pg_plan_advice: Plan Stability and User Planner Control for PostgreSQL?

Robert Haas
VP, Chief Architect, Database Servers
2026.pgconf.dev

What Is `pg_plan_advice`?

- New `contrib` module that is intended to ship with PostgreSQL 19.
- Implements a mini-language that can be used to describe key planner decisions.
- `pg_plan_advice` both *generates* and *accepts* this mini-language.
  - When a query is planned, you can ask the system to tell you what decisions were made, and get back a plan advice string.
  - Later, you can use that plan advice string to ask the system to make those same decisions again.
  - Or you can edit the string to ask the system to make different decisions next time.

Isn’t This Just Planner Hints?

- You can absolutely use plan advice in the way that people use hints in other database systems.
- However, my goal was to create a *plan stability feature*.
- Inevitably, if you provide a way to ask the planner to do the same thing, somebody can use the same mechanism to ask it to a do different thing.
- I think that’s OK. Overriding the planner’s judgement is risky, but it’s useful in case of emergency, and it’s great for development and testing.

Prior Work

- I am not the first person to do work in this area.
- Multiple out-of-core extensions exist for both hints and plan stability.
- I'm not an expert on any of those extensions, but also don't mean to dismiss them.
- My belief is that `pg_plan_advice` offers some advantages over prior art, but others might disagree:
  - I don't know of another system that integrates hinting and plan stability in the way that `pg_plan_advice` does.
  - `pg_hint_plan` has to duplicate a lot of core planner code due to lack of hooks and other extensibility mechanisms, whereas `pg_plan_advice` modifies core to add extensibility (which should benefit `pg_hint_plan`, too).

Engineering Challenges

- *Advice Language Design*. Which planner decisions do we want to try to control, and which are out of scope? What syntax should we choose?
- *Advice Generation*. In certain situations, the finished plan doesn't contain enough information to reconstruct what decisions the planner made.
- *Advice Enforcement*. How does the user supply an advice string? How do we get the planner to generate a plan that conforms to the advice string?
- *Code Churn*. Avoid changing too much core planner code. Avoid introducing too many different/unfamiliar concepts.

EDB POSTGRES AI

Advice Language Design

Advice Language: Example

```
EXPLAIN (COSTS OFF, PLAN_ADVICE)
SELECT * FROM gt_fact f JOIN gt_dim d ON f.dim_id = d.id ORDER BY d.id;

 Gather Merge
   Workers Planned: 1
   ->  Sort
         Sort Key: f.dim_id
         ->  Parallel Hash Join
               Hash Cond: (f.dim_id = d.id)
               ->  Parallel Seq Scan on gt_fact f
               ->  Parallel Hash
                     ->  Parallel Seq Scan on gt_dim d

 Generated Plan Advice:
   JOIN_ORDER(f d) HASH_JOIN(d) SEQ_SCAN(f d) GATHER_MERGE((f d))
```

Advice Language: Core Ideas

- Existing hinting systems such as `pg_hint_plan` generally use a syntax of this form:

  ```
  OPERATOR_NAME(relation_alias1 relation_alias2 relation_alias3)
  ```

  So I started with that.

- But we need something that is unambiguous and round-trip safe.

Relation Aliases Need Not Be Unique

```
rhaas=# select * from x, x;
ERROR:  table name "x" specified more than once

rhaas=# select * from x where exists (select * from x);
 a
---
(0 rows)

rhaas=# select * from (x join (values (1)) on true) y, x;
 a | column1 | a
---+---------+---
(0 rows)
```

…plus `x` could be a partitioned table, and we might need to talk about each partition individually.

Relation Identifier Syntax

- General syntax is

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

- Components that are not needed can be omitted.
- `occurrence_number` defaults to 1 (and nearly always is 1).
- `partition_schema` and `partition_name` are not needed for unpartitioned tables.
- `plan_name` is not needed for the top-level query (which is nameless).
- So it's usually simple, but when needed, we can write something like:

  ```
  SEQ_SCAN(orders#3/public.orders2026@expr_1)
  ```

Join Orders Can Be Bushy - Use Nested Parentheses

```
Merge Join
-> Nested Loop
   -> Hash Join
      -> Merge Join
         -> Nested Loop
            -> Seq Scan on apple
            -> Index Scan on banana
         -> Index Scan on carrot
      -> Hash
         -> Seq Scan on daikon
   -> Index Scan on eggplant
-> Hash Join
   -> Seq Scan on fennel
   -> Hash
      -> Seq Scan on garlic
```

```
JOIN_ORDER(apple banana carrot
   daikon eggplant
   (fennel garlic))
MERGE_JOIN_PLAIN(carrot
   (fennel garlic))
```

EDB POSTGRES AI

Advice Generation

Reconstructing Planner Decisions from the Final Plan

- For the most part, you can tell what the planner did by looking at the final plan.
- But when I started developing this feature, I discovered that there were various corner cases where the final plan is ambiguous.
- For this project to work out, those cases had to be fixed.

Example: Result Nodes Were Not Identifiable

```
explain (costs off, plan_advice)
select * from int8_tbl t1 left join int8_tbl t2 on false left join
int8_tbl t3 on false;

Nested Loop Left Join
   Join Filter: false
   ->  Nested Loop Left Join
         Join Filter: false
         ->  Seq Scan on int8_tbl t1
         ->  Result
               Replaces: Scan on t2
               One-Time Filter: false
   ->  Result
         Replaces: Scan on t3
         One-Time Filter: false
```

```
JOIN_ORDER(t1 t2 t3)
NESTED_LOOP_PLAIN(t2 t3)
SEQ_SCAN(t1)
NO_GATHER(t1 t2 t3)
```

Ambiguity Problems

`f2bae51dfd5` Keep track of what RTIs a Result node is scanning.

- Otherwise, we can't tell what part of the original query corresponds to a `Result` node that appears in the final plan.

`0d4391b265f` Store information about elided nodes in the final plan.

- `Subquery Scan` nodes are often omitted from the plan, making the boundary between the main query level and subqueries ambiguous.
- `Append` and `MergeAppend` nodes are omitted when they have just one child, which can make cases involving partitioned tables and especially partitionwise join ambiguous.

`7358abcc607` Store information about `Append` node consolidation in the final plan.

- `Append` or `MergeAppend` nodes inserted due to set operations can be merged with `Append` or `MergeAppend` nodes inserted to deal with partitioning, leading to confusion.

`26255a32073` Add an `alternative_plan_name` field to `PlannerInfo`.

- In certain cases, we clone a subquery and plan both copies, and the plan tree did not capture the relationship between the original and the copy.

Order-of-Operations Problems

`8c49a484e8e` Assign each subquery a unique name prior to planning it.
`adbad833f3d` Store information about range-table flattening in the final plan.

- The problems on the previous slide have to do with preserving information until planning is complete.
- But I also ran into some problems where we didn't make certain decisions early enough.
- We must be able to compute the relation identifier for a given relation before that relation has already been planned.
- Without these commits, you can't do that.

EDB POSTGRES AI

Advice Enforcement

Where Do We Get The Advice?

- Simplest method - set explicitly for each query:
```
SET pg_plan_advice.advice = 'advice_string_goes_here';
```

- `pg_stash_advice` method - automatic matching by query ID:
```
SELECT pg_create_advice_stash('my_stash');
SELECT pg_set_stashed_advice('my_stash', '123456789',
'advice_string_goes_here');
SET pg_stash_advice.stash_name = 'my_stash';
```

- Pluggable! You can write your own “advisor” module that will be called each time a query is planned and may choose to provide advice.

How Do We Enforce The Advice?

- Plan as normal.
- Disable any paths that are incompatible with the supplied advice.
- Surviving paths should comply with the advice!
- Two problems:
  - Before PostgreSQL 19, the only method we had for disabling paths was to set `enable_seqscan`, `enable_hashjoin`, etc. to false, which affected the entire plan.
  - Before PostgreSQL 18, disabling a path just meant adding a large constant to its cost. So it was still theoretically possible for a disabled path to win on cost.

New Planner Concept: `pgs_mask`

- Bitmask of allowable path generation strategies.
- Can be set on multiple levels - whole plan, per-relation, even more fine-grained for joins.
- Now used internally to implement `enable_seqscan`, `enable_nestloop`, etc. by setting the mask for the entire plan.
- `pg_plan_advice` (or other modules) can manipulate this mask at a fine-grained level to get rid of particular strategies for just certain parts of the plan.

Enforcement via Negative Constraints Is Tricky

- Suppose the user says `GATHER((foo bar)).`
- User wants a `Gather` node positioned in the plan so that it is above `foo` and `bar` and no other relations.
- What do we need to do to enforce this?

Enforcement Rules for `Gather((foo bar))`

- When planning the join that includes exactly `foo` and `bar`, disallow:
  - `Gather Merge`
  - non-parallel plans
- When the set of relations overlaps with `foo` and `bar` but is not exactly equal, disallow:
  - `Gather`
  - `Gather Merge`
- When considering possible join orders, disallow any that join `foo` or `bar` to anything other than each other before they've been joined to each other.

General Thoughts on Escape Attempts

- In general: For any advice tag type, if there's a way for the planner to work around the restrictions that you attempt to enforce without technically violating them, it will generally attempt to do so.
  - Costing favors avoiding restrictions rather than adhering to them.
- A surprisingly large amount of `pg_plan_advice`'s enforcement logic is devoted to problems of this type.
- Enforcement of scan advice is *mostly* straightforward, but pretty much everything else requires much more complicated rules than I initially anticipated.
- In particular, almost all non-scan advice involves implicit negative join order constraints.

EDB POSTGRES AI

Future Work

What Comes Next?

- Cardinality hints.
- Control over aggregation strategy or other areas not presently covered.
- Experimentation with the “advisor” hook that lets other plugins integrate with `pg_plan_advice`.
  - e.g. Randomize plan generation and see if we can generate any plans that fail or produce different results.
  - e.g. Automatically “learn” good plans and then use plan advice to regenerate those plans.
  - e.g. Automatically feed cardinalities from one planning cycle into future cycles (like `pg_plan_advsr` or `aqo`).

Thank You!

Any questions?