



The Future of Plan Management and Feedback Loops

Presented by Lukas Fittl

Agenda for today

- **Control plans** with `pg_plan_advice`
- **Managed plans** with `pg_stash_advice`
- **Monitor plans** with `pg_stat_plans` + `pg_plan_advice`
- What could Plan Management look like?
- What could Plan Feedback Loops look like?



Plan Control with pg_plan_advice

Why do we want to control query plans?



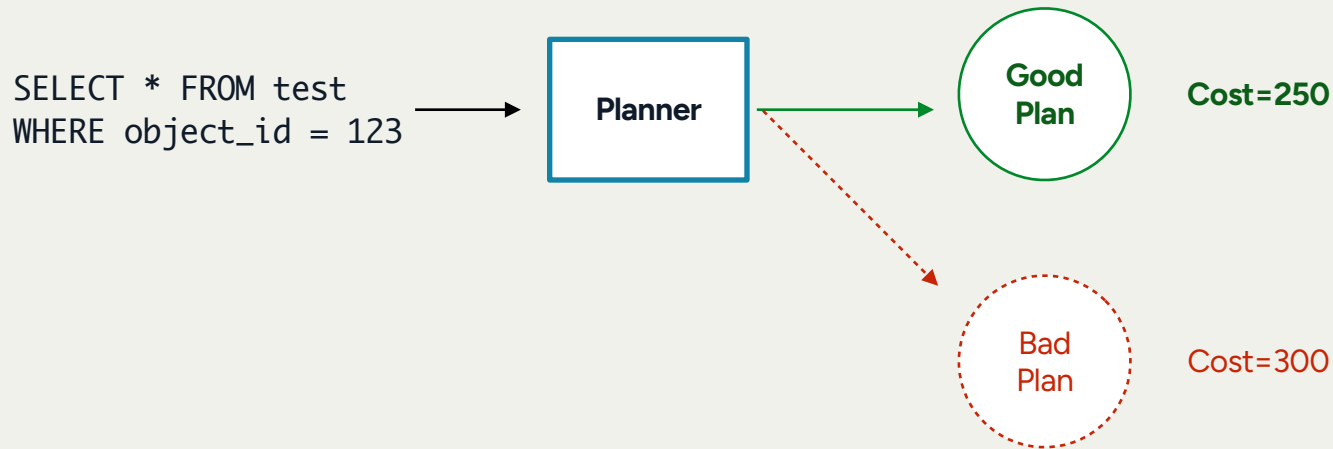
Control is necessary for debugging

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

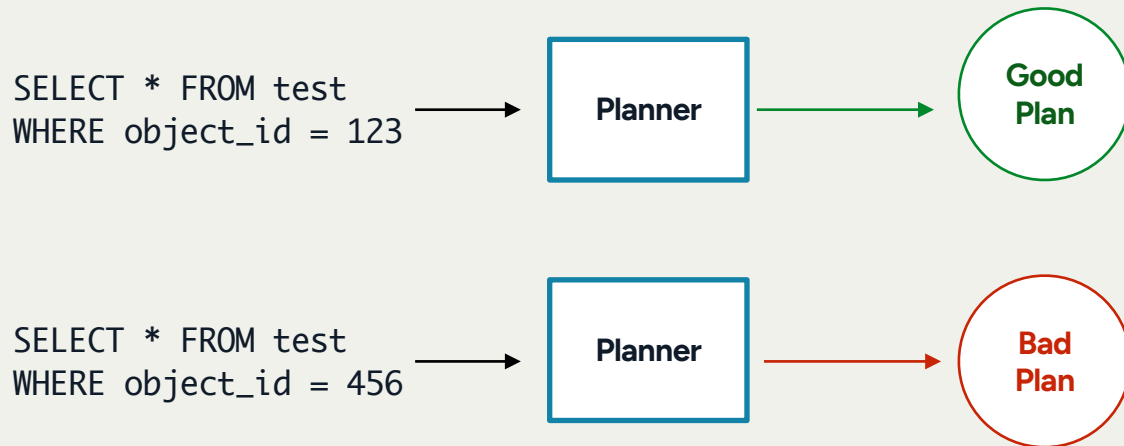
Control is necessary for debugging



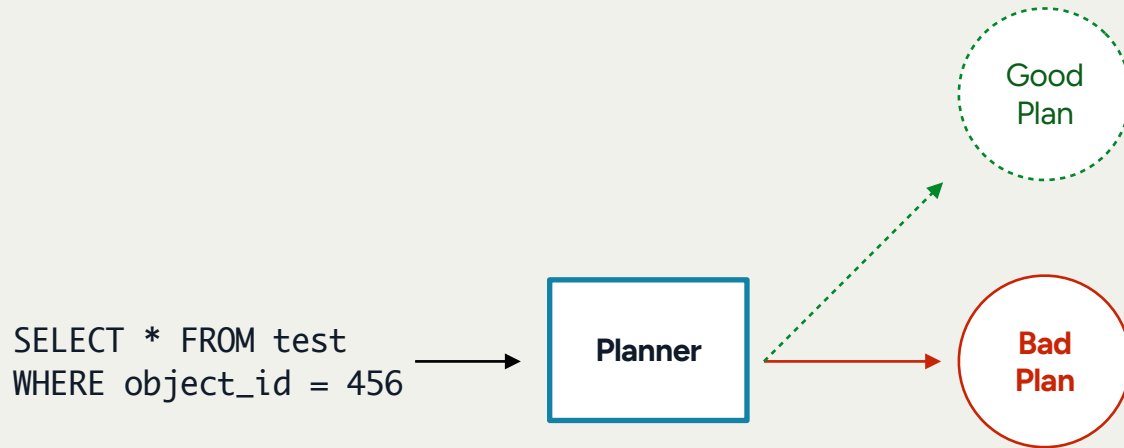
Control is necessary for debugging



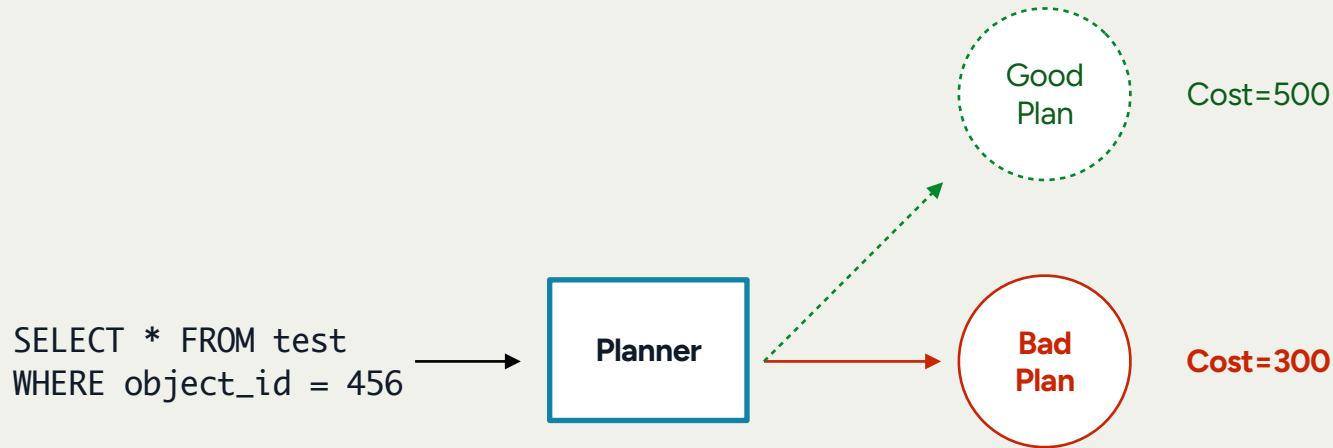
Control is necessary for debugging



Control is necessary for debugging



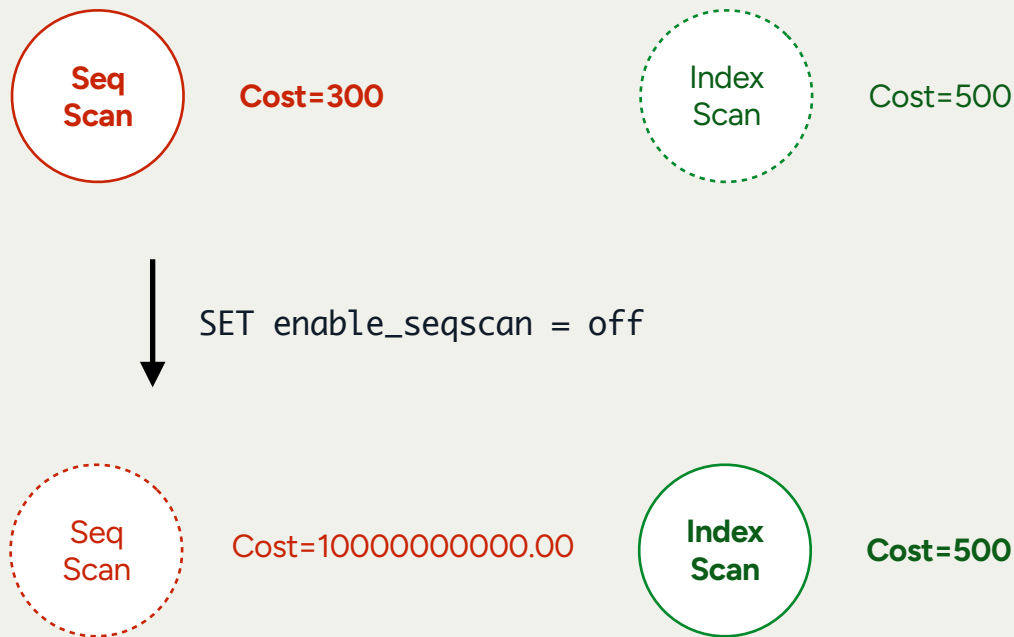
Control is necessary for debugging



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



The old way: `pg_hint_plan`

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;
```

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:
```





The Postgres 19 way: **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!

New Experimental Extension #1: [pg_advice_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);
```



The text found between `/*+` and `*/` is passed to `pg_plan_advice`, which parses it with the same parser it uses for the `pg_plan_advice` setting.

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
- Partition advice has to be set for each partition, not just the parent table
- And some more issues we may find in Postgres 19 release testing (please help out!)



Basic Plan Management with pg_stash_advice

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

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.

pg_hint_plan has a 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.

[Prev](#)

[Up](#)

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

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 ...;
```

pg_stash_advice Limitations:

- Query IDs can have hash collisions with entirely unrelated queries
- Query IDs can be the same for queries referencing different tables, e.g. when using the same alias for simple queries (PG18+)
- No mechanism for invalidating advice when the schema changes
- Stashed advice is not copied to replicas (but it is crash safe)

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

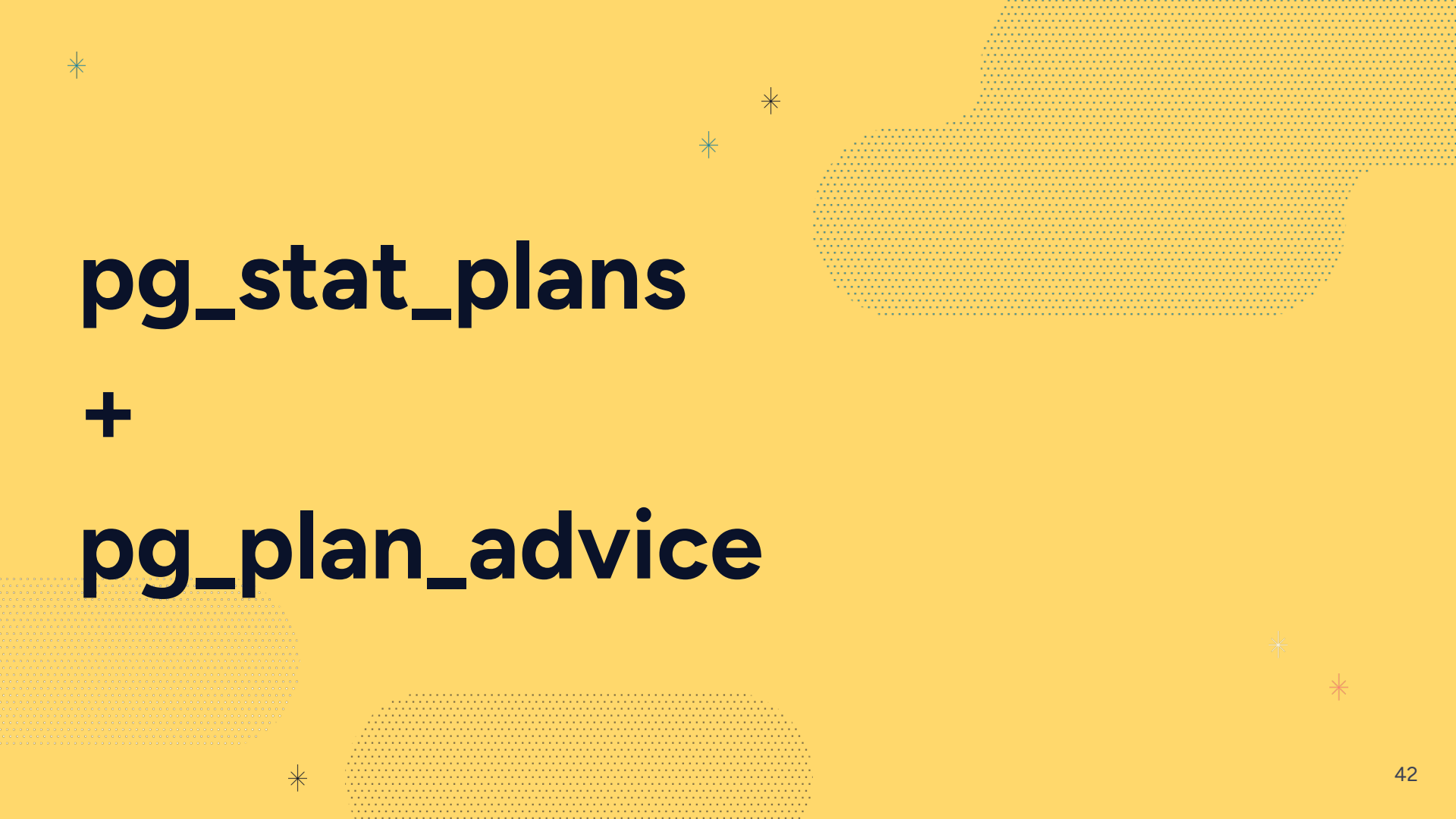
Seq Scan on t1 (cost=0.00..35.50 rows=2550 width=4)

Generated Plan Advice:

SEQ_SCAN(t1)

NO_GATHER(t1)

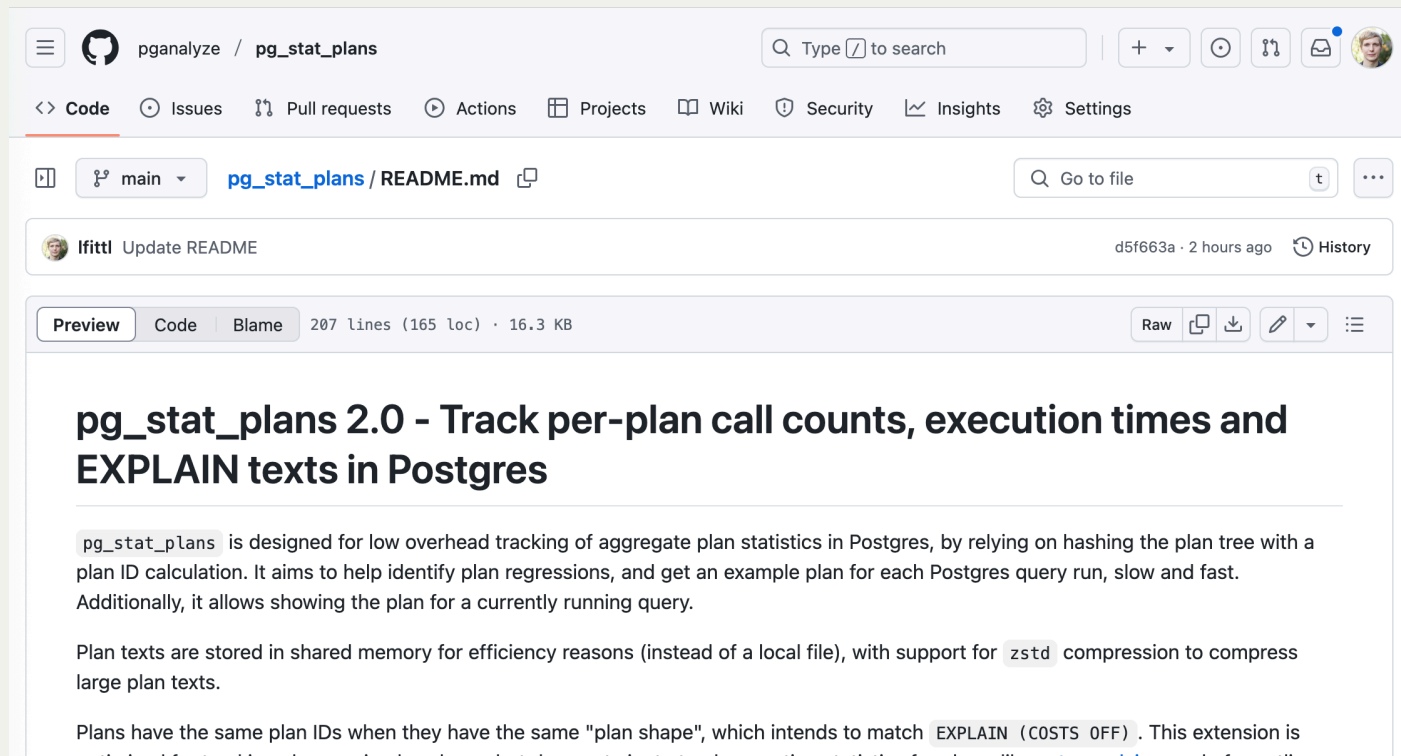
(4 rows)



pg_stat_plans + pg_plan_advice

What is pg_stat_plans?

The new `pg_stat_plans` was initially [released](#) in September 2025:



The screenshot shows the GitHub repository for `pganalyze / pg_stat_plans`. The repository is on the `main` branch, and the selected file is `pg_stat_plans / README.md`. The commit is by `Ifitti` with the message "Update README" and a timestamp of "2 hours ago". The file statistics show 207 lines (165 loc) and 16.3 KB. The README content is as follows:

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

A new pg_stat_plans



And 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)
10     10      extension (unmaintained), with a goal of upstreaming parts of this extension into the core Postgres project over time.
11     11      -**Experimental**. May still change in incompatible ways without notice. Not (yet) recommended for production use.
12     12      -
13     13      ## Supported PostgreSQL versions
14     14
15     15      Requires at least Postgres 16.
```

A new pg_stat_plans



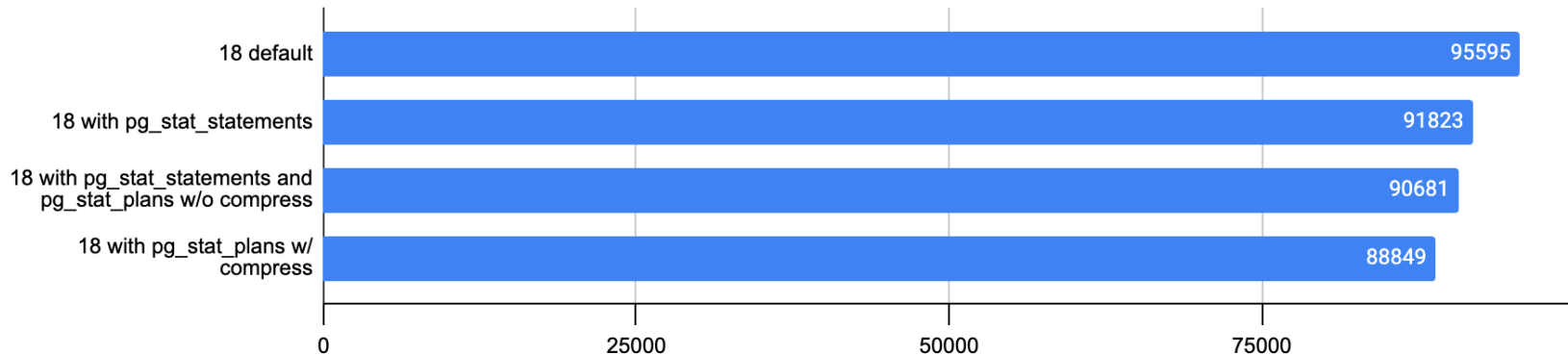
Cumulative statistics on which query ID used which plan, how often (calls), and how long it took (total_exec_time).

```
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          46
              |                    Index Cond: (server_id = '00000000-0000-0000-0000-000000000000'::uuid)
```

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

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 Gündüz!



df7cb 2 hours ago

Thanks @lfittl, everything works now 😊 .

pg_stat_plans 2.1.1 has been released!

What's new:

- Support for Postgres 19 beta 4
- 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

Integrating with pg_plan_advice

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) + +

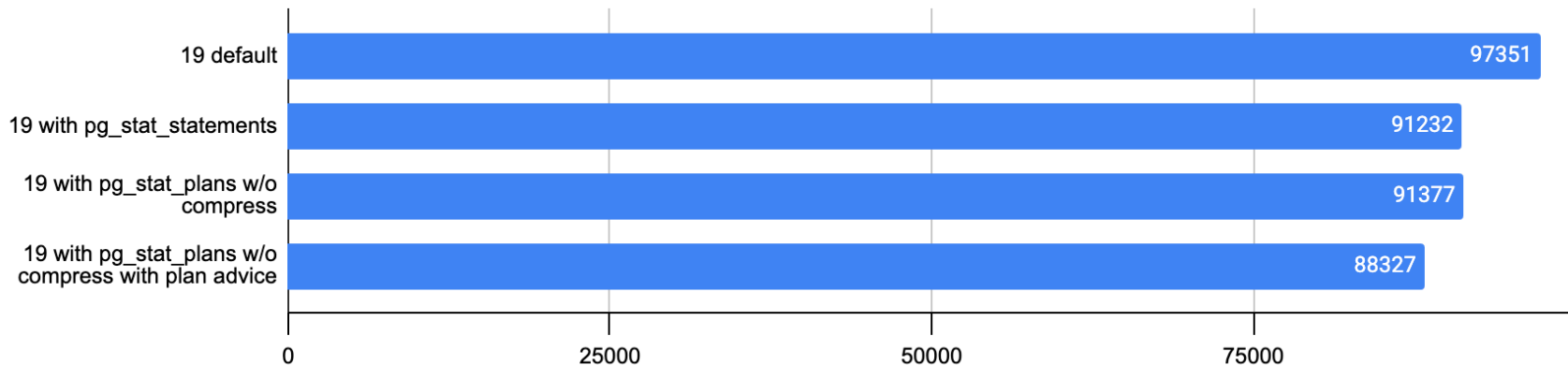
(3 rows)

pg_stat_plans.plan_advice = on



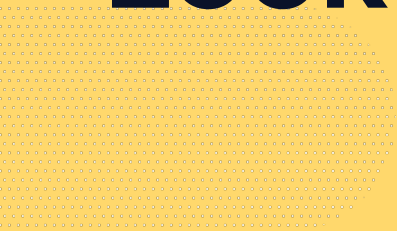
This has a minor overhead, similar to turning compression on:

Postgres 19 - pg_stat_plans - TPS (pgbench -T 60 -S, best of 3)





What Could Plan Management Look Like?



"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.

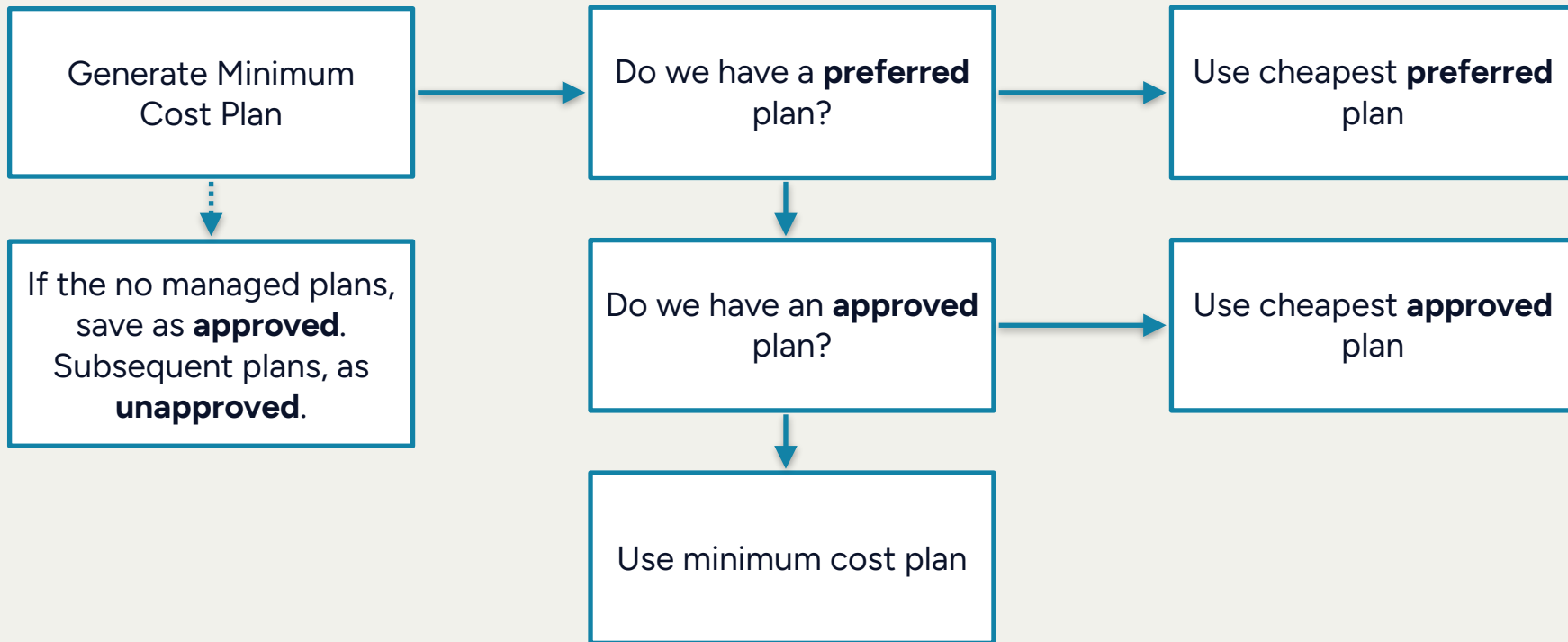
Approving better plans

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 basic idea of approved-plan style management



New Experimental Extension #2: [pg_plan_mgmt](#)

pg_plan_mgmt



Experimental. This is a first iteration. The SQL interface, GUCs and the on-disk format may still change, and it has not seen production use.

Plan baselines for PostgreSQL, built on `contrib/pg_plan_advice`, with no core patches.

Every distinct plan the optimizer produces for a managed statement is captured, identified by the advice string ("outline") `pg_plan_advice` generates for it. The first plan of a statement is **Approved**, later ones **Unapproved**. With baselines in use, the planner's minimum-cost plan runs only if it is **Approved** or **Preferred**; otherwise the cheapest enforceable **Preferred** plan runs, else the cheapest enforceable **Approved** one, else the minimum-cost plan. Plans can also be **Rejected** or disabled.

Requirements

PostgreSQL 19 or later, with `pg_plan_advice` installed.

```
SET pg_plan_mgmt.capture_plan_baselines = manual;  
EXPLAIN SELECT ...;           -- or run the statements  
SET pg_plan_mgmt.capture_plan_baselines = off;
```

```
SET pg_plan_mgmt.use_plan_baselines = on;  
EXPLAIN SELECT ...;
```

Hash Join

...

Supplied Plan Advice:

JOIN_ORDER(f d) /* matched */

...

Plan Management: Approved plan used instead of minimum-cost plan

Baseline Plan: 1 (Approved), Minimum-Cost Plan: 2 (Unapproved)

```
SELECT query_id, plan_no, status, enabled, valid, sql_text, search_path,  
role_name, plan_outline, estimated_total_cost, last_used, plan_count  
FROM plan_mgmt.plans;
```

```
SELECT plan_mgmt.set_plan_status(query_id, plan_no, 'Approved');  
-- or Preferred, Rejected, Unapproved
```

```
SELECT plan_mgmt.set_plan_enabled(query_id, plan_no, false);  
SELECT plan_mgmt.delete_plan(query_id, plan_no);
```

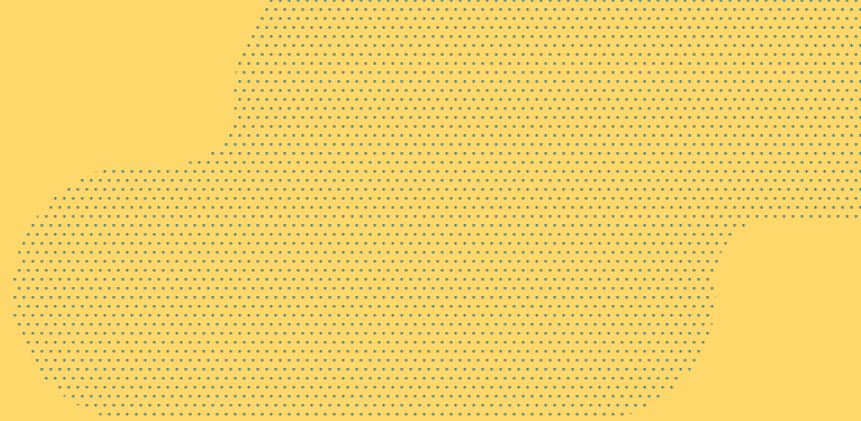
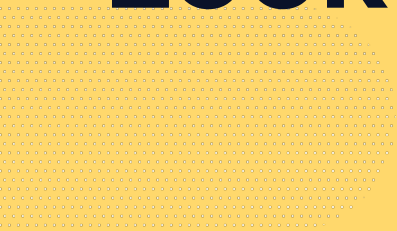
```
SELECT * FROM plan_mgmt.validate_plans('update');  
SELECT plan_mgmt.reset();
```

Open questions:

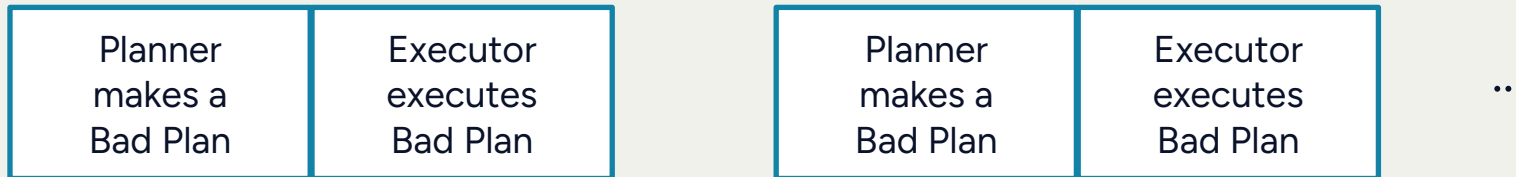
- Is this actually a useful paradigm overall?
- Is the separation between approved and preferred useful?
- Is it too expensive to plan at least twice?
- How can we make this safe for non-superusers, especially in light of a function to re-validate plans that might run other user's queries?



What Could Plan Feedback Loops Look Like?

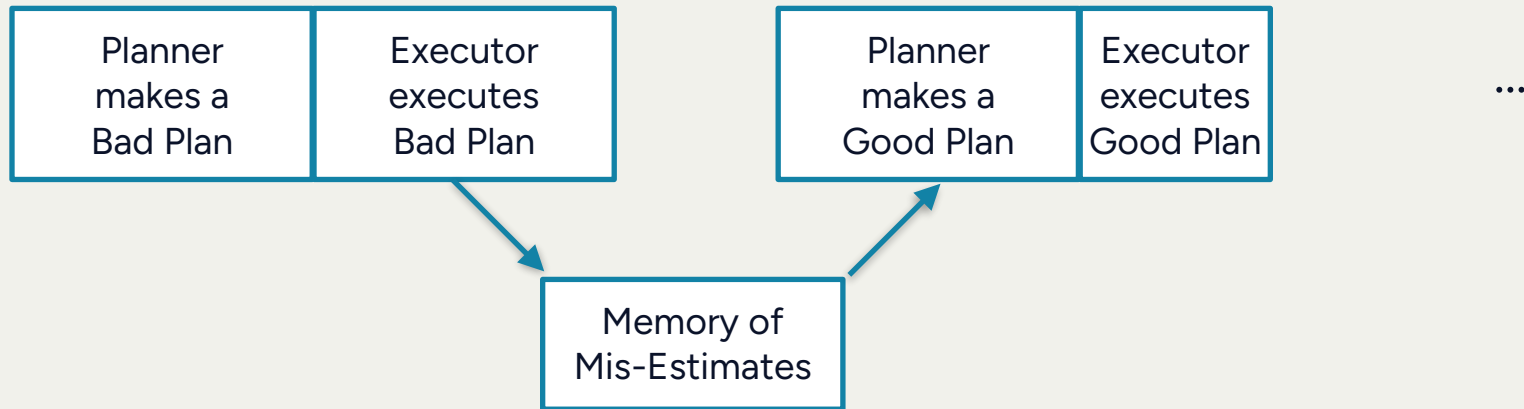


What Could Plan Feedback Loops Look Like?



Postgres will happily execute a bad plan.
Again, and again.

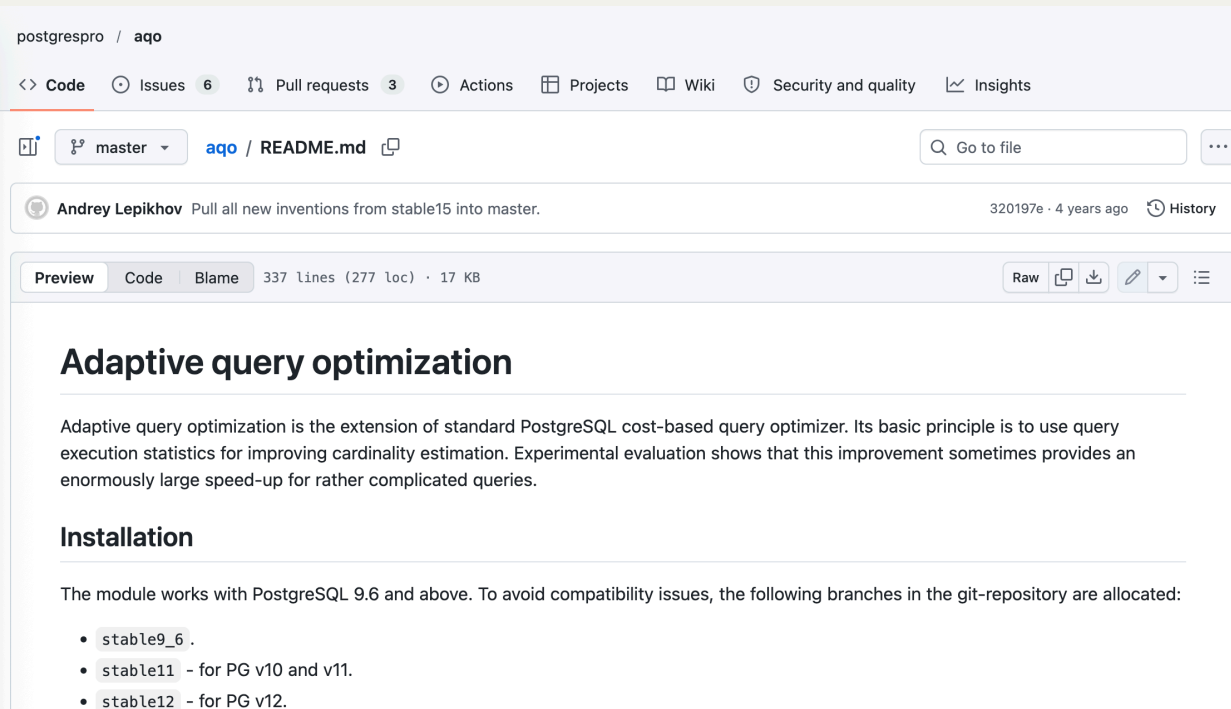
What Could Plan Feedback Loops Look Like?



We need some way to inform the planner of what actually happened in the past.

What Could Plan Feedback Loops Look Like?

AQO is an existing approach:



The screenshot shows the GitHub repository for PostgreSQL, specifically the 'aqo' directory. The repository is owned by 'postgrespro'. The main branch is 'master'. The file 'aqo / README.md' is selected. The commit history shows a pull request by 'Andrey Lepikhov' titled 'Pull all new inventions from stable15 into master.' dated '320197e · 4 years ago'. The README content is displayed below the commit history.

postgrespro / aqo

<> Code 6 Issues 6 Pull requests 3 Actions Projects Wiki Security and quality Insights

master aqo / README.md Go to file

Andrey Lepikhov Pull all new inventions from stable15 into master. 320197e · 4 years ago History

Preview Code Blame 337 lines (277 loc) · 17 KB Raw Copy Download Edit

Adaptive query optimization

Adaptive query optimization is the extension of standard PostgreSQL cost-based query optimizer. Its basic principle is to use query execution statistics for improving cardinality estimation. Experimental evaluation shows that this improvement sometimes provides an enormously large speed-up for rather complicated queries.

Installation

The module works with PostgreSQL 9.6 and above. To avoid compatibility issues, the following branches in the git-repository are allocated:

- stable9_6.
- stable11 - for PG v10 and v11.
- stable12 - for PG v12.

What Could Plan Feedback Loops Look Like?

And there are a good number of research papers:

Learned Offline Query Planning via Bayesian Optimization

Jeffrey Tao 

University of Pennsylvania
jefftao@seas.upenn.edu

Natalie Maus 

University of Pennsylvania
nmaus@seas.upenn.edu

Haydn Jones 

University of Pennsylvania
haydnj@seas.upenn.edu

Yimeng Zeng 

University of Pennsylvania
yimengz@seas.upenn.edu

Jacob R. Gardner 

University of Pennsylvania
jacobrg@seas.upenn.edu

Ryan Marcus 

University of Pennsylvania
rcmarcus@seas.upenn.edu

Kepler: Robust Learning for Faster Parametric Query Optimization

LYRIC DOSHI*, Google, USA

VINCENT ZHUANG*, Google, USA

GAURAV JAIN, Google, USA

RYAN MARCUS, University of Pennsylvania, USA

HAOYU HUANG, Google, USA

DENIZ ALTINBÜKEN, Google, USA

EUGENE BREVDO, Google, USA

CAMPBELL FRASER, Google, USA

Bao: Making Learned Query Optimization Practical

Ryan Marcus

MIT & Intel Labs

ryanmarcus@csail.mit.edu

Parimarjan Negi

MIT

pnegi@csail.mit.edu

Hongzi Mao

MIT

hongzi@csail.mit.edu

Nesime Tatbul

MIT & Intel Labs

tatbul@csail.mit.edu

Mohammad Alizadeh

MIT

alizadeh@csail.mit.edu

Tim Kraska

MIT

kraska@csail.mit.edu

What Could Plan Feedback Loops Look Like?

```
typedef enum RowsOverrideKind
{
    ROWS_OVERRIDE_ABSOLUTE,    /* use value as the row count estimate */
    ROWS_OVERRIDE_MULTIPLY,    /* multiply the standard estimate by value */
} RowsOverrideKind;

typedef struct RowsOverride
{
    pg_node_attr(no_copy_equal, no_read, no_query_jumble)

    NodeTag    type;

    /* relids of the base or join relation this override applies to */
    Relids     relids;

    /* how to combine value with the planner's own estimate */
    RowsOverrideKind kind;

    /* the row count or multiplier */
    double     value;
} RowsOverride;
```

What if we invented a mechanism like `pgs_mask`, but for row overrides?

And then use `build_simple_rel_hook` to attach them.

New Experimental Extension #3: pg_plan_feedback

pg_plan_feedback

Experimental. This extension depends on an [uncommitted Postgres patch](#) and is not ready for production use. Its interfaces, settings and stored data may change without notice.

Learns row count corrections from query execution and feeds them back to the Postgres planner.

After a query runs, `pg_plan_feedback` compares the planner's row estimate for each base relation scan with the actual row count, and remembers the ratio per query ID and relation. The next time the same query is planned, it hands the planner a multiplicative correction for each relation through the core `add_rows_override()` mechanism. It never forces a plan shape: it only corrects cardinalities, and the cost model decides the rest. When an estimate is already good, the correction converges to 1 and stops being applied.

This version corrects base relations only (table scans, partition children, `UNION ALL` arms, and subquery, CTE and function scans).

```
SET pg_plan_feedback.mode = observe;  
EXPLAIN (ANALYZE, TIMING OFF, SUMMARY OFF, BUFFERS OFF)  
SELECT c.name, o.amount  
FROM orders o JOIN customers c ON c.id = o.customer_id  
WHERE o.status = 'open' AND o.region = 'EU';
```

```
    Nested Loop  (cost=0.28..22430.84 rows=102 width=24) (actual rows=10000.00 loops=1)  
      -> Seq Scan on orders o  (cost=0.00..22353.00 rows=102 width=16) (actual  
rows=10000.00 loops=1)  
        Filter: ((status = 'open'::text) AND (region = 'EU'::text))  
        Rows Removed by Filter: 990000  
      -> Index Scan using customers_pkey on customers c  (cost=0.28..0.76 rows=1 width=16)  
(actual rows=1.00 loops=10000)  
        Index Cond: (id = o.customer_id)  
        Index Searches: 10000
```

```
SELECT queryid, relation, correction, observations, applied,  
       last_estimated_rows AS est, last_actual_rows AS actual, min_factor, max_factor  
FROM pg_plan_feedback;
```

queryid	relation	correction	observations	applied	est	actual	min_factor	max_factor
-5973088281084997252	o	98.04	1	0	102	10000	98.04	98.04

queryid	relation	correction	observations	applied
-5973088281084997252	o	98.04	2	0

```
SET pg_plan_feedback.mode = apply;  
EXPLAIN (ANALYZE, TIMING OFF, SUMMARY OFF, BUFFERS OFF) SELECT ... ;
```

```
Hash Join (cost=29.50..22408.86 rows=10000 width=24) (actual rows=10000.00 loops=1)  
  Hash Cond: (o.customer_id = c.id)  
    -> Seq Scan on orders o (cost=0.00..22353.00 rows=10000 width=16) (actual rows=10000.00  
loops=1)  
      Filter: ((status = 'open'::text) AND (region = 'EU'::text))  
      Rows Removed by Filter: 990000  
      Plan Feedback: x98.0  
    -> Hash (cost=17.00..17.00 rows=1000 width=16) (actual rows=1000.00 loops=1)  
        -> Seq Scan on customers c (cost=0.00..17.00 rows=1000 width=16) (actual rows=1000.00  
loops=1)  
      Plan Feedback:  
        o: x98.0
```



Thank you!



pganalyze



Learn more at pganalyze.com

Contact me: lukas@pganalyze.com