Explaining EXPLAIN:
An Introduction to PostgreSQL EXPLAIN Plans

SCaLE 19x | 2022.07.28 | Richard Yen

About Me

- Support Engineer at EnterpriseDB since 2015
- Previously a DBA and Web Developer
- Been using PostgreSQL since v. 7.4

Why is my query slow?

EDB™

Tell Postgres to EXPLAIN!

EDB

Our Roadmap

- What does EXPLAIN do?
- How does EXPLAIN work?
- How do I interpret EXPLAIN output (scans, joins, etc.)?
- Some non-trivial real-world examples of how EXPLAIN can help

What does EXPLAIN do?

- Explains what Postgres plans to do for a query (EXPLAIN)
- Explains what Postgres did for a query (EXPLAIN ANALYZE)

What doesn't EXPLAIN do?

- Won’t explain why the query planner made some choice
- Won’t tell you about query performance being affected by another session
- Won’t tell you about stuff happening outside the database (i.e., in the OS)
- Won’t tell you about external environmental factors (i.e., network latency)

How does the query planner work?

- Cost-based approach
- Uses Table/Index Statistics
  - Stored in `pg_statistic` (don’t look there)
  - Can be viewable by looking `pg_stats` (for the adventurous)
  - Refreshed with `ANALYZE` (not to be confused with `EXPLAIN ANALYZE`)
- Tuned by Configuration
  - `enable_*` parameters
  - `*_cost` parameters

Config Parameters

EDB

Cost Parameters

- cpu_index_tuple_cost
- cpu_operator_cost
- cpu_tuple_cost
- jit_above_cost
- jit_inline_above_cost
- jit_optimize_above_cost
- parallel_setup_cost
- parallel_tuple_cost
- random_page_cost
- seq_page_cost

Join Parameters

- enable_bitmapscan
- enable_gathermerge
- enable_hashjoin
- enable_mergejoin
- enable_nestloop
- enable_partitionwise_join

Scan Parameters

- enable_indexonlyscan
- enable_indexscan
- enable_seqscan
- enable_tidscan

Other Parameters

- enable_hashagg
- enable_parallel_append
- enable_parallel_hash
- enable_partition_pruning
- enable_partitionwise_aggregate
- enable_material
- enable_sort

```
SELECT *
  FROM pg_settings
 WHERE name LIKE '%cost'
    OR name LIKE 'enable%';
```

What does EXPLAIN do?

```
bash $ pgbench -i && psql
<...>
postgres=# EXPLAIN SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON (a.bid=b.bid) WHERE a.aid < 100000;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..4141.00 rows=99999 width=461)
   Join Filter: (a.bid = b.bid)
   ->  Seq Scan on pgbench_branches b  (cost=0.00..1.01 rows=1 width=364)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..2890.00 rows=99999 width=97)
         Filter: (aid < 100000)
(5 rows)
```

???

Cost Calculation

```
Nested Loop  (cost=0.00..4141.00 rows=99999 width=461)
  Join Filter: (a.bid = b.bid)
  ->  Seq Scan on pgbench_branches b  (cost=0.00..1.01 rows=1 width=364)
  ->  Seq Scan on pgbench_accounts a  (cost=0.00..2890.00 rows=99999 width=97)
```

```
cost = ( #blocks * seq_page_cost ) + ( #records * cpu_tuple_cost ) + ( #records * cpu_filter_cost )

postgres=# select pg_relation_size('pgbench_accounts');
 pg_relation_size
------------------
         13434880

block_size      = 8192   (8kB, typical OS)
#blocks         = 1640   (relation_size / block_size)
#records        = 100000
seq_page_cost   = 1      (default)
cpu_tuple_cost  = 0.01   (default)
cpu_filter_cost = 0.0025 (default)

cost = ( 1640 * 1 ) + ( 100000 * 0.01 ) + ( 100000 * 0.0025 ) = 2890
```

For Realz (w/ ANALYZE)

```
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON (a.bid=b.bid) WHERE a.aid < 100000;
                                                   QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..4141.00 rows=99999 width=461) (actual time=0.039..56.582 rows=99999 loops=1)
   Join Filter: (a.bid = b.bid)
   ->  Seq Scan on pgbench_branches b  (cost=0.00..1.01 rows=1 width=364) (actual time=0.025..0.026 rows=1 loops=1)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..2890.00 rows=99999 width=97) (actual time=0.008..25.752 rows=99999 loops=1)
         Filter: (aid < 100000)
         Rows Removed by Filter: 1
 Planning Time: 0.306 ms
 Execution Time: 61.031 ms
(8 rows)
```

For Realz Realz (w/ ANALYZE & BUFFERS)

```
postgres=# EXPLAIN (BUFFERS, ANALYZE) SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON (a.bid=b.bid)
             WHERE a.aid < 100000;
                                                    QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..4141.00 rows=99999 width=461) (actual time=0.039..56.582 rows=99999 loops=1)
   Join Filter: (a.bid =      )
   Buffers: shared hit=3 read=1638
   ->  Seq Scan on pgbench_branches b  (cost=0.00..1.01 rows=1 width=364) (actual time=0.025..0.026 rows=1 loops=1)
       Buffers: shared hit=1
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..2890.00 rows=99999 width=97) (actual time=0.008..25.752 rows=99999
loops=1)
         Filter: (aid < 100000)
         Rows Removed by Filter: 1
         Buffers: shared hit=2 read=1638
 Planning Time: 0.306 ms
 Execution Time: 61.031 ms
(8 rows)
```

The Building Blocks:

Joins & Scans

EDB™

Our Example (with more rows)

```
postgres=# INSERT INTO pgbench_branches (bid, bbalance, filler) VALUES (generate_series(2,100000),1,'');
INSERT 0 99999
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON (a.bid=b.bid) WHERE a.aid <
100000;
```

Our Example (with more rows)

```
postgres=# INSERT INTO pgbench_branches (bid, bbalance, filler) VALUES (generate_series(2,100000),1,'');
INSERT 0 99999
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON (a.bid=b.bid) WHERE a.aid <
100000;
                                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=1676.90..4830.08 rows=99999 width=461) (actual time=147.289..229.678 rows=99999 loops=1)
   Hash Cond: (a.bid = b.bid)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..2890.00 rows=99999 width=97) (actual time=0.020..26.903 rows=99999
loops=1)
         Filter: (aid < 100000)
         Rows Removed by Filter: 1
   ->  Hash  (cost=1656.40..1656.40 rows=1640 width=364) (actual time=63.742..63.743 rows=100000 loops=1)
         Buckets: 32768 (originally 2048)  Batches: 4 (originally 1)  Memory Usage: 3841kB
         ->  Seq Scan on pgbench_branches b  (cost=0.00..1656.40 rows=1640 width=364) (actual time=0.014..22.897
rows=100000 loops=1)
 Planning Time: 0.278 ms
 Execution Time: 234.480 ms
(10 rows)
```

Our Example (with Hash Join off)

```
postgres=# set enable_hashjoin to off;
SET
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON (a.bid=b.bid) WHERE a.aid < 100000;
                                                                        QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------
-------------------------
 Nested Loop  (cost=0.29..5389.80 rows=99999 width=461) (actual time=0.056..104.242 rows=99999 loops=1)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..2890.00 rows=99999 width=97) (actual time=0.014..32.394 rows=99999 loops=1)
         Filter: (aid < 100000)
         Rows Removed by Filter: 1
   ->  Memoize  (cost=0.29..0.38 rows=1 width=364) (actual time=0.000..0.000 rows=1 loops=99999)
         Cache Key: a.bid
         Hits: 99998  Misses: 1  Evictions: 0  Overflows: 0  Memory Usage: 1kB
         ->  Index Scan using pgbench_branches_pkey on pgbench_branches b  (cost=0.28..0.37 rows=1 width=364) (actual
time=0.014..0.014 rows=1 loops=1)
               Index Cond: (bid = a.bid)
 Planning Time: 0.292 ms
 Execution Time: 109.068 ms
(11 rows)
```

Our Example (seeking less rows)

```
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON (a.bid=b.bid) WHERE a.aid < 100;
```

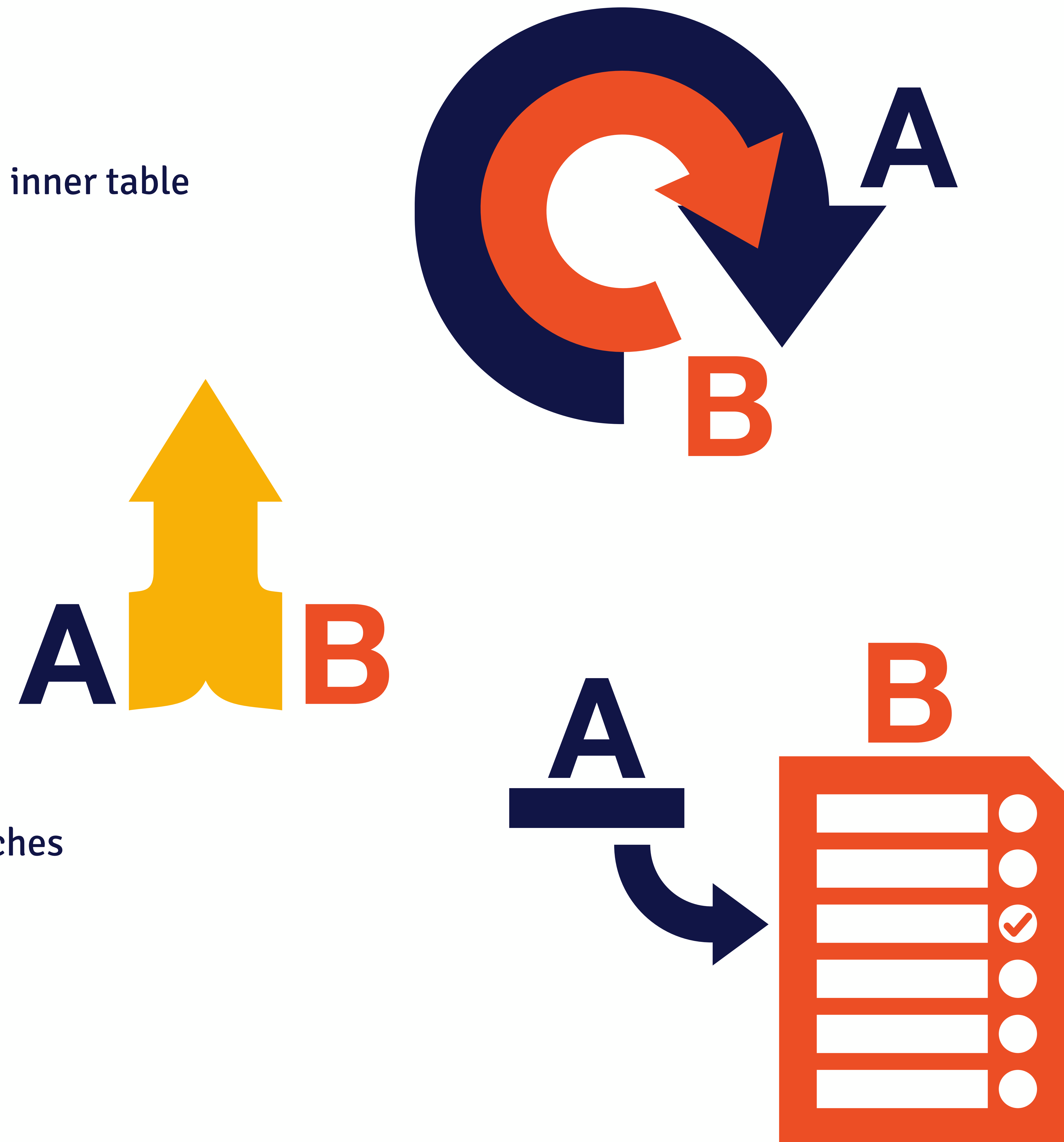
Our Example (seeking less rows)

```
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON (a.bid=b.bid) WHERE a.aid < 100;
                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=14.60..16.13 rows=99 width=194) (actual time=0.094..0.130 rows=99 loops=1)
   Merge Cond: (b.bid = a.bid)
   ->  Index Scan using pgbench_branches_pkey on pgbench_branches b  (cost=0.29..4247.29 rows=100000 width=97) (actual
time=0.013..0.014 rows=2 loops=1)
   ->  Sort  (cost=14.31..14.55 rows=99 width=97) (actual time=0.071..0.079 rows=99 loops=1)
         Sort Key: a.bid
         Sort Method: quicksort  Memory: 38kB
         ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts a  (cost=0.29..11.03 rows=99 width=97) (actual
time=0.010..0.033 rows=99 loops=1)
               Index Cond: (aid < 100)
 Planning Time: 0.931 ms
 Execution Time: 0.205 ms
(10 rows)
```

A word on Joins

EDB

- **Nested Loops**
  - For each row in outer table, scan for matching rows in the inner table
  - Fast to start, best for small tables
- **Merge Join**
  - Zipper-operation on **sorted** data sets
  - Good for large tables
  - High startup cost if additional sort is required
- **Hash Join**
  - Build hash of inner table values, scan outer table for matches
  - Only usable for equality conditions
  - High startup cost, but fast execution



Scan Improvements

```
postgres=# UPDATE pgbench_accounts SET bid = aid;
UPDATE 100000
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts WHERE bid = 1;
                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..5778.24 rows=199939 width=97) (actual time=19.322..45.161 rows=1 loops=1)
   Filter: (bid = 1)
   Rows Removed by Filter: 99999
 Planning Time: 0.101 ms
 Execution Time: 45.191 ms
(5 rows)

postgres=# CREATE INDEX pgba_bid_idx ON pgbench_accounts (bid);
CREATE INDEX
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts WHERE bid = 1;
                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
---------
 Index Scan using pgba_bid_idx on pgbench_accounts  (cost=0.29..8.31 rows=1 width=97) (actual time=0.076..0.077 rows=1
loops=1)
   Index Cond: (bid = 1)
 Planning Time: 0.312 ms
 Execution Time: 0.119 ms
(4 rows)
```

The Fastest Scan

```
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_accounts where aid < 1000;
                                                            QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
---------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.43..47.87 rows=939 width=97) (actual
time=0.371..0.721 rows=999 loops=1)
   Index Cond: (aid < 1000)
 Planning Time: 0.226 ms
 Execution Time: 0.815 ms
(4 rows)

postgres=# EXPLAIN ANALYZE SELECT aid FROM pgbench_accounts where aid < 1000;
                                                            QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
---------
 Index Only Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.43..28.87 rows=939 width=4) (actual
time=0.022..0.169 rows=999 loops=1)
   Index Cond: (aid < 1000)
   Heap Fetches: 0
 Planning Time: 0.161 ms
 Execution Time: 0.237 ms
(5 rows)
```

Index Scan Costs

```
postgres=# show random_page_cost;
 random_page_cost
------------------
 4
(1 row)
postgres=# EXPLAIN SELECT * FROM pgbench_accounts WHERE aid < 1000;
                                            QUERY PLAN
---------------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.29..50.30 rows=1029 width=97)
   Index Cond: (aid < 1000)
(2 rows
postgres=# SET random_page_cost = 100;
postgres=# EXPLAIN SELECT * FROM pgbench_accounts WHERE aid < 1000;
                                             QUERY PLAN
----------------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.29..434.30 rows=1029 width=97)
   Index Cond: (aid < 1000)
(2 rows)
postgres=# SET random_page_cost = 1000;
postgres=# EXPLAIN SELECT * FROM pgbench_accounts WHERE aid < 1000;
                              QUERY PLAN
----------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..2890.00 rows=1029 width=97)
   Filter: (aid < 1000)
(2 rows)
```

Index Scan Isn't Always Better

If you have 10,000 rows and you want aid < 10,000, you're going to scan the entire table anyways

Scan Types

EDB

Sequential Scan

- Scan the whole table
- Can be chosen if query planner thinks it will retrieve many matching rows

Index Scan

- Scan all/some rows in index; look up rows in heap
- Causes random seek

Index Only Scan

- Scan all/some rows in index
- No need to look up rows in heap

Bitmap Heap Scan

- Scan index, building a bitmap of pages to visit
- Look up only relevant pages in heap for rows

EDB

EXPLAIN-ing the Unexpected

Other things EXPLAIN can show you

Bad Statistics

```
$ pgbench -T 300 && psql
postgres=# CREATE INDEX foo ON pgbench_history (aid);
CREATE INDEX
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_history WHERE aid < 100;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on pgbench_history  (cost=0.00..2346.00 rows=35360 width=50) (actual time=0.221..22.91 rows=170 loops=1)
   Filter: (aid < 100)
   Rows Removed by Filter: 159911
 Planning Time: 0.610 ms
 Execution Time: 24.292 ms
(6 rows)
postgres=# ANALYZE;
ANALYZE
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_history WHERE aid < 100;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Index Scan using foo on pgbench_history  (cost=0.42..579.09 rows=153 width=50) (actual time=0.017..1.913 rows=170 loops=1)
   Index Cond: (aid < 100)
 Planning Time: 0.167 ms
 Execution Time: 3.507 ms
(5 rows)
```

VACUUM and ANALYZE often!

autovacuum will help you with that 👍🏻

EDB™

CREATE STATISTICS

```
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    [ ( statistics_kind [, ... ] ) ]
    ON column_name, column_name [, ...]
    FROM table_name
```

Insufficient Memory Allocation

```
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_history WHERE delta < 0 ORDER BY delta;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=12225.68..12424.74 rows=79623 width=50) (actual time=1187.391..1763.319 rows=79875 loops=1)
   Sort Key: delta
   Sort Method: external merge  Disk: 2664kB
   ->  Seq Scan on pgbench_history  (cost=0.00..3021.01 rows=79623 width=50) (actual time=0.023..593.128 rows=79875 loops=1)
         Filter: (delta < 0)
         Rows Removed by Filter: 80206
 Planning Time: 0.082 ms
 Execution Time: 2312.374 ms
(8 rows)
postgres=# SHOW work_mem ;
 work_mem
----------
 4MB
(1 row)
postgres=# SET work_mem = '16 MB';
SET
postgres=# EXPLAIN ANALYZE SELECT * FROM pgbench_history WHERE delta < 0 ORDER BY delta;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=9502.68..9701.74 rows=79623 width=50) (actual time=1128.871..1662.322 rows=79875 loops=1)
   Sort Key: delta
   Sort Method: quicksort  Memory: 9313kB
   ->  Seq Scan on pgbench_history  (cost=0.00..3021.01 rows=79623 width=50) (actual time=0.021..569.691 rows=79875 loops=1)
         Filter: (delta < 0)
         Rows Removed by Filter: 80206
 Planning Time: 0.083 ms
 Execution Time: 2187.715 ms
(8 rows)
```

Index Definition Mismatch

```
postgres=# CREATE INDEX fillertext_idx ON pgbench_history (aid, substring(filler,1,1));
postgres=# EXPLAIN SELECT * FROM pgbench_history WHERE aid = 10000 AND left(filler,1) = 'b';
                              QUERY PLAN
--------------------------------------------------------------------------
 Bitmap Heap Scan on pgbench_history  (cost=4.44..12.26 rows=1 width=47)
   Recheck Cond: (aid = 10000)
   Filter: ("left"((filler)::text, 1) = 'b'::text)
   Heap Blocks: exact=2
   ->  Bitmap Index Scan on fillertext_idx  (cost=0.00..4.43 rows=2 width=0)
         Index Cond: (aid = 10000)
(6 rows)
postgres=# EXPLAIN SELECT * FROM pgbench_history WHERE aid = 10000 AND substring(lower(filler),1,1) = 'b';
                              QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on pgbench_history  (cost=4.44..12.26 rows=1 width=47)
   Recheck Cond: (aid = 10000)
   Filter: ("substring"(lower((filler)::text), 1, 1) = 'b'::text)
   Heap Blocks: exact=2
   ->  Bitmap Index Scan on fillertext_idx  (cost=0.00..4.43 rows=2 width=0)
         Index Cond: (aid = 10000)
(6 rows)
```

Index Definition Mismatch

```
postgres=# CREATE INDEX fillertext_idx ON pgbench_history (aid, substring(filler,1,1));
postgres=# EXPLAIN SELECT * FROM pgbench_history WHERE aid = 10000 AND left(filler,1) = 'b';
postgres=# EXPLAIN SELECT * FROM pgbench_history WHERE aid = 10000 AND substring(lower(filler),1,1) = 'b';


postgres=# EXPLAIN SELECT * FROM pgbench_history WHERE aid = 10000 AND substring(filler,1,1) = 'b';
                                  QUERY PLAN
-------------------------------------------------------------------------------------
 Index Scan using fillertext_idx on pgbench_history  (cost=0.42..8.44 rows=1 width=47)
   Index Cond: ((aid = 10000) AND ("substring"((filler)::text, 1, 1) = 'b'::text))
(2 rows)
```

Index Definition Mismatch (another example)

```
postgres=# CREATE INDEX idx_org_dept ON org ((info -> 'dept'::text) ->> 'name'::text));
CREATE TABLE

postgres=# explain SELECT * FROM org where 'aa'::text IN (SELECT jsonb_array_elements(info -> 'dept') ->> 'name');
                              QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on org  (cost=0.00..719572.55 rows=249996 width=1169)
   Filter: (SubPlan 1)
   SubPlan 1
     ->  Result  (cost=0.00..2.27 rows=100 width=32)
           ->  ProjectSet  (cost=0.00..0.52 rows=100 width=32)
                 ->  Result  (cost=0.00..0.01 rows=1 width=0)

postgres=# explain SELECT * FROM organization where 'aa'::text IN (info -> 'dept' ->> 'name');
                                         QUERY PLAN
----------------------------------------------------------------------------------------------
 Index Scan using idx_org_dept on org  (cost=0.42..8.44 rows=1 width=1169)
   Index Cond: ('aa'::text = ((info -> 'dept'::text) ->> 'name'::text))
(2 rows)
```

Some Other Situations

EDB

- Prepared Statements
  - `PREPARE foo AS SELECT * FROM pgbench_accounts WHERE aid = $1;`
  - First 5 executions use a custom plan (taking into account $1)
  - After that, a generic plan is used (often not very efficient)
  - Can adjust `plan_cache_mode` in v. 12 and later
- Join order
- JIT (Just-In-Time Compilation)
- from_collapse_limit/ join_collapse_limit
- ORMs

Unexpected Behavior

```
postgres=# \d mytable
                    Table "public.mytable"
 Column |            Type             | Collation | Nullable | Default
--------+-----------------------------+-----------+----------+---------
 col_a  | numeric                     |           | not null |
 col_b  | numeric                     |           | not null |
 col1   | character varying(128)      |           |          |
 col2   | character varying(512)      |           |          |
 col3   | character varying(128)      |           |          |
 col4   | timestamp without time zone |           |          |
 col5   | character varying(128)      |           |          |

postgres=# EXPLAIN (ANALYZE, BUFFERS) UPDATE mytable SET col1 = 'A', col2 = 'text', (...) WHERE col_a = '3443949' AND col_b = '2222696';
                                                      QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Update on mytable  (cost=0.43..8.45 rows=1 width=1364) (actual time=0.167..0.167 rows=0 loops=1)
   Buffers: shared hit=10
   ->  Index Scan using "mytable_idx" on mytable (cost=0.43..8.45 rows=1 width=1364) (actual time=0.074..0.074 rows=1 loops=1)
         Index Cond: ((mytable.col_a = '3443949'::numeric) AND (mytable.col_b = '2222696'::numeric))
         Buffers: shared hit=4
 Planning time: 0.480 ms
 Execution time: 0.252 ms
(8 rows)
```

40.922 ms ????

Unexpected Behavior

```
postgres=# \d mytable
                   Table "public.mytable"
 Column |            Type             | Collation | Nullable | Default
--------+-----------------------------+-----------+----------+---------
 col_a  | numeric                     |           | not null |
 col_b  | numeric                     |           | not null |
 col1   | character varying(128)      |           |          |
 col2   | character varying(512)      |           |          |
 col3   | character varying(128)      |           |          |
 col4   | timestamp without time zone |           |          |
 col5   | character varying(128)      |           |          |
```

40.922 ms !!!!

```
duration: 40.922 ms  statement: EXPLAIN (ANALYZE, BUFFERS) UPDATE mytable SET col1 = 'A', col2 = 'text', (...) WHERE col_a = '3443949' AND
col_b = '2222696';


Update on mytable (cost=0.00..89304.06 rows=83 width=1364) (actual time=889.070..889.070 rows=0 loops=1)
 ->  Seq Scan on mytable  (cost=0.00..89304.06 rows=83 width=1364) (actual time=847.736..850.867 rows=1 loops=1)
       Filter: ((mytable.col_a)::double precision = '3443949'::double precision) AND ((mytable.col_b)::double precision =
'2222696'::double precision))
       Rows Removed by Filter: 3336167
```

auto_explain

- Prints `EXPLAIN` plans to your log
- Can do `EXPLAIN ANALYZE` (and `BUFFERS`, `FORMAT`, etc.)
- Can even log trigger statistics and nested statements
- Can be done on a per-session basis with `LOAD auto_explain`;
- Creates additional I/O on disk

Thank You

EDB supercharges Postgres to help our customers overcome these challenges.

EDB™