Using the Postgres query planner to debug bad plans and speed up queries

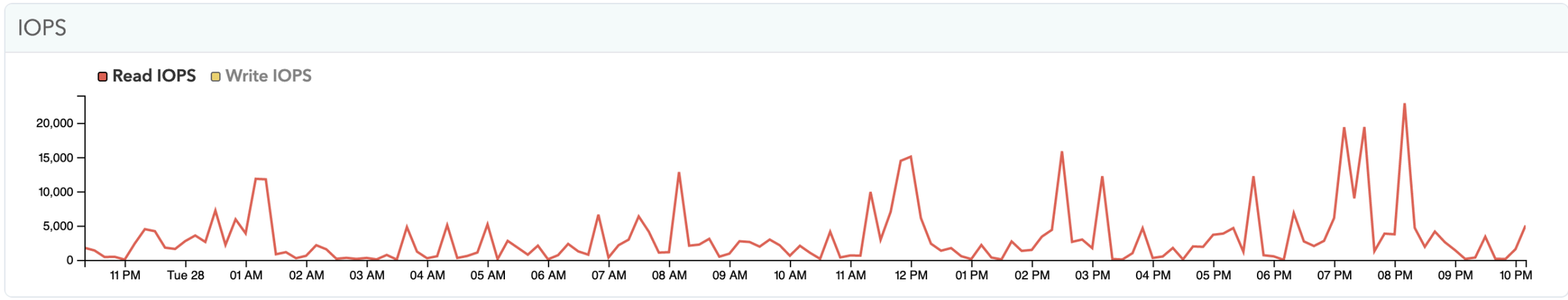
@LukasFittl
hachyderm.io/@lukas

1. Going from query to query plan
2. How the Postgres planner works
3. Cost estimation and selectivity
4. Scan/Join planning
5. Parameterized index scans
6. Bounded sorts
7. Tracking “bad” plans with auto_explain
8. Working with EXPLAIN (ANALYZE, BUFFERS)
9. Pinning plans with pg_hint_plan, or Aurora QPM
10. Guiding the planner to the right plan

1. Going from query to query plan
2. How the Postgres planner works
3. Cost estimation and selectivity
4. Scan/Join planning
5. Parameterized index scans
6. Bounded sorts
7. Tracking “bad” plans with auto_explain
8. Working with EXPLAIN (ANALYZE, BUFFERS)
9. Pinning plans with pg_hint_plan, or Aurora QPM
10. Guiding the planner to the right plan

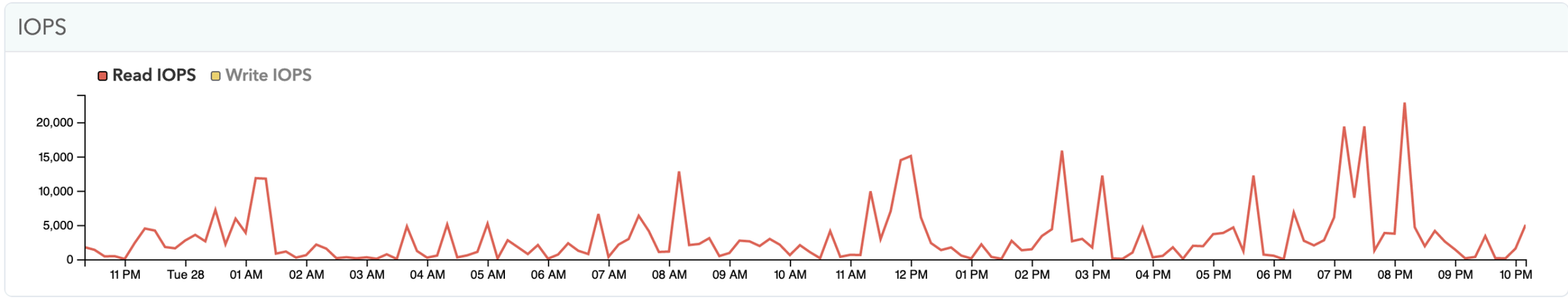
pganalyze

Going from query to query plan



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

pganalyze

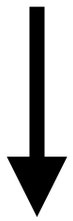


SELECT INSERT, UPDATE, DELETE DDL & other Compare to 7 days ago Search...

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)
```

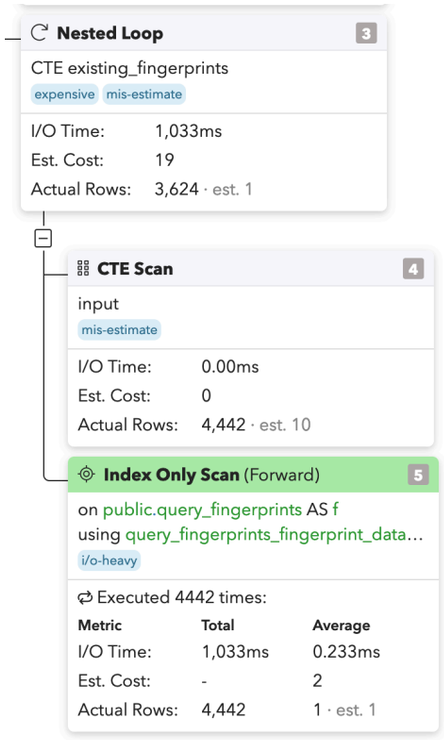


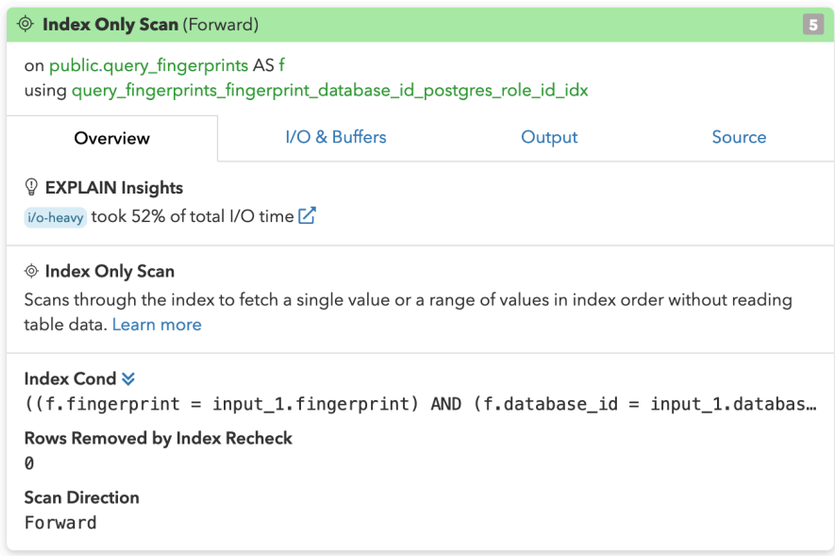
EXPLAIN

```
->  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
```

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

auto_explain + pganalyze

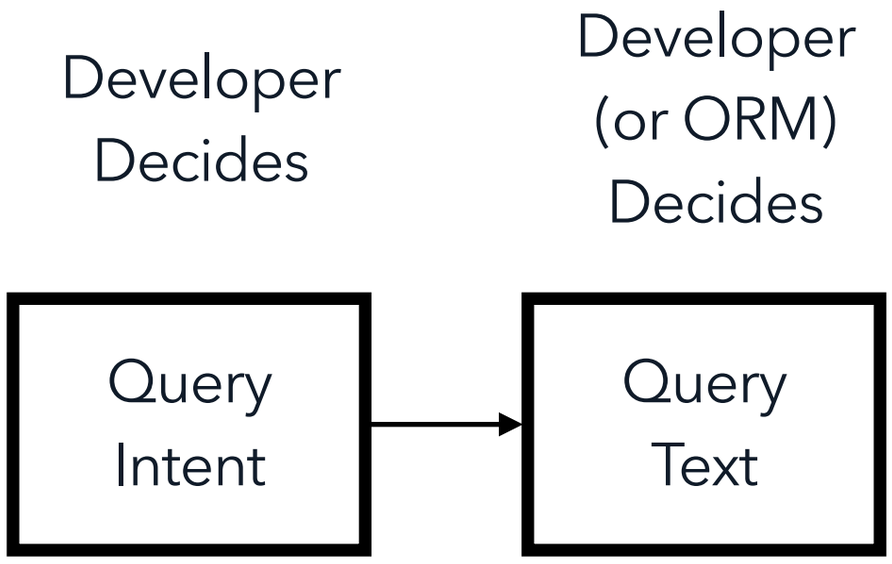




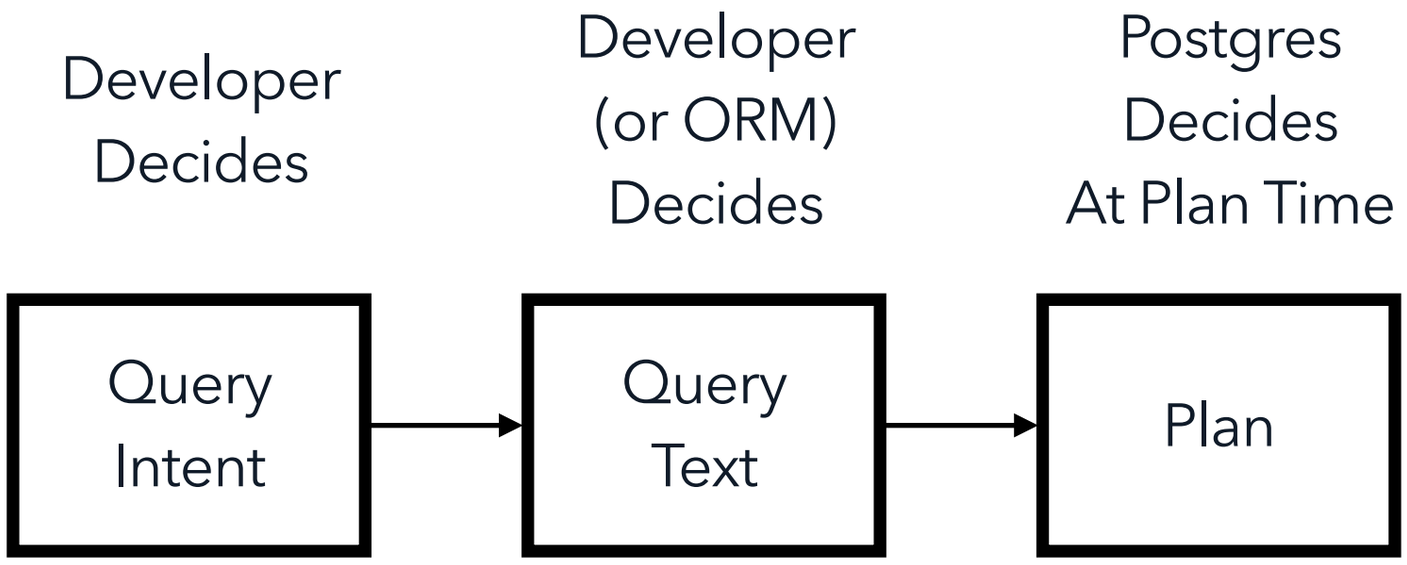
pganalyze

Developer
Decides

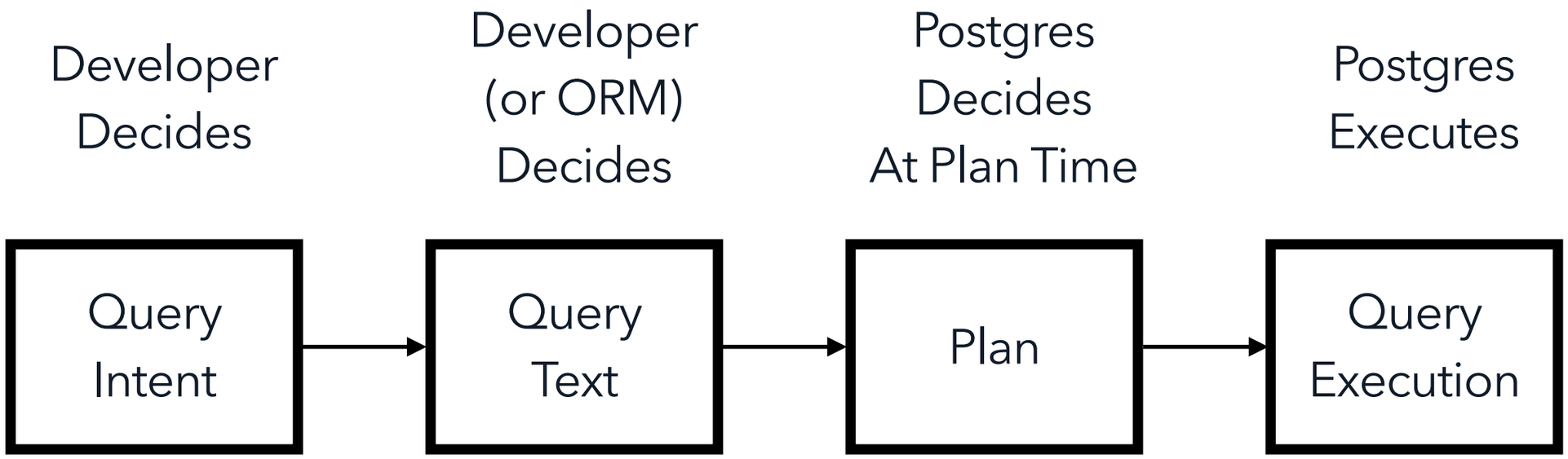
Query
Intent


Developer
Decides
Developer
(or ORM)
Decides
Query
Intent
Query
Text

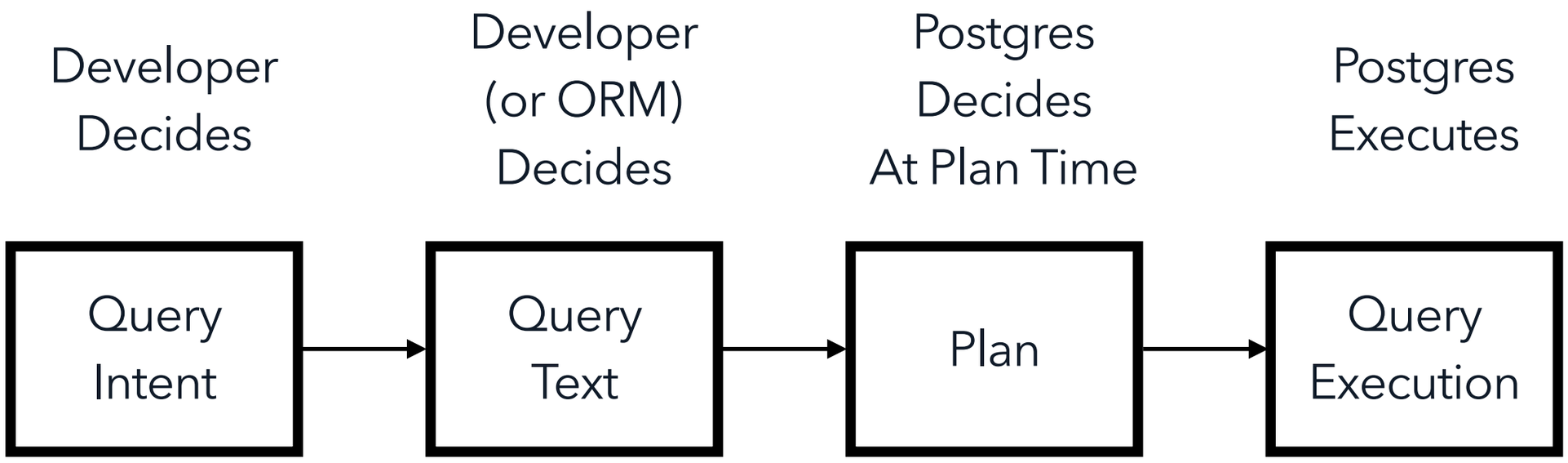
pganalyze


Developer
Decides
Developer
(or ORM)
Decides
Postgres
Decides
At Plan Time
Query
Intent
Query
Text
Plan


pganalyze


Developer
Decides
Developer
(or ORM)
Decides
Postgres
Decides
At Plan Time
Postgres
Executes
Query
Intent
Query
Text
Plan
Query
Execution

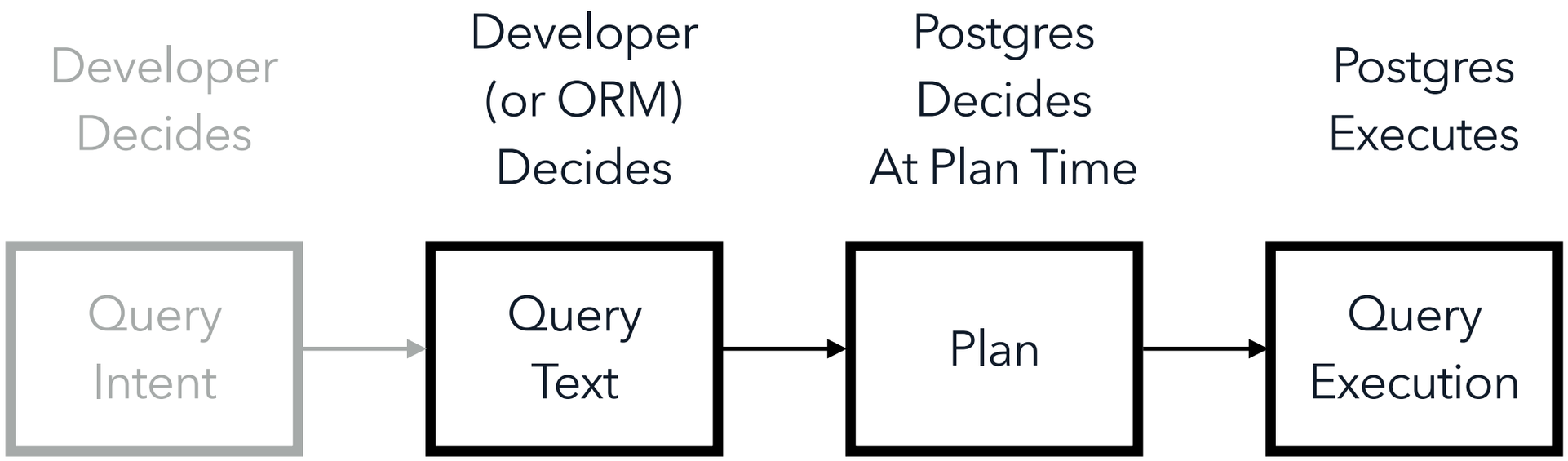
pganalyze



Change by:

- Changing data model
- Changing app logic

pganalyze



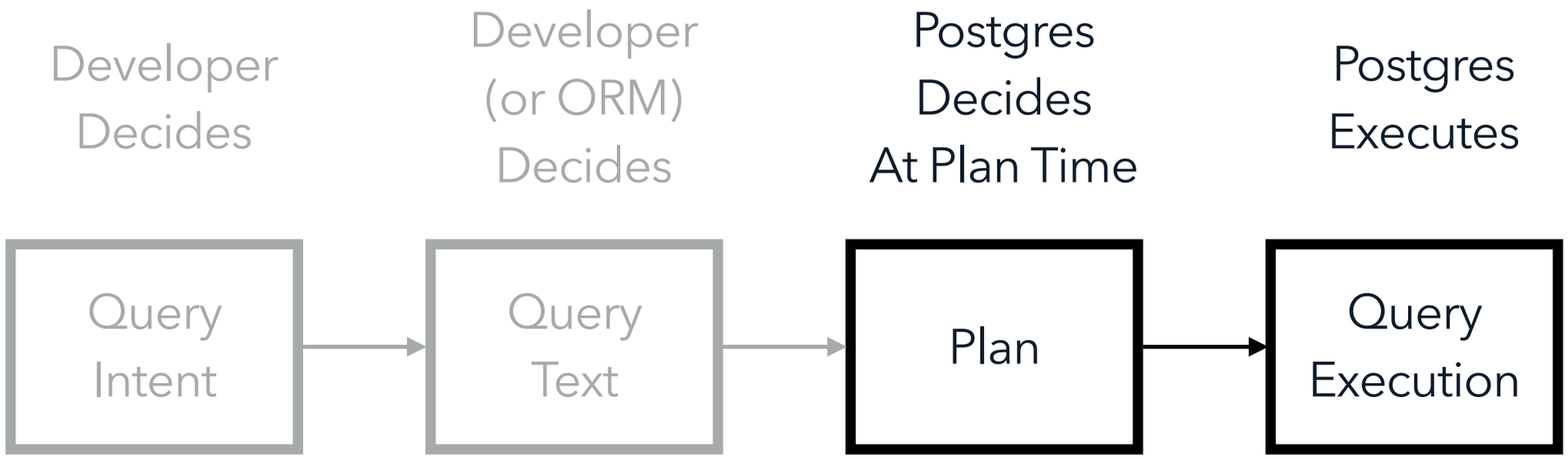
Change by:

- Changing data model
- Changing app logic

Change by:

- Rewriting query logic
- Adding more column filters

pganalyze



Change by:
- Changing data model
- Changing app logic

Change by:
- Rewriting query logic
- Adding more column filters

Change by:
- Improving statistics
- Changing settings
- Running ANALYZE or VACUUM
- Planner hints

pganalyze

Developer Decides

Query Intent

Change by:

- Changing data model
- Changing app logic

Developer (or ORM) Decides

Query Text

Change by:

- Rewriting query logic
- Adding more column filters

Postgres Decides At Plan Time

Plan

Postgres Executes

Query Execution

Change by:

- Improving statistics
- Changing settings
- Running ANALYZE or VACUUM
- Planner hints

What We'll Focus On Today!

pganalyze

pganalyze

How the Postgres planner works

“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

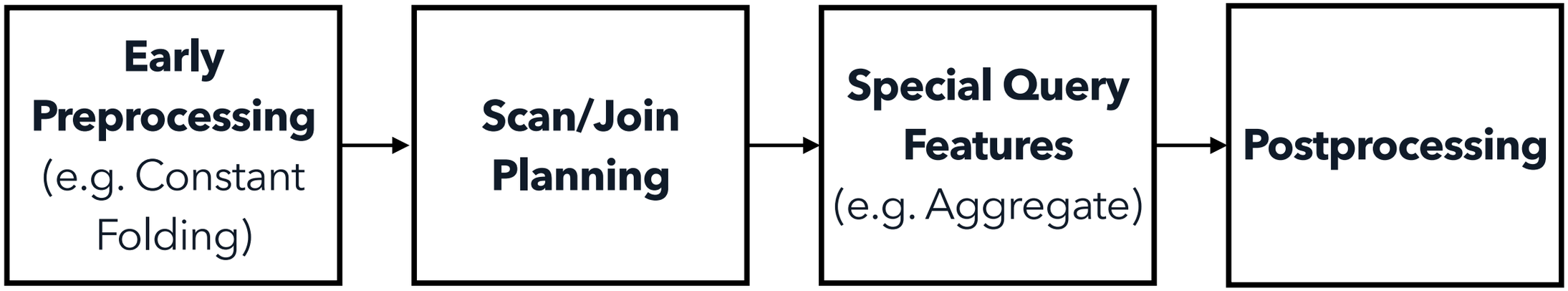
Postgres planner responsibilities:

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:

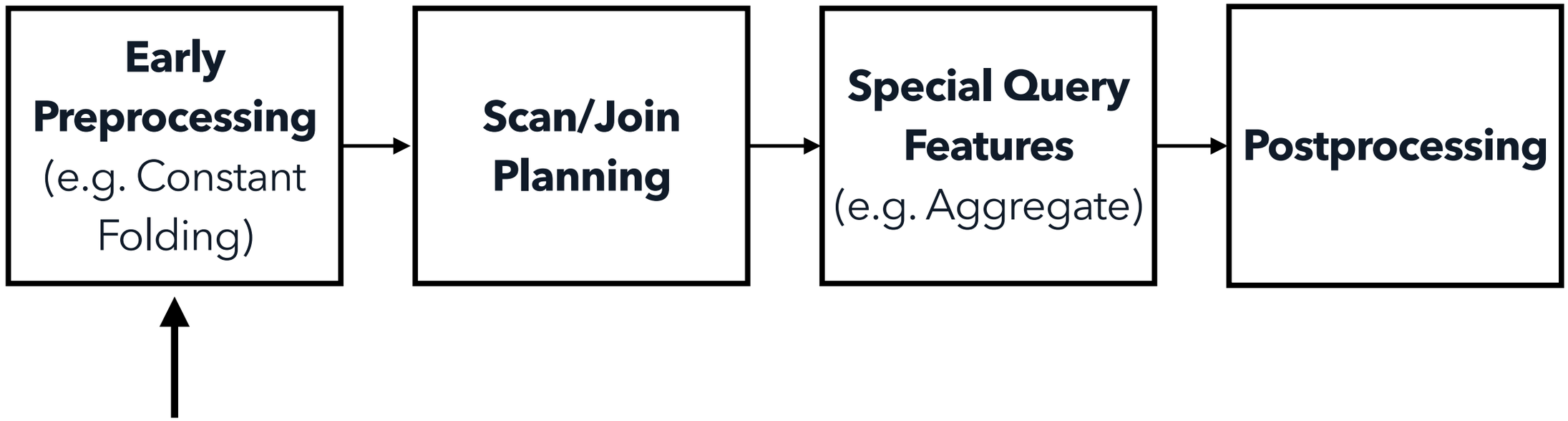
- Find all possible query plans
  (it discards seemingly worse plans quickly)
- Change a plan when its expectations don’t hold true
  (e.g. a lot more rows match than expected)
- Keep track of execution performance
  (it will happily keep producing slow queries)

Phases of Planning



pganalyze

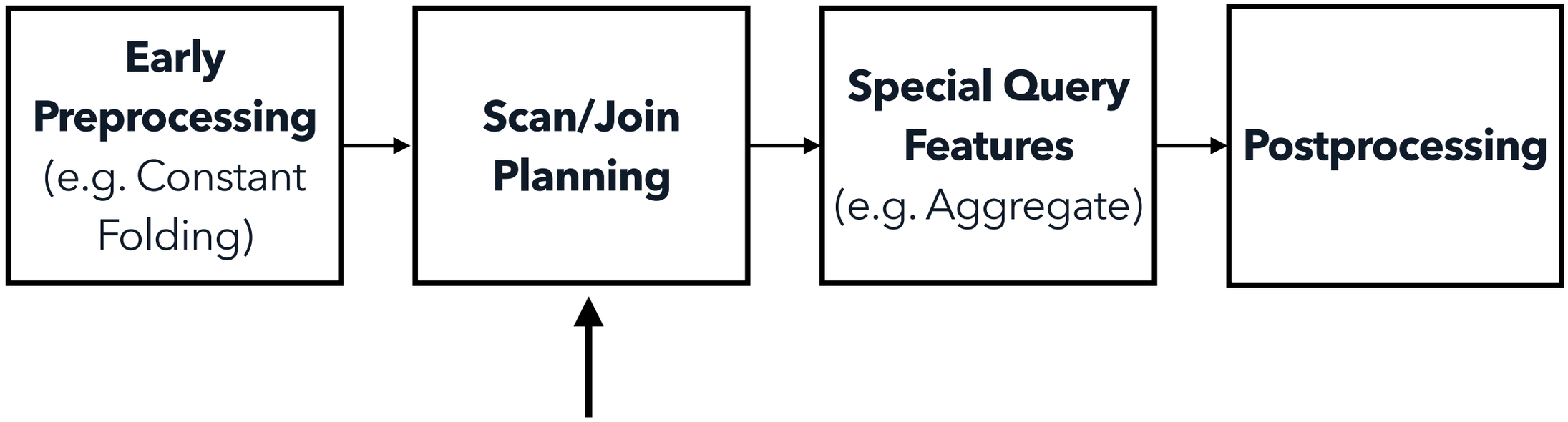
Phases of Planning



If you're surprised **why the planner transformed your query in a certain way** (e.g. removed sub-SELECT), this is the part to understand.

pganalyze

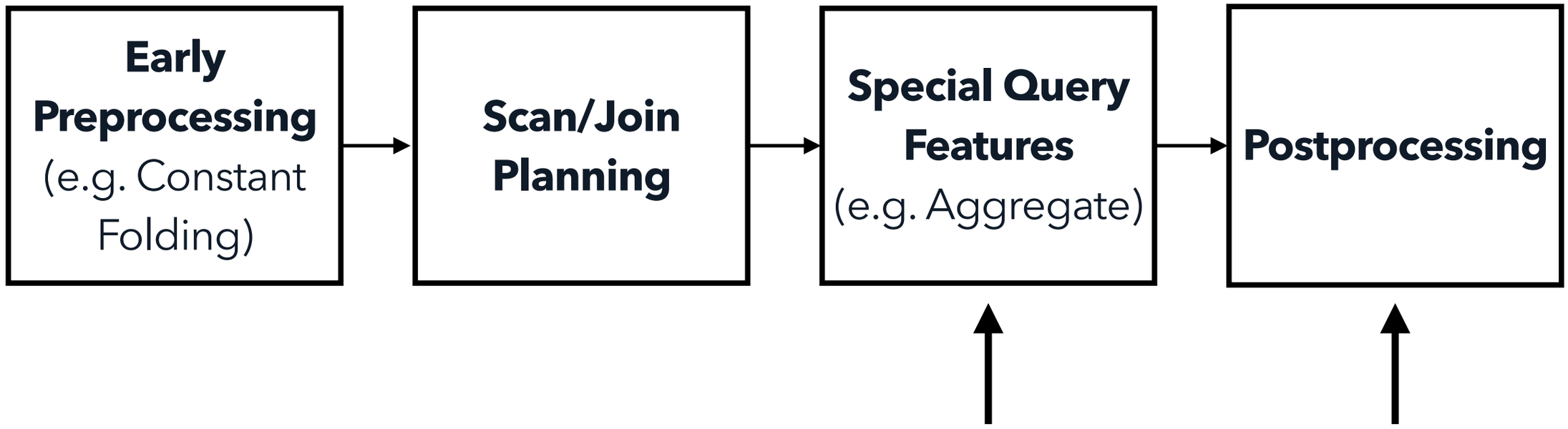
Phases of Planning



This is where a lot of the **variability in different plans for the same query exists.**

pganalyze

Phases of Planning



pganalyze

pganalyze

Cost estimation and selectivity

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

```
->  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)

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

Total cost:

What the planner aims to minimize

```
->  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

pganalyze

What Is “Cost”?

Not a specific unit,

think of it as the “currency” that the planner operates in when it does **cost-based search**

What is the cost of a Sequential Scan?

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?

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);
```

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);
```

Selectivity is the hard part

- Tom Lane

```
/*
 * 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)
{
…
}
```

Selectivity also determines

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

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

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

rows = total_rows * **selectivity**

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

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”**?

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

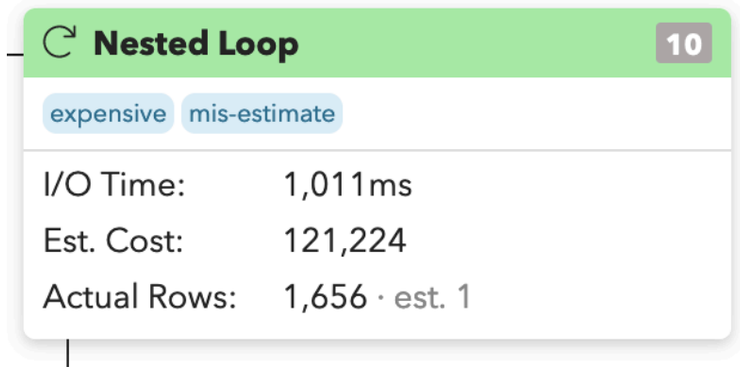
(and often wrong)

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.
     * …
     */
```

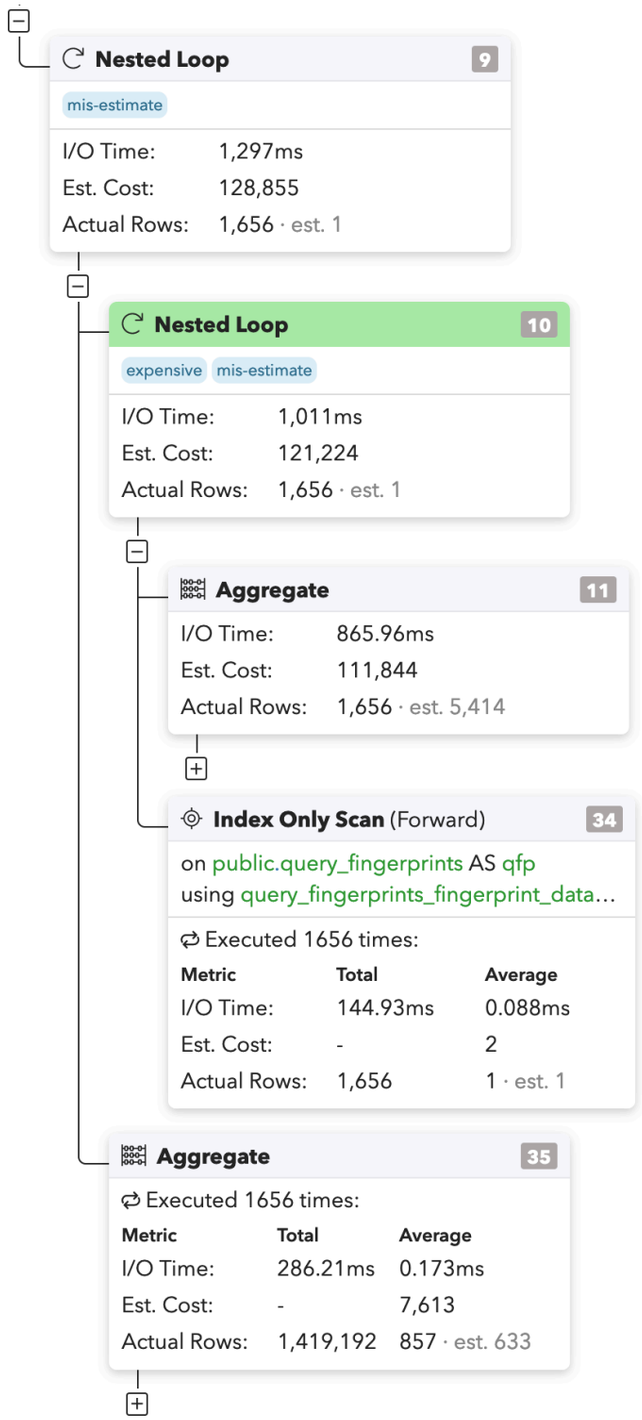
To improve join selectivity (in some cases),
increase the both table column's statistics targets,
to collect more **MCVs**



pganalyze EXPLAIN Insight: Mis-estimate

```
->  Nested Loop  (cost=0.42..121224.18 rows=1 width=53)
                 (actual rows=1656)
```

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

Scan/Join planning

"The aim of the planner is to produce a *join rel* containing all the *base rels* of the query"

- Tom Lane

Step 1: Find the cheapest path for base relations

Go through each base rel, and find:
(1) The cheapest path
(2) The cheapest path if the base rel is parameterized
(see “Parameterized Index Scan”)

(**Path** = Simplified version of a plan node)

```
SELECT *
  FROM a
  JOIN b ON (..)
  JOIN c ON (..)
 WHERE b.x = 1
   AND c.y = 2
```

“a”, “b” and “c” are base relations (“base rels”)

```
SELECT *
  FROM a
  JOIN b ON (..)
  JOIN c ON (..)
 WHERE b.x = 1
   AND c.y = 2
```

A = Sequential Scan

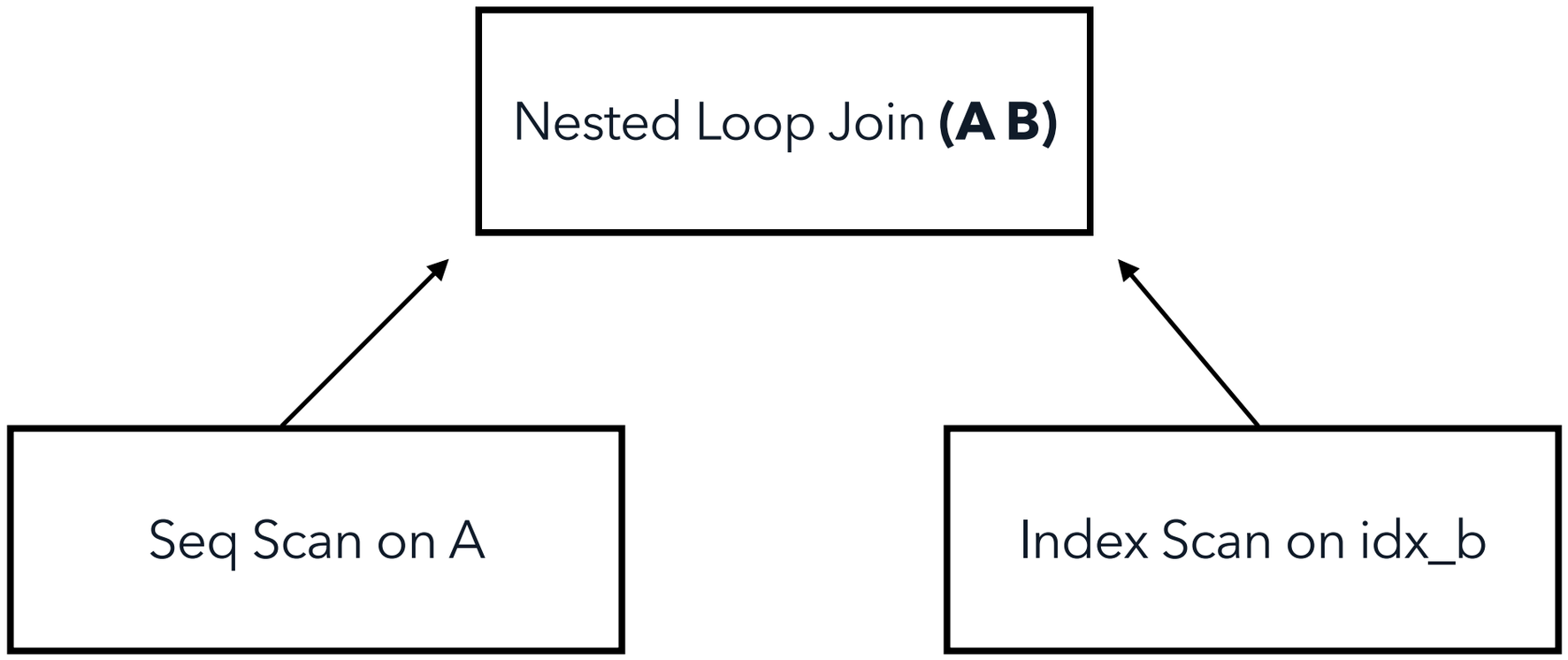
B = Index Scan on btree (x)

C = Index Scan on btree (y)

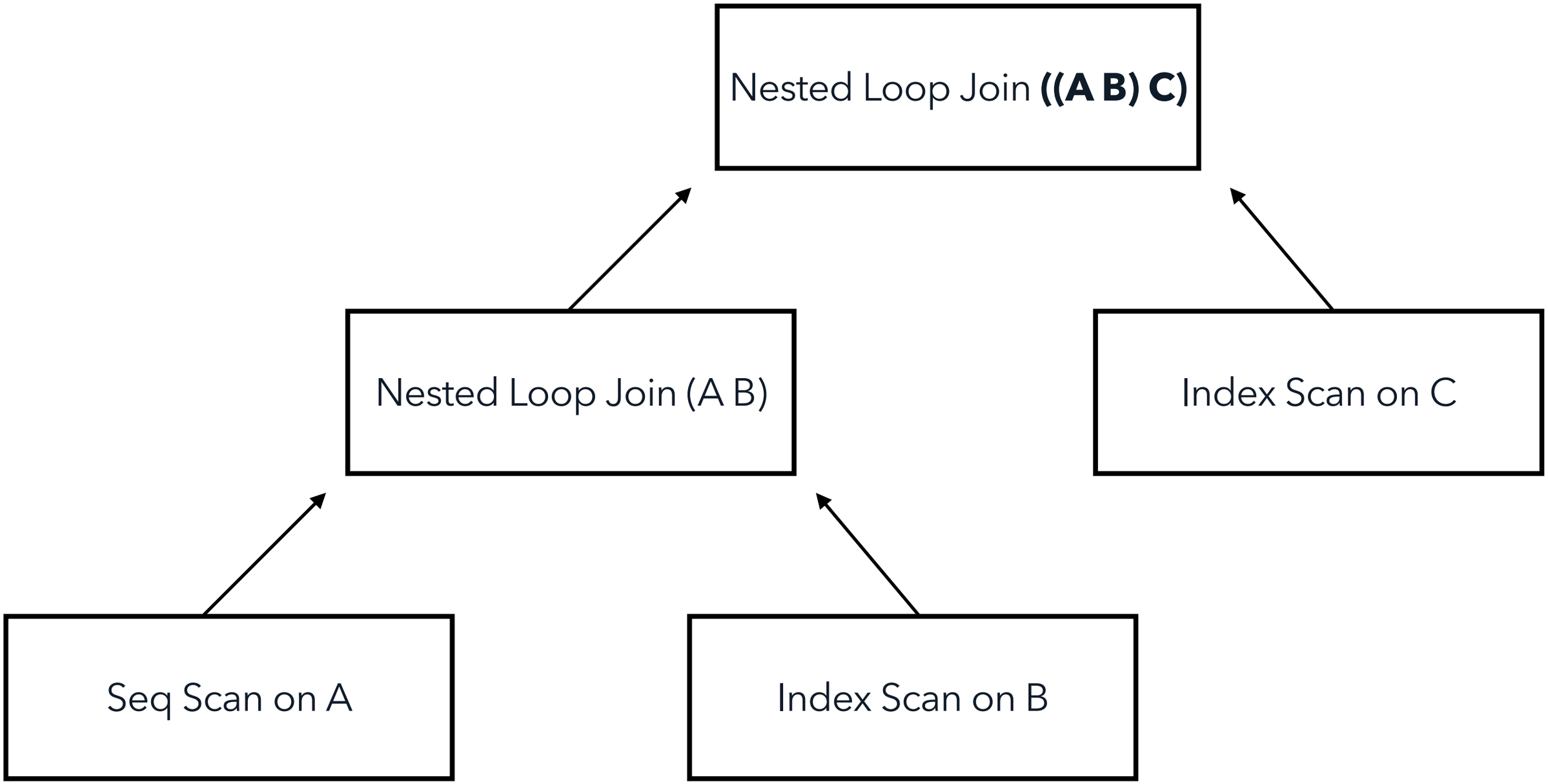
Step 2: Find the best (or closest to best) join order

"Finding the best ordering of pairwise joins is the hard part"

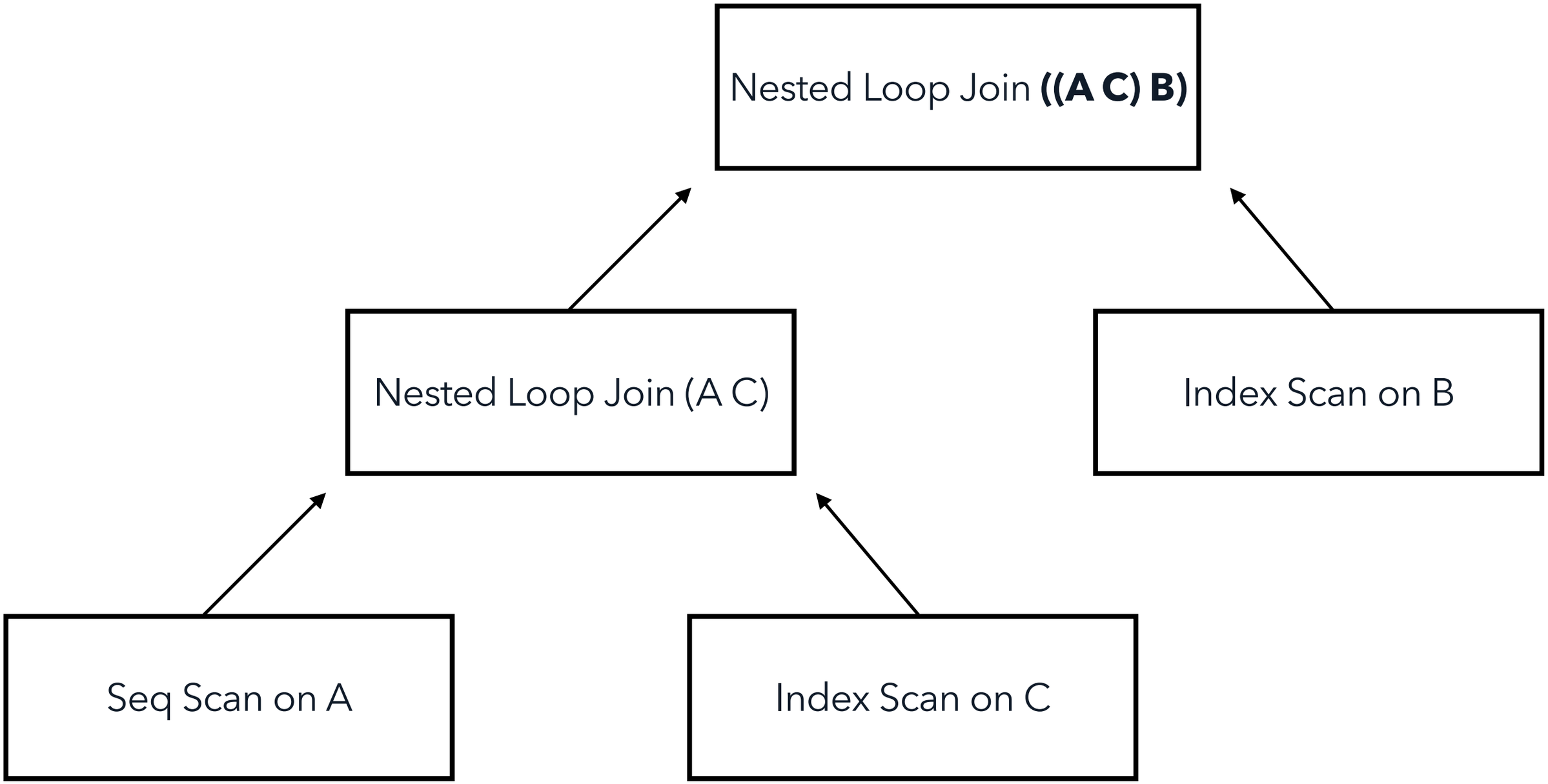
- Tom Lane



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

pganalyze

((A B) C)

= Join Order

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

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:

(experimental pganalyze feature in development)

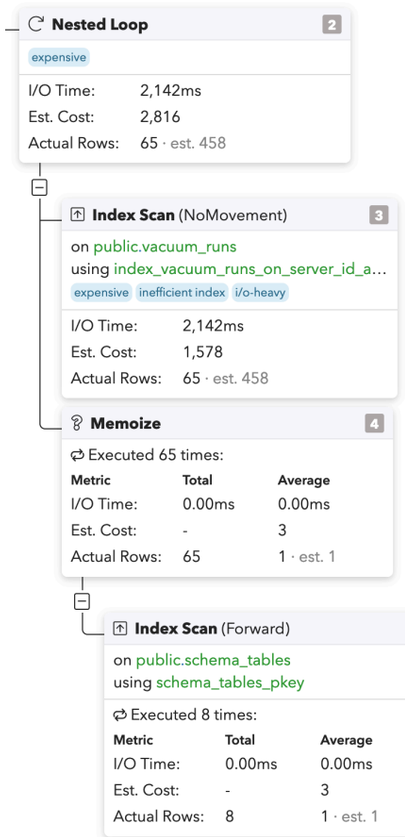
EXPLAINs

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

pganalyze

Parameterized Index Scans

```
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)
```

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

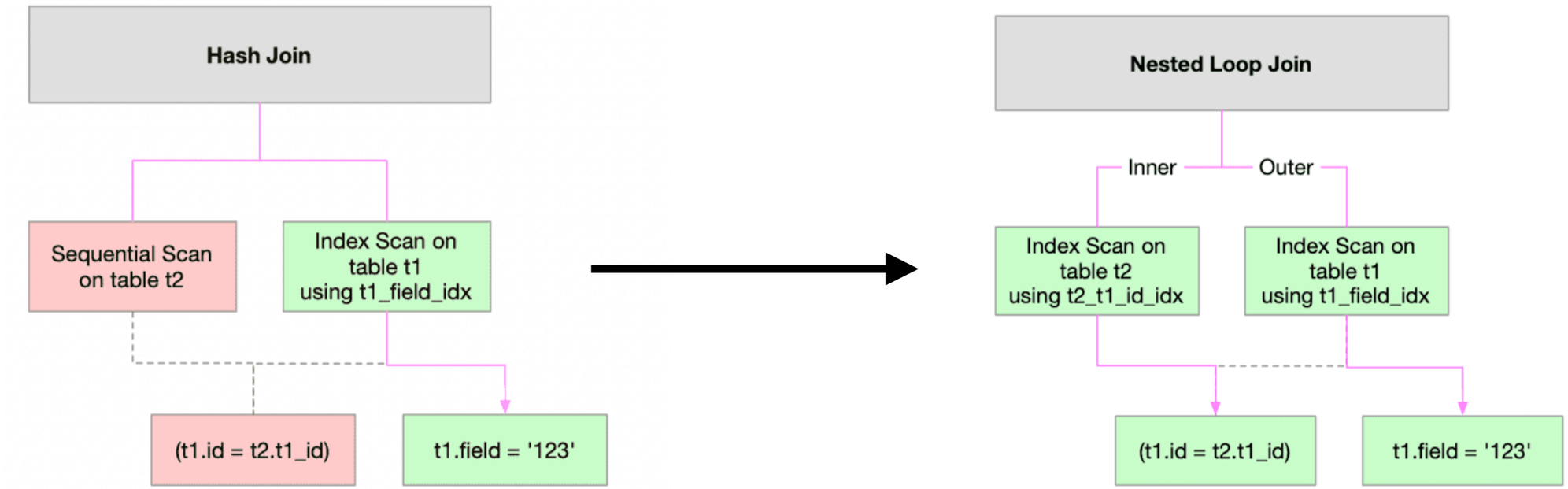
1. Have an expression that uses fixed constant values
   (e.g. "WHERE NOT deleted_at")
2. Have a parameter value (or constant) passed from the client
   (e.g. "WHERE user_id = $1")
3. Filter based on another table's output, as part of a JOIN
   (e.g. "JOIN orgs ON (orgs.id = user.org_id)")

=> (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)
```



Parameterized Index Scan

pganalyze

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

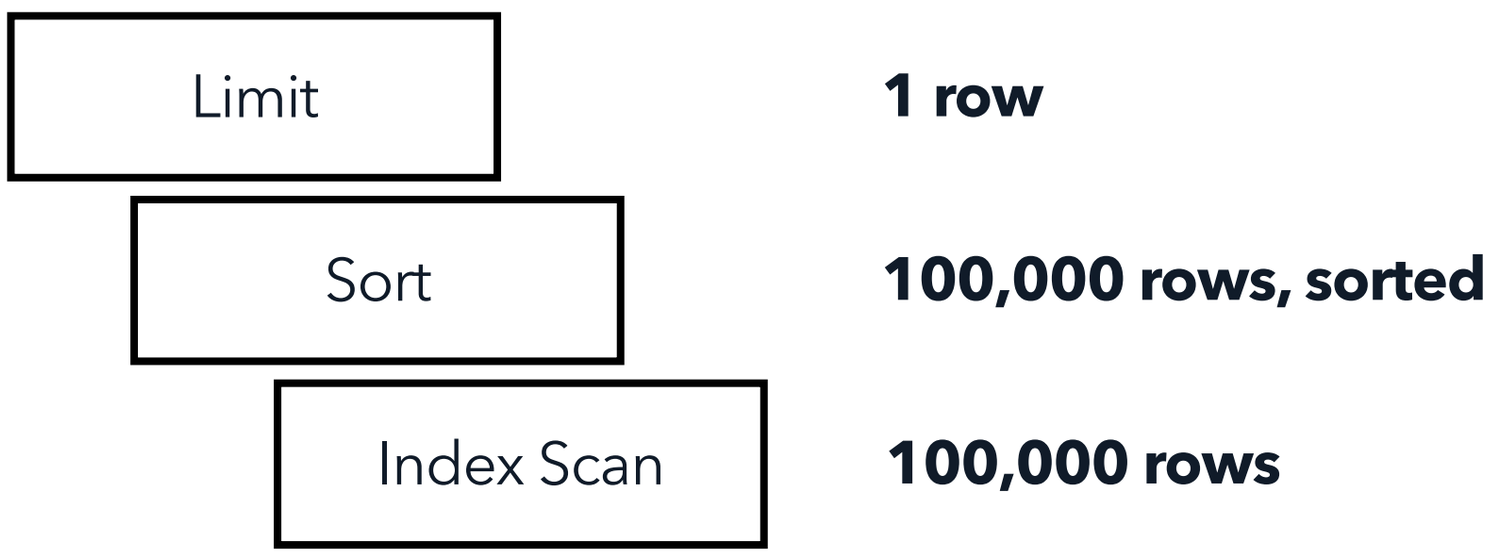
(Join order matters!)

pganalyze

Bounded Sorts

```
SELECT *
  FROM users
 WHERE company = "Lumon Industries"
 ORDER BY created_at DESC
 LIMIT 1;

CREATE INDEX ON users(company);
```



pganalyze

```
SELECT *
  FROM users
 WHERE company = "Lumon Industries"
 ORDER BY created_at DESC
 LIMIT 1;

CREATE INDEX ON users(company, created_at);
```

Limit	**1 row**
Index Scan	**1 row**

B-Tree indexes are sorted, and thus we only need to

find the left-most (or right-most) index entry

to find the result for our ORDER BY … LIMIT 1

Easiest way to get a 100x speed improvement
on a query that does ORDER BY x LIMIT n?

Push the bounded sort into the index lookup.
(by creating the right indexes
and keeping your sorts simple)

Bounded sorts gone wrong:

```
SELECT *
  FROM foo
 WHERE a = 1
 ORDER BY b
 LIMIT 1
```

Planner thinks that it will fill the LIMIT
after reading a small percentage of index on **b**
(and incorrectly chooses index on **b**, over index on **a**)

"This query is my arch nemesis."

Robert Haas at Postgres Open '13

Ruby on Rails loves ORDER BY id:

```
> User.where(email: 'user@example.com').first

SELECT "users".*
  FROM "users"
 WHERE "users"."email" = 'user@example.com'
ORDER BY "users"."id" ASC
LIMIT 1
```

If this bad behavior is a problem,
you can workaround by adding a
(seemingly unnecessary)
multi-column index:

```
CREATE INDEX ON users (email, id);
```

1. Going from query to query plan
2. How the Postgres planner works
3. Cost estimation and selectivity
4. Scan/Join planning
5. Parameterized index scans
6. Bounded sorts
7. Tracking “bad” plans with auto_explain
8. Working with EXPLAIN (ANALYZE, BUFFERS)
9. Pinning plans with pg_hint_plan, or Aurora QPM
10. Guiding the planner to the right plan

pganalyze

Tracking “bad” plans with auto_explain

If we run “EXPLAIN ANALYZE”, we can get the query plan as it is right now - but that doesn’t tell us:

1. **What the actual execution times were** (was the data in cache?)
2. **What the plan was at the time of a problem** (could a recent ANALYZE have changed it?)

auto_explain helps us
track outlier plans through the Postgres logs

F.4. auto_explain

The `auto_explain` module provides a means for logging execution plans of slow statements automatically, without having to run **EXPLAIN** by hand. This is especially helpful for tracking down un-optimized queries in large applications.

The module provides no SQL-accessible functions. To use it, simply load it into the server. You can load it into an individual session:

```
LOAD 'auto_explain';
```

(You must be superuser to do that.) More typical usage is to preload it into some or all sessions by including `auto_explain` in **session_preload_libraries** or **shared_preload_libraries** in `postgresql.conf`. Then you can track unexpectedly slow queries no matter when they happen. Of course there is a price in overhead for that.

F.4.1. Configuration Parameters

There are several configuration parameters that control the behavior of `auto_explain`. Note that the default behavior is to do nothing, so you must set at least `auto_explain.log_min_duration` if you want any results.

```
LOG:  duration: 3.651 ms  plan:
  Query Text: SELECT count(*)
              FROM pg_class, pg_index
              WHERE oid = indrelid AND indisunique;
  Aggregate  (cost=16.79..16.80 rows=1 width=0) (…)
    ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (…)
          Hash Cond: (pg_class.oid = pg_index.indrelid)
          ->  Seq Scan on pg class  (cost=0.00..9.55 rows=255 width=4) (…)
->  Hash  (cost=3.02..3.02 rows=92 width=4) (…)
      Buckets: 1024  Batches: 1  Memory Usage: 4kB
      ->  Seq Scan on pg_index  (…)
            Filter: indisunique
```

Recommended config:

```
shared_preload_libraries = "auto_explain,…"

auto_explain.log_min_duration = 1000ms
auto_explain.log_format = json
auto_explain.log_timing = off
auto_explain.log_analyze = on
auto_explain.log_buffers = on

( + you may turn on extra outputs)
```

Can I run auto_explain in production?

Yes!

But, turn log_timing off.

Timing data collection can be expensive on plans with many Nested Loops.

Recommended to turn it off for most systems:

auto_explain.log_timing = off

(If you do this, its safe to use log_analyze=on)

Maybe changes in Postgres 17?

Sampling-based timing for EXPLAIN ANALYZE

Edit | Comment/Review ▾ | Change Status ▾

Title	Sampling-based timing for EXPLAIN ANALYZE
Topic	Monitoring & Control
Created	2023-01-02 11:39:39
Last modified	2023-01-31 16:14:42 (1 month, 3 weeks ago)
Latest email	2023-03-14 18:37:57 (2 weeks ago)
Status	2023-03: Needs review 2023-01: Moved to next CF
Target version	
Authors	Lukas Fittl (lfittl)
Reviewers	Jelte Fennema (jeltef)
Committer	
Links	
Emails	**Sampling-based timing for EXPLAIN ANALYZE** ✕ First at 2023-01-02 11:36:04 by Lukas Fittl <lukas at fittl.com> Latest at 2023-03-14 18:37:57 by Greg Stark <stark at mit.edu> Latest attachment (v2-0001-Add-TIMING-SAMPLING-option-for-EXPLAIN-ANALYZE.patch) at 2023-01-17 10:50:40 from Lukas Fittl <lukas at fittl.com>

How do I make sense of auto_explain output?

Need to associate it back to queries,
and track it over time.

Two methods:

1. Use the Postgres queryid to link data
2. Track it with pganalyze

Use the Postgres queryid to link data

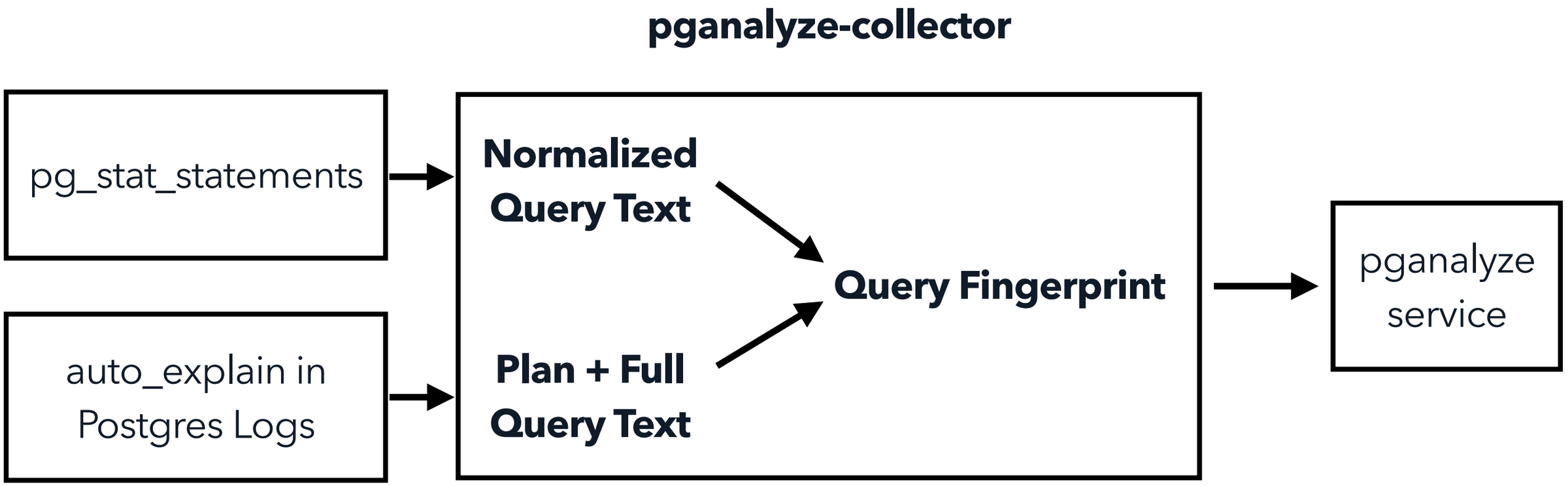
Since Postgres 14, the pg_stat_statements “queryid” is used in:

1. log_line_prefix
2. pg_stat_activity
3. pg_stat_statements itself

To track auto_explain logs for a particular query:

1. Add queryid to your log_line_prefix
2. Find slow queries using pg_stat_statements
3. Grep your logfiles for log lines that match the queryid

Tracking auto_explain output with pganalyze's Automated EXPLAIN feature



pganalyze

Tracking auto_explain output with pganalyze's Automated EXPLAIN feature

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 (...), up…	pgaweb_workers	3.57ms	14396.92	26.24%	16.98%
INSERT INTO query_stats_35d_20230328 (...) SELECT ... F…	pgaweb_workers	142.15ms	246.60	8.10%	11.57%
INSERT INTO schema_index_stats_35d_20230329 (...) SELEC…	pgaweb_workers	49.76ms	477.59	5.34%	7.84%
WITH total_times AS (...), table_queries AS (...), fing…	pgaweb_workers	125.33ms	181.84	9.86%	7.52%
WITH data AS (...), existing_rows AS (...), update_rows…	pgaweb_workers	3.29ms	5285.23	8.91%	5.73%
WITH data(server_id, query_id, schema_table_scan_id, sc…	pgaweb_workers	18.22ms	795.45	5.39%	4.78%
INSERT INTO schema_table_stats_35d_20230329 (...) SELEC…	pgaweb_workers	18.44ms	483.57	1.66%	2.94%
WITH slow_queries AS (...) SELECT ... FROM slow_queries…	pgaweb_workers	1,519.23ms	5.36	4.25%	2.69%
WITH data AS (...), stats AS (...), save_log_lines AS (…	pgaweb_workers	2.50ms	3254.23	0.20%	2.68%
INSERT INTO public.query_stats_hourlies_35d_20230328 (…	pgaweb_workers	3.61ms	1488.99	1.92%	1.77%
WITH data AS (...), existing_rows AS (...), insert_rows…	pgaweb_workers	1.02ms	5285.26	2.22%	1.77%
WITH input AS (...), existing_fingerprints AS (...), up…	pgaweb_workers	16.11ms	331.39	2.16%	1.76%
SELECT ... FROM schema_indices JOIN schema_tables ON sc…	pgaweb_workers	10.94ms	479.84	2.78%	1.73%

Tracking auto_explain output with pganalyze's Automated EXPLAIN feature

`WITH input AS (...), existing_fingerprints AS (...), update_queries AS (...) SELECT ... FROM upda...`

Query #43873801 · Fingerprint: 197b7e0f2ee6aa27 · Role: pgaweb_workers

Avg Time: 3.57ms

Calls Per Minute: 14,396.92 / min

Compare to 7 days ago

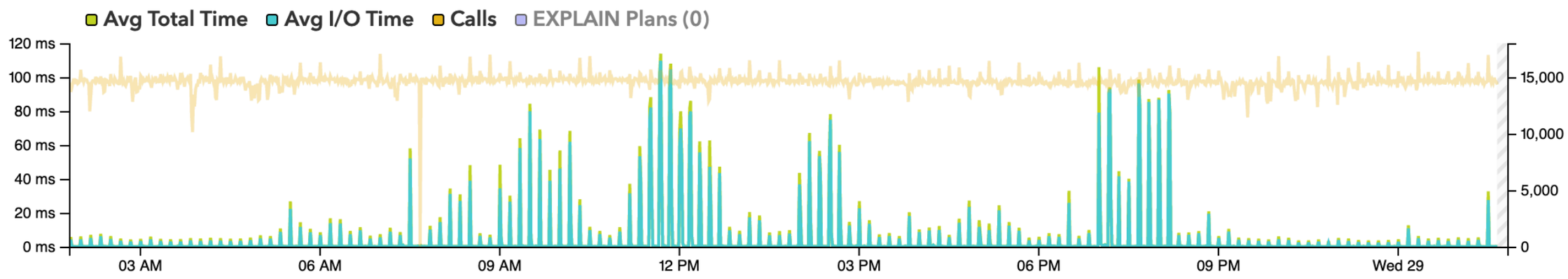
Overview | Index Advisor ✓ | Query Samples 5+ | EXPLAIN Plans 5+ | Query Tags 0 | Log Entries 100+

SQL Statement

```
WITH input AS (
  SELECT *
    FROM unnest($1::int[], $2::bytea[], $3::uuid[], $4::date[], $5::int[]) AS _( database_id, fingerprint, postgres_role_id, last_occurred
_...
```

Show full query text

Avg Time & Calls



pganalyze

Tracking auto_explain output with pganalyze's Automated EXPLAIN feature

```
WITH input AS (...), existing_fingerprints AS (...), update_queries AS (...) SELECT
... FROM upda...
```

Query #43873801 · Fingerprint: 197b7e0f2ee6aa27 · Role: pgaweb_workers

Avg Time: 3.57ms

Calls Per Minute: 14,396.92 / min

Compare to 7 days ago

Overview | Index Advisor ✓ | Query Samples 5+ | EXPLAIN Plans 5+ | Query Tags 0 | Log Entries 100+

EXPLAINs

EXECUTED AT	PLAN	EST. COST	RUNTIME	I/O READ TIME		READ FROM DISK	PLAN NODES
2023-03-29 01:50:14am PDT	a39f918	23	3,562.79ms	3,375.98ms	95%	37.7 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:45:49am PDT	a35d9f1	23	1,036.31ms	904.17ms	87%	31.3 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:41:19am PDT	a38a7f3	23	2,027.00ms	1,983.19ms	98%	24.2 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:40:34am PDT	a33fd4f	23	5,236.96ms	4,311.54ms	82%	28.1 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:31:17am PDT	a38a7f3	23	1,164.77ms	1,121.74ms	96%	24.9 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:30:55am PDT	a3e345f	23	1,071.48ms	894.88ms	84%	24.8 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:30:33am PDT	a3da1b6	23	5,401.36ms	4,951.23ms	92%	22.2 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:22:41am PDT	a3916e8	23	4,377.70ms	3,919.18ms	90%	57.8 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:21:08am PDT	a3bada2	23	1,261.88ms	1,110.78ms	88%	31.3 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:20:31am PDT	a35b2d1	23	1,801.60ms	1,383.91ms	77%	95.9 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:19:15am PDT	a358406	23	1,692.40ms	1,462.34ms	86%	47.1 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:11:08am PDT	a38a7f3	23	1,023.54ms	976.16ms	95%	22.5 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:10:43am PDT	a3fa506	23	1,123.63ms	917.88ms	82%	29.4 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:10:12am PDT	a310e10	23	1,710.74ms	1,646.93ms	96%	16.4 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:00:51am PDT	a33fc3b	23	3,621.43ms	3,081.02ms	85%	88.6 MB	Aggregate · Append · Hash Joi…
2023-03-29 01:00:04am PDT	a3e7f62	23	1,522.59ms	1,073.36ms	70%	25.6 MB	Aggregate · Append · Hash Joi…

Tracking auto_explain output with pganalyze's Automated EXPLAIN feature

```
WITH input AS (...), existing_fingerprints AS (...), update_queries AS (...) SELECT
... FROM upda...
```

Query #43873801 · Fingerprint: 197b7e0f2ee6aa27 · Role: pgaweb_workers

Avg Time 3.57ms

Calls Per Minute 14,396.92 / min

Compare to 7 days ago

Overview | Index Advisor ✓ | Query Samples 5+ | EXPLAIN Plans 5+ | Query Tags 0 | Log Entries 100+

EXPLAIN Plan

2023-03-29 01:50:14am PDT · Fingerprint: a39f918

3,562.79ms	3,375.98ms	37.7 MB	23
Runtime	I/O Read Time	Read From Disk	Total Est. Cost

✔ Analyze ✔ Verbose ✔ Costs ✔ Buffers ✖ Timing ✖ Summary

Compare Query Plans | View EXPLAIN Source

EXPLAIN Insights

Plan node 3 was estimated to be expensive (cost 17 vs avg cost 2) and had rows under-estimated by a factor of 5278
Plan node 9 took 94% of total I/O time
Plan node 2 had rows under-estimated by a factor of 528
Plan node 4 had rows under-estimated by a factor of 528
Plan node 8 had rows under-estimated by a factor of 5278
Plan node 13 had rows under-estimated by a factor of 528
Plan node 14 had rows under-estimated by a factor of 5278
Plan node 15 had rows under-estimated by a factor of 5278

```
WITH input AS (
SELECT *
FROM unnest($1::int[], $2::bytea[], $3::uuid[], $4::date[], $5::int[]) AS _(
database_id, fingerprint, postgres_role_id, last_occurred_...
```

Show full query text

Aggregate 1

I/O Time: 3,376ms
Est. Cost: 23
Actual Rows: 101 · est. 11

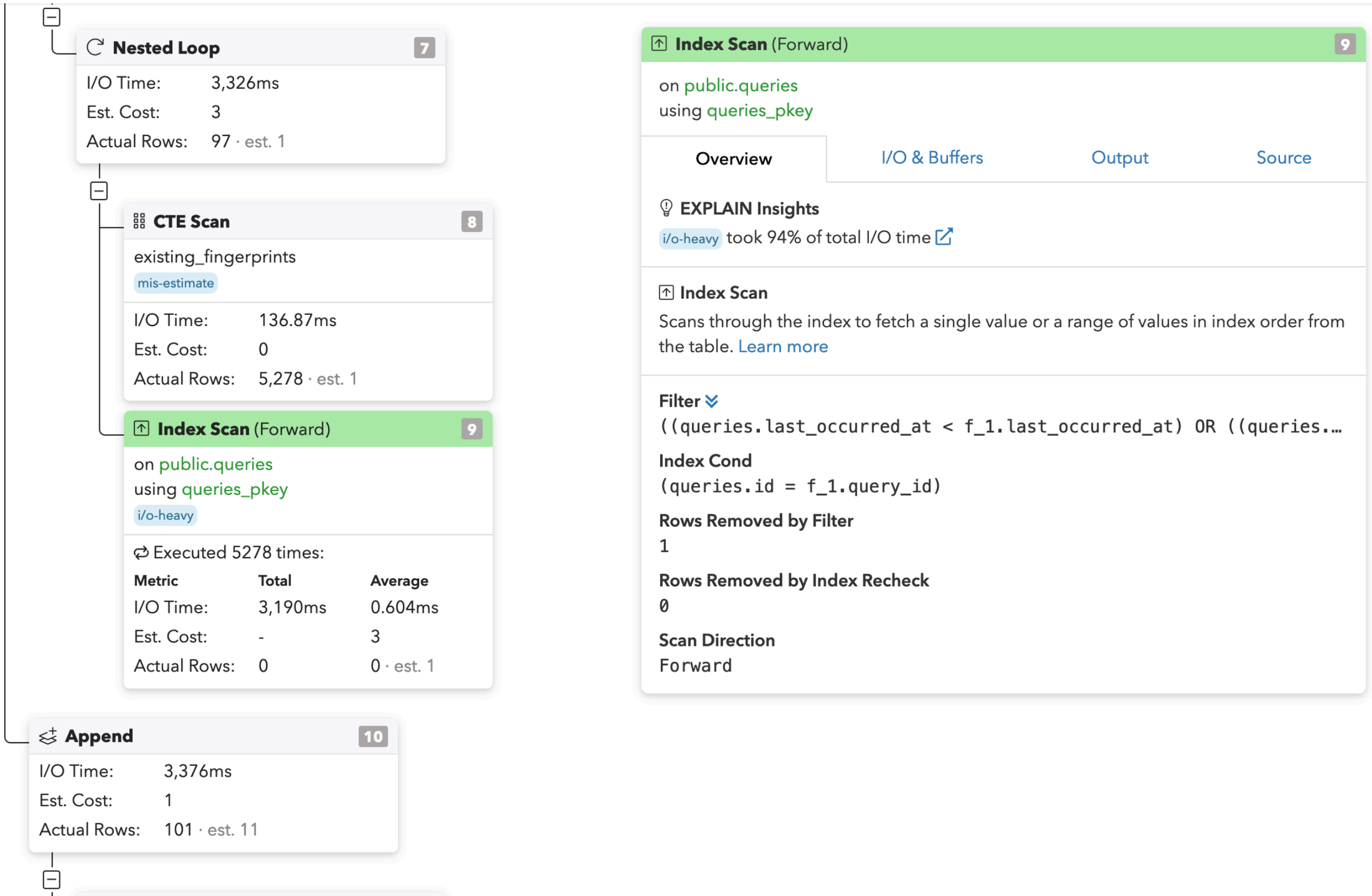
Aggregate 1

Overview | I/O & Buffers | Output | Source

Aggregate

Performs grouped or ungrouped aggregation on output of its child. Learn more

Tracking auto_explain output with pganalyze's Automated EXPLAIN feature



pganalyze

Working with **EXPLAIN (ANALYZE, BUFFERS)**

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

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)

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

Be careful with hit counters in loops!

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

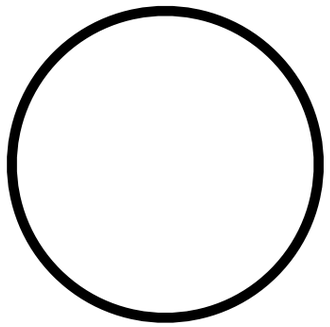
This does not mean 1200 buffers were loaded.

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

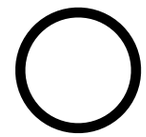
pganalyze

Pinning plans with **pg_hint_plan** or **Aurora QPM**

Your Data Used To Look Like This:

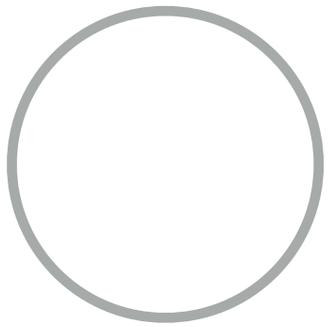


But New Rows Look Like This:

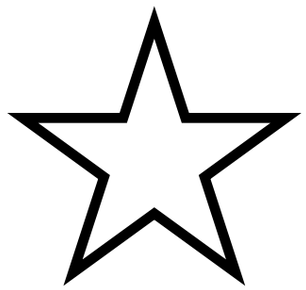


pganalyze

Your Data Used
To Look Like This:



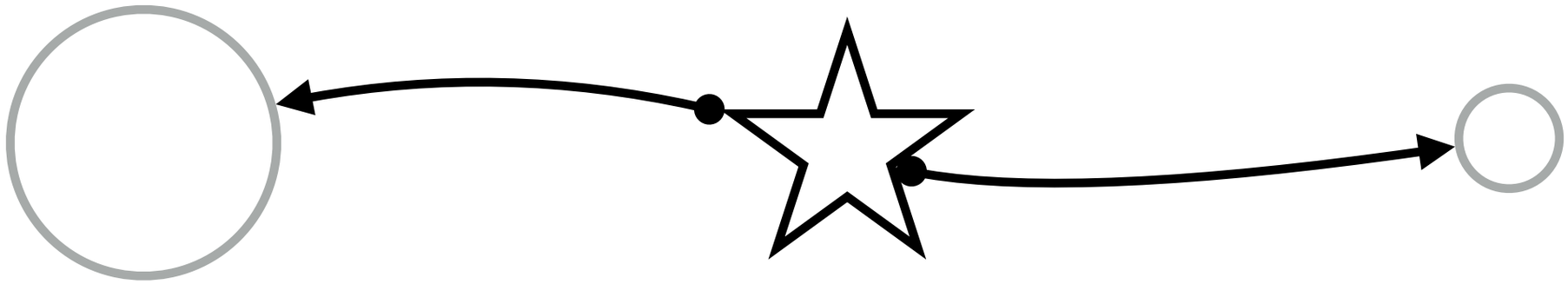
Today:



But New Rows
Look Like This:

Query Plan A:

Query Plan B:



Based on Selectivity and Cost Estimates,
Postgres will choose one or the other

pganalyze

Option 1: Use pg_hint_plan to direct Postgres to a particular query plan

README.md

pg_hint_plan 1.6

`pg_hint_plan` makes it possible to tweak PostgreSQL execution plans using so-called "hints" in SQL comments, like `/*+ SeqScan(a) */`.

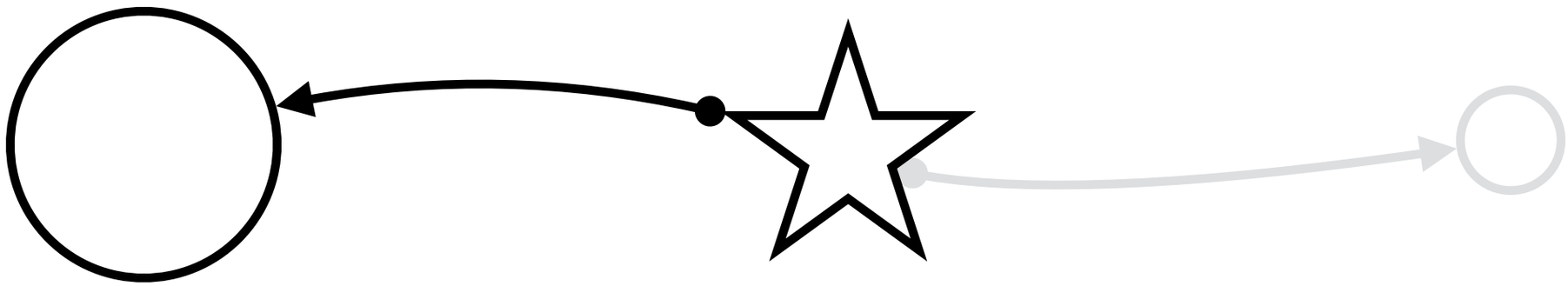
PostgreSQL uses a cost-based optimizer, which utilizes data statistics, not static rules. The planner (optimizer) esitimates costs of each possible execution plans for a SQL statement then the execution plan with the lowest cost finally be executed. The planner does its best to select the best best execution plan, but is not always perfect, since it doesn't count some properties of the data, for example, correlation between columns.

For more details, please see the various documentations available in the **docs/** directory:

1. Description
2. The hint table
3. Installation
4. Unistallation
5. Details in hinting
6. Errors
7. Functional limitations
8. Requirements
9. Hints list

Query Plan A:

Query Plan B:



Modify query text:
/*+ HashJoin(a b) */ SELECT ...

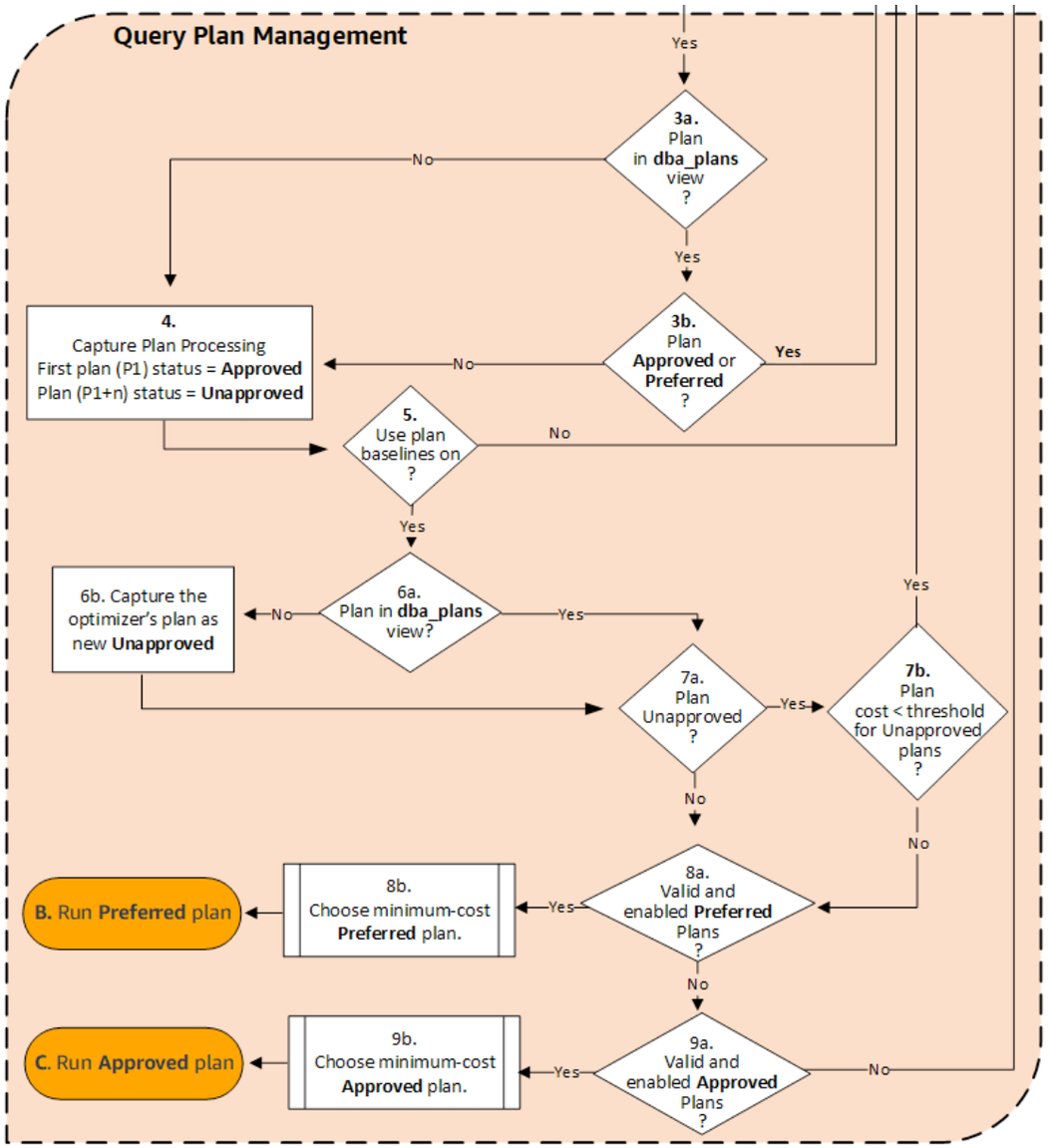
Or use the **hint table**
(matches on query text)

pganalyze

```
SELECT * FROM hint_plan.hints;

-[ RECORD 1 ]-----+-----------------------------
id                | 1
norm_query_string | SELECT * FROM t1 WHERE t1.id = ?;
application_name  |
hints             | SeqScan(t1)
-[ RECORD 2 ]-----+-----------------------------
id                | 2
norm_query_string | SELECT id FROM t1 WHERE t1.id = ?;
application_name  |
hints             | IndexScan(t1)
```

Option 2: Use Aurora Query Plan Management (if applicable)




```
SELECT sql_hash, plan_hash, status, enabled, stmt_name
  FROM apg_plan_mgmt.dba_plans;

 sql_hash   | plan_hash |   status   | enabled | stmt_name
------------+-----------+------------+---------+-----------
 1984047223 | 512153379 | Approved   | t       | rangequery
 1984047223 | 512284451 | Unapproved | t       | rangequery
(2 rows)
```

pganalyze

How Aurora calculates SQL hashes is essential!

```
/*Leading comment*/
EXPLAIN SELECT /* Query 1 */ * FROM t
WHERE x > 7 AND y = 1;
```

```
SELECT /* Query 1 */ * FROM t
WHERE x > CONST AND y = CONST;
```

https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraPostgreSQL.Optimize.Start.html#AuroraPostgreSQL.Optimize.Start.hash-and-normalization

“The exact same SQL statement can produce different hashes depending on how its arguments get bound by the client.

This leads to baffling behavior where it can seem like QPM isn't working at all.”

- Chris Kiehl about Aurora QPM in How to influence query planning in Postgresql

Neither pg_hint_plan or Aurora QPM deal with bad selectivity estimates.

They deal with whole plans, and perform **per-query** adjustments.

Guiding the planner to the right plan

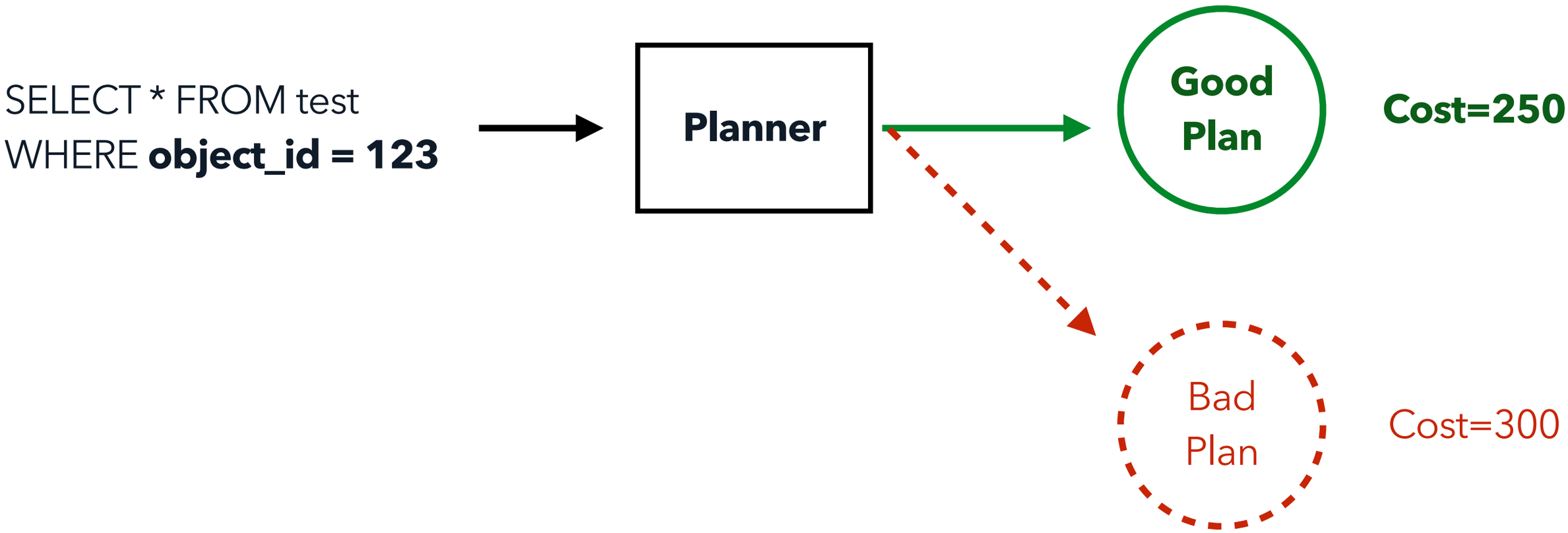
To Understand

Why A “Bad” Plan Was Chosen

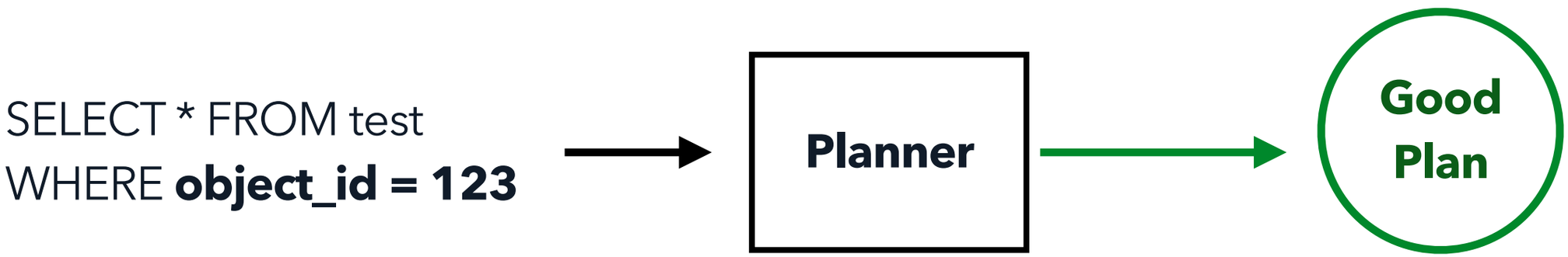
Start By Forcing The Good Plan

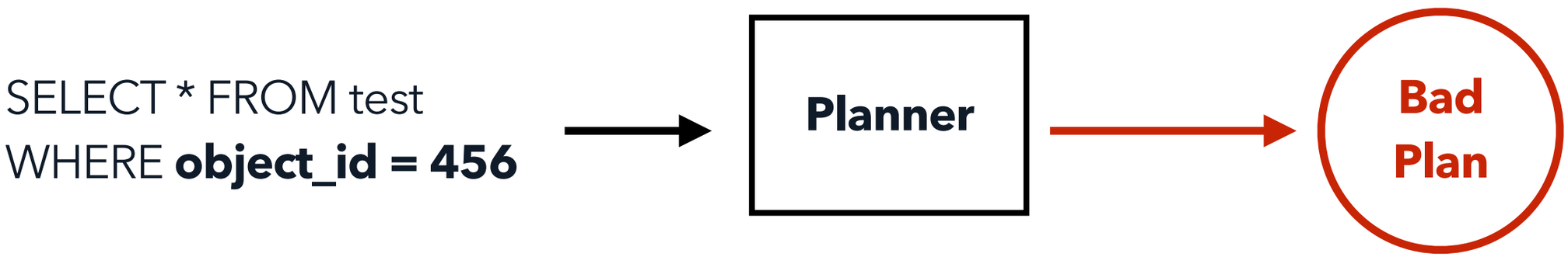
SELECT * FROM test
WHERE **object_id = 123**

pganalyze


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

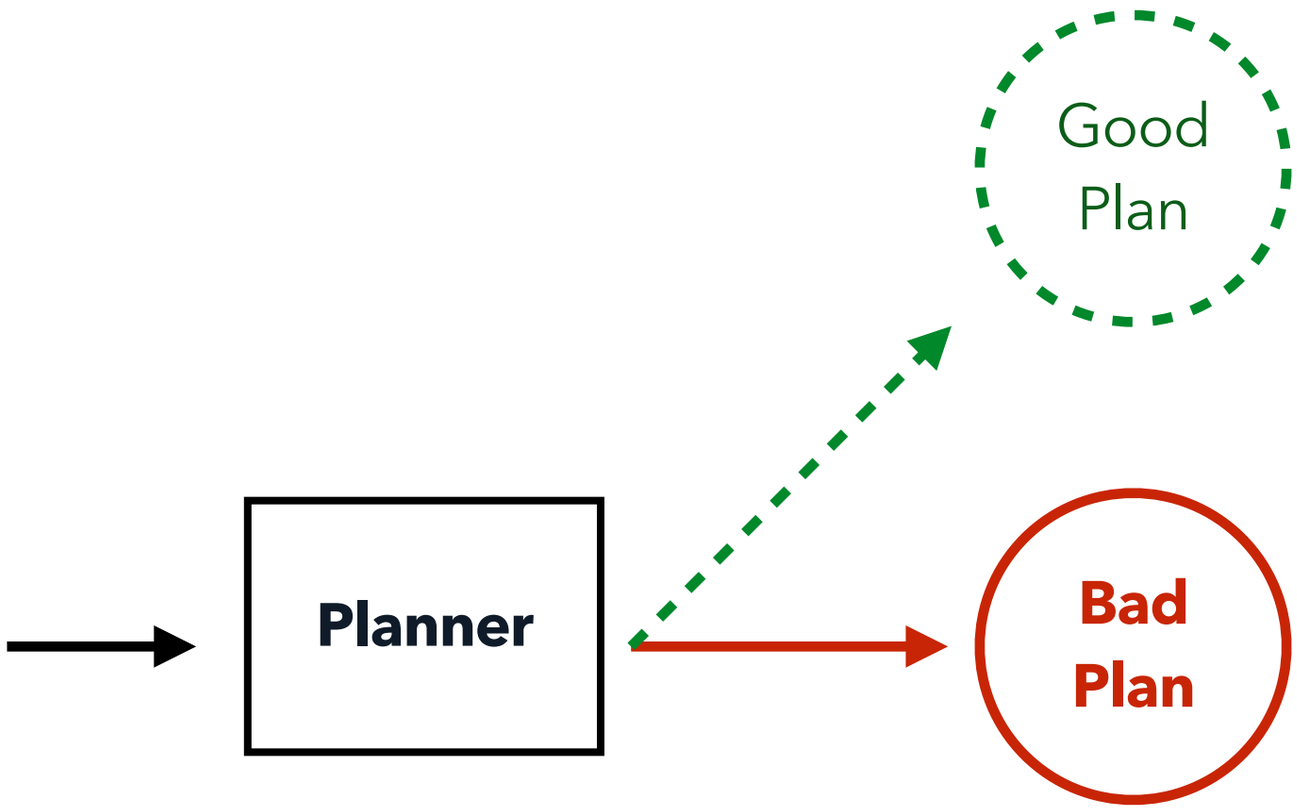
pganalyze


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

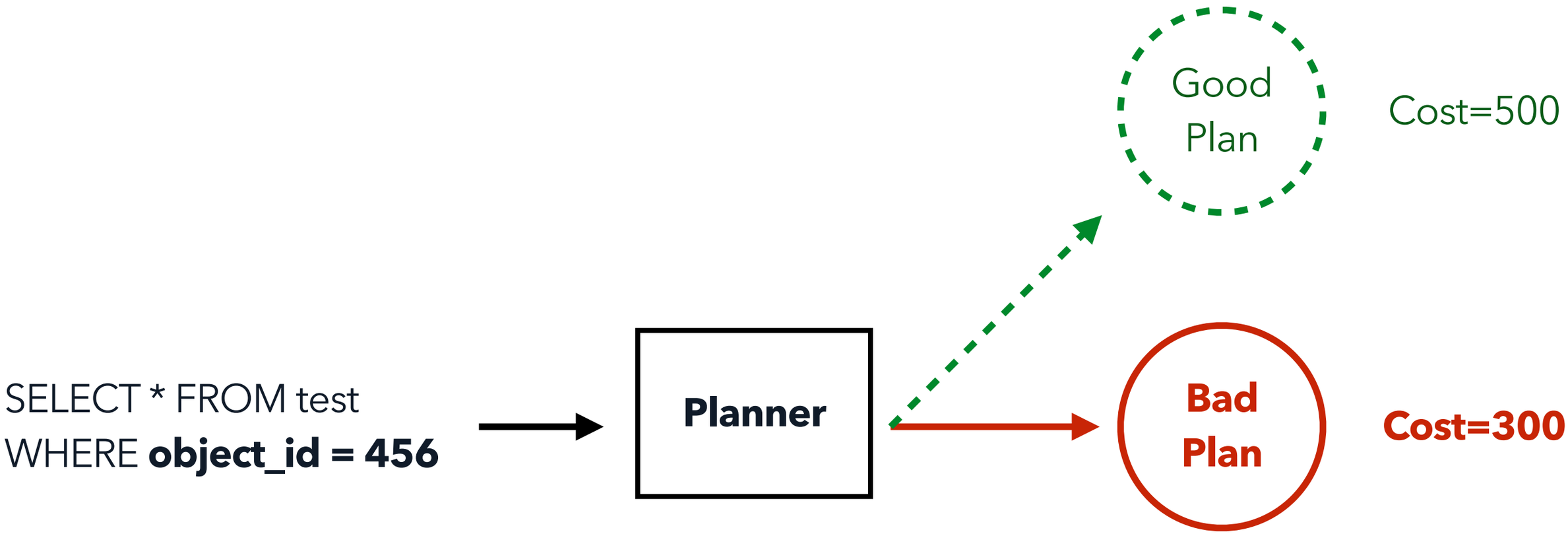

SELECT * FROM test
WHERE **object_id = 456**
Planner
Bad
Plan

pganalyze

SELECT * FROM test
WHERE **object_id = 456**



pganalyze


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

pganalyze

The easiest test:

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

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;
```

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)

```
/*+ 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)

Good plan:

1,451,807 cost

```
->  Nested Loop  (cost=0.99..1451807.35 rows=119623 wid
      ->  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)
```

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"

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

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**

We will send you an email with a recording of this webinar tomorrow!

Feel free to get in touch with us at pganalyze.com/contact