pganalyze

Postgres Plan Monitoring and Management in Practice

Presented by Lukas Fittl

Agenda for today

- **Plan Monitoring** - pg_stat_plans
- **Plan Control** - pg_hint_plan vs pg_plan_advice
- **Plan Management -** hint table vs pg_stash_advice

Plan Monitoring

Query Plans
can change over time

Query Plans can change over time

pganalyze

Let's start with a simple query:

```
SELECT databases.*
FROM databases
WHERE
  databases.server_id = $1
  AND databases.hidden = $2
ORDER BY databases.id ASC
LIMIT $3
```

Query Plans can change over time

pganalyze

Sometimes we get a good plan:

```
 Limit  (cost=137.54..137.55 rows=4 width=152) (actual time=0.029..0.029 rows=2 loops=1)
   ->  Sort  (cost=137.54..137.55 rows=4 width=152) (actual time=0.028..0.028 rows=2 loops=1)
         Sort Key: id
         Sort Method: quicksort  Memory: 25kB
         ->  Index Scan using index_databases_on_server_id_and_datname on databases
(cost=0.56..137.50 rows=4 width=152) (actual time=0.018..0.023 rows=2 loops=1)
               Index Cond: (server_id = 'XXX'::uuid)
               Filter: (NOT hidden)
               Rows Removed by Filter: 3
 Planning Time: 0.096 ms
 Execution Time: 0.046 ms
```

Query Plans can change over time

pganalyze

And sometimes we get a bad plan:

```
 Limit  (cost=1000.58..18048.33 rows=1000 width=152) (actual time=537.544..539.169 rows=1
loops=1)
   ->  Gather Merge  (cost=1000.58..162851.91 rows=9494 width=152) (actual time=537.543..539.167
rows=1 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Parallel Index Scan using databases_pkey on databases  (cost=0.56..160756.04
rows=3956 width=152) (actual time=368.390..535.075 rows=0 loops=3)
               Filter: ((NOT hidden) AND (server_id = 'YYY'::uuid))
               Rows Removed by Filter: 982622
 Planning Time: 0.088 ms
 Execution Time: 539.213 ms
```

Query Plans can change over time

pganalyze

Reasons for bad plans suddenly appearing:

- Different input values change selectivity
- ANALYZE changed the table statistics
- Table data changed
- Indexes changed
- Postgres version upgrades (rare, but it happens!)

Query Plans can change over time

pganalyze

We can't run (and look at)
EXPLAIN on every single query.

Plan Statistics are about capturing what happens over time, so we can **proactively identify bad plans.**

Query Plans can change over time

pganalyze

"I'm a huge fan of Postgres. This one is "user error", but we still got bit pretty hard.

A query plan changed, on a frequently-run query (~1k/sec) on a large table (~2B rows) without warning. Went from sub-millisecond to multi-second.

The PG query planner is generally very good, but also very opaque."
- Scott Hardy on Hacker News (2021)

How to capture plan statistics

How to capture plan statistics

pganalyze

Query ID

Differentiates by query structure.

Plan ID

Differentiates by plan shape.

How to capture plan statistics

pganalyze

Plan Shape
~ EXPLAIN (COSTS OFF)

```
Seq Scan on users
   Filter: (lower((email)::text) = '...'::text)
```

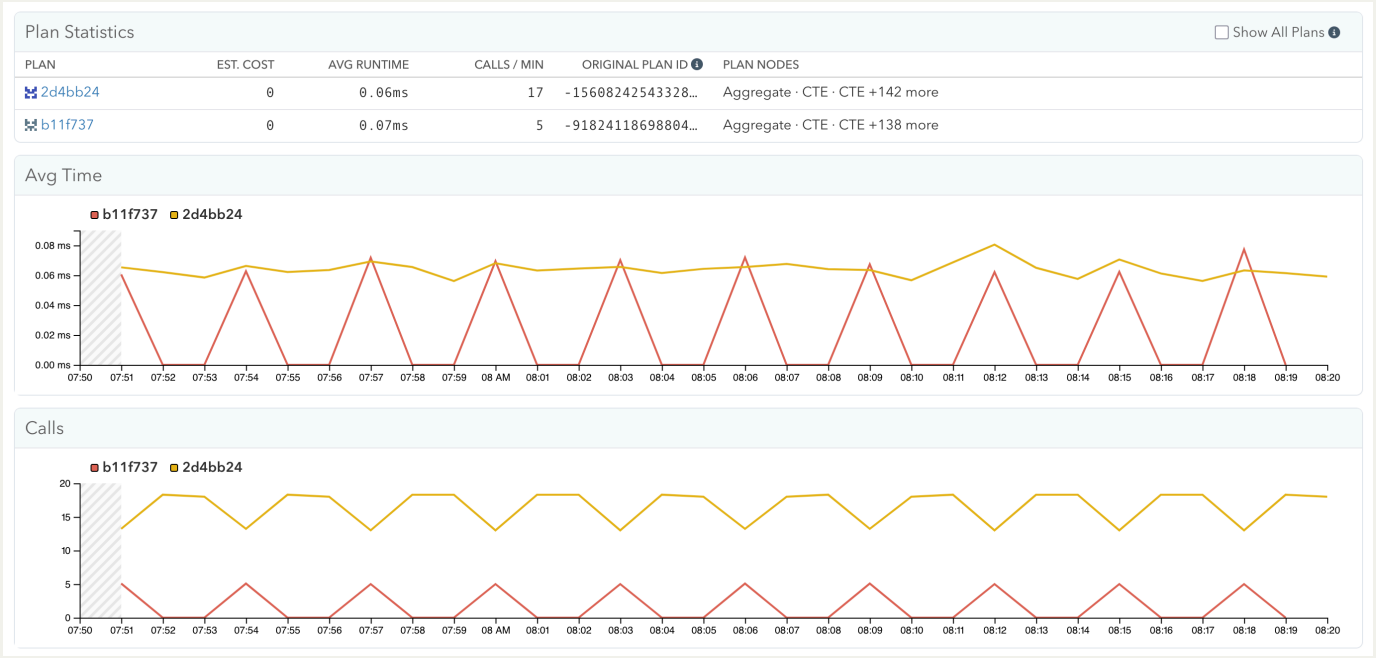
vs

```
 Bitmap Heap Scan on users
   Recheck Cond: (lower((email)::text) = '...'::text)
   ->  Bitmap Index Scan on index_users_lower_email
         Index Cond: (lower((email)::text) = '...'::text)
```

How to capture plan statistics

pganalyze

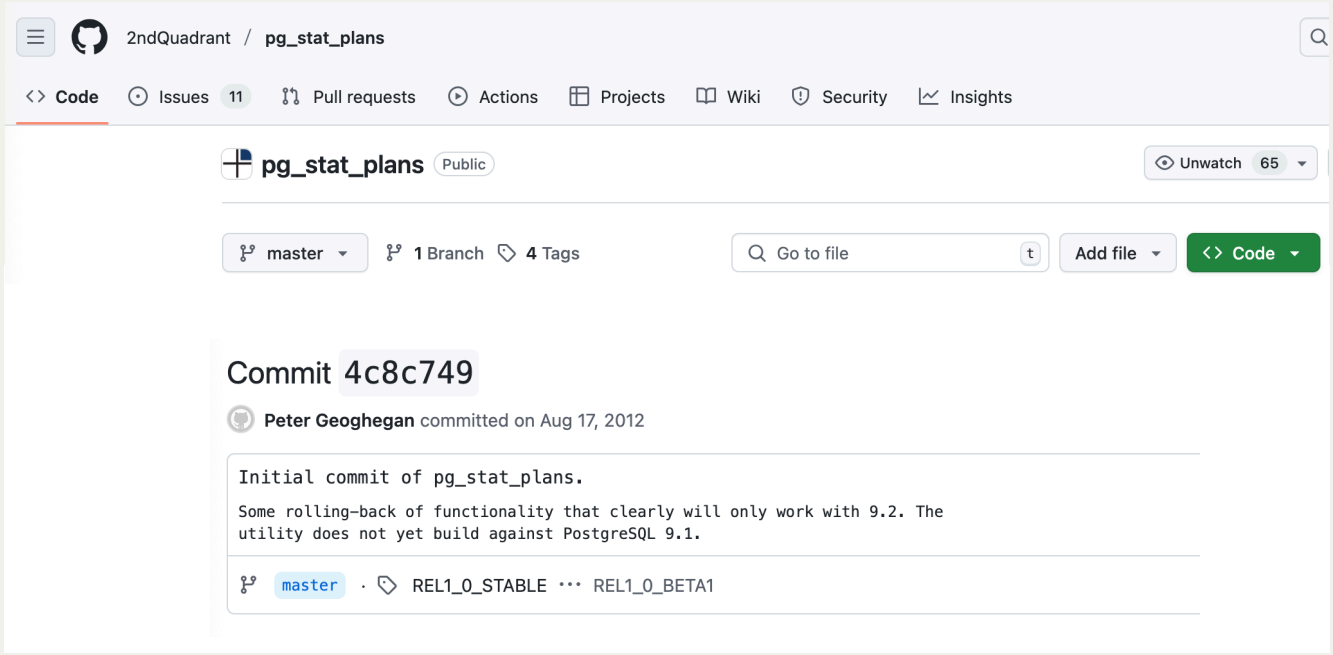
Plan IDs let us track plan usage over time



How to capture plan statistics

pganalyze

This is not a new idea.



How to capture plan statistics

pganalyze

The old pg_stat_plans is unmaintained.

There are open-source alternatives, but they have high overhead.

How to capture plan statistics

pganalyze

pg_store_plans

master pg_store_plans / pg_store_plans.c

Code Blame 2485 lines (2153 loc) · 67.3 KB

```
107   typedef enum pgspVersion
1240
1241            normalized_plan = pgsp_json_normalize(plan);
1242            shorten_plan = pgsp_json_shorten(plan);
1243            elog(DEBUG3, "pg_store_plans: Normalized plan: %s", normalized_plan);
1244            elog(DEBUG3, "pg_store_plans: Shorten plan: %s", shorten_plan);
1245            elog(DEBUG3, "pg_store_plans: Original plan: %s", plan);
1246            plan_len = strlen(shorten_plan);
1247
1248            key.planid = hash_any((const unsigned char *)normalized_plan,
1249                                                  strlen(normalized_plan));
1250            pfree(normalized_plan);
1251
```

Calculates the EXPLAIN text for every execution to hash it for the plan ID

~20% overhead in some cases

How to capture plan statistics

pganalyze

pg_stat_monitor

main pg_stat_monitor / pg_stat_monitor.c

Code Blame 4041 lines (3486 loc) · 116 KB ·

```
707
708             /* Extract the plan information in case of SELECT statement */
709             if (queryDesc->operation == CMD_SELECT && pgsm_enable_query_plan)
710             {
711                     int                     rv;
712                     MemoryContext oldctx;
713
714                     /*
715                      * Making sure it is a per query context so that there's no memory
716                      * leak when executor ends.
717                      */
718                     oldctx = MemoryContextSwitchTo(queryDesc->estate->es_query_cxt);
719
720                     rv = snprintf(plan_info.plan_text, PLAN_TEXT_LEN, "%s", pgsm_explain(queryDesc));
721
722                     /*
723                      * If snprint didn't write anything or there was an error, let's keep
724                      * planinfo as NULL.
725                      */
726                     if (rv > 0)
727                     {
728                             plan_info.plan_len = (rv < PLAN_TEXT_LEN) ? rv : PLAN_TEXT_LEN - 1;
729                             plan_info.planid = pgsm_hash_string(plan_info.plan_text, plan_info.plan_len);
730                             plan_ptr = &plan_info;
731                     }
732
```

Calculates the EXPLAIN text for every execution to hash it for the plan ID (if enabled)

How to capture plan statistics

pganalyze

In 2024, AWS launched
aurora_plan_stats for Aurora
(but it has some scaling problems)

And Microsoft has plan IDs
in **Query Store for Azure Postgres**
(but its not documented how it works)

Can Postgres do better here?

Low-overhead Plan IDs

Low-overhead Plan IDs

pganalyze

Plan ID calculation must be fast

It should happen with every planning cycle.

Low-overhead Plan IDs

pganalyze

ExplainPrintPlan + hash(big text)

Low-overhead Plan IDs

pganalyze

~~ExplainPrintPlan + hash(big text)~~

Low-overhead Plan IDs

pganalyze

We need a tree walk + "jumble"

Query ID = Walk post parse-analysis trees
    Plan ID = Walk plan tree

This is not trivial.

Because of expressions in plan trees

```
typedef struct IndexScan
{
    Scan        scan;
    /* OID of index to scan */
    Oid         indexid;
    /* list of index quals (usually OpExprs) */
    List       *indexqual;

} IndexScan;
```

e.g. Index Quals are "Usually" OpExpr
(but could be any node, and we want to make a hash of it)

We can maintain "what is significant" on the plannodes.h structs

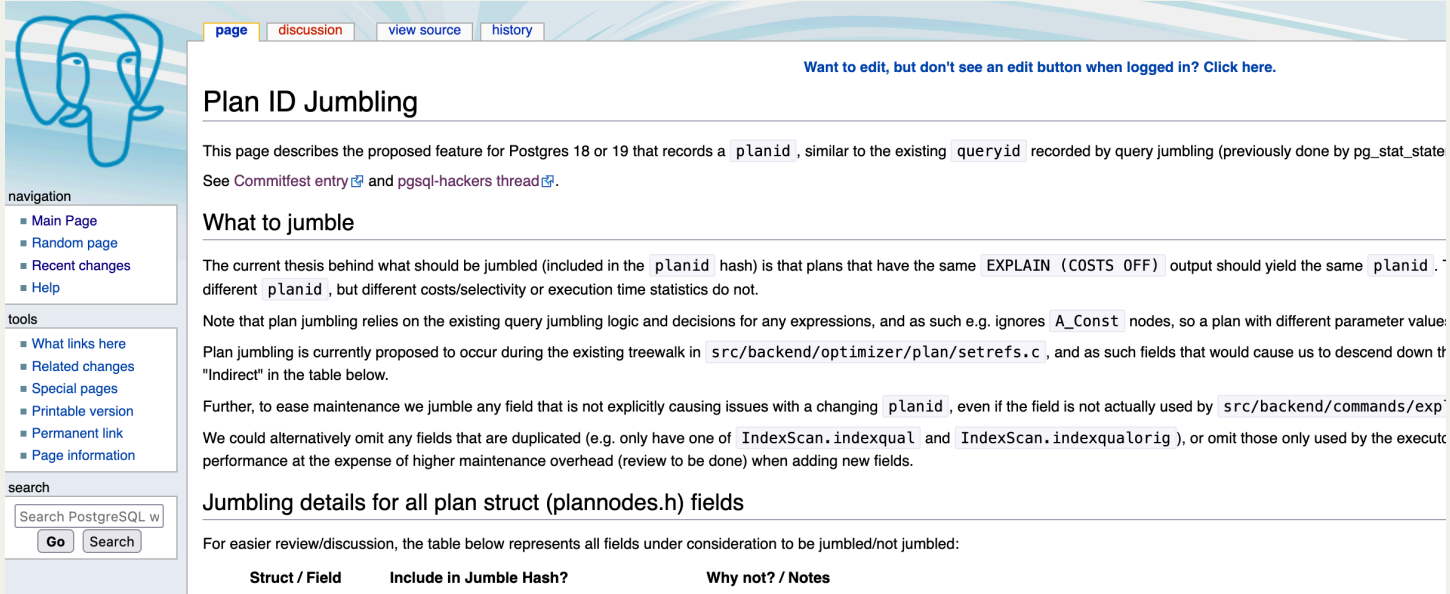
Extensions have to do some extra legwork

```
1059,7 +1059,7 @@ typedef struct Memoize
     * The maximum number of entries that the planner expects will fit in th
     * cache, or 0 if unknown
     */
-    uint32          est_entries;
+    uint32          est_entries pg_node_attr(query_jumble_ignore);

    /* paramids from param_exprs */
    Bitmapset  *keyparamids;
1156,7 +1156,7 @@ typedef struct Agg
    Oid                *grpCollations pg_node_attr(array_size(numCols));

    /* estimated number of groups in input */
-    long            numGroups;
+    long            numGroups pg_node_attr(query_jumble_ignore);

    /* for pass-by-ref transition data */
    uint64          transitionSpace;
```

Low-overhead Plan IDs

pganalyze

In Postgres 18, we started effort to define what a "Plan ID" could be in upstream Postgres



page | discussion | view source | history

Want to edit, but don't see an edit button when logged in? Click here.

Plan ID Jumbling

This page describes the proposed feature for Postgres 18 or 19 that records a `planid`, similar to the existing `queryid` recorded by query jumbling (previously done by pg_stat_state

See Commitfest entry and pgsql-hackers thread.

What to jumble

The current thesis behind what should be jumbled (included in the `planid` hash) is that plans that have the same `EXPLAIN (COSTS OFF)` output should yield the same `planid`. T different `planid`, but different costs/selectivity or execution time statistics do not.

Note that plan jumbling relies on the existing query jumbling logic and decisions for any expressions, and as such e.g. ignores `A_Const` nodes, so a plan with different parameter value

Plan jumbling is currently proposed to occur during the existing treewalk in `src/backend/optimizer/plan/setrefs.c`, and as such fields that would cause us to descend down th "Indirect" in the table below.

Further, to ease maintenance we jumble any field that is not explicitly causing issues with a changing `planid`, even if the field is not actually used by `src/backend/commands/exp`

We could alternatively omit any fields that are duplicated (e.g. only have one of `IndexScan.indexqual` and `IndexScan.indexqualorig`), or omit those only used by the executo performance at the expense of higher maintenance overhead (review to be done) when adding new fields.

Jumbling details for all plan struct (plannodes.h) fields

For easier review/discussion, the table below represents all fields under consideration to be jumbled/not jumbled:

Struct / Field	Include in Jumble Hash?	Why not? / Notes

navigation
- Main Page
- Random page
- Recent changes
- Help

tools
- What links here
- Related changes
- Special pages
- Printable version
- Permanent link
- Page information

search
Search PostgreSQL w
Go | Search

Low-overhead Plan IDs

pganalyze

No in-core Plan ID work has been committed (yet).

But we did get a related **improvement in Postgres 18** we can build on.

Low-overhead Plan IDs

pganalyze

PG18: Allow plugins to set a 64-bit plan identifier in PlannedStmt

projects / postgresql.git / commit

summary | shortlog | log | commit | commitdiff | tree
(parent: 8a3e401) | patch

Allow plugins to set a 64-bit plan identifier in PlannedStmt

```
author    Michael Paquier <michael@paquier.xyz>
          Mon, 24 Mar 2025 04:23:42 +0000 (13:23 +0900)
committer Michael Paquier <michael@paquier.xyz>
          Mon, 24 Mar 2025 04:23:42 +0000 (13:23 +0900)
commit    2a0cd38da5ccf70461c51a489ee7d25fcd3f26be
tree      000fe6d92b36523695dcb368d699ecf2ecd0f191      tree
parent    8a3e4011f02dd2789717c633e74fefdd3b648386      commit | diff
```

Allow plugins to set a 64-bit plan identifier in PlannedStmt

This field can be optionally set in a PlannedStmt through the planner

```
typedef struct PlannedStmt
{
    pg_node_attr(no_equal, no_query_jumble)

    NodeTag     type;

    /* select|insert|update|delete|merge|utility */
    CmdType     commandType;

    /* query identifier (copied from Query) */
    uint64      queryId;

    /* plan identifier (can be set by plugins) */
    uint64      planId;
```

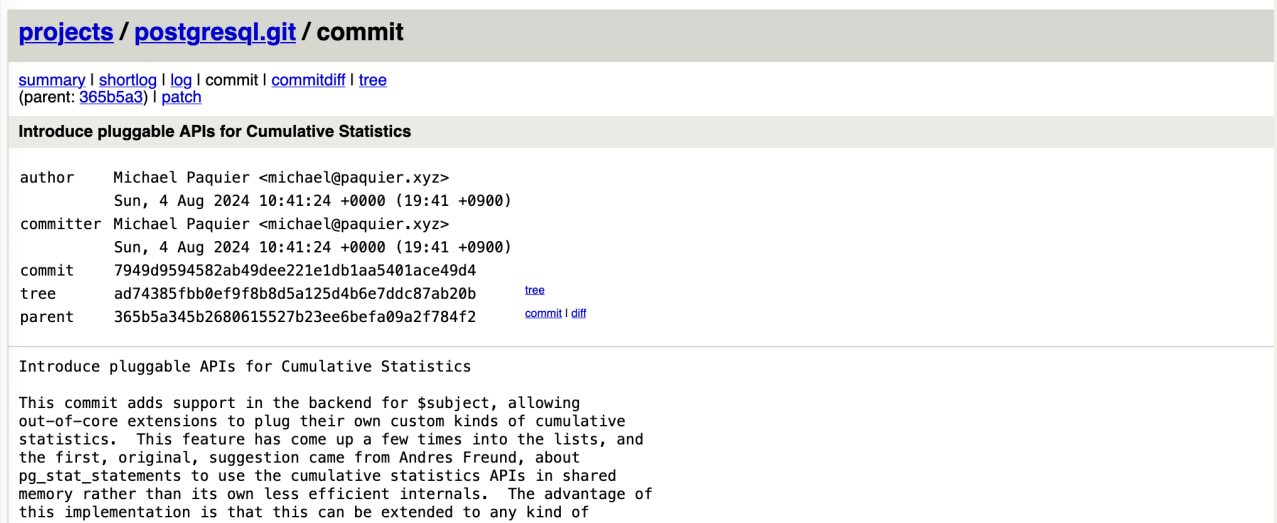
In Postgres 18, you can now write an extension that sets **PlannedStmt.planId** in a **planner_hook,** and then uses it in **ExecutorFinish_hook** to track statistics.

Low-overhead Plan IDs

pganalyze

We got one other key improvement in PG 18:
Pluggable cumulative statistics

PG18: Introduce pluggable APIs for Cumulative Statistics



projects / postgresql.git / commit

summary | shortlog | log | commit | commitdiff | tree
(parent: 365b5a3) | patch

Introduce pluggable APIs for Cumulative Statistics

```
author    Michael Paquier <michael@paquier.xyz>
          Sun, 4 Aug 2024 10:41:24 +0000 (19:41 +0900)
committer Michael Paquier <michael@paquier.xyz>
          Sun, 4 Aug 2024 10:41:24 +0000 (19:41 +0900)
commit    7949d9594582ab49dee221e1db1aa5401ace49d4
tree      ad74385fbb0ef9f8b8d5a125d4b6e7ddc87ab20b    tree
parent    365b5a345b2680615527b23ee6befa09a2f784f2    commit | diff
```

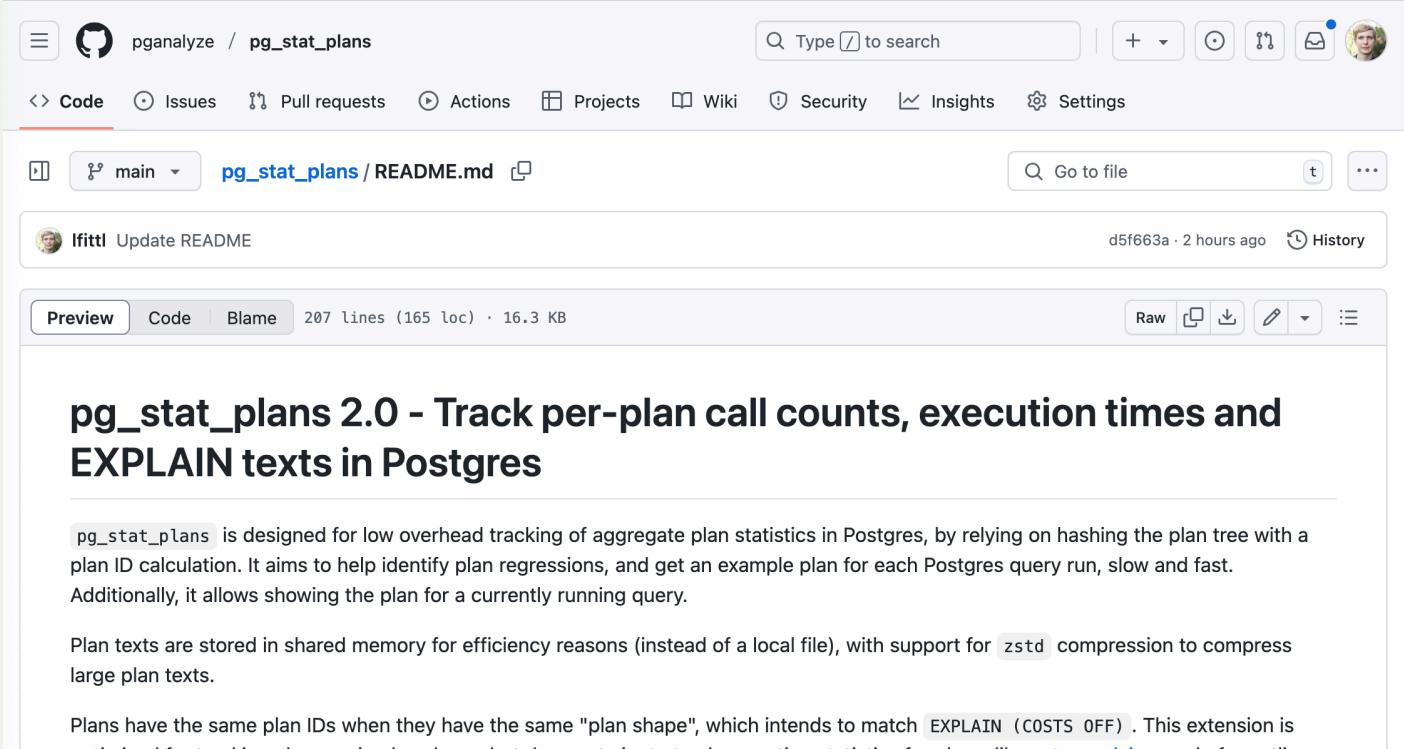
```
Introduce pluggable APIs for Cumulative Statistics

This commit adds support in the backend for $subject, allowing
out-of-core extensions to plug their own custom kinds of cumulative
statistics.  This feature has come up a few times into the lists, and
the first, original, suggestion came from Andres Freund, about
pg_stat_statements to use the cumulative statistics APIs in shared
memory rather than its own less efficient internals.  The advantage of
this implementation is that this can be extended to any kind of
```

A new pg_stat_plans

A new pg_stat_plans

pganalyze

Initially released in September 2025:



A new pg_stat_plans

pganalyze

And now no longer marked experimental:

Remove experimental marker (#17)

The design of pg_stat_plans has been sufficiently reviewed and discussed (and aligns with the direction of pg_stat_statements) that significant user-facing API changes seem less likely. Further, additional stress testing and review has been done on concurrency and scaling issues. Therefore, drop the experimental marker.

main (#17) 1 parent e82aba9 commit 1cb1ef2

1 file changed -2

Search within code

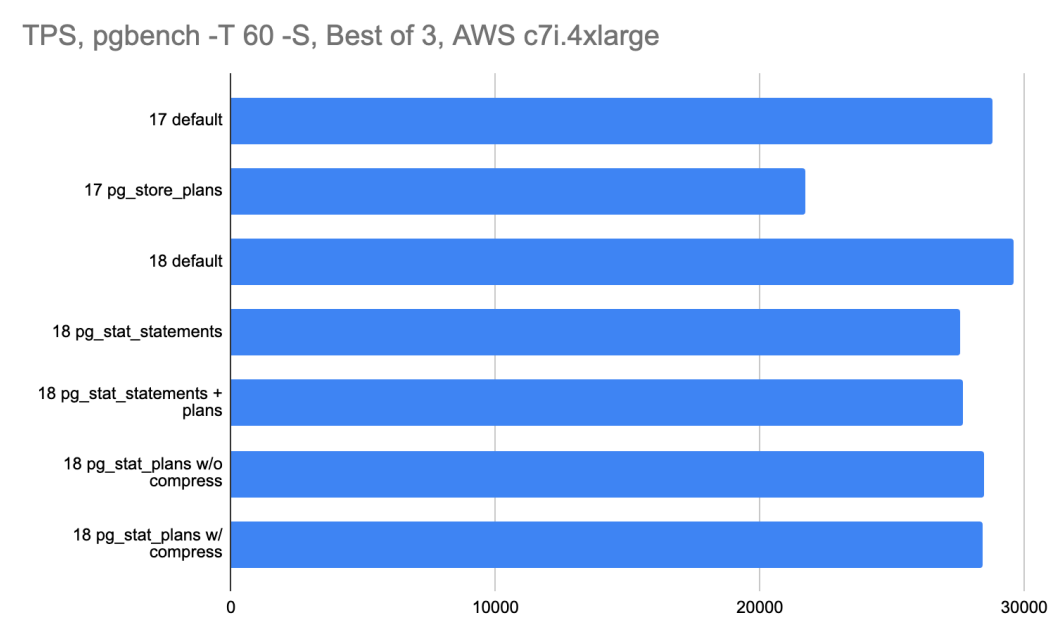
README.md -2

```
@@ -8,8 +8,6 @@ Plans have the same plan IDs when they have the same "plan shape", which intends
 8  8
 9  9   This project is inspired by multiple Postgres community projects, including the original [pg_stat_plans](https://github.com/2ndQuadrant/pg_stat_plans)
        extension (unmaintained), with a goal of upstreaming parts of this extension into the core Postgres project over time.
10 10
11    - **Experimental**. May still change in incompatible ways without notice. Not (yet) recommended for production use.
12    -
13 11   ## Supported PostgreSQL versions
14 12
15 13   Requires at least Postgres 16.
```

A new pg_stat_plans

pganalyze

Overhead is noticeably lower than existing extensions (higher is better)



Disclaimer:
Benchmark from October 2025, not yet updated

A new pg_stat_plans

pganalyze

Cumulative statistics on **which query ID used which plan, how often (calls)**, and **how long it took (total_exec_time).**

```
SELECT * FROM pg_stat_plans;
```

```
-[ RECORD 1 ]---+-----------------------------------------------------------------------------------------------------
userid          | 10
dbid            | 16391
toplevel        | t
queryid         | -2322344003805516737
planid          | -1865871893278385236
calls           | 1
total_exec_time | 0.047708
plan            | Limit
                |   ->  Sort
                |         Sort Key: database_stats_35d.frozenxid_age DESC
                |         ->  Bitmap Heap Scan on database_stats_35d_20250514 database_stats_35d
                |               Recheck Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
                |               Filter: ((frozenxid_age IS NOT NULL) AND (collected_at = '2025-05-14 14:30:00'::time
                |               ->  Bitmap Index Scan on database_stats_35d_20250514_server_id_idx
                |                     Index Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
```

A new pg_stat_plans

pganalyze

Plan ID calculated with tree walk after planning
+ copying code from Postgres

```
SELECT * FROM pg_stat_plans;
```

```
-[ RECORD 1 ]---+-----------------------------------------------------------------------------------------------
userid          | 10
dbid            | 16391
toplevel        | t
queryid         | -2322344003805516737
planid          | -1865871893278385236
calls           | 1
total_exec_time | 0.047708
plan            | Limit
                |   ->  Sort
                |         Sort Key: database_stats_35d.frozenxid_age DESC
                |         ->  Bitmap Heap Scan on database_stats_35d_20250514 database_stats_35d
                |               Recheck Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
                |               Filter: ((frozenxid_age IS NOT NULL) AND (collected_at = '2025-05-14 14:30:00'::time
                |               ->  Bitmap Index Scan on database_stats_35d_20250514_server_id_idx
                |                     Index Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
```

A new pg_stat_plans

pganalyze

Plan Text stored in Dynamic Shared Memory,
not a file on disk. Optionally compressed with zstd.

```
SELECT * FROM pg_stat_plans;
```

```
-[ RECORD 1 ]---+-------------------------------------------------------------------------------------------------
userid          | 10
dbid            | 16391
toplevel        | t
queryid         | -2322344003805516737
planid          | -1865871893278385236
calls           | 1
total_exec_time | 0.047708
plan            | Limit
                |   ->  Sort
                |         Sort Key: database_stats_35d.frozenxid_age DESC
                |         ->  Bitmap Heap Scan on database_stats_35d_20250514 database_stats_35d
                |               Recheck Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
                |               Filter: ((frozenxid_age IS NOT NULL) AND (collected_at = '2025-05-14 14:30:00'::time
                |               ->  Bitmap Index Scan on database_stats_35d_20250514_server_id_idx
                |                     Index Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
```

A new pg_stat_plans

pganalyze

Compression of plan texts is important - EXPLAIN output can be large!

```
Limit
  ->  Sort
        Sort Key: servers.integrated_explain DESC, servers.integrated_log_insights DESC, servers.last_snapshot_at, servers.last_test_snapshot_at
        ->  Index Scan using unique_servers_organization_id_system_columns on servers
              Index Cond: (organization_id = '...'::uuid)
              Filter: ((ANY (organization_id = (hashed SubPlan 1).col1)) OR (ANY (id = (hashed SubPlan 2).col1)) OR (ANY (id = (hashed SubPlan 3).col1)))
              SubPlan 1
                ->  Nested Loop
                      Join Filter: (organization_role_assignments.organization_role_id = organization_roles.organization_role_id)
                      ->  Nested Loop
                            ->  Hash Join
                                  Hash Cond: (organization_role_assignments.organization_membership_id = organization_memberships.organization_membership_id)
                                  ->  Seq Scan on organization_role_assignments
                                  ->  Hash
                                        ->  Seq Scan on organization_memberships
                                              Filter: ((deleted_at IS NULL) AND accepted AND (user_id = 1))
                            ->  Index Scan using unique_organization_role_permissions_role_id on organization_permissions
                                  Index Cond: (organization_role_id = organization_role_assignments.organization_role_id)
                                  Filter: view
                      ->  Index Only Scan using organization_roles_pkey on organization_roles
                            Index Cond: (organization_role_id = organization_permissions.organization_role_id)
              SubPlan 2
                ->  Nested Loop
                      ->  Nested Loop
                            Join Filter: (organization_role_assignments_1.organization_role_id = organization_permissions_1.organization_role_id)
                            ->  Seq Scan on organization_permissions organization_permissions_1
                                  Filter: (view AND (server_id IS NOT NULL))
                            ->  Materialize
                                  ->  Hash Join
                                        Hash Cond: (organization_role_assignments_1.organization_membership_id = organization_memberships_1.organization_membership_id)
                                        ->  Seq Scan on organization_role_assignments organization_role_assignments_1
                                        ->  Hash
                                              ->  Seq Scan on organization_memberships organization_memberships_1
                                                    Filter: ((deleted_at IS NULL) AND accepted AND (user_id = 1))
                      ->  Index Only Scan using organization_roles_pkey on organization_roles organization_roles_1
                            Index Cond: (organization_role_id = organization_role_assignments_1.organization_role_id)
              SubPlan 3
                ->  Nested Loop
                      ->  HashAggregate
                            Group Key: organization_permissions_2.database_id
                            ->  Nested Loop
                                  ->  Nested Loop
                                        ->  Hash Join
                                              Hash Cond: (organization_role_assignments_2.organization_role_id = organization_permissions_2.organization_role_id)
                                              ->  Seq Scan on organization_role_assignments organization_role_assignments_2
                                              ->  Hash
                                                    ->  Seq Scan on organization_permissions organization_permissions_2
                                                          Filter: (view AND (database_id IS NOT NULL))
                                        ->  Index Scan using organization_memberships_pkey on organization_memberships organization_memberships_2
                                              Index Cond: (organization_membership_id = organization_role_assignments_2.organization_membership_id)
                                              Filter: ((deleted_at IS NULL) AND accepted AND (user_id = 1))
                                  ->  Index Only Scan using organization_roles_pkey on organization_roles organization_roles_2
                                        Index Cond: (organization_role_id = organization_role_assignments_2.organization_role_id)
                      ->  Index Scan using databases_pkey on databases
```

A new pg_stat_plans

pganalyze

Compression of plan texts is important - EXPLAIN output can be large!

```
Limit
  ->  Sort
        Sort Key: servers.integrated_explain DESC, servers.integrated_log_insights DESC, servers.last_snapshot_at, servers.last_test_snapshot_at
        ->  Index Scan using unique_servers_organization_id_system_columns on servers
              Index Cond: (organization_id = '...'::uuid)
              Filter: ((ANY (organization_id = (hashed SubPlan 1).col1)) OR (ANY (id = (hashed SubPlan 2).col1)) OR (ANY (id = (hashed SubPlan 3).col1)))
              SubPlan 1
                ->  Nested Loop
                      Join Filter: (organization_role_assignments.organization_role_id = organization_roles.organization_role_id)
                      ->  Nested Loop
                            ->  Hash Join
                                  Hash Cond: (organization_role_assignments.organization_membership_id = organization_memberships.organization_membership_id)
                                  ->  Seq Scan on organization_role_assignments
                                  ->  Hash
                                        ->  Seq Scan on organization_memberships
                                              Filter: ((deleted_at IS NULL) AND accepted AND (user_id = 1))
                            ->  Index Scan using unique_organization_role_permissions_role_id on organization_permissions
                                  Index Cond: (organization_role_id = organization_role_assignments.organization_role_id)
                                  Filter: view
                      ->  Index Only Scan using organization_roles_pkey on organization_roles
                            Index Cond: (organization_role_id = organization_permissions.organization_role_id)
              SubPlan 2
                ->  Nested Loop
                      ->  Nested Loop
                            Join Filter: (organization_role_assignments_1.organization_role_id = organization_permissions_1.organization_role_id)
                            ->  Seq Scan on organization_permissions organization_permissions_1
                                  Filter: (view AND (server_id IS NOT NULL))
                            ->  Materialize
                                  ->  Hash Join
                                        Hash Cond: (organization_role_assignments_1.organization_membership_id = organization_memberships_1.organization_membership_id)
                                        ->  Seq Scan on organization_role_assignments organization_role_assignments_1
                                        ->  Hash
                                              ->  Seq Scan on organization_memberships organization_memberships_1
                                                    Filter: ((deleted_at IS NULL) AND accepted AND (user_id = 1))
```

4727 B uncompressed (~**2,000** entries in 10MB shared memory)

855 B compressed zstd (~**12,000** entries in 10MB shared memory)

602 B compressed zstd + dictionary (~**17,000** entries in 10MB shared memory)

A new pg_stat_plans

pganalyze

Get the plan for a currently running query

(no progress tracking, just the plan that's being used)

```
SELECT * FROM pg_stat_plans_activity;
```

```
  pid  |       plan_id        |                                                    plan
-------+----------------------+------------------------------------------------------------------------------------------------
 83994 | -5449095327982245076 | Merge Join
       |                      |   Merge Cond: ((a.datid = p.dbid) AND (a.usesysid = p.userid) AND (a.query_id = p.queryid) AND (a.plan_id
       |                      |   ->  Sort
       |                      |         Sort Key: a.datid, a.usesysid, a.query_id, a.plan_id
       |                      |         ->  Function Scan on pg_stat_plans_get_activity a
       |                      |   ->  Sort
       |                      |         Sort Key: p.dbid, p.userid, p.queryid, p.planid
       |                      |         ->  Function Scan on pg_stat_plans p
       |                      |               Filter: (toplevel IS TRUE)
 87168 |  4721228144609632390 | Sort
       |                      |   Sort Key: q.id
       |                      |   ->  Nested Loop
       |                      |         ->  Index Scan using index_query_runs_on_server_id on query_runs q
       |                      |               Index Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
       |                      |               Filter: ((started_at IS NULL) AND (finished_at IS NULL))
       |                      |         ->  Index Scan using databases_pkey on databases db
       |                      |               Index Cond: (id = q.database_id)
 81527 |  3819832514333472635 | Result
```

pg_stat_plans 2.1 is released today!

What's new:

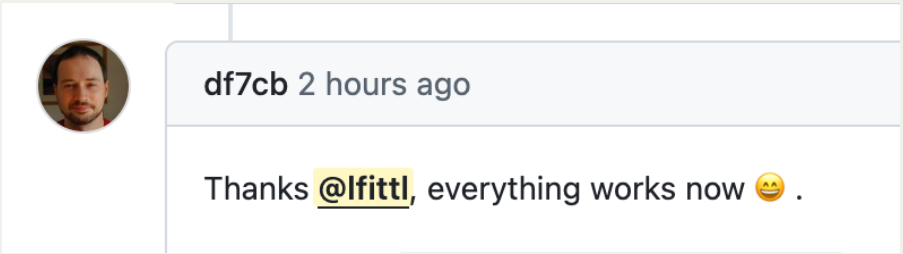
- Support for Postgres 19
- Correctness/scaling improvements (no longer experimental!)
- Memory use for plan texts controlled through new setting (pg_stat_plans.max_plan_memory) now defaults to 16MB
- Optional pg_plan_advice integration

A new pg_stat_plans

pganalyze

Now available in PGDG for Debian/Ubuntu, with RPMs coming in the future:

```
apt install pg-stat-plans
```



With big thanks to Christoph Berg!

Plan Control

Why do we want to control query plans?

Control is necessary for debugging

Control is necessary for debugging

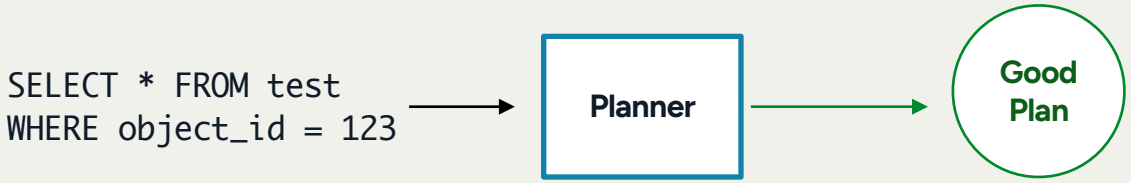
pganalyze

To Understand

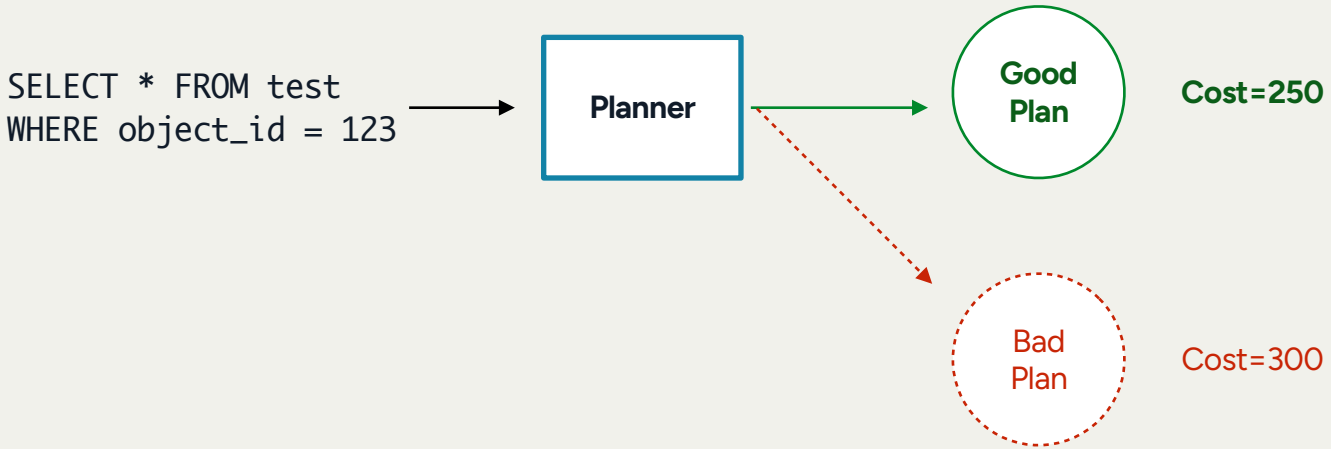
Why A “Bad” Plan Was Chosen

Start By Forcing The Good Plan

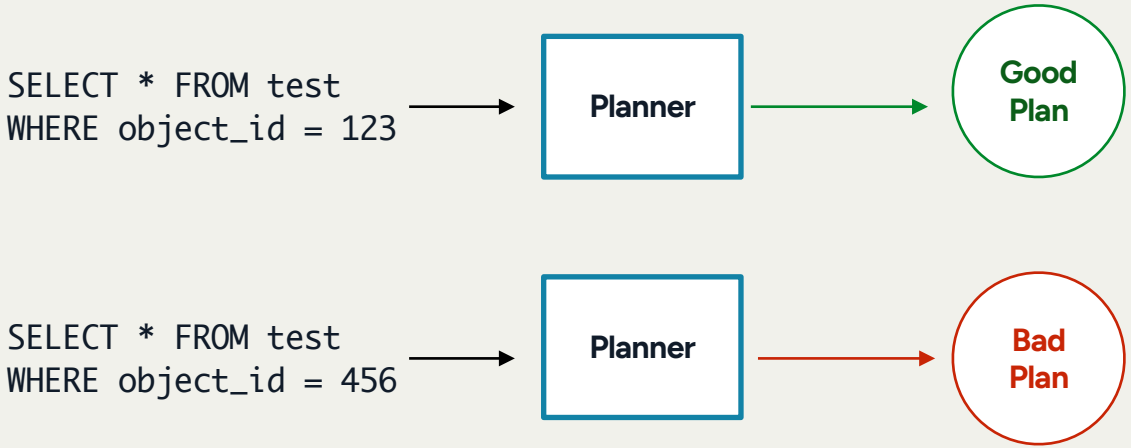
Control is necessary for debugging



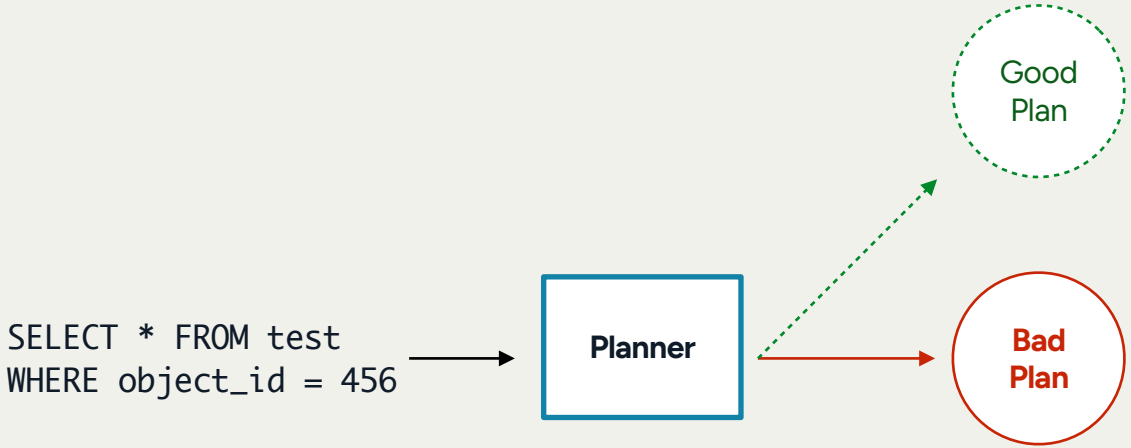
Control is necessary for debugging



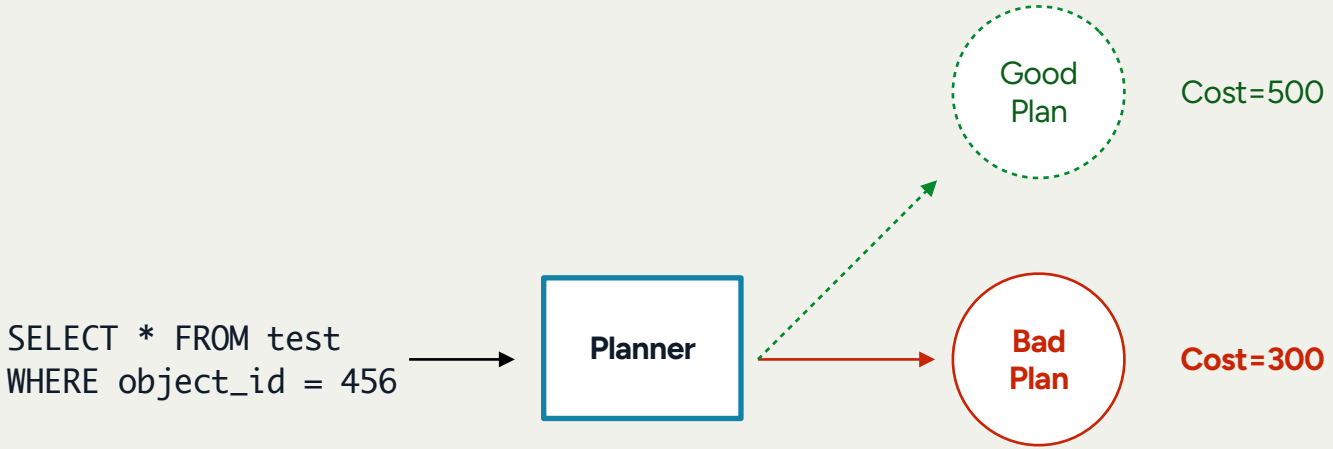
Control is necessary for debugging



Control is necessary for debugging



Control is necessary for debugging



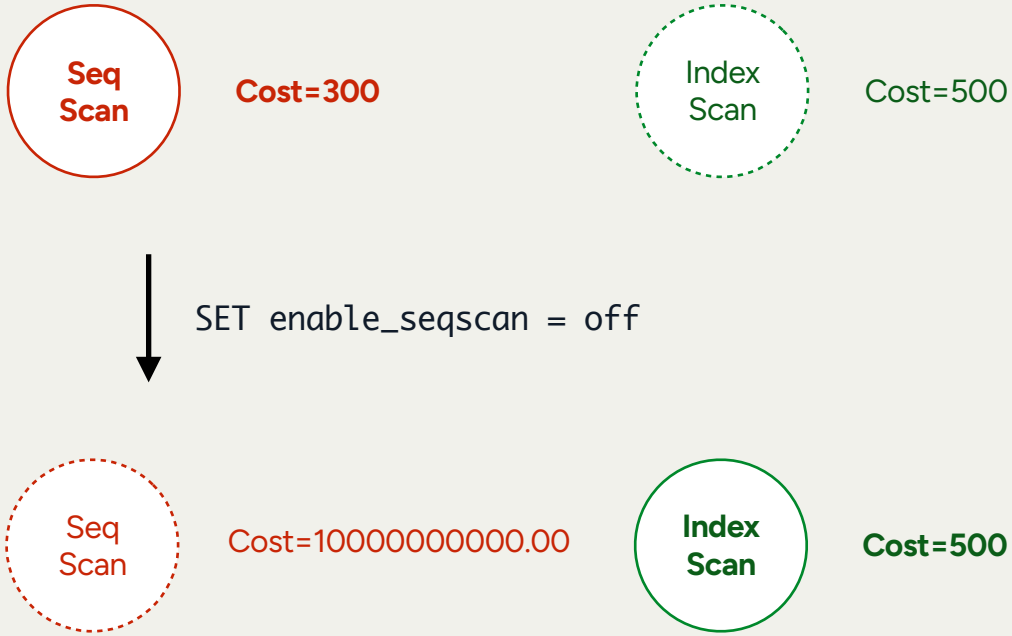
Control is necessary for debugging

pganalyze

The easiest test:

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

Control is necessary for debugging



Control is necessary for debugging

pganalyze

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

pg_hint_plan

For more complicated cases,

Utilize pg_hint_plan to force the good plan

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

Hints are enabled via comments, or the per-query hint table

```
 /*+
  HashJoin(a b)
  SeqScan(a)
 */
EXPLAIN SELECT *
  FROM pgbench_branches b
  JOIN pgbench_accounts a ON b.bid = a.bid
  ORDER BY a.aid;
```

```
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 width=0)
      ->  Index Scan using schema_tables_database_id_schema_name_table_name_
            (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 sch
            (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 sch
          (cost=0.43..372676.50 rows=23553966 width=8)
```

How to debug pg_hint_plan not working as expected

```
SET pg_hint_plan.debug_print = true;

/*+ NestedLoop(table1 table2) */ EXPLAIN SELECT * FROM …;

INFO:  pg_hint_plan: hint syntax error at or near "NestedLoop".
DETAIL:  Unrecognized hint keyword "NestedLoop".

                        QUERY PLAN
-----------------------------------------------------------
…
```

How to debug pg_hint_plan not working as expected

```
SET pg_hint_plan.debug_print = true;
SET client_min_messages = LOG;
```

```
/*+ NestLoop(table1 table2) IndexScan(table3) */
EXPLAIN SELECT * FROM table1
JOIN table2 ON (table2_id = table2.id) WHERE table1_id = '123';

LOG:  pg_hint_plan:
used hint:
NestLoop(table1 table2)
not used hint:
IndexScan(table3)
duplication hint:
error hint:
```

pganalyze

pg_plan_advice

pg_plan_advice

pganalyze

pg_plan_advice is the foundation for a new mechanism of achieving plan stability and control in Postgres 19+, [authored by Robert Haas](#).

Wednesday, March 04, 2026

pg_plan_advice: Plan Stability and User Planner Control for PostgreSQL?

I'm proposing a very ambitious patch set for PostgreSQL 19. Only time will tell whether it ends up in the release, but I can't resist using this space to give you a short demonstration of what it can do. The patch set introduces three new contrib modules, currently called `pg_plan_advice`, `pg_collect_advice`, and `pg_stash_advice`.

`pg_plan_advice` allows you to generate a "plan advice" string that describes the overall shape of the plan. You can then use that plan advice string to ensure that the same plan is recreated, or you can vary it to cause a different plan to be generated. Here's an example excerpted from the regression tests:

```
LOAD 'pg_plan_advice';
// ...test setup omitted for brevity...
```

About Me

Robert Haas

View my complete profile

Blog Archive

▼ 2026 (6)

- ▶ June (1)
- ▼ March (2)
  - Hacking Workshop for April/May 2026
  - pg_plan_advice: Plan Stability and User Planner Co...
- ▶ February (1)
- ▶ January (2)

pg_plan_advice

pganalyze

The core user-facing mechanic is very similar: We have a query whose plan we want to control, and we pass Postgres a string to take care of it.

Advice tags are similar in spirit to query hints, but hook in a at a different level.

Postgres 19 added the in-core mechanism in the planner, and **pg_plan_advice adds targeting to specific parts of a query** + the ability to set the active advice:

```
SET pg_plan_advice.advice = 'SEQ_SCAN(big_table)';
SELECT * FROM big_table WHERE id = 1;
```

But why is advice not set with a query comment?

pg_plan_advice

pganalyze

Using comments for hints/advice is challenging:

1. They don't work well in certain cases (functions, ECPG)
   => This is very hard to fix

2. Postgres treats comments as whitespace in its parser
   => This can likely be done, but it'd have made the patch a lot more complex

Luckily, pg_plan_advice is extensible!

pg_advice_comment

pganalyze

Introducing pg_advice_comment (released today), so we can stop talking about "But why is it not a hint-like query comment".

README License

pg_advice_comment - A little helper for pg_plan_advice to set advice from hint-style comments

Supply `pg_plan_advice` plan advice from a comment embedded in the query.

`pg_advice_comment` registers an *advisor* with `pg_plan_advice`. Before a query is planned, it scans the raw query text for a leading hint-style comment:

```
/*+ SEQ_SCAN(foo) JOIN_ORDER(foo bar) */
SELECT * FROM foo JOIN bar USING (id);
```

pg_plan_advice

Otherwise **pg_plan_advice** is very similar to **pg_hint_plan:**

Index Scans:
`INDEX_SCAN(table index_name)`

Join Order:
`JOIN_ORDER(a (b c) d)`

Join Method:
HASH_JOIN(a)

Where is the second table?
"Join method advice specifies the relation, or set of relations, that should appear on the inner side of a join using the named join method"

pg_plan_advice

Otherwise **pg_plan_advice** is very similar to **pg_hint_plan:**

Index Scans:
`INDEX_SCAN(table index_name)`

Join Order:
`JOIN_ORDER(a (b c) d)`

Join Method:
HASH_JOIN(a)

Where is the second table?
"Join method advice specifies the relation, or set of relations, that should appear on the inner side of a join using the named join method"

pg_plan_advice

pganalyze

Two more cool things about pg_plan_advice:

(1) It supports targeting complex queries very well:

F.30.3. Advice Targets

An *advice target* uniquely identifies a particular instance of a particular relation involved in a particular query. In simple cases, such as the examples shown above, the advice target is simply the relation alias. However, a more complex syntax is required when subqueries are used, when tables are partitioned, or when the same relation alias is mentioned more than once in the same subquery (e.g., `(foo JOIN bar ON foo.a = bar.a) x JOIN foo ON x.b = foo.b`). Any combination of these three things can occur simultaneously: a relation could be mentioned more than once, be partitioned, and be used inside of a subquery.

Because of this, the general syntax for a relation identifier is:

```
alias_name#occurrence_number/partition_schema.partition_name@plan_name
```

All components except for the `alias_name` are optional and are included only when required. When a component is omitted, the

pg_plan_advice

pganalyze

Two more cool things about pg_plan_advice:

(2) It gives you feedback on advice use via EXPLAIN:

```
SET pg_plan_advice.advice = 'JOIN_ORDER(x f d)';
EXPLAIN (COSTS OFF)
        SELECT * FROM join_fact f JOIN join_dim d ON f.dim_id = d.id;
                     QUERY PLAN
-----------------------------------------------------
 Nested Loop
   Disabled: true
   ->  Seq Scan on join_fact f
   ->  Index Scan using join_dim_pkey on join_dim d
         Index Cond: (id = f.dim_id)
 Supplied Plan Advice:
   JOIN_ORDER(x f d) /* partially matched */
```

pg_plan_advice

pganalyze

And some limitations:

- No control over row/selectivity estimates (being talked about for future releases)
- Partition advice has to be set for each partition, not just the parent table
- And some more issues we may find in Postgres 19 beta testing (please help out!)

Plan Management

How can I get Postgres to use a different plan **without changing my application?**

Aurora Query Plan Management

"With Aurora PostgreSQL query plan management, you can control how and when query execution plans change."

QPM has a mechanism to capture plan baselines, and approve/disapprove plans.

Rumors are its partially based on pg_hint_plan, and has limitations because of that.

But its hard to know for sure, since its not open-source.

Approving better plans

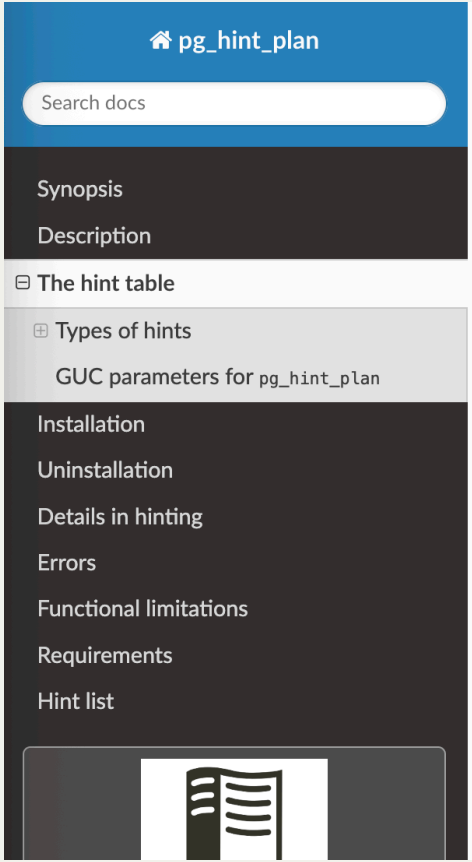
The following example demonstrates how to change the status of managed plans to approved using the `apg_plan_mgmt.evolve_plan_baselines` function.

```
SELECT apg_plan_mgmt.evolve_plan_baselines (
   sql_hash,
   plan_hash,
   min_speedup_factor := 1.0,
   action := 'approve'
)
FROM apg_plan_mgmt.dba_plans WHERE status = 'Unapproved';
```

```
NOTICE:     rangequery (1,10000)
NOTICE:     Baseline   [ Planning time 0.761 ms, Execution time 13.261 ms]
NOTICE:     Baseline+1 [ Planning time 0.204 ms, Execution time 8.956 ms]
NOTICE:     Total time benefit: 4.862 ms, Execution time benefit: 4.305 ms
NOTICE:     Unapproved -> Approved
evolve_plan_baselines
----------------------
0
(1 row)
```

pg_hint_plan has a hint table

pganalyze



The hint table

Hints can be specified in a comment, still this can be inconvenient in the case where queries cannot be edited. In the case, hints can be placed in a special table named `"hint_plan.hints"`. The table consists of the following columns:

column	description
`id`	Unique number to identify a row for a hint. This column is filled automatically by sequence.
`query_id`	A unique query ID, generated by the backend when the GUC compute_query_id is er
`application_name`	The value of `application_name` where sessions can apply a hint. The hint in the example below applies to sessions connected from psql. An empty string implies that all sessions will apply the hint.
`hints`	Hint phrase. This must be a series of hints excluding surrounding comment marks.

The following example shows how to operate with the hint table.

pg_hint_plan has a hint table

pganalyze

```
INSERT INTO hint_plan.hints(query_id, application_name, hints)
  VALUES (-7164653396197960701, '', 'SeqScan(t1)');
```

Hints always apply when:

- pg_hint_plan.enable_hint_table is on
- The query ID and application name matches
- You're executing a query in the database the extension is created

pg_hint_plan has a hint table

```
INSERT INTO hint_plan.hints(query_id, application_name, hints)
  VALUES (-7164653396197960701, '', 'SeqScan(t1)');
```

Query ID

You can get the Query ID from:

1. **pg_stat_statements**
2. **EXPLAIN (VERBOSE)**

```
EXPLAIN (VERBOSE) SELECT * FROM t1;
 ...
 Query Identifier: -7164653396197960701
(3 rows)
```

pg_stash_advice

pganalyze

pg_stash_advice: New contrib module in Postgres 19, companion to pg_plan_advice

F.33. pg_stash_advice — store and automatically apply plan advice

The `pg_stash_advice` extension allows you to stash **plan advice** strings in dynamic shared memory where they can be automatically applied. An `advice stash` is a mapping from **query identifiers** to plan advice strings. Whenever a session is asked to plan a query whose query ID appears in the relevant advice stash, the plan advice string is automatically applied to guide planning. Note that advice stashes are stored in dynamically allocated shared memory. This means that it is important to be mindful of memory consumption when deciding how much plan advice to stash. Optionally, advice stashes and their contents can automatically be persisted to disk and reloaded from disk; see `pg_stash_advice.persist`, below.

In order to use this module, you will need to execute `CREATE EXTENSION pg_stash_advice` in at least one database, so that you have access to the SQL functions to manage advice stashes. You will also need the `pg_stash_advice` module to be loaded in all sessions where you want this module to automatically apply advice. It will usually be best to do this by adding `pg_stash_advice` to **shared_preload_libraries** and restar the server.

Once you have met the above criteria, you can create advice stashes using the `pg_create_advice_stash` function described below and set the plan advice for a given query ID in a given stash using the `pg_set_stashed_advice` function. Then, you need only configure `pg_stash_advice.stash_name` to point to the chosen advice stash name. For some use cases, rather than setting this on a system-wide basis, yo may find it helpful to use `ALTER DATABASE ... SET` or `ALTER ROLE ... SET` to configure values that will apply only to a database or only to a certain role. Likewise, it may sometimes be better to set the stash nar

pg_stash_advice

pganalyze

1. Create a new advice stash:

```
SELECT pg_create_advice_stash('mystash');
```

2. Insert an entry into the stash, based on the query ID:

```
SELECT pg_set_stashed_advice(
  'mystash', '-71646533961979607O1', 'SEQ_SCAN(t1)');
```

3. Activate the stash to be used

```
SET pg_stash_advice.stash_name = 'mystash';
ALTER ROLE myrole SET pg_stash_advice.stash_name = 'mystash';
ALTER DATABASE ...;
```

How do I know what hint / plan advice to use?

pganalyze

With pg_hint_plan, you're out of luck (write your own hints)

pg_plan_advice

pganalyze

With pg_plan_advice, we get an ***extremely* useful feature**, that shows us the plan advice that would be needed to reproduce a query plan:

```
EXPLAIN (PLAN_ADVICE) SELECT * FROM t1;

                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..35.50 rows=2550 width=4)
 Generated Plan Advice:
   SEQ_SCAN(t1)
   NO_GATHER(t1)
(4 rows)
```

pg_stat_plans.plan_advice = on

pganalyze

Extensions can request advice generation from pg_plan_advice, and then retrieve the advice that would be needed.

pg_stat_plans now optionally shows you the advice for a given query plan:

```
SELECT planid, plan, plan_advice FROM pg_stat_plans LIMIT 1;
```

```
-[ RECORD 1 ]-------------------------------------------
planid      | -5265070082406572764
plan        | Index Scan using databases_pkey on databases
            |   Index Cond: (id = 10054854)
plan_advice | INDEX_SCAN(databases public.databases_pkey)
            | NO_GATHER(databases)
```

pg_stat_plans.plan_advice = on

pganalyze

This way, we know **which advice to use to reproduce a good plan:**

Same query:

```
SELECT $1 AS one FROM "servers" WHERE "servers"."organization_id" = $2 AND (servers.last_snapshot_at > now() - $3::interval) LIMIT $4
```

Different plans:

```
SELECT planid, plan, plan_advice FROM pg_stat_plans WHERE queryid = 3878558321776717777;
```

```
        planid         |                                   plan                                    |                 plan_advice
-----------------------+---------------------------------------------------------------------------+-----------------------------------------------------------
  5978205042974804435  | Limit                                                                    +| BITMAP_HEAP_SCAN(servers)                                +
                       |   ->  Bitmap Heap Scan on servers                                        +| NO_GATHER(servers)
                       |         Recheck Cond: (organization_id = '...'::uuid)                    +|
                       |         Filter: (last_snapshot_at > (now() - 'P1D'::interval))           +|
                       |         ->  Bitmap Index Scan on index_servers_on_organization_id        +|
                       |               Index Cond: (organization_id = '...'::uuid)                 |
 -2886304953344121656  | Limit                                                                    +| SEQ_SCAN(servers)                                        +
                       |   ->  Seq Scan on servers                                                +| NO_GATHER(servers)
                       |         Filter: ((organization_id = '...'::uuid) AND (last_snapshot_at > (now() - 'P1D'::interval)))  |
  5714062015821715388  | Limit                                                                    +| INDEX_SCAN(servers public.index_servers_on_organization_id)+
                       |   ->  Index Scan using index_servers_on_organization_id on servers       +| NO_GATHER(servers)
                       |         Index Cond: (organization_id = '...'::uuid)                      +|
                       |         Filter: (last_snapshot_at > (now() - 'P1D'::interval))            |
(3 rows)
```

Postgres is made by people.

Your feedback can help shape how this all works.

pganalyze

Thank you!

Learn more at pganalyze.com

Contact us: lukas@pganalyze.com