Tracking plan shapes over time with Plan IDs

+ a new pg_stat_plans

lukas@pganalyze.com

1. Why do we need a Plan ID?
2. Status Quo of Plan Statistics
3. In-core Plan IDs vs Extensions
4. A new pg_stat_plans

pganalyze

Why do we need a Plan ID?

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)
```

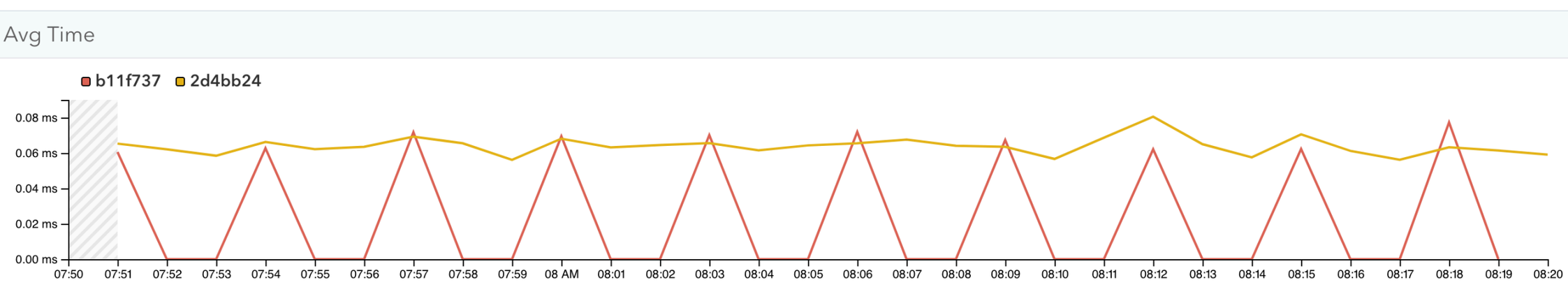
Plan IDs let us track plan usage over time

Plan Statistics

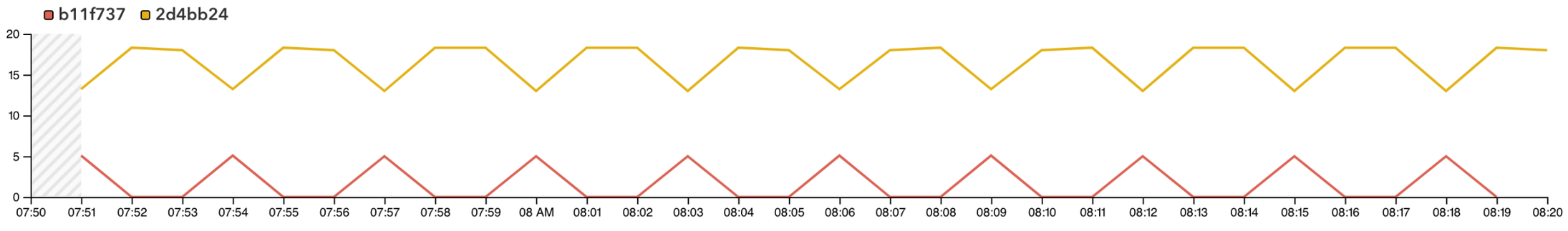
Show All Plans

PLAN	EST. COST	AVG RUNTIME	CALLS / MIN	ORIGINAL PLAN ID	PLAN NODES
2d4bb24	0	0.06ms	17	-15608242543328…	Aggregate · CTE · CTE +142 more
b11f737	0	0.07ms	5	-91824118698804…	Aggregate · CTE · CTE +138 more

Avg Time



Calls



Plan IDs let us detect regressions, quickly

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

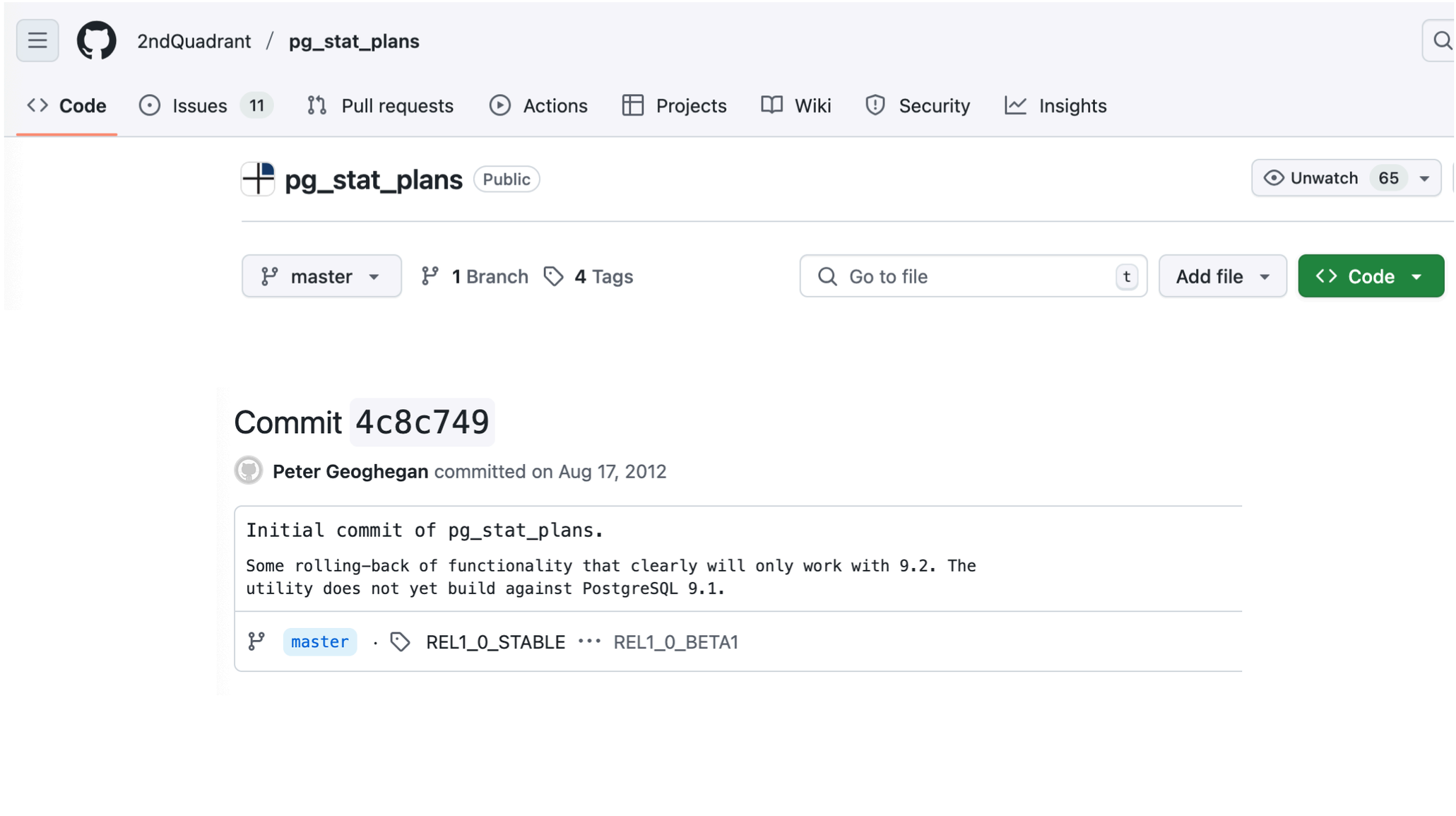
A query plan changed, on a frequently-run query (~1k/sec) on a large table (~2B rows) without warning. Went from sub-millisecond to multi-second.

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

pganalyze

Status Quo of Plan Statistics

This is not a new idea.



The old pg_stat_plans is unmaintained.

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

pg_store_plans

master pg_store_plans / pg_store_plans.c

Code Blame 2485 lines (2153 loc) · 67.3 KB

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

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

~20% overhead in some cases

pg_stat_monitor

main pg_stat_monitor / pg_stat_monitor.c

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

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

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

In 2024, AWS launched
aurora_plan_stats for Aurora.

And Microsoft has plan IDs
in **Query Store for Azure Postgres.**

Can Postgres do better here?

pganalyze

In-core Plan IDs vs Extensions

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 messy out-of-core.

```
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 jumble it)

In core its easy to maintain "what is significant" on the plannodes.h structs

```
@@ -1059,7 +1059,7 @@ typedef struct Memoize
1059  1059          * The maximum number of entries that the planner expects will fit in the
1060  1060          * cache, or 0 if unknown
1061  1061          */
1062       -        uint32          est_entries;
     1062  +        uint32          est_entries pg_node_attr(query_jumble_ignore);
1063  1063
1064  1064          /* paramids from param_exprs */
1065  1065          Bitmapset  *keyparamids;
@@ -1156,7 +1156,7 @@ typedef struct Agg
1156  1156          Oid                *grpCollations pg_node_attr(array_size(numCols));
1157  1157
1158  1158          /* estimated number of groups in input */
1159       -        long            numGroups;
     1159  +        long            numGroups pg_node_attr(query_jumble_ignore);
1160  1160
```

Input needed on what is significant

navigation

- Main Page
- Random page
- Recent changes
- Help

tools

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

search

Search PostgreSQL w

Go Search

page | discussion | view source | history

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

Plan ID Jumbling

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

See Commitfest entry and pgsql-hackers thread.

What to jumble

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

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

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 "Indirect" in the table below.

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

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 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
Plan (abstract)		
type	Yes	
disabled_nodes	No	Costing/selectivity information should be ignored
startup_cost	No	Costing/selectivity information should be ignored

In core we also have a tree walk we can re-use, in setrefs.c

9 ■■■■■ src/backend/optimizer/plan/setrefs.c

```
@@ -19,6 +19,7 @@
 #include "catalog/pg_type.h"
 #include "nodes/makefuncs.h"
 #include "nodes/nodeFuncs.h"
+#include "nodes/queryjumble.h"
 #include "optimizer/optimizer.h"
 #include "optimizer/pathnode.h"
 #include "optimizer/planmain.h"
@@ -1315,6 +1316,14 @@ set_plan_refs(PlannerInfo *root, Plan *plan, int rtoffset)
         plan->lefttree = set_plan_refs(root, plan->lefttree, rtoffset);
         plan->righttree = set_plan_refs(root, plan->righttree, rtoffset);

+        /*
+         * If enabled, append significant information to the plan identifier
+         * jumble (we do this here since we're already walking the tree in a
+         * near-final state)
+         */
+        if (IsPlanIdEnabled())
+                JumbleNode(root->glob->plan_jumble_state, (Node *) plan);
```

This is all PG 19 discussion material.

But we did get
key improvements in 18
we can build on.

PG18: 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
```

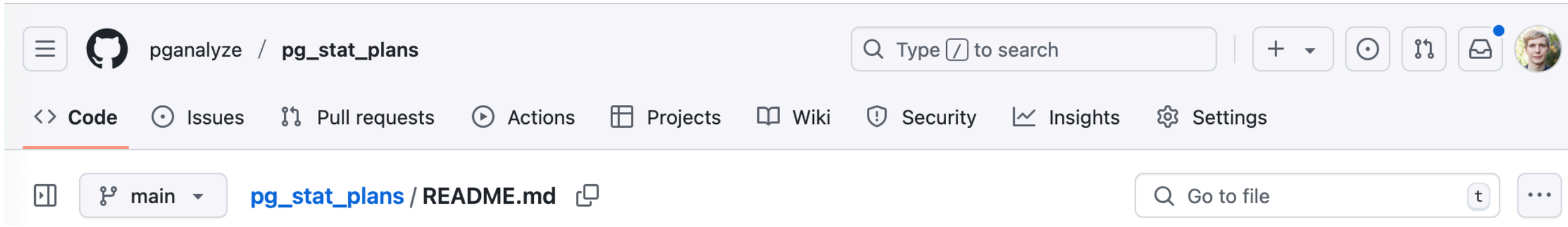
```
Allow plugins to set a 64-bit plan identifier in PlannedStmt

This field can be optionally set in a PlannedStmt through the planner
hook, giving extensions the possibility to assign an identifier related
to a computed plan.  The backend is changed to report it in the backend
entry of a process running (including the extended query protocol), with
semantics and APIs to set or get it similar to what is used for the
existing query ID (introduced in the backend via 4f0b0966c8).  The plan
ID is reset at the same timing as the query ID.  Currently, this
information is not added to the system view pg_stat_activity; extensions
can access it through PgBackendStatus.

Some patches have been proposed to provide some features in the planning
area, where a plan identifier is used as a key to know the plan involved
(for statistics, plan storage and manipulations, etc.), and the point of
this commit is to provide an anchor in the backend that extensions can
rely on for future work.   The reset of the plan identifier is
controlled by core and follows the same pattern as the query identifier
added in 4f0b0966c8.

The contents of this commit are extracted from a larger set proposed
originally by Lukas Fittl, that Sami Imseih has proposed as an
independent change, with a few tweaks sprinkled by me.

Author: Lukas Fittl <lukas@fittl.com>
Author: Sami Imseih <samimseih@gmail.com>
Reviewed-by: Bertrand Drouvot <bertranddrouvot.pg@gmail.com>
Reviewed-by: Michael Paquier <michael@paquier.xyz>
Discussion: https://postgr.es/m/CAP53Pkyow59ajFMHGpmb1BK9WHDypaWtUsS_5DoYUEfsa_Hktg@mail.gmail.com
Discussion: https://postgr.es/m/CAA5RZ0vyWd4r35uUBUmhngv8XqeiJUkJDDKkLf5LCoWxv-t_pw@mail.gmail.com
```

```
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.

pganalyze

A new pg_stat_plans



github.com/pganalyze/pg_stat_plans

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 optimized for tracking changes in plan shape, but does not aim to track execution statistics for plans, like auto_explain can do for outliers.

This project is inspired by multiple Postgres community projects, including the original pg_stat_plans extension (unmaintained), with a goal of upstreaming parts of this extension into the core Postgres project over time.

Experimental. May still change in incompatible ways without notice. Not (yet) recommended for production use.

PG18: Introduce pluggable APIs for Cumulative Statistics

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

```
Introduce pluggable APIs for Cumulative Statistics

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

The stats kinds are divided into two parts:
- The in-core "builtin" stats kinds, with designated initializers, able
to use IDs up to 128.
- The "custom" stats kinds, able to use a range of IDs from 128 to 256
(128 slots available as of this patch), with information saved in
TopMemoryContext.  This can be made larger, if necessary.

There are two types of cumulative statistics in the backend:
- For fixed-numbered objects (like WAL, archiver, etc.).  These are
attached to the snapshot and pgstats shmem control structures for
efficiency, and built-in stats kinds still do that to avoid any
redirection penalty.  The data of custom kinds is stored in a first
array in snapshot structure and a second array in the shmem control
structure, both indexed by their ID, acting as an equivalent of the
builtin stats.
- For variable-numbered objects (like tables, functions, etc.).  These
are stored in a dshash using the stats kind ID in the hash lookup key.

Internally, the handling of the builtin stats is unchanged, and both
```

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 data
                |               Recheck Cond: (server_id = '00000000-0000-0000-000
```

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
```

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

```
SELECT * FROM pg_stat_plans;
```

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

```
plan                | Limit
                    |   ->  Sort
                    |         Sort Key: database_stats_35d.frozenxid_age DESC
                    |         ->  Bitmap Heap Scan on database_stats_35d_20250514 data
                    |               Recheck Cond: (server_id = '00000000-0000-0000-000
                    |               Filter: ((frozenxid_age IS NOT NULL) AND (collecte
                    |               ->  Bitmap Index Scan on database_stats_35d_202505
                    |                     Index Cond: (server_id = '00000000-0000-000
```

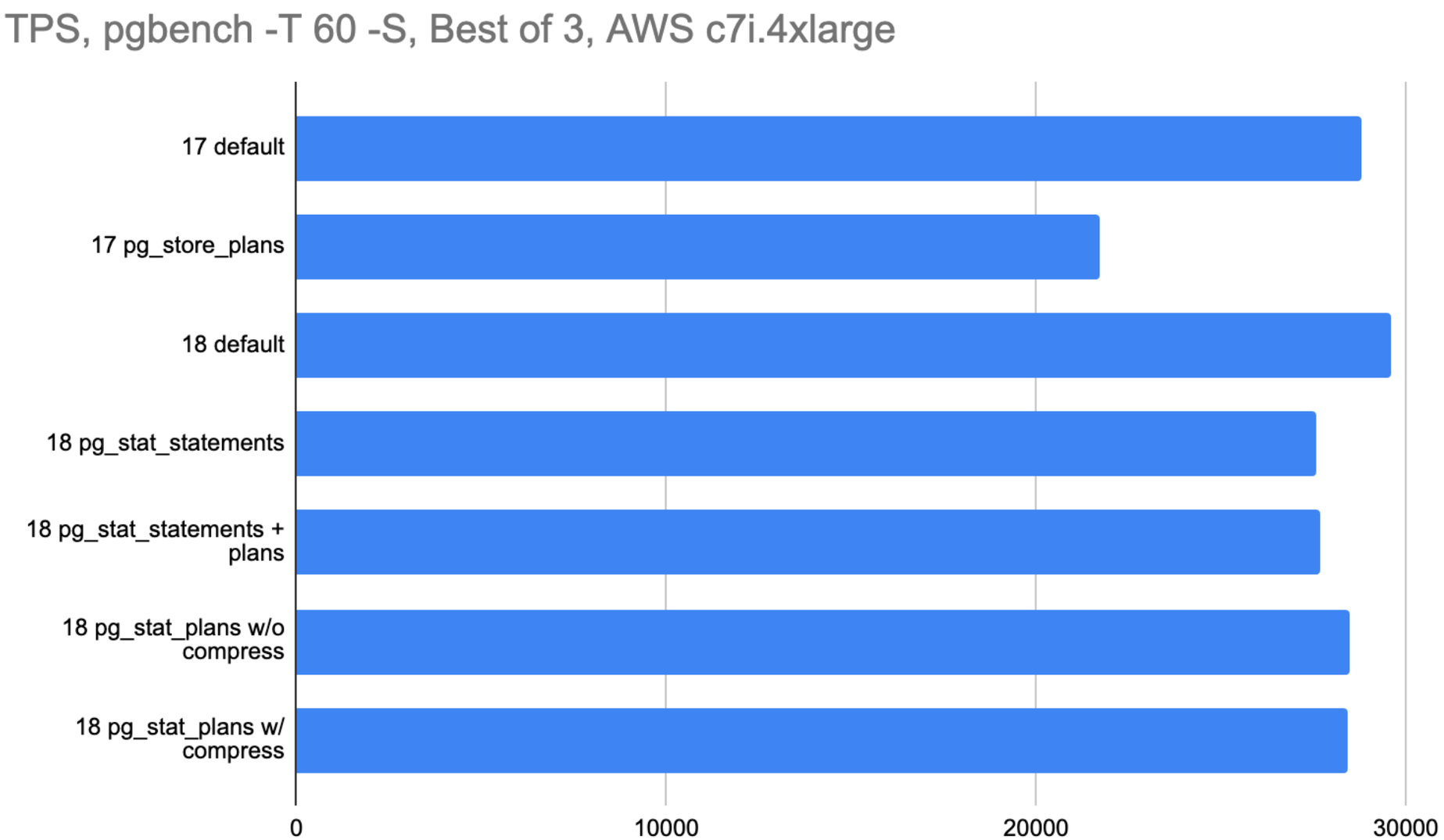
SELECT * FROM pg_stat_plans_activity;

```
  pid   |       plan_id          |                                                    plan
--------+------------------------+-----------------------------------------------------------
 83994  | -5449095327982245076   | Merge Join
        |                        |   Merge Cond: ((a.datid = p.dbid) AND (a.usesysid = p.userid)
        |                        |   ->  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
        |                        |               Index Cond: (server_id = '00000000-0000-0000-00
        |                        |               Filter: ((started_at IS NULL) AND (finished_at
```

Get the plan for a currently running query

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

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



Next steps for pg_stat_plans 2.0

- Plan text compression improvements
- Stabilize extension (test/benchmark)
- Partial support for older releases

Open questions

- How do we handle Append nodes in plan IDs?
- What metrics should we capture per-plan?
- Worth supporting other EXPLAIN formats?
- Should a future pg_stat_statements version handle plans too, or should we keep it separate?

Thank you!