What's in a Plan?

And how did it get there, anyway?

- Robert Haas | 2018-06-01

Plan Contents: Structure Definition

```
typedef struct Plan
{
        NodeTag         type;

        /* estimated execution costs for plan (see costsize.c for more info) */
        Cost            startup_cost;   /* cost expended before fetching any tuples */
        Cost            total_cost;             /* total cost (assuming all tuples fetched) */

        /* planner's estimate of result size of this plan step */
        double          plan_rows;              /* number of rows plan is expected to emit */
        int                     plan_width;             /* average row width in bytes */

        /*
         * information needed for parallel query
         */
        bool            parallel_aware; /* engage parallel-aware logic? */
        bool            parallel_safe;  /* OK to use as part of parallel plan? */

        /*
         * Common structural data for all Plan types.
         */
        int                     plan_node_id;   /* unique across entire final plan tree */
        List       *targetlist;         /* target list to be computed at this node */
        List       *qual;                       /* implicitly-ANDed qual conditions */
        struct Plan *lefttree;          /* input plan tree(s) */
        struct Plan *righttree;
        List       *initPlan;           /* Init Plan nodes (un-correlated expr
                                                                 * subselects) */

        /*
         * Information for management of parameter-change-driven rescanning
         */
        Bitmapset  *extParam;
        Bitmapset  *allParam;
} Plan;
```

Plan Contents: By Category

- Node Tag
- Costing Information
- Parallel Query Support
- Target List & Qual
- Left & Right Subtrees
- InitPlans
- extParam & allParam
- Type-specific information

Costing Information

- PostgreSQL first generates paths representing possible query plans; winning paths are converted to plans.
- Costs are important at the path stage because they let us determine which paths are best, but we save the information in the final plan.

```
/*
 * estimated execution costs for plan
 */
Cost                startup_cost;
Cost                total_cost;

/*
 * planner's estimate of result size
 */
double              plan_rows;
int                 plan_width;    /* in bytes/row */
```

Costing Information: Uses

- EXPLAIN.
- For a hash join or hashed subplan, row count and width are used to set the initial size of the hash table.
- For a hash join, should we fetch the first outer tuple before or after building the hash table?
- Decide between AlternativeSubPlans.
- Decide between custom plans and generic plans.

Parallel Query

```
/* engage parallel-aware logic? */
bool                parallel_aware;

/* OK to use as part of parallel plan? */
bool                parallel_safe;

/* unique across entire final plan tree */
int                 plan_node_id;
```

Parallel Query: Motivation

- Why do we need the parallel_aware flag?

```
Gather
-> Merge Join
  -> Parallel Index Scan on a
  -> Index Scan on b
```

- Why do we need the plan_node_id?

```
Gather
-> Append
  -> Parallel Seq Scan on p1
  -> Parallel Seq Scan on p2
  -> Parallel Seq Scan on p3
```

Target List, Filter, Left & Right Subtrees (1)

```
/* target list to be computed at this node */
List        *targetlist;

/* implicitly-ANDed qual conditions */
List        *qual;

/* input plan tree(s) */
struct Plan *lefttree;
struct Plan *righttree;
```

Target List, Filter, Left & Right Subtrees (2)

```
Merge Left Join
  Output: a.q2, b.q1
  Merge Cond: (a.q2 = (COALESCE(b.q1, '1'::bigint)))
  Filter: (COALESCE(b.q1, '1'::bigint) > 0)
  ->  Sort
        Output: a.q2
        Sort Key: a.q2
        ->  Seq Scan on public.int8_tbl a
              Output: a.q2
  ->  Sort
        Output: b.q1, (COALESCE(b.q1, '1'::bigint))
        Sort Key: (COALESCE(b.q1, '1'::bigint))
        ->  Seq Scan on public.int8_tbl b
              Output: b.q1, COALESCE(b.q1, '1'::bigint)
```

Left, Right, Center Right, Center Left?

```
Append
-> Seq Scan on foo
-> Seq Scan on bar
-> Seq Scan on baz
-> Seq Scan on quux
```

InitPlans & SubPlans

```
regression=# explain (costs off, verbose) select f1,
(select odd from tenk1 where unique1 = f1) from int4_tbl
where f1 = (select min(abs(f1)) from int4_tbl);

 Seq Scan on public.int4_tbl
   Output: int4_tbl.f1, (SubPlan 1)
   Filter: (int4_tbl.f1 = $1)
   InitPlan 2 (returns $1)
     ->  Aggregate
           Output: min(abs(int4_tbl_1.f1))
           ->  Seq Scan on public.int4_tbl int4_tbl_1
                 Output: int4_tbl_1.f1
   SubPlan 1
     ->  Index Scan using tenk1_unique1 on public.tenk1
           Output: tenk1.odd
           Index Cond: (tenk1.unique1 = int4_tbl.f1)
```

InitPlans, not SubPlans!

- Each Plan node carries a list of associated initPlans.
- SubPlans are not listed; they just appear in expressions. The executor builds a per-node list at runtime.

```
List *initPlan;   /* Init Plan nodes (un-correlated
                   * expr subselects) */
```

extPararam & allParam

```
/*
 * Information for parameter-change-driven rescanning
 *
 * extParam includes the paramIDs of all external
 * PARAM_EXEC params affecting this plan node or its
 * children.  setParam params from the node's
 * initPlans are not included, but their extParams
 * are.
 *
 * allParam includes all the extParam paramIDs, plus
 * the IDs of local params that affect the node (i.e.,
 * the setParams of its initplans). These are _all_
 * the PARAM_EXEC params that affect this node.
 */
Bitmapset  *extParam;
Bitmapset  *allParam;
```

extParam & allParam: Example

```
explain (verbose, costs off)
select 1 = all (select (select 1));

 Result
   Output: (SubPlan 2)
   SubPlan 2
     ->  Materialize ← extParam empty, allParam = {$0}
           Output: ($0)
           InitPlan 1 (returns $0)
             ->  Result
                   Output: 1
           ->  Result
                 Output: $0
```

extParams & allParams: Execution

- allParam is used to decide which nodes to reset when we need to rescan.
- For example, we can rescan a sort either by rereading the existing output or by throwing away the old output, regenerating the input, and sorting again.
- If the sort's input depends on a parameter which has changed, we need to do the latter; otherwise it's faster to do the former.
- extParam is also used for this purpose … barely. It's mostly used when assembling the final plan, rather than at execution time.

Where's the Parameter?

```
Nested Loop
-> Seq Scan on int4_tbl
-> Append
  -> Index Scan using t3i on t3 a
     Index Cond: (expensivefunc(x) = int4_tbl.f1)
  -> Index Scan using t3i on t3 b
     Index Cond: (expensivefunc(x) = int4_tbl.f1)
```

Where's the Parameter?

```
Nested Loop
-> Seq Scan on int4_tbl
-> Append ← extParam = allParam = {$0}
   -> Index Scan using t3i on t3 a ← here too
      Index Cond: (expensivefunc(x) = int4_tbl.f1)
   -> Index Scan using t3i on t3 b ← and also here
      Index Cond: (expensivefunc(x) = int4_tbl.f1)
```

EXPLAIN vs. Reality – So Far

- parallel_safe flag is not displayed.
- plan_node_id is not displayed.
- InitPlans and SubPlans are displayed in the same way, but only InitPlans are really attached that way.
- extParam and allParam are not displayed, although you can infer something about them from the InitPlan display (and from knowledge of how Nested Loops work).

Expression Deparsing: It's all a lie!

```
Nested Loop Left Join
  Output: "*VALUES*".column1, i1.f1, (666)
  Join Filter: ("*VALUES*".column1 = i1.f1)
  -> Values Scan on "*VALUES*"
     Output: "*VALUES*".column1
  -> Materialize
     Output: i1.f1, (666)
     -> Nested Loop Left Join
        Output: i1.f1, 666
        -> Seq Scan on public.int4_tbl i1
           Output: i1.f1
        -> Index Only Scan using tenk1_unique2 on
public.tenk1 i2
           Output: i2.unique2
           Index Cond: (i2.unique2 = i1.f1)
```

Expression Deparsing: The lie exposed!

```
Nested Loop Left Join
  Output: OUTER.1, INNER.1, INNER.2
  Join Filter: (OUTER.1 = INNER.1)
  -> Values Scan on "*VALUES*"
     Output: "*VALUES*".column1
  -> Materialize
     Output: OUTER.1, OUTER.2
     -> Nested Loop Left Join
        Output: OUTER.1, 666
        -> Seq Scan on public.int4_tbl i1
           Output: i1.f1
        -> Index Only Scan using tenk1_unique2 on
public.tenk1 i2
           Output: i2.unique2
           Index Cond: (i2.unique2 = $0)
```


ENTERPRISEDB

Expression Deparsing: Explained

- When we initially generated paths, references to table columns (internally called “Var” nodes) and expressions in target list and expressions refer to the table that will really provide the value.
- But at execution time, it’s not useful to know the original source of the value – we need to know from where we can obtain it.
- One of the last stages of planning is to replace Vars and expressions with Vars that refer to the “outer” or “inner” plan.

Thanks

- Any Questions?