

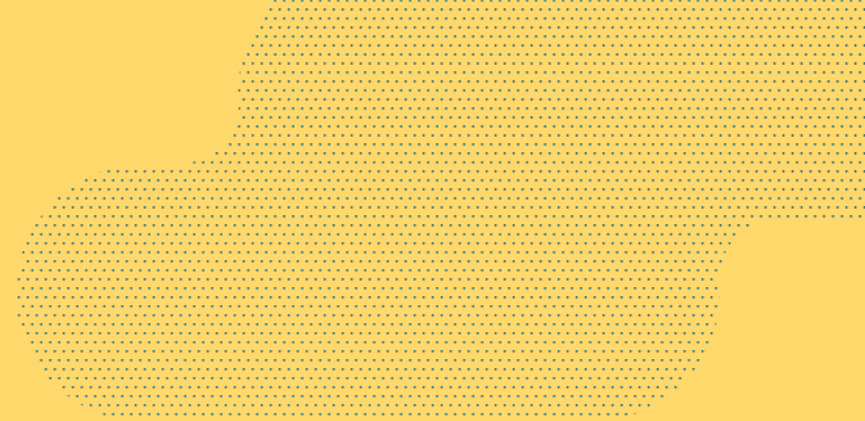


# Postgres Plan Monitoring and Management in Practice

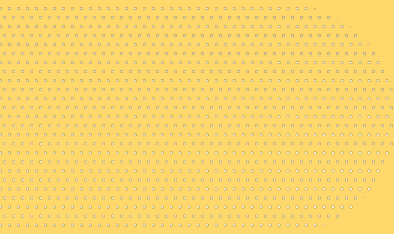
Presented by Lukas Fittl

# 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
- +
- **New in pganalyze:** Inefficient Index Use insights



# Plan Monitoring





# Query Plans can change over time

# Query Plans can change over time



**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
```

## 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



## 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
```

## 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!)

**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



"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

## **Query ID**

Differentiates by query structure.

## **Plan ID**

Differentiates by plan shape.

## 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)
```

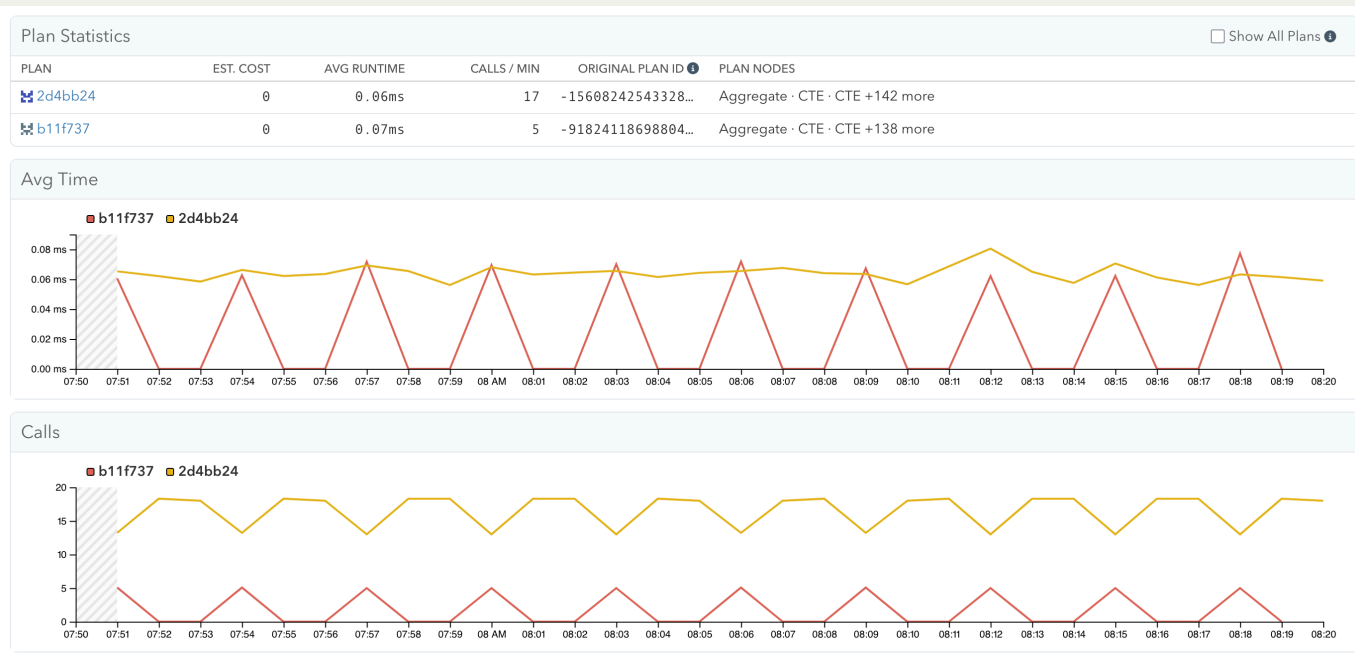
```
-> Bitmap Index Scan on index_users_lower_email
```

```
Index Cond: (lower((email)::text) = '...'::text)
```

# How to capture plan statistics



## Plan IDs let us track plan usage over time



# How to capture plan statistics



This is not a new idea.

A screenshot of a GitHub repository page for '2ndQuadrant / pg\_stat\_plans'. The page shows the repository name, navigation tabs (Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights), and a commit history section. The commit shown is 'Commit 4c8c749' by Peter Geoghegan, committed on Aug 17, 2012. The commit message reads: 'Initial commit of pg\_stat\_plans. Some rolling-back of functionality that clearly will only work with 9.2. The utility does not yet build against PostgreSQL 9.1.' The repository is public and has 65 watchers. The current branch is 'master', with 1 branch and 4 tags. There are buttons for 'Go to file', 'Add file', and 'Code'.

**The old `pg_stat_plans`  
is unmaintained.**

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

# How to capture plan statistics



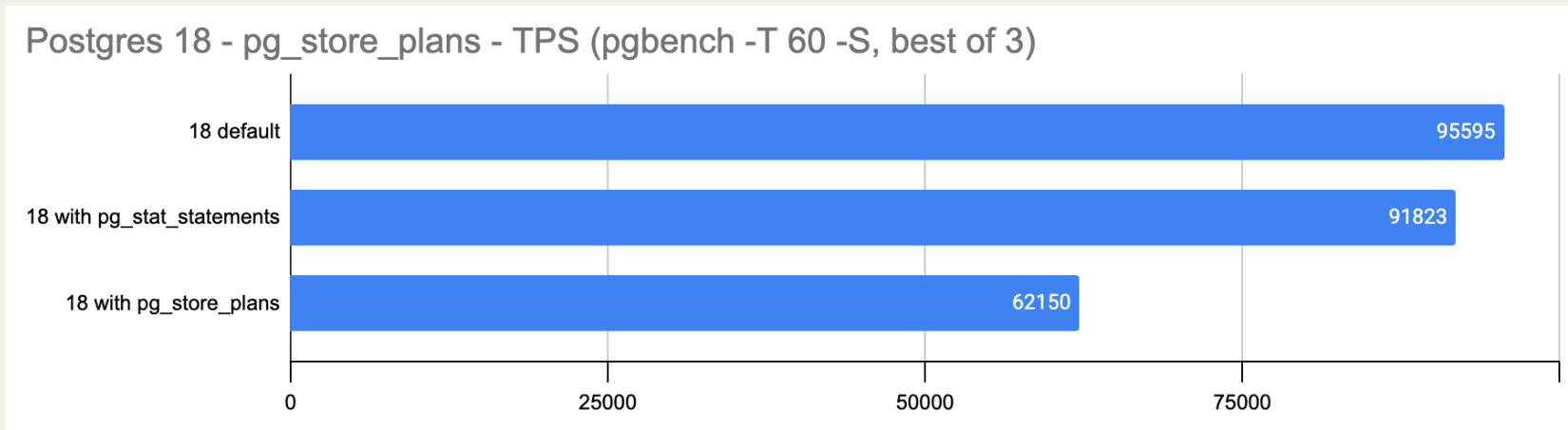
## pg\_store\_plans

```
master pg_store_plans / pg_store_plans.c
Code Blame 2485 lines (2153 loc) · 67.3 KB
107     typedef enum pgspVersion
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

## pg\_store\_plans



# How to capture plan statistics



## pg\_stat\_monitor

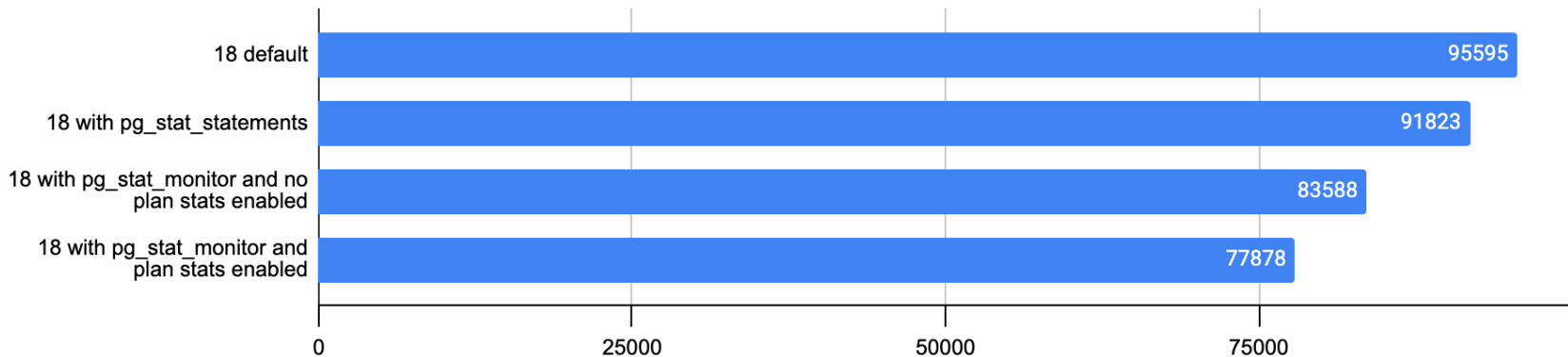
```
Code Blame 4041 lines (3486 loc) · 116 KB · 🔍
707
708     /* Extract the plan information in case of SELECT statement */
709     if (queryDesc->operation == CMD_SELECT && pgsm_enable_query_plan)
710     {
711         int rv;
712         MemoryContext oldctx;
713
714         /*
715          * Making 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 snprintf 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

## pg\_stat\_monitor

Postgres 18 - pg\_stat\_monitor - TPS (pgbench -T 60 -S, best of 3)



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

## **Plan ID calculation must be fast**

It should happen with every planning cycle.

**ExplainPrintPlan + hash(big text)**

~~ExplainPrintPlan + hash(big text)~~

## 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;

```

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

```
} IndexScan;
```

# 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 the  
    * 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 */
```

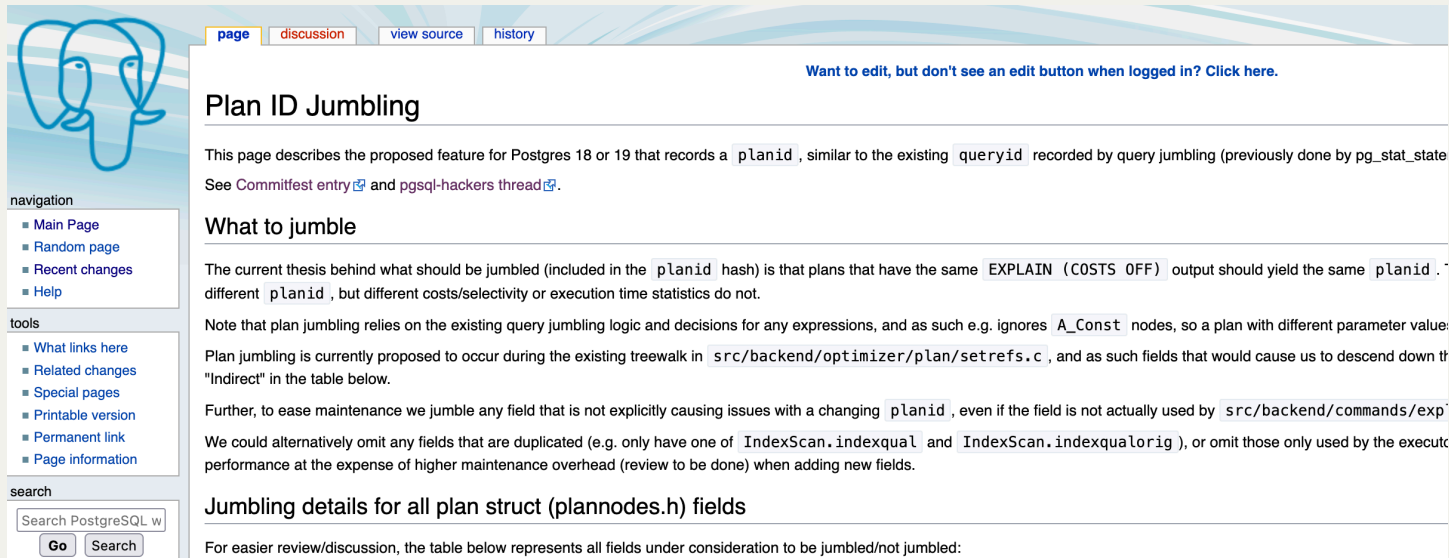
```
long           numGroups;
```

```
long           numGroups pg_node_attr(query_jumble_ignore);
```

```
/* for pass-by-ref transition data */
```

```
uint64         transitionSpace;
```

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



The screenshot shows a wiki page for "Plan ID Jumbling" on the PostgreSQL website. The page includes a navigation sidebar with links like "Main Page", "Random page", and "Help". The main content area has tabs for "page", "discussion", "view source", and "history". The text describes a proposed feature for Postgres 18 or 19 that records a `planid` field, similar to the existing `queryid`. It discusses the "What to jumble" criteria, noting that plans with the same `EXPLAIN (COSTS OFF)` output should yield the same `planid`, even if costs/selectivity or execution time statistics differ. It also mentions that plan jumbling relies on existing query jumbling logic and decisions for any expressions, and as such e.g. ignores `A_Const` nodes. The page further explains that 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 the "Indirect" in the table below. It also notes that 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`. Finally, it states that 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 execut 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
----------------	-------------------------	------------------

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

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

## 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 hook, giving extensions the possibility to assign an identifier related

```
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.

## 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

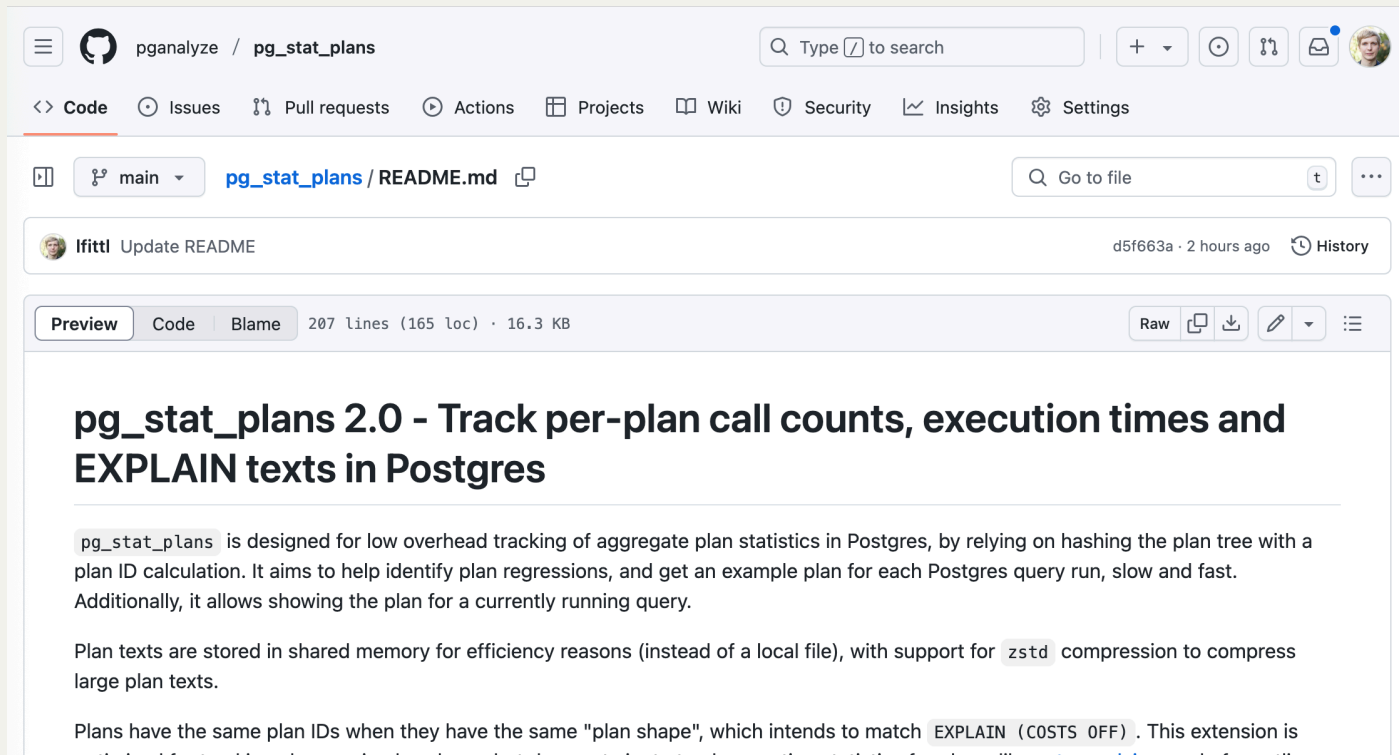
```
author    Michael Paquier <michael@paquier.xyz>
          Sun, 4 Aug 2024 10:41:24 +0000 (19:41 +0900)
committer Michael Paquier <michael@paquier.xyz>
          Sun, 4 Aug 2024 10:41:24 +0000 (19:41 +0900)
commit    7949d9594582ab49dee221e1db1aa5401ace49d4
tree      ad74385fbb0ef9f8b8d5a125d4b6e7ddc87ab20b   tree
parent    365b5a345b2680615527b23ee6bfa09a2f784f2   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`

## Initially released in September 2025:



The screenshot shows the GitHub repository page for `pganalyze / pg_stat_plans`. The repository is on the `main` branch, and the selected file is `pg_stat_plans / README.md`. The commit history shows a recent update by `lfittl` titled "Update README" with commit ID `d5f663a` from 2 hours ago. The file content is displayed in a preview mode, showing the title `pg_stat_plans 2.0 - Track per-plan call counts, execution times and EXPLAIN texts in Postgres`. The README text describes the extension's purpose: tracking aggregate plan statistics in Postgres with low overhead, identifying regressions, and showing the plan for a currently running query. It also mentions that plan texts are stored in shared memory for efficiency and support for `zstd` compression. The extension is designed to match the "plan shape" of `EXPLAIN (COSTS OFF)`.

pganalyze / pg\_stat\_plans

Type ↗ to search

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main pg\_stat\_plans / README.md Go to file

lfittl Update README d5f663a · 2 hours ago History

Preview Code Blame 207 lines (165 loc) · 16.3 KB Raw Copy Download Edit

## pg\_stat\_plans 2.0 - Track per-plan call counts, execution times and EXPLAIN texts in Postgres

`pg_stat_plans` is designed for low overhead tracking of aggregate plan statistics in Postgres, by relying on hashing the plan tree with a plan ID calculation. It aims to help identify plan regressions, and get an example plan for each Postgres query run, slow and fast. Additionally, it allows showing the plan for a currently running query.

Plan texts are stored in shared memory for efficiency reasons (instead of a local file), with support for `zstd` compression to compress large plan texts.

Plans have the same plan IDs when they have the same "plan shape", which intends to match `EXPLAIN (COSTS OFF)`. This extension is

## 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



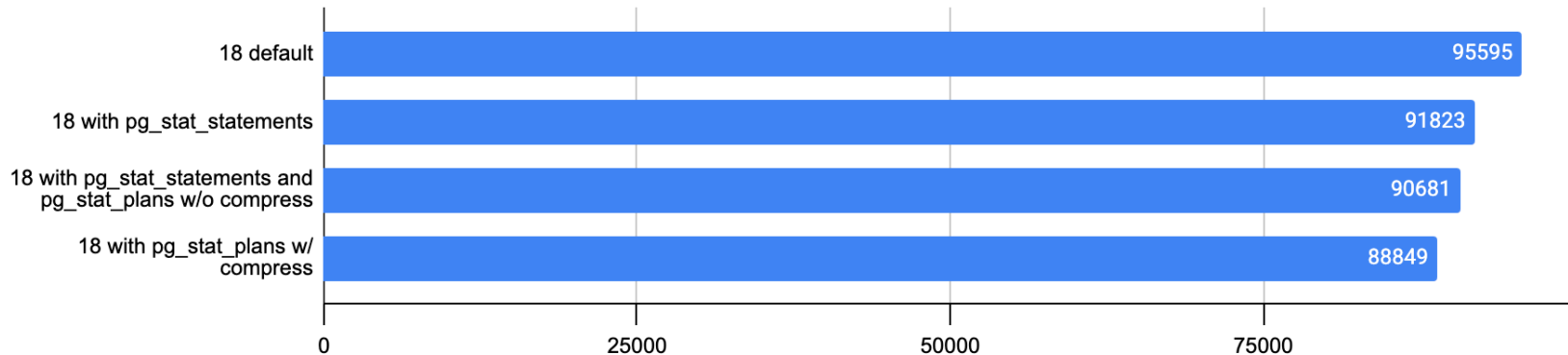
▼ README.md 📄 ↕

-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.
```

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

Postgres 18 - pg\_stat\_plans - TPS (pgbench -T 60 -S, best of 3)



# A new pg\_stat\_plans



**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 38  
            |           Index Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
```

# A new pg\_stat\_plans



## 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



**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          40  
              |           Index Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
```

# A new pg\_stat\_plans



## 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).coll1)) OR (ANY (id = (hashed SubPlan 2).coll1)) OR (ANY (id = (hashed SubPlan 3).coll1)))
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_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_1 organization_role_assignments_1
        -> Hash
          -> Seq Scan on organization_memberships_1 organization_memberships_1
          Filter: ((deleted_at IS NULL) AND accepted AND (user_id = 1))
        -> Index Only Scan using organization_roles_pkey on organization_roles_1 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_2 organization_role_assignments_2
          -> Hash
            -> Seq Scan on organization_permissions_2 organization_permissions_2
            Filter: (view AND (database_id IS NOT NULL))
          -> Index Scan using organization_memberships_pkey on organization_memberships_2 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_2 organization_roles_2
          Index Cond: (organization_role_id = organization_role_assignments_2.organization_role_id)
      -> Index Scan using databases_pkey on databases
```

## 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).coll1)) OR (ANY (id = (hashed SubPlan 2).coll1)) OR (ANY (id = (hashed SubPlan 3).coll1)))
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_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_1 organization_role_assignments_1
        -> Hash
          -> Seq Scan on organization_memberships_1 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



**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 this month!

## 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



**Now available in PGDG for Debian/Ubuntu and RPM-based systems:**

```
apt install pg-stat-plans  
yum install pg_stat_plans_18
```

**With big thanks to Christoph Berg and Devrim Gunduz!**

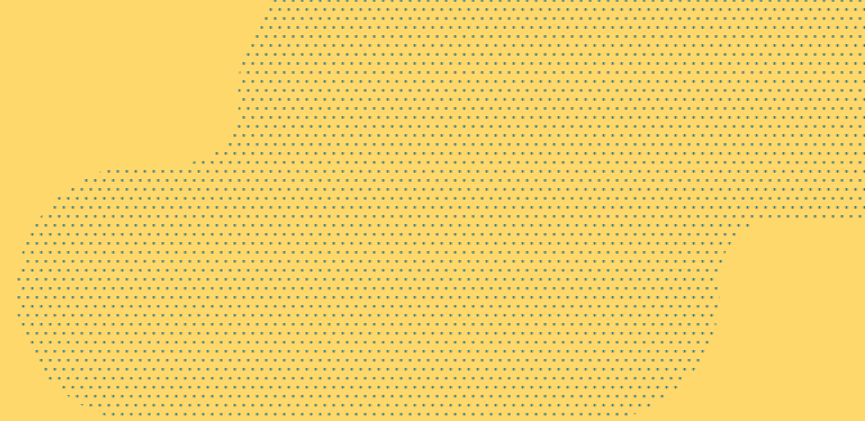


df7cb 2 hours ago

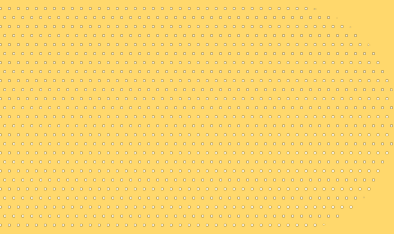
Thanks [@lfittl](#), everything works now 😊 .

## Integrates with pganalyze Plan Statistics





# Plan Control



# Why do we want to control query plans?



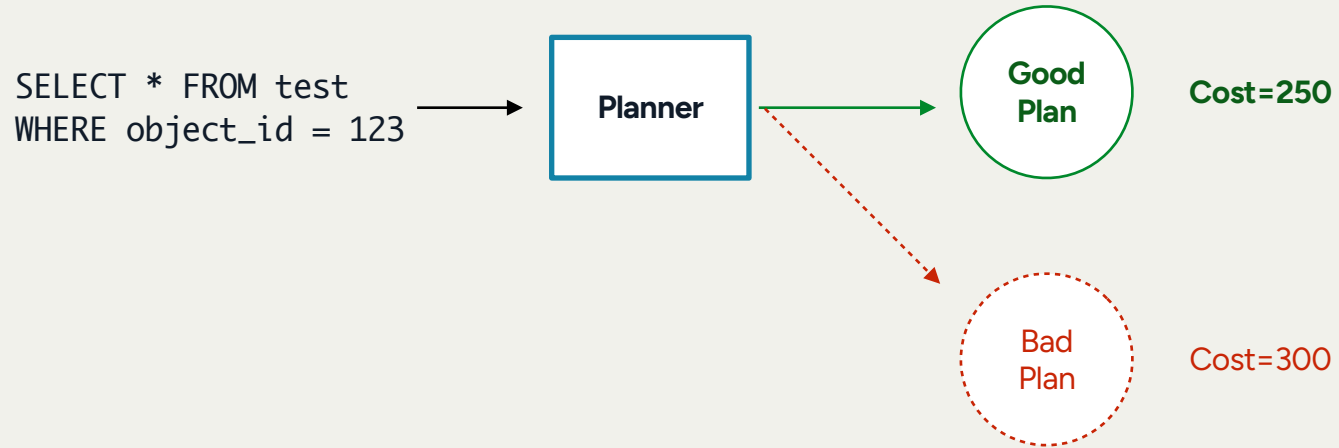
# Control is necessary for debugging

To Understand  
**Why A “Bad” Plan Was Chosen**  
Start By Forcing The Good Plan

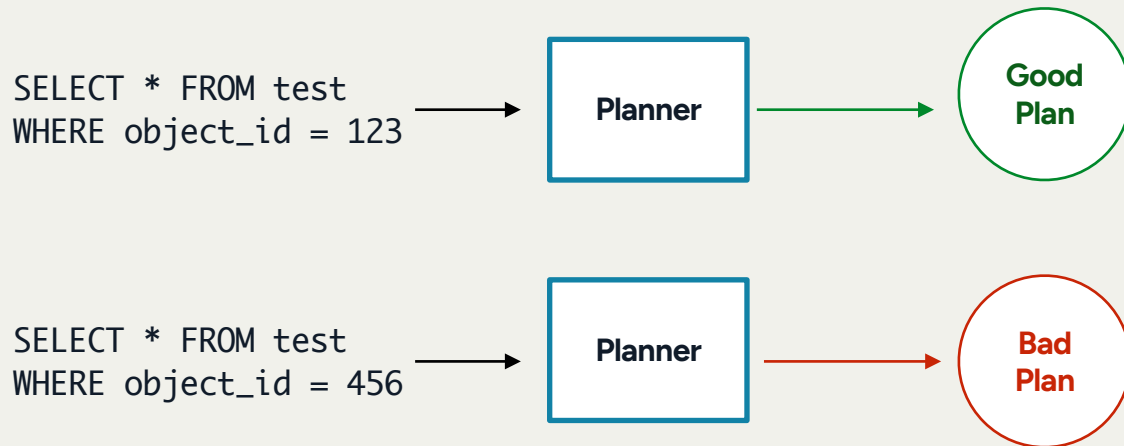
# Control is necessary for debugging



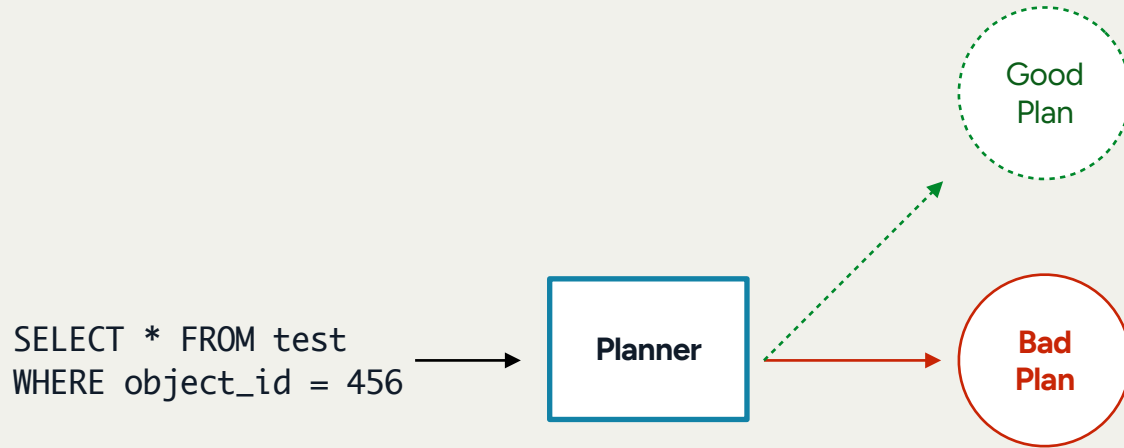
# Control is necessary for debugging



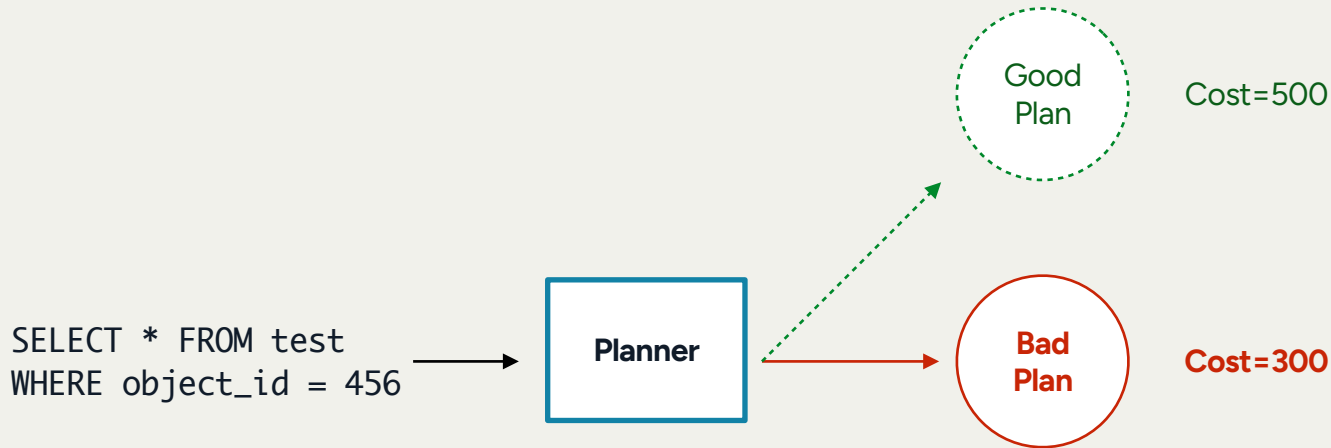
# Control is necessary for debugging



# Control is necessary for debugging



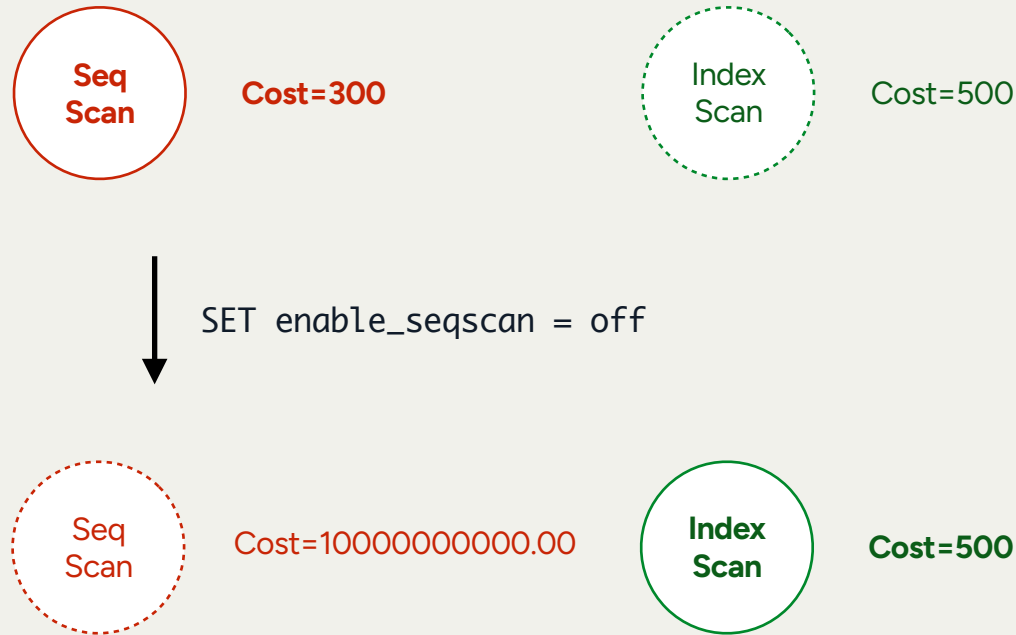
# Control is necessary for debugging



## The easiest test:

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

# Control is necessary for debugging



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

# Example: Inefficient Index Use

EXPLAIN Plan Select plan

Show  Est. Cost  Runtime  Rows  Buffers  Reads  Writes

Plan	Runtime
1 <a href="#">Index Scan (Forward) on schema_indices</a> <a href="#">query advisor/wrong index use</a>	19,161.14ms
on public.schema_indices using schema_indices_table_id_idx	
Filter: ((schema_indices.created_at <= \$1::timestamp with time zone) AND (schema_indices.database_id...	
Rows Removed by Filter: 27419	
Scan Direction: Forward	

Runtime	I/O Read Time
19,161.15ms	19,121.66ms
Read From Disk	Total Est. Cost
32.3 MB	261,059

Jun 18 09:31:53pm PDT · Plan Fingerprint: [e3aaa2d](#)

### Insights

**Query Advisor**

**1 Wrong Index Used**

- The index schema\_indices\_table\_id\_idx was used
- 27,419 rows were removed by filter after the index scan

[Create Workbook](#)

# Example: Inefficient Index Use

## Query #9873506765 - Wrong Index Use Testing








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

### All Query Plans

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

### Query Plans

Filter by Parameter Set... ▾

	PLAN	VARIANT	PARAMETER SET	EST. COST	RUNTIME
<input checked="" type="checkbox"/>	 ad34442	Baseline	Parameter Set 1	255,632	20.56ms
<input type="checkbox"/>	 a9916eb	Baseline	Parameter Set 2	1,038	11.72ms
<input type="checkbox"/>	 676ff34	Baseline	Parameter Set 3	205,188	180.80ms
<input checked="" type="checkbox"/>	 23b27c4	With removed_at index	Parameter Set 1	302,505	 0.62ms
<input type="checkbox"/>	 23b27c4	With removed_at index	Parameter Set 2	1,041	11.89ms
<input type="checkbox"/>	 23b27c4	With removed_at index	Parameter Set 3	210,188	132.84ms

+ New Query Variant

 All Query Plans

 Baseline

 With removed\_at index 

Query Advisor

No insights found

Baseline Query

[#9873506765](#)

Owner

panalyze

Workbook description

No description

Last edit

Jun 24 01:15:32pm PDT

# 20.56ms => 0.62ms

## Query #9873506765 - Wrong Index Use Testing

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

### Plan Comparison

Select plans

Compare **ⓘ**  Est. Cost  Runtime  Rows  Actual  Estimated  Buffers  Reads  Writes

Plan A	Plan B	Plan A	Plan B
Baseline - Parameter Set 1	With removed_at index - Parameter Set 1	Actual Rows	Actual Rows
-> Gather		0	
-> Bitmap Heap Scan on schema...		0	
-> Bitmap Index Scan <sup>2</sup>		27,448	
	-> Index Scan <sup>1</sup> on schema_indices		0

### Summary

Node Details

Node Source

#### Plan A: Baseline - Parameter Set 1

Seen At	Total Est. Cost	Runtime
Jun 24 01:13pm	255,632	20.56ms
<b>Plan Fingerprint</b>	<b>Read From Disk</b>	<b>I/O Read Time</b>
ad34442	0 B	0.00ms

#### Plan B: With removed\_at index - Parameter Set 1

Seen At	Total Est. Cost	Runtime
Jun 24 01:15pm	302,505	0.62ms
<b>Plan Fingerprint</b>	<b>Read From Disk</b>	<b>I/O Read Time</b>
23b27c4	0 B	0.00ms

### Index usage

#### A B Index

- ✓ 1. schema\_indices\_database\_id\_removed\_at\_idx
- ✓ 2. schema\_indices\_table\_id\_idx

# 255,632 cost units => 302,505 cost units

## Query #9873506765 - Wrong Index Use Testing






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

### All Query Plans

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

### Query Plans

Filter by Parameter Set... ▾

	PLAN	VARIANT	PARAMETER SET	EST. COST	RUNTIME
<input checked="" type="checkbox"/>	 ad34442	Baseline	Parameter Set 1	255,632	20.56ms
<input type="checkbox"/>	 a9916eb	Baseline	Parameter Set 2	1,038	11.72ms
<input type="checkbox"/>	 676ff34	Baseline	Parameter Set 3	205,188	180.80ms
<input checked="" type="checkbox"/>	 23b27c4	With removed_at index	Parameter Set 1	302,505	 0.62ms
<input type="checkbox"/>	 23b27c4	With removed_at index	Parameter Set 2	1,041	11.89ms
<input type="checkbox"/>	 23b27c4	With removed_at index	Parameter Set 3	210,188	132.84ms

+ New Query Variant

 All Query Plans

 Baseline

 With removed\_at index 

Query Advisor

No insights found

Baseline Query

[#9873506765](#)

Owner

panalyze

Workbook description

No description

Last edit

Jun 24 01:15:32pm PDT

# The best permanent fix is to reconsider indexes

Indexes							
Total Index Size	7.3 GB	Index Write Overhead ⓘ	0.17				
Name	Definition	Constraint	Valid?	First Seen	Last Used	Index Size	Index Write Overhead
<a href="#">schema_indices_database_id_idx</a>	btree (database_id) WHERE (removed_at IS NULL)		Yes ✓	2 years ago	10 hours ago	428.6 MB	0.01
<a href="#">schema_indices_database_id_removed_at_idx</a>	btree (database_id, removed_at)		Yes ✓	8 months ago	10 hours ago	443.4 MB	0.07
<a href="#">schema_indices_pkey</a>	btree (id)	PRIMARY KEY (id)	Yes ✓	2 years ago	10 hours ago	2.8 GB	0.06
<a href="#">schema_indices_table_id_idx</a>	btree (table_id) WHERE ((removed_at IS NULL) AND (NOT is_valid))		Yes ✓	2 years ago	10 hours ago	984 kB	0.01
<a href="#">schema_indices_table_id_name_idx</a>	btree (table_id, name) WHERE (removed_at IS NULL)	UNIQUE	Yes ✓	2 years ago	10 hours ago	3.6 GB	0.02

Tiny partial index incorrectly used



# 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;
```

# Example: Forcing index use

With removed\_at index

With parameter names ▾

SQL Query

Unified Diff

Split Diff

```
1 /*+ IndexScan(schema_indices schema_indices database id removed at idx) */
2 SELECT schema_indices.*
3 FROM schema_indices
4 WHERE
5     schema_indices.database_id = $database_id
6     AND schema_indices.removed_at IS NULL
7     AND schema_indices.is_valid = $is_valid
8     AND schema_indices.created_at <= $created_at
9
```

```
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:
```



# pg\_plan\_advice

**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)

**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 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?**

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!**

Introducing [pg\\_advice\\_comment](#) (released this month), 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);
```



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)
```

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"

## 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

## 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

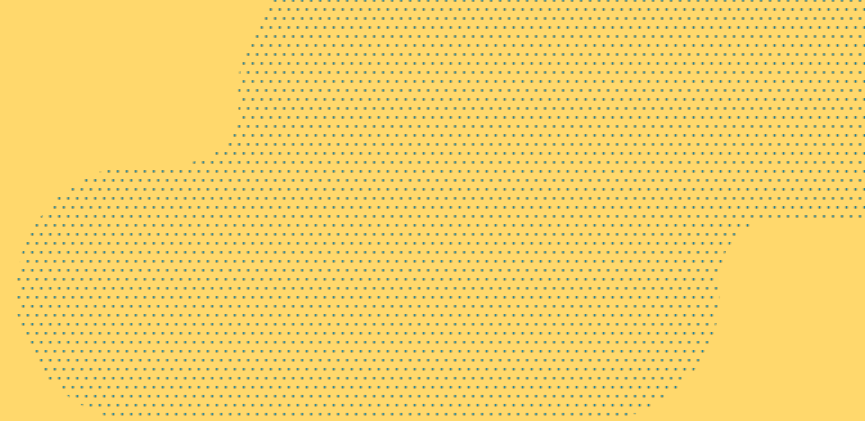
      Index Cond: (id = f.dim\_id)

Supplied Plan Advice:

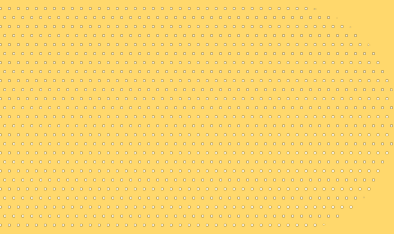
  JOIN\_ORDER(x f d) /\* partially matched \*/

## 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?**

"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)
```



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

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

```
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: New contrib module in Postgres 19, companion to pg\_plan\_advice

[Documentation](#) → [PostgreSQL 19](#)

Development Versions: [19](#) / [devel](#)

This documentation is for an unsupported version of PostgreSQL.

You may want to view the same page for the **current** version, or one of the other supported versions listed above instead.

F.33. pg\_stash\_advice — store and automatically apply plan advice

[Prev](#)

[Up](#)

Appendix F. Additional Supplied Modules and Extensions

[Home](#)

### F.33. pg\_stash\_advice — store and automatically apply plan advice

[F.33.1. Functions](#)

[F.33.2. Configuration Parameters](#)

[F.33.3. Author](#)

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 restarting 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, you 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 name

## 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', '-7164653396197960701', '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?

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

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)
```

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)
```

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 = 387855832177671777;
```

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

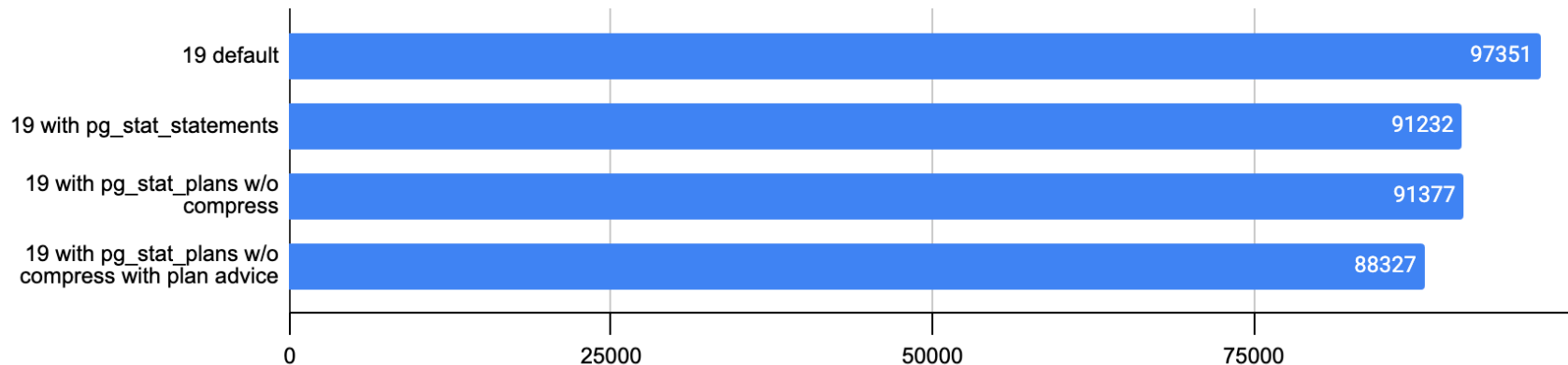
(3 rows)

# pg\_stat\_plans.plan\_advice = on



This has a minor overhead, similar to turning compression on:

Postgres 19 - pg\_stat\_plans - TPS (pgbench -T 60 -S, best of 3)





# New in pganalyze: Inefficient Index Use insights



# New Query Advisor Insight: Inefficient Index Use



[pganalyze.com/docs/query-advisor/insights/inefficient-index-use](https://pganalyze.com/docs/query-advisor/insights/inefficient-index-use)

## Overview

### Severity



Info

### Check Frequency



Daily

### Last Updated

2026-06-25 08:26:12am PDT

### State



Triggered

Acknowledge

### Description

Tuning opportunity found for query [#44015156](#)

< Insight 1 of 1: Inefficient Index Use >

Found in plan: e3aaa2d

**Pattern Detected:** An index scan is reading many rows that are later discarded by a filter, suggesting a different index would be a better match for this query. Try using a `pg_hint_plan` hint to force the planner to use a different existing index.

[Learn more](#) about this insight.

## Current Query

Impact ■■■■

```
1 /*job:Issues::CheckUpWorker, line:/app/services/checks/schema/index_invalid.rb:15:in
   'block in
   Checks::Schema::IndexInvalid#call', sentry_trace_id:13aa3ed33d244659b01cedb9e32d1fd6, traces
   tate:pganalyze=t:1782259230.0350888*/
2 SELECT schema_indices.*
3 FROM schema_indices
4 WHERE
5   schema_indices.database_id = $1
6   AND schema_indices.removed_at IS NULL
7   AND schema_indices.is_valid = $2
8   AND schema_indices.created_at <= $3
9
```

## Suggested Query Rewrite [Show Preview](#)

Create Workbook

The planner picked an index that doesn't match the filter conditions of this query, so Postgres reads many rows from the index and then discards most of them. A different existing index on the filter column(s) could let the scan return fewer rows.

One way to address this is to force a specific index using a `pg_hint_plan` `IndexScan` hint, covered in the steps below.

### Identify a better index

The planner picked `schema_indices_table_id_idx`, but the filter conditions of this query (`created_at`, `database_id`) aren't covered by that index, so Postgres reads many rows and discards them at the Filter step.

Alternative indexes on this table that could be used instead:

- `schema_indices_database_id_removed_at_idx`

Whether one of these is faster than the current choice depends on how selective your filter values are in practice

# New Query Advisor Insight: Inefficient Index Use



[pganalyze.com/docs/query-advisor/insights/inefficient-index-use](https://pganalyze.com/docs/query-advisor/insights/inefficient-index-use)

EXPLAIN Plan ⚙ Select plan

Show  Est. Cost  Runtime  Rows  Buffers  Reads  Writes

Plan	Actual Rows	Est. Rows
1 <a href="#">🔍</a> Index Scan (Forward) on schema_indices <a href="#">query_advisor/inefficient_index_use</a>	0	1,882

on public.schema\_indices using schema\_indices\_table\_id\_idx  
Filter: ((schema\_indices.created\_at <= \$1::timestamp with time zone) AND (schema\_indices.database\_id = \$2::integer))  
Rows Removed by Filter: 27419  
Scan Direction: Forward

Runtime 19,161.15ms I/O Read Time 19,121.66ms  
Read From Disk 32.3 MB Total Est. Cost 261,059

Jun 18 09:31:53pm PDT · Plan Fingerprint: [e3aaa2d](#)

### Insights

**Query Advisor**

**1** [Inefficient Index Use](#)

- The index schema\_indices\_table\_id\_idx was used
- 27,419 rows were removed by filter after the index scan

[Create Workbook](#)

# New Query Advisor Insight: Inefficient Index Use



## Use pganalyze Workbooks to validate with EXPLAIN ANALYZE + pg\_hint\_plan

Query #9873506765 - Wrong Index Use Testing

Overview Compare Plans Parameter Sets Activity

All Query Plans

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

Query Plans Filter by Parameter Set...

PLAN	VARIANT	PARAMETER SET	EST. COST	RUNTIME
<input checked="" type="checkbox"/> ad34442	Baseline	Parameter Set 1	255,632	20.56ms
<input type="checkbox"/> a9916eb	Baseline	Parameter Set 2	1,038	11.72ms
<input type="checkbox"/> 676ff34	Baseline	Parameter Set 3	205,188	180.80ms
<input checked="" type="checkbox"/> 23b27c4	With removed_at index	Parameter Set 1	302,505	0.62ms
<input type="checkbox"/> 23b27c4	With removed_at index	Parameter Set 2	1,041	11.89ms
<input type="checkbox"/> 23b27c4	With removed_at index	Parameter Set 3	210,188	132.84ms

20.56ms => 0.62ms

+ New Query Variant

All Query Plans

- Baseline
- With removed\_at index

Query Advisor

No insights found

Baseline Query

#9873506765

Owner

pganalyze

Workbook description

No description

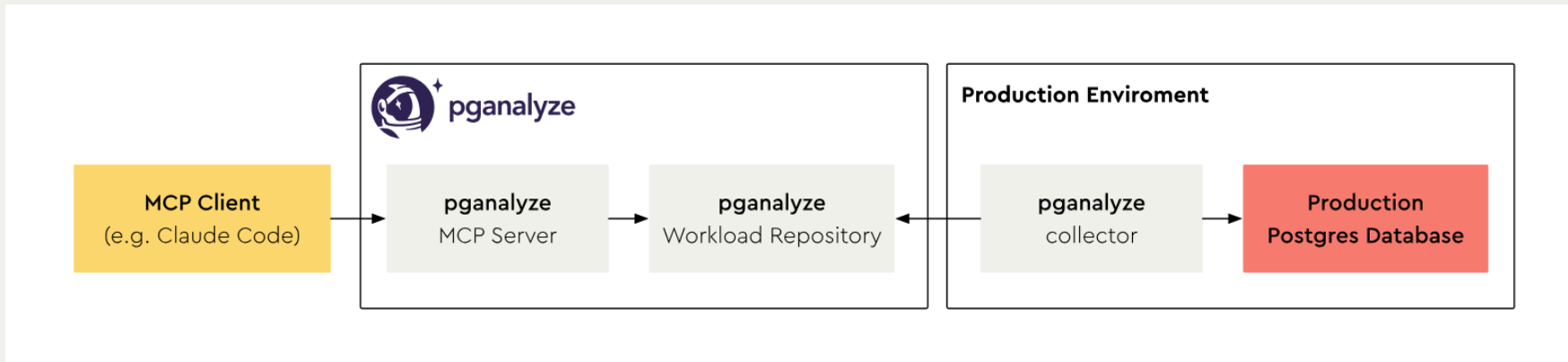
Last edit

Jun 24 01:15:32pm PDT

# New Query Advisor Insight: Inefficient Index Use



Use pganalyze MCP Server to quickly access insight information



[pganalyze.com/docs/mcp](https://pganalyze.com/docs/mcp)

# New Query Advisor Insight: Inefficient Index Use



## Use pganalyze MCP Server to quickly access insight information

does pganalyze report any index use problems for queries on prod-db-main?

- I'll check pganalyze for index use problems on prod-db-main. Let me load the relevant tools first.
- I'll find the prod-db-main server first.  
Called pganalyze-staging (ctrl+o to expand)
- Found it (qlosi5joffcpcfppzmz6aufoiga). Now let me check for index-use issues.  
Called pganalyze-staging (ctrl+o to expand)
- Large result – let me have a subagent extract the index-use findings without flooding context.
- **Agent**(Extract index-use issues)
  - └ Done (8 tool uses · 54.6k tokens · 52s)  
(ctrl+o to expand)
- Yes. pganalyze reports **4 inefficient index use problems** on prod-db-main (all on database 23, severity info, under the "Advisor Insights" check):

Issue	Query	Samples	Max runtime	Insight
24880 ( <a href="https://staging.pganalyze.com/databases/23/checks/queries/advisor_insight/24880">https://staging.pganalyze.com/databases/23/checks/queries/advisor_insight/24880</a> )	#44015156 – SELECT schema_indices.* FROM schema_indices WHERE ...	2	19,161 ms	InefficientIndexUse
24881 ( <a href="https://staging.pganalyze.com/databases/23/checks/queries/advisor_insight/24881">https://staging.pganalyze.com/databases/23/checks/queries/advisor_insight/24881</a> )	#43933950 – WITH scans_and_clauses AS (...) ...	10	14,236 ms	InefficientIndexUse
24882 ( <a href="https://staging.pganalyze.com/databases/23/checks/queries/advisor_insight/24882">https://staging.pganalyze.com/databases/23/checks/queries/advisor_insight/24882</a> )	#43957759 – WITH scans_and_clauses AS (...) ...	10	4,235 ms	InefficientIndexUse
24883 ( <a href="https://staging.pganalyze.com/databases/23/checks/queries/advisor_insight/24883">https://staging.pganalyze.com/databases/23/checks/queries/advisor_insight/24883</a> )	#43957758 – WITH scans_and_clauses AS (...) ...	1	947 ms	InefficientIndexUse

These are the "wrong/inefficient index use" insights (the feature on your current wrong-index-use-insight branch). The top one (#44015156 on schema\_indices, ~19s) is the highest-impact.

For broader context, the same server also has other index-related findings that are not inefficient-index-use:

- **Missing indexes:** 14 issues (notably 6 insights on issues, 3 on users)
- **Unused indexes:** 14 issues
- **Invalid indexes:** none

Want me to pull the EXPLAIN / advisor detail for any of these – e.g. #44015156 – to see what index it's picking vs. what it should use?

# New Query Advisor Insight: Inefficient Index Use



## Coming Soon: Validate Insights using Workbooks & pganalyze MCP Server

Reach out to join early access!

- The rewrite applies the classic NestedLoop fix: it hoists `schema_tables` (the `at` reference) into a `__schema_tables AS MATERIALIZED` CTE and joins against that instead. Materializing breaks the row-count misestimate that was feeding the nested loop, which should push the planner toward a hash join and avoid discarding those 268k rows.

Let me create this as a variant and benchmark it against the baseline. I'll use the `$alias`-style query and tag it with the applied insight from the baseline result (596).

Called `pganalyze-local` (ctrl+o to expand)

- Benchmark complete. Both the baseline and the insight-driven rewrite ran successfully against database 10054854. Here's how they compare:

**Benchmark: NestedLoop insight rewrite**

Workbook: <http://localhost:5100/databases/10054854/workbooks/269>

**Benchmark: NestedLoop insight rewrite**

Workbook: <http://localhost:5100/databases/10054854/workbooks/269>

\*  
**Thank you!**



## WHAT'S NEW IN PGANALYZE

- MCP Server — now in Public Preview (Claude Code, Codex & Cursor)
- Server Groups — cluster-wide Query Performance across primaries & replicas
- Plan Statistics — track which query plans run over time, powered by pg\_stat\_plans
- Query Advisor — new insights for inefficient index use and queries spilling to disk

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

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