EXPLAIN: Understand the query plan

15-Feb-2019 @ PGConf India

- Jeevan Chalke
jeevan.chalke@enterprisedb.com

Who Am I?

- My name is Jeevan Chalke
- I’m a Technical Architect at EnterpriseDB
  - Working from Pune, India office
- My PostgreSQL contribution includes
  - Reviews
  - Several bug fixes
  - Aggregate Pushdown to remote server (v10)
  - Partition-wise Aggregates (v11)
- Email
  - jeevan.chalke@enterprisedb.com

Agenda

- Brief on optimizer/planner
- Paths and Plan
- EXPLAIN and EXPLAIN ANALYZE
  - Scans - seq, index, bitmap
  - Joins
  - Sort
  - Limit
  - Parallelism
  - Grouping
  - Partitioning
  - FDW
- Options used with EXPLAIN

Brief on optimizer/planner

- Statistics
  - ANALYZE table
- Creates different Paths
  - The way query can be evaluated to give the desired output
- Gives metrics to them
- Chooses the cheapest path
- Converts Path to the Plan
- Executor then executes the query according to the chosen plan

Paths and Plan

- Costs
  - cpu_tuple_cost
  - cpu_operator_cost
  - ...
- Rows
  - Estimated
  - Actual
- Width
- Time
  - Actual
  - Planning
  - Execution

Basic EXPLAIN with seq scan, setup

```
# create table mytab1(a int, b int, c int);
# create table mytab2(x int, y int);

# insert into mytab1 select i, i%50, i%100
  from generate_series(1, 1000000) i;
# insert into mytab2 select i, i%30 from generate_series(1, 1000) i;

# analyze mytab1;
# analyze mytab2;

# select relname, relpages, reltuples from pg_class
  where relname like 'mytab%';
 relname | relpages | reltuples
---------+----------+-----------
 mytab1  |     5406 |     1e+06
 mytab2  |        5 |      1000
(2 rows)

# select name, setting from pg_settings
  where name in ('cpu_tuple_cost', 'seq_page_cost');
      name       | setting
-----------------+---------
 cpu_tuple_cost | 0.01
 seq_page_cost  | 1
(2 rows)
```

Basic EXPLAIN with seq scan

- Syntax:
  - EXPLAIN [(option [, ...])] statement
  - EXPLAIN [ANALYZE] [VERBOSE] statement

```
# explain
  select * from mytab1;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on mytab1  (cost=0.00..15406.00 rows=1000000 width=12)
(1 row)
```

- For a seq scan; the planner has to do two things
  - Read all the pages
  - Read all the tuples from each page
- So the cost will be

  5406 (`relpages`) * 1 (`seq_page_cost`) +

  1000000 (`reltuples`) * 0.01 (`cpu_tuple_cost`) =

  15406.00

With WHERE clause

```
# explain
  select * from mytab1 where a < 800000;
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on mytab1  (cost=0.00..17906.00 rows=800411 width=12)
   Filter: (a < 800000)
(2 rows)
```

- Filter is added in the output showing the condition
- Number of rows reduced
- Cost increased; cpu_operator_cost = 0.0025 (+2500)

With an Index

- Let’s add an index on mytab1 and run the same query again

```
# create index myidx1 on mytab1(a);

# explain
  select * from mytab1 where a < 800000;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on mytab1  (cost=0.00..17906.00 rows=800411 width=12)
   Filter: (a < 800000)
(2 rows)
```

- Same result, why?

Let’s understand EXPLAIN ANALYZE first...

EXPLAIN ANALYZE

- Gives the EXPLAIN plan
- Executes the query; discards the output
- Gives actual timing and row details
- Gives a few more finer details; like loops and allows using BUFFERS option
- Provides summary showing planning and execution time
- Risky with DML commands
  - For example, an EXPLAIN ANALYZE DELETE... will actually delete the rows from a table

With an Index (continued...)

```
# explain (analyze)
  select * from mytab1 where a < 800000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on mytab1  (cost=0.00..17906.00 rows=800411 width=12) (actual
time=0.019..173.306 rows=799999 loops=1)
   Filter: (a < 800000)
   Rows Removed by Filter: 200001
 Planning Time: 0.127 ms
 Execution Time: 201.754 ms
(5 rows)

# set enable_seqscan to off;  -- Forces Index scan

# explain (analyze)
  select * from mytab1 where a < 800000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using myidx1 on mytab1  (cost=0.42..27130.62 rows=800411
width=12) (actual time=0.226..372.299 rows=799999 loops=1)
   Index Cond: (a < 800000)
 Planning Time: 0.098 ms
 Execution Time: 399.717 ms
(4 rows)
```

With an Index (continued...)

```
# set enable_seqscan to on;

# explain (analyze)
  select * from mytab1 where a > 800000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using myidx1 on mytab1  (cost=0.42..6767.22 rows=199588
width=12) (actual time=0.135..115.500 rows=200000 loops=1)
   Index Cond: (a > 800000)
 Planning Time: 0.156 ms
 Execution Time: 123.236 ms
(4 rows)
```

- So, having an index doesn't always improve the performance.
- But it does in most of the cases and you have to figure it out by looking at your data.

With an Index, with specific column

```
# explain (analyze)
  select a from mytab1 where a > 800000;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using myidx1 on mytab1  (cost=0.42..6767.22 rows=199588
width=4) (actual time=0.060..77.517 rows=200000 loops=1)
   Index Cond: (a > 800000)
   Heap Fetches: 200000
 Planning Time: 0.091 ms
 Execution Time: 85.114 ms
(5 rows)
```

- Index Scan is changed to Index Only Scan
- Required column is part of an index itself so no need to scan the actual table

- Available planner options
  - enable_seqscan, enable_indexscan, enable_indexonlyscan, enable_bitmapscan

With Bitmap Index/Heap Scan

```
# create index myidx2 on mytab1(c);

# explain (costs off)
  select a, c from mytab1 where a > 800000 or c <= 100000;
                    QUERY PLAN
-------------------------------------------------
 Bitmap Heap Scan on mytab1
   Recheck Cond: ((a > 800000) OR (c <= 100000))
   ->  BitmapOr
         ->  Bitmap Index Scan on myidx1
               Index Cond: (a > 800000)
         ->  Bitmap Index Scan on myidx2
               Index Cond: (c <= 100000)
(7 rows)
```

- Two-step scanning
  - Find final rows in the output, locate them (Inner Plan)
    - ANDs or Ors the multiple conditions as needed
  - Actually, fetches only those rows (Outer Plan)
- Outer plan sorts the rows physical location and then fetches those

JOINs

```
# explain (analyze, costs off)
  select a, x from mytab1 t1 left join mytab2 t2 on (t1.a = t2.x);
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Left Join (actual time=0.670..298.946 rows=1000000 loops=1)
   Hash Cond: (t1.a = t2.x)
   ->  Seq Scan on mytab1 t1 (actual time=0.017..102.551 rows=1000000 ..
   ->  Hash (actual time=0.642..0.642 rows=1000 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 44kB
         ->  Seq Scan on mytab2 t2 (actual time=0.009..0.349 rows=1000 ..
 Planning Time: 0.294 ms
 Execution Time: 330.789 ms
(8 rows)

# explain (costs off)
  select a from mytab1 t1 where a in (select x from mytab2);
                   QUERY PLAN
-----------------------------------------------
 Merge Semi Join
   Merge Cond: (t1.a = mytab2.x)
   ->  Index Only Scan using myidx1 on mytab1 t1
   ->  Sort
         Sort Key: mytab2.x
         ->  Seq Scan on mytab2
(6 rows)
```

JOINs summary

- Merge Join
  - Sorts, and then merges
  - Faster for bigger data-set
- Hash Join
  - Works with equality constraints
  - Faster provided we have enough memory
  - Mostly used where one table is smaller
- Nested Loop
  - For smaller data-set
  - Mostly used for cross joins
- Join Types: Left/Right/Full/Anti/Semi/Inner
- Planner parameters
  - enable_hashjoin, enable_mergejoin, enable_nestloop

Sort

```
# explain (analyze, buffers)
  select * from mytab1 t1 where a < 200000 order by b;
                                  QUERY PLAN
-------------------------------------------------------------------------------
Sort  (cost=27679.73..28177.42 rows=199079 width=12) (actual
time=147.275..178.516 rows=199999 loops=1)
   Sort Key: b
   Sort Method: external merge  Disk: 4320kB
   Buffers: shared hit=1631, temp read=540 written=543
   ->  Index Scan using myidx1 on mytab1 t1  (cost=0.42..6752.31 rows=199079
width=12) (actual time=0.035..67.219 rows=199999 loops=1)
         Index Cond: (a < 200000)
         Buffers: shared hit=1631
 Planning Time: 0.130 ms
 Execution Time: 190.342 ms
(9 rows)
```

- Sort Key
- Sort Method is external merge sort
- Buffers option, temp read/written in blocks (8K)

Sort (continued...)

```
# set work_mem to '16MB';

# explain (analyze, buffers)
  select * from mytab1 t1 where a < 200000 order by b;
                                  QUERY PLAN
-------------------------------------------------------------------------------
Sort  (cost=24274.23..24771.92 rows=199079 width=12) (actual
time=127.852..157.664 rows=199999 loops=1)
   Sort Key: b
   Sort Method: quicksort  Memory: 15853kB
   Buffers: shared hit=1631
   ->  Index Scan using myidx1 on mytab1 t1  (cost=0.42..6752.31 rows=199079
width=12) (actual time=0.035..78.813 rows=199999 loops=1)
         Index Cond: (a < 200000)
         Buffers: shared hit=1631
 Planning Time: 0.150 ms
 Execution Time: 174.043 ms
(9 rows)
```

- Sort Method is quicksort due to increased work_mem sorting now done in-memory.

Limit

```
# explain (analyze, timing off)
  select * from mytab1 t1 where b < 20 limit 10;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.00..0.45 rows=10 width=12) (actual rows=10 loops=1)
   ->  Seq Scan on mytab1 t1  (cost=0.00..17906.00 rows=401200 width=12)
(actual rows=10 loops=1)
         Filter: (b < 20)
 Planning Time: 0.110 ms
 Execution Time: 0.096 ms
(5 rows)
```

- Scan stops as soon as LIMIT is reached
- Also, notice new option timing

Parallelism

```
# create table mytab3(x int, y int);
# insert into mytab3 select i, i%30 from generate_series(1, 10000000) i;
# analyze mytab3;
# set parallel_leader_participation to off;
# set parallel_tuple_cost = 0.1;
```

- Look for parameters related to Parallelism
  - parallel_leader_participation, parallel_setup_cost, parallel_tuple_cost, max_parallel_workers, max_parallel_workers_per_gather, etc.
- New nodes in EXPLAIN
  - Gather or Gather Merge
- Workers

Parallelism (continued...)

```
# explain (analyze, verbose, summary off)
  select x from mytab3 t1 where x < 2000000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=0.43..55317.41 rows=1982958 width=4) (actual
time=9.104..1052.454 rows=1999999 loops=1)
   Output: x
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Index Only Scan using myidx3 on public.mytab3 t1
(cost=0.43..55317.41 rows=991479 width=4) (actual time=0.102..661.237
rows=1000000 loops=2)
         Output: x
         Index Cond: (t1.x < 2000000)
         Heap Fetches: 1999999
         Worker 0: actual time=0.106..657.655 rows=1063596 loops=1
         Worker 1: actual time=0.099..664.820 rows=936403 loops=1
(10 rows)
```

- Gather collects tuples from all the workers
- Gather's result is always unsorted

Grouping

```
# explain (analyze, verbose)
  select count(*) from mytab2 t1 group by y having sum(x) > 17000;
                                QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate  (cost=22.50..22.88 rows=10 width=12) (actual
time=0.480..0.486 rows=5 loops=1)
   Output: count(*), y
   Group Key: t1.y
   Filter: (sum(t1.x) > 17000)
   Rows Removed by Filter: 25
   ->  Seq Scan on public.mytab2 t1  (cost=0.00..15.00 rows=1000 width=8)
(actual time=0.013..0.108 rows=1000 loops=1)
         Output: x, y
 Planning Time: 0.081 ms
 Execution Time: 0.545 ms
(9 rows)
```

- Group keys are displayed, and Filter shows the Having clause
- Parameter enable_hashagg can be used to disable HashAggregate

Grouping with parallelism

```
# explain verbose
  select count(*) from mytab3 t1 group by y having sum(x) > 17000;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=132749.06..132756.89 rows=10 width=12)
   Output: count(*), y
   Group Key: t1.y
   Filter: (sum(t1.x) > 17000)
   ->  Gather Merge  (cost=132749.06..132756.06 rows=60 width=20)
         Output: y, (PARTIAL count(*)), (PARTIAL sum(x))
         Workers Planned: 2
         ->  Sort  (cost=131749.04..131749.11 rows=30 width=20)
               Output: y, (PARTIAL count(*)), (PARTIAL sum(x))
               Sort Key: t1.y
               ->  Partial HashAggregate  (cost=131748.00..131748.30 rows=30
width=20)
                     Output: y, PARTIAL count(*), PARTIAL sum(x)
                     Group Key: t1.y
                     ->  Parallel Seq Scan on public.mytab3 t1
(cost=0.00..94248.00 rows=5000000 width=8)
                           Output: x, y
(15 rows)
```

MIN()/MAX() aggregates

```
# explain (analyze, costs off) select min(b) from mytab1 t1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate (actual time=214.125..214.125 rows=1 loops=1)
   ->  Seq Scan on mytab1 t1 (actual time=0.018..104.833 rows=1000000 loops=1)
 Planning Time: 0.134 ms
 Execution Time: 214.161 ms
(4 rows)

# explain (analyze, costs off) select max(a) from mytab1 t1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Result (actual time=3.365..3.366 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit (actual time=3.356..3.358 rows=1 loops=1)
           ->  Index Only Scan Backward using myidx1 on mytab1 t1 (actual
time=3.353..3.353 rows=1 loops=1)
                 Index Cond: (a IS NOT NULL)
                 Heap Fetches: 1
 Planning Time: 0.282 ms
 Execution Time: 3.478 ms
(8 rows)
```

- With an index, min/max executes really fast

Partitioning

```
# create table pagg_tab (a int, b int, c text, d int) partition by list(c);
# create table pagg_tab_p1 partition of pagg_tab
  for values in ('0000', '0001', '0002', '0003');
# create table pagg_tab_p2 partition of pagg_tab
  for values in ('0004', '0005', '0006', '0007');
# create table pagg_tab_p3 partition of pagg_tab
  for values in ('0008', '0009', '0010', '0011');
# insert into pagg_tab select i%20, i%30, to_char(i%12, 'FM0000'), i%30
  from generate_series(0, 2999) i;
# analyze pagg_tab;
# set enable_partitionwise_join to on;
# set enable_partitionwise_aggregate to on;
```

- Look for parameters related to Partitioning
  - enable_partition_pruning, enable_partitionwise_aggregate, enable_partitionwise_join
- New node Append

Partitioning (continued...)

```
# explain (analyze, costs off)
  select * from pagg_tab;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Append (actual time=0.016..0.650 rows=3000 loops=1)
   ->  Seq Scan on pagg_tab_p1 (actual time=0.015..0.132 rows=1000 loops=1)
   ->  Seq Scan on pagg_tab_p2 (actual time=0.008..0.130 rows=1000 loops=1)
   ->  Seq Scan on pagg_tab_p3 (actual time=0.012..0.137 rows=1000 loops=1)
 Planning Time: 0.189 ms
 Execution Time: 0.803 ms
(6 rows)
```

- Append of all children scanned sequentially

Partition-wise Join and Aggregation

```
# explain (costs off) select t1.c, count(*) from pagg_tab t1, pagg_tab t2
  where t1.c = t2.c and t1.c in ('0000', '0004') group by (t1.c);
                          QUERY PLAN
------------------------------------------------------------------
 Append
   ->  HashAggregate
         Group Key: t1.c
         ->  Hash Join
               Hash Cond: (t1.c = t2.c)
               ->  Seq Scan on pagg_tab_p1 t1
                     Filter: (c = ANY ('{0000,0004}'::text[]))
               ->  Hash
                     ->  Seq Scan on pagg_tab_p1 t2
   ->  HashAggregate
         Group Key: t1_1.c
         ->  Hash Join
               Hash Cond: (t1_1.c = t2_1.c)
               ->  Seq Scan on pagg_tab_p2 t1_1
                     Filter: (c = ANY ('{0000,0004}'::text[]))
               ->  Hash
                     ->  Seq Scan on pagg_tab_p2 t2_1
(17 rows)
```

- Both partition-wise join and aggregate kick in, along with partition pruning
- Useful when join condition and aggregation key involves partition key

Foreign Scan

```
# explain (analyze, verbose, costs off)
  select count(*) from fprt1 t1 inner join fprt2 t2 on (t1.a = t2.b) group by t1.a;
                                                                QUERY PLAN

-------------------------------------------------------------------------------------
-----------------------------------------------
 Append (actual time=0.799..1.612 rows=84 loops=1)
   ->  Foreign Scan (actual time=0.798..0.802 rows=42 loops=1)
         Output: (count(*)), t1.a
         Relations: Aggregate on ((public.ftprt1_p1 t1) INNER JOIN
(public.ftprt2_p1 t2))
         Remote SQL: SELECT count(*), r5.a FROM (public.fprt1_p1 r5 INNER JOIN
public.fprt2_p1 r8 ON (((r5.a = r8.b)))) GROUP BY 2
   ->  Foreign Scan (actual time=0.799..0.802 rows=42 loops=1)
         Output: (count(*)), t1_1.a
         Relations: Aggregate on ((public.ftprt1_p2 t1) INNER JOIN
(public.ftprt2_p2 t2))
         Remote SQL: SELECT count(*), r6.a FROM (public.fprt1_p2 r6 INNER JOIN
public.fprt2_p2 r9 ON (((r6.a = r9.b)))) GROUP BY 2
 Planning Time: 5.245 ms
 Execution Time: 1.865 ms
(11 rows)
```

- Foreign Scan and Remote SQL
- Both join and aggregation is pushed down on the remote server

Options

- ANALYZE [boolean]
  - Executes the query. Shows actual run times and other statistics
- VERBOSE [boolean]
  - Detailed output
- COSTS [boolean]
  - Shows estimated statistics and rows. Default TRUE
- BUFFERS [boolean]
  - Displays information on buffer usage. Used only with ANALYZE
- TIMING [boolean]
  - Show actual times. Used only with ANALYZE, default TRUE
- SUMMARY [boolean]
  - Gives summary on planning and execution times
- FORMAT { TEXT | XML | JSON | YAML }
  - Output display format. Default TEXT
- Explain works with SELECT, INSERT, UPDATE, DELETE, VALUES, EXECUTE, DECLARE, CREATE TABLE AS, or CREATE MATERIALIZED VIEW queries

Format

```
# explain (format json, verbose off,
  analyze, summary, timing, costs,
  buffers off)
  select * from mytab1;
            QUERY PLAN
-------------------------------------
 [                                  +
   {                                +
     "Plan": {                      +
       "Node Type": "Seq Scan",     +
       "Parallel Aware": false,     +
       "Relation Name": "mytab1",   +
       "Alias": "mytab1",           +
       "Startup Cost": 0.00,        +
       "Total Cost": 15406.00,      +
       "Plan Rows": 1000000,        +
       "Plan Width": 12,            +
       "Actual Startup Time": 0.018,+
       "Actual Total Time": 154.808,+
       "Actual Rows": 1000000,      +
       "Actual Loops": 1            +
     },                             +
     "Planning Time": 0.081,        +
     "Triggers": [                  +
     ],                             +
     "Execution Time": 198.105      +
   }                                +
 ]
(1 row)
```

```
# explain (format xml, verbose off,
  analyze, summary, timing, costs,
  buffers off)
  select * from mytab1;
                          QUERY PLAN
-------------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">+
   <Query>                                               +
     <Plan>                                              +
       <Node-Type>Seq Scan</Node-Type>                   +
       <Parallel-Aware>false</Parallel-Aware>            +
       <Relation-Name>mytab1</Relation-Name>             +
       <Alias>mytab1</Alias>                             +
       <Startup-Cost>0.00</Startup-Cost>                 +
       <Total-Cost>15406.00</Total-Cost>                 +
       <Plan-Rows>1000000</Plan-Rows>                    +
       <Plan-Width>12</Plan-Width>                       +
       <Actual-Startup-Time>0.016</Actual-Startup-Time>  +
       <Actual-Total-Time>133.351</Actual-Total-Time>    +
       <Actual-Rows>1000000</Actual-Rows>                +
       <Actual-Loops>1</Actual-Loops>                    +
     </Plan>                                             +
     <Planning-Time>0.095</Planning-Time>                +
     <Triggers>                                          +
     </Triggers>                                         +
     <Execution-Time>171.051</Execution-Time>            +
   </Query>                                              +
 </explain>
(1 row)
```

References

- EXPLAIN
  - https://www.postgresql.org/docs/11/sql-explain.html
- Using EXPLAIN
  - https://www.postgresql.org/docs/11/using-explain.html
- Other web links
  - http://www.postgresqltutorial.com/postgresql-explain/
  - https://momjian.us/main/writings/pgsql/optimizer.pdf
  - https://wiki.postgresql.org/images/4/45/Explaining_EXPLAIN.pdf
  - https://www.dalibo.org/_media/understanding_explain.pdf
- Query Planning Gone Wrong by Robert Haas
  - http://rhaas.blogspot.com/2013/05/query-planning-gone-wrong.html