What’s in a Plan?

• Robert Haas | 2019-10-18

Overview

- Volcano-Style Execution
- The Plan Data Structure Generally
- Specialty Information (Costing, Parallel Query)
- Core Information (Target List, Filter Qual, Subtrees)
- Parameters, InitPlans, SubPlans
- Expression Deparsing

Volcano-Style Execution

- A PostgreSQL plan is a tree of Plan nodes.
- Tuples are “pulled” from the top of the tree, which pulls from progressively lower levels of the tree; the nodes at the bottom pull from base relations.
- The first system that I know of which used a system of this type is called Volcano (early 1990s), and so we refer to this as Volcano-style execution.
- Data flow in EXPLAIN plans is from more deeply indented levels to less deeply indented levels.

Volcano-Style Plan

```
explain (costs off) select * from tenk1 t1 left join
(tenk1 t2 join tenk1 t3 on t2.thousand = t3.unique2) on
t1.hundred = t2.hundred and t1.ten + t2.ten = t3.ten
where t1.unique1 = 1;
```

```
 Nested Loop Left Join
   ->  Index Scan using tenk1_unique1 on tenk1 t1
         Index Cond: (unique1 = 1)
   ->  Nested Loop
         Join Filter: ((t1.ten + t2.ten) = t3.ten)
         ->  Bitmap Heap Scan on tenk1 t2
               Recheck Cond: (t1.hundred = hundred)
               ->  Bitmap Index Scan on tenk1_hundred
                     Index Cond: (hundred = t1.hundred)
         ->  Index Scan using tenk1_unique2 on tenk1 t3
               Index Cond: (unique2 = t2.thousand)
```

Plan Data 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 Data 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 Data Structure: 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 also 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, Qual, Left & Right Subtrees (1)

- Target List: The list of columns or expressions that this node will produce.
- Filter or “Qual” Condition: A test that will be performed on each generated row; those that fail are discarded.
- Left and Right Subtrees: The inputs to the current plan node.
  - For example, the inputs to a join are the two relations being joined.
  - Many plan nodes have only one input, or none at all.

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

```
/* 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, Qual, Left & Right Subtrees (3)

```
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)
```

Plans With Many Inputs

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

Parameters

- In complex plans, it's hard to stick to strictly Volcano-style execution.
- For some kinds of plan constructs, we need a more flexible way to move data around.
- A parameter is a container for a single value which can be set by one part of the plan and then later used elsewhere.
- The planner is responsible for arranging the plan so that parameters are set before use, and updated when necessary.
- Parameters are numbered ($0, $1, etc.).

InitPlans & SubPlans (1)

- An InitPlan or SubPlan is a planning construct that is used by certain kinds of queries.
- Specifically, either an InitPlan or a SubPlan will be created when a subquery is used in a part of the query other than the FROM clause.
- Which one of these gets created depends on whether the subquery depends on the outer query level, as well as on exactly how the subquery is used.
- An InitPlan concludes by setting a parameter. It is typically run just once; once the parameter is set, it holds onto the assigned to it.

InitPlans & SubPlans (2)

```
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)
```

Plan nodes list InitPlans, not SubPlans!

- Each Plan node carries a list of associated initPlans.
- SubPlans are not directly attached to the Plan; they just appear in expressions.
- At runtime, the executor finds all the attached SubPlan structures and puts them into a list.

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

extParam & 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

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

 Seq Scan on public.int4_tbl ← allParam = {$0}
   Output: int4_tbl.f1
   Filter: (int4_tbl.f1 = $0)
   InitPlan 2 (returns $0)
     ->  Aggregate
           Output: min(abs(int4_tbl_1.f1))
           ->  Seq Scan on public.int4_tbl int4_tbl_1
                 Output: int4_tbl_1.f1
```

Hidden Parameters

```
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)
```

Hidden Parameters Revealed

```
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)
```

extParams & allParams: Execution

- allParam is used to decide which nodes to reset when we need to rescan.
- For example, a parameterized index scan needs to produce different results if the parameter changes.
- Some nodes, like Sort and Materialize, cache the data they output so that they can cheaply produce the same output again.
- But, if any of the parameters listed in allParam change, then the node needs to throw away any cached data and reread its input.
- As the input will have changed due to the different parameter, the output will also change.

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)
```

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.

ENTERPRISEDB

Thanks

- Any Questions?