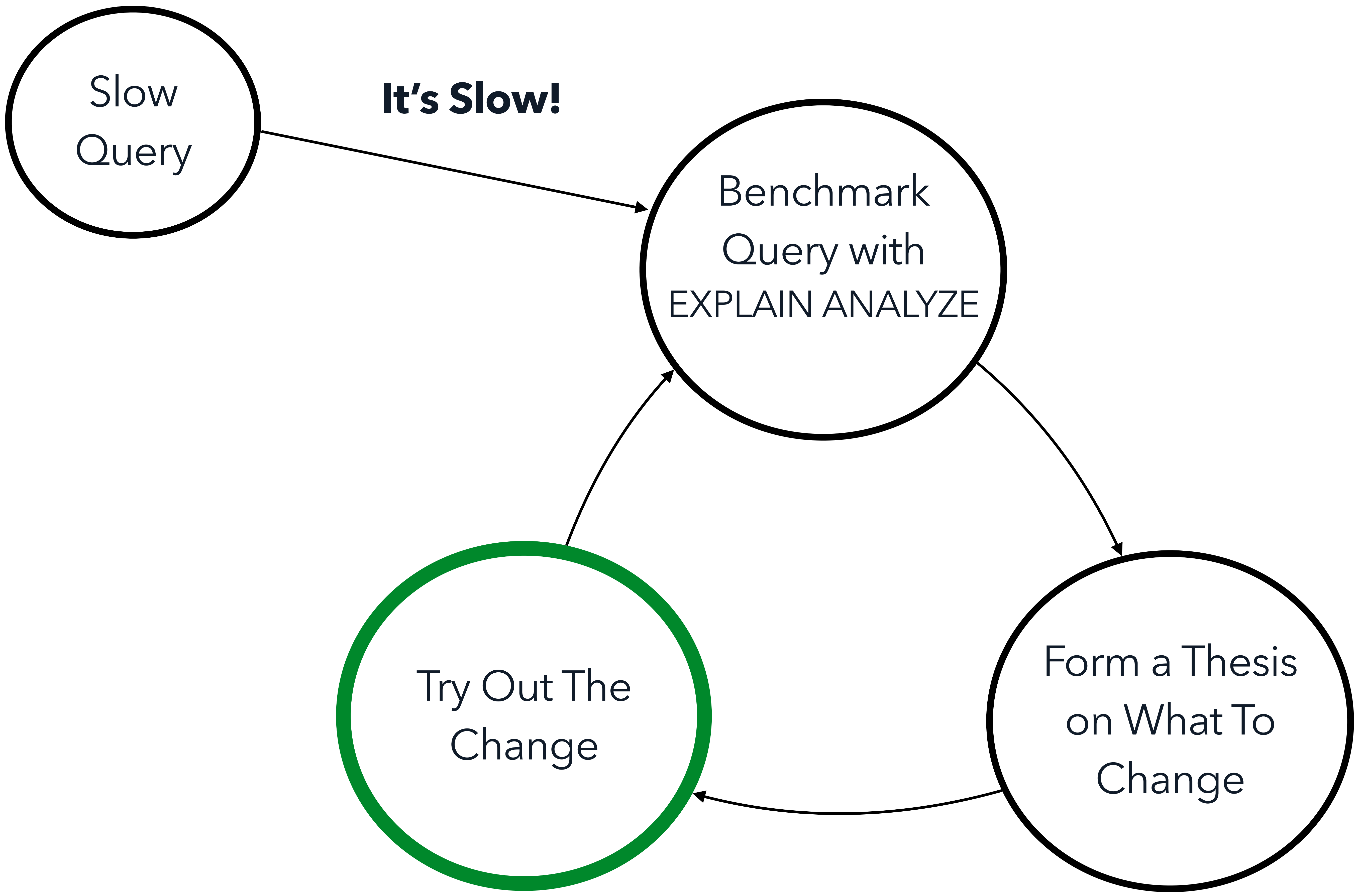
(Join order matters!)

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

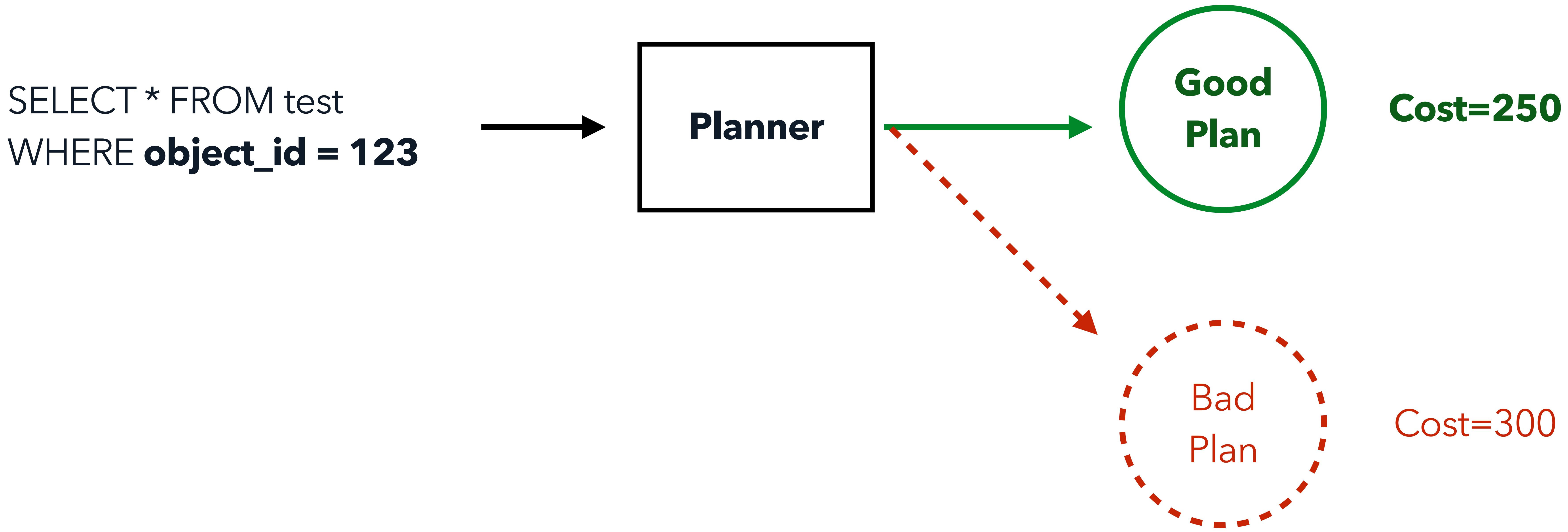
To Understand

**Why A “Bad” Plan Was Chosen**

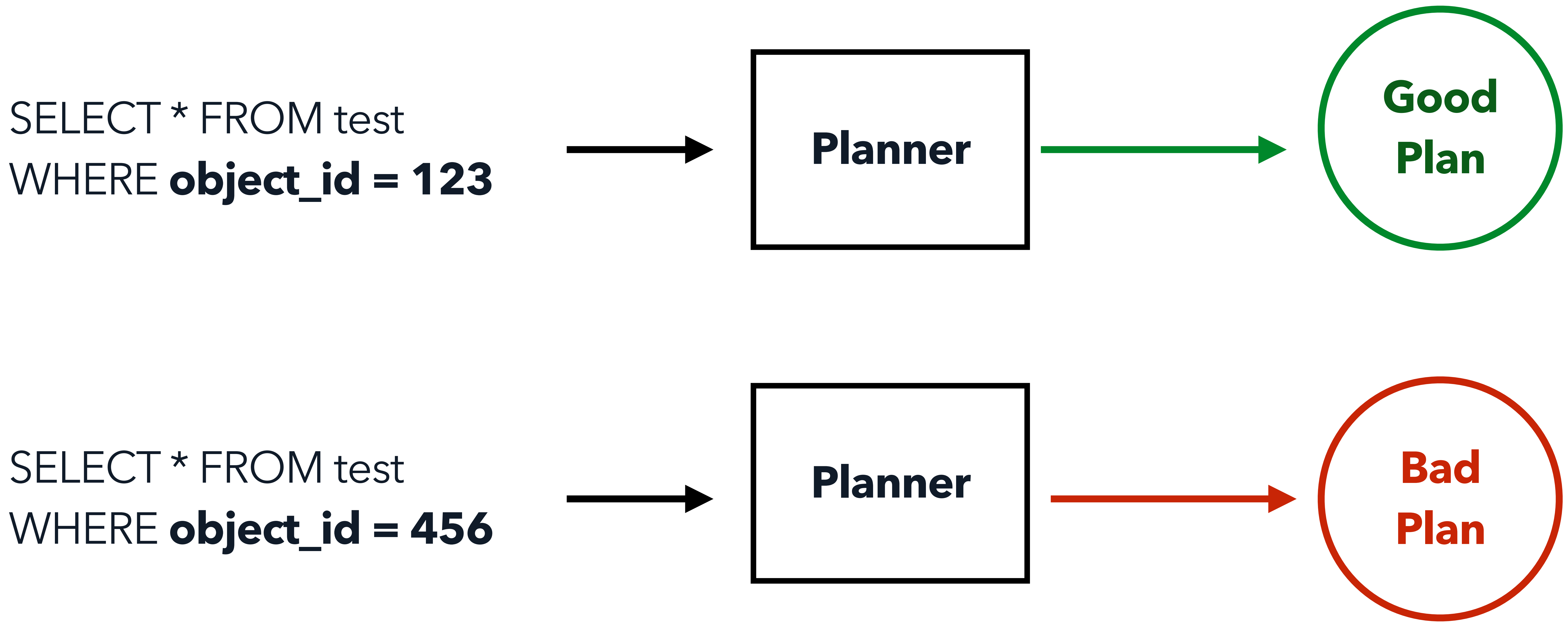
Start By Forcing The Good Plan

pganalyze

SELECT * FROM test
WHERE **object_id = 123**

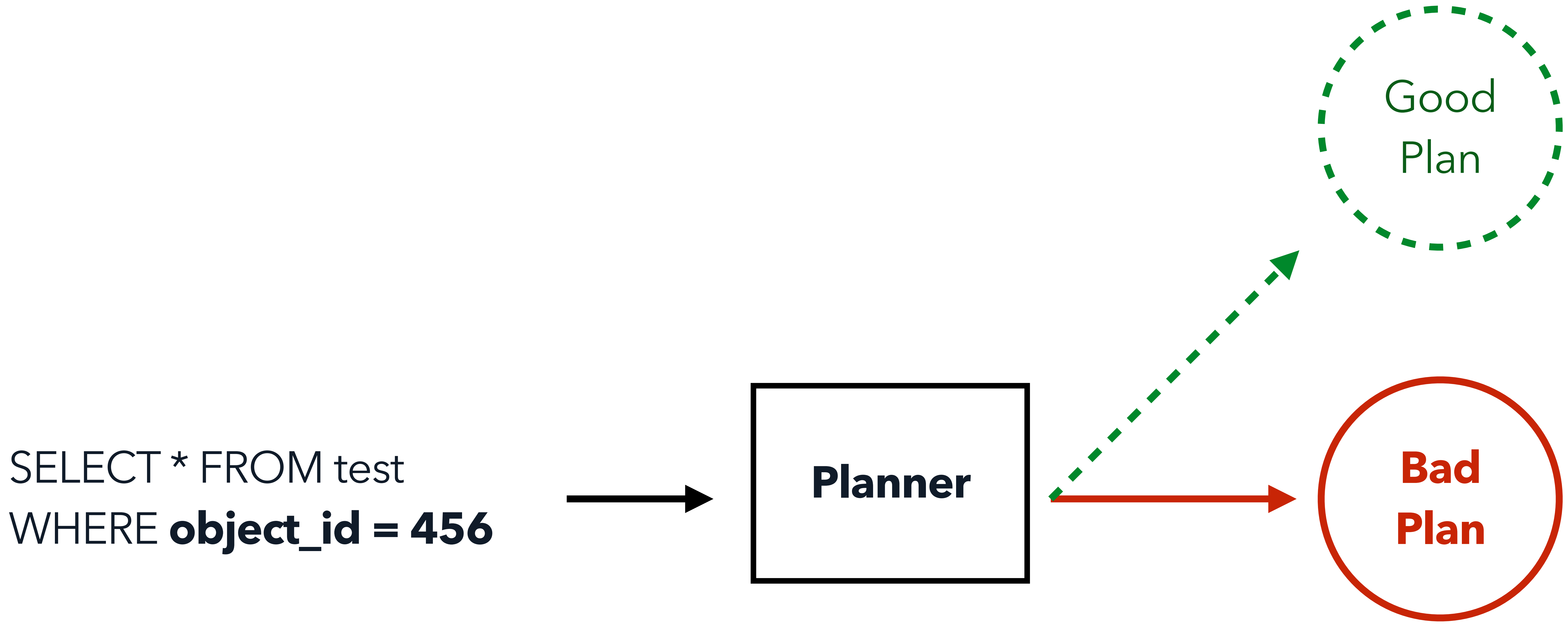

SELECT * FROM test
WHERE **object_id = 123**
Planner
Good Plan
Cost=250
Bad Plan
Cost=300

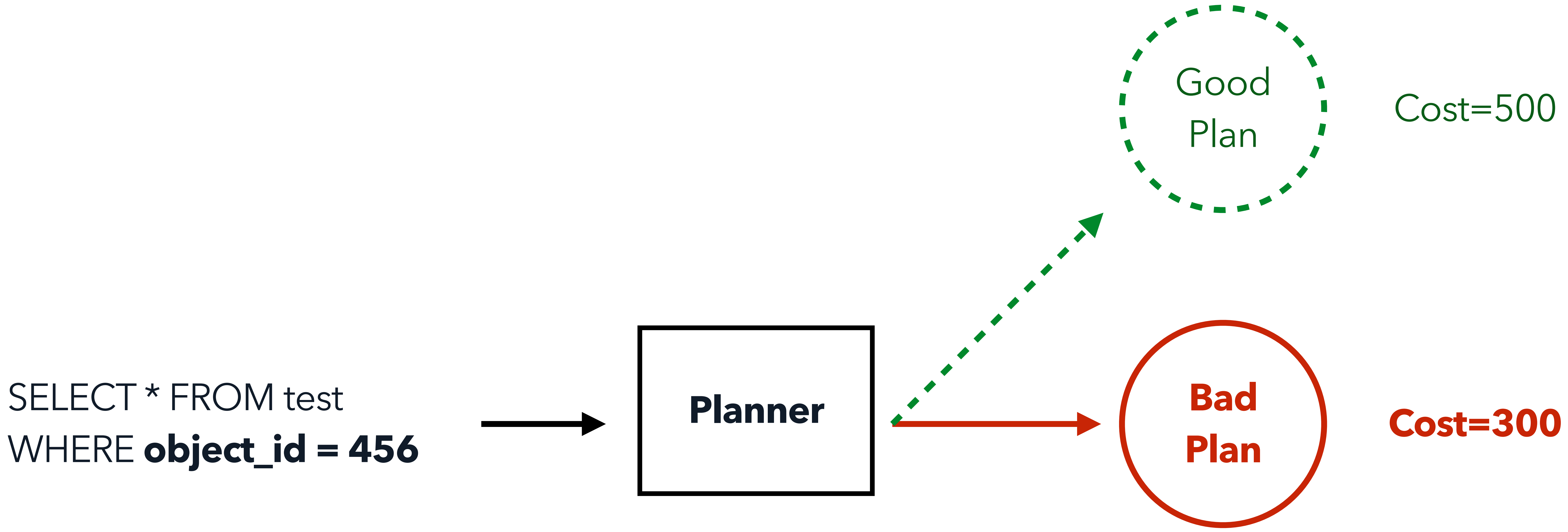
pganalyze



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

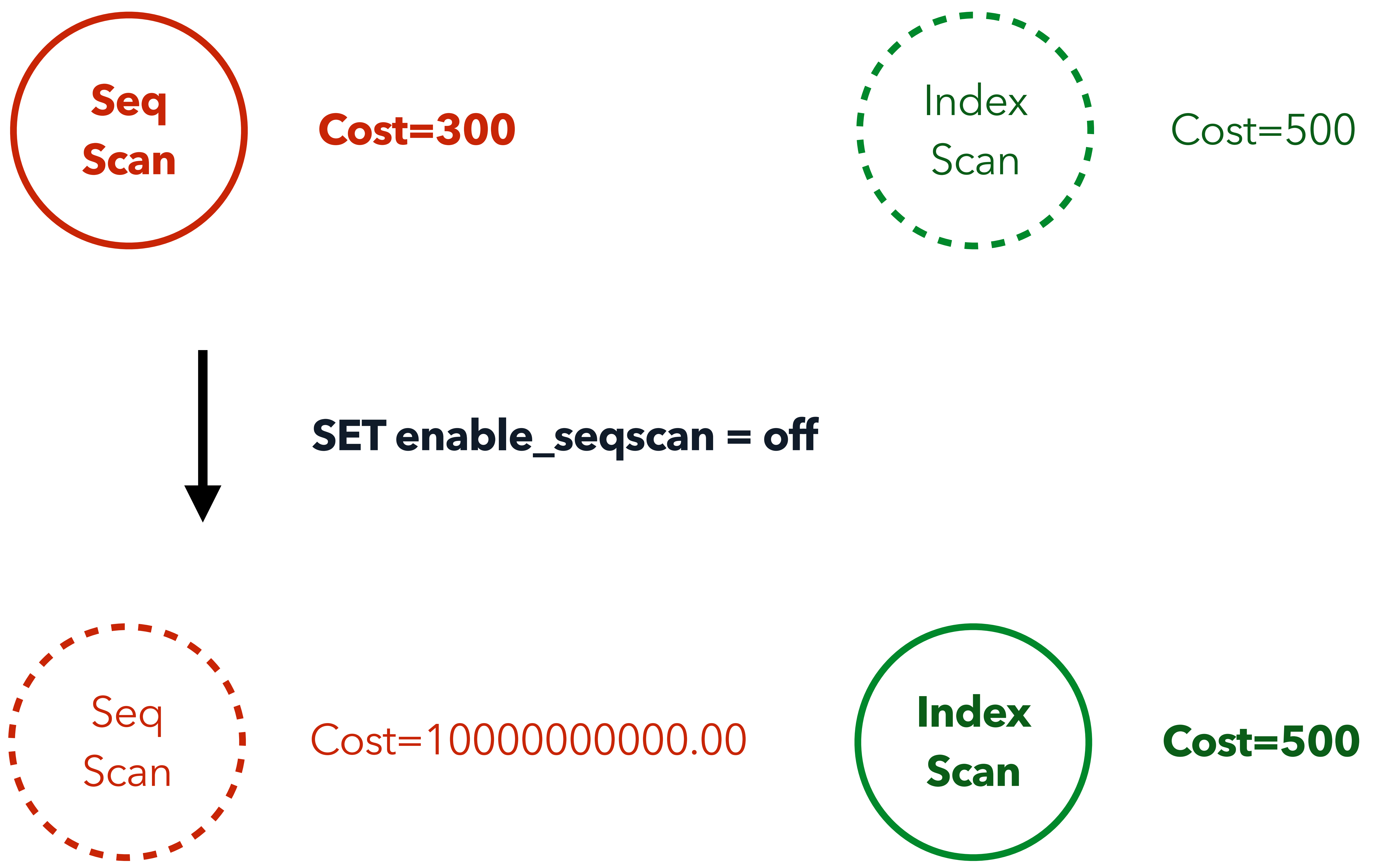

SELECT * FROM test
WHERE **object_id = 456**
Planner
Good Plan
Cost=500
Bad Plan
Cost=300

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

pganalyze

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;
```

For more complicated cases,

**Utilize pg_hint_plan to force the good plan**

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

pganalyze

```
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
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 width=0)
      ->  Index Scan using schema_tables_database_id_schema_name_table_name_idx on
        (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 width=0)
      ->  Index Only Scan using index_schema_column_stats_on_table_id on schema_co
        (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 width=0)
      ->  Index Only Scan using index_schema_column_stats_on_table_id on schema_co
          (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

If you can, choose
**Better Statistics** or
**Rewriting Queries**
over
**Planner Hints**

pganalyze

# Thank you!

Try out pganalyze:

PGANALYZE.COM

---

Reach out for any questions:

lukas@pganalyze.com

---