Beyond Joins and Indexes

BRUCE MOMJIAN

As a follow up to the presentation, *Explaining the Postgres Query Optimizer*, this talk shows the non-join and non-index operations that the optimizer can choose.
My previous talk, *Explaining the Postgres Query Optimizer*, covered:

- Query optimization basics
- Optimizer statistics
- Join methods
- Scan methods, including indexes
- Limit

https://momjian.us/main/presentations/performance.html#optimizer
<table>
<thead>
<tr>
<th>No.</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Result</td>
</tr>
<tr>
<td>2.</td>
<td>Values Scan</td>
</tr>
<tr>
<td>3.</td>
<td>Function Scan</td>
</tr>
<tr>
<td>4.</td>
<td>Incremental Sort</td>
</tr>
<tr>
<td>5.</td>
<td>Unique</td>
</tr>
<tr>
<td>6.</td>
<td>Append</td>
</tr>
<tr>
<td>7.</td>
<td>Merge Append</td>
</tr>
<tr>
<td>8.</td>
<td>Subquery Scan</td>
</tr>
<tr>
<td>9.</td>
<td>HashSetOp</td>
</tr>
<tr>
<td>10.</td>
<td>SetOp</td>
</tr>
<tr>
<td>11.</td>
<td>Materialize</td>
</tr>
<tr>
<td>12.</td>
<td>Memoize</td>
</tr>
<tr>
<td>13.</td>
<td>Group</td>
</tr>
<tr>
<td>14.</td>
<td>Aggregate</td>
</tr>
<tr>
<td>15.</td>
<td>Group Aggregate</td>
</tr>
<tr>
<td>16.</td>
<td>Hash Aggregate</td>
</tr>
<tr>
<td>17.</td>
<td>Mixed Aggregate</td>
</tr>
<tr>
<td>18.</td>
<td>Window Aggregate</td>
</tr>
<tr>
<td>19.</td>
<td>Parallel Seq Scan</td>
</tr>
<tr>
<td>20.</td>
<td>Partial Aggregate</td>
</tr>
<tr>
<td>21.</td>
<td>Gather</td>
</tr>
<tr>
<td>22.</td>
<td>Finalize Aggregate</td>
</tr>
<tr>
<td>23.</td>
<td>Gather Merge</td>
</tr>
<tr>
<td>24.</td>
<td>Parallel Append</td>
</tr>
<tr>
<td>25.</td>
<td>Parallel Hash</td>
</tr>
<tr>
<td>26.</td>
<td>Parallel Hash Join</td>
</tr>
<tr>
<td>27.</td>
<td>CTE Scan</td>
</tr>
<tr>
<td>28.</td>
<td>WorkTable Scan</td>
</tr>
<tr>
<td>29.</td>
<td>Recursive Union</td>
</tr>
<tr>
<td>30.</td>
<td>Project Set</td>
</tr>
<tr>
<td>31.</td>
<td>LockRows</td>
</tr>
<tr>
<td>32.</td>
<td>Sample Scan</td>
</tr>
<tr>
<td>33.</td>
<td>Table Function Scan</td>
</tr>
<tr>
<td>34.</td>
<td>Foreign Scan</td>
</tr>
<tr>
<td>35.</td>
<td>Tid Scan</td>
</tr>
<tr>
<td>36.</td>
<td>Insert</td>
</tr>
<tr>
<td>37.</td>
<td>Update</td>
</tr>
<tr>
<td>38.</td>
<td>Delete</td>
</tr>
<tr>
<td>39.</td>
<td>Merge</td>
</tr>
<tr>
<td>40.</td>
<td>Semi Join</td>
</tr>
<tr>
<td>41.</td>
<td>Anti Join</td>
</tr>
<tr>
<td>42.</td>
<td>SubPlan</td>
</tr>
<tr>
<td>43.</td>
<td>Others</td>
</tr>
</tbody>
</table>
My previous talk covered:

- enable_seqscan
- enable_bitmapscan
- enable_indexscan
- enable_indexonlyscan
- enable_nestloop
- enable_hashjoin
- enable_mergejoin
- enable_sort

This talk will cover:

- enable_incremental_sort
- enable_material

- enable_memoize
- enable_hashagg
- enable_gathermerge
- enable_parallel_append
- enable_parallel_hash
- enable_tidscan

Not covered:

- enable_async_append
- enable_partition_pruning
- enable_partitionwise_join
- enable_partitionwise_aggregate

https://www.postgresql.org/docs/current/runtime-config-query.html

* https://momjian.us/main/presentations/performance.html#partitioning
1. Result

```sql
-- This disables EXPLAIN cost output
\set EXPLAIN 'EXPLAIN (COSTS OFF)'

:EXPLAIN SELECT 1;
  QUERY PLAN
  -----------
  Result
```

All the queries used in this presentation are available at [https://momjian.us/main/writings/pgsql/beyond.sql](https://momjian.us/main/writings/pgsql/beyond.sql).
2. Values Scan

```
:EXPLAIN VALUES (1), (2);
QUERY PLAN
---------------------------
Values Scan on "*VALUES*"
```

Causes are in blue, optimizer choices are in red.
3. Function Scan

```
:EXPLAIN SELECT * FROM generate_series(1,4);
    QUERY PLAN

----------------------------------
 Function Scan on generate_series
```
4. Incremental Sort

CREATE TABLE large (x) AS SELECT generate_series(1, 1000000);
ANALYZE large;
CREATE INDEX i_large ON large (x);

ALTER TABLE large ADD COLUMN y INTEGER;

:EXPLAIN SELECT * FROM large ORDER BY x, y;

QUERY PLAN

-----------------------------------------
Incremental Sort
Sort Key: x, y
Presorted Key: x
-> Index Scan using i_large on large
Incremental Sort

```
3  6
3  3
3  2
3  4
3  12
3  5
4  8
4  6
4  11
4  3
```

Incremental Sort

```
3  2
3  3
3  4
3  5
3  6
3  12
4  3
4  6
4  8
4  11
```
5. Unique, First Example

```
:EXPLAIN SELECT DISTINCT * FROM generate_series(1, 10) ORDER BY 1;

QUERY PLAN

---------------------------------------------
Unique
  -> Sort
    Sort Key: generate_series
  -> Function Scan on generate_series
```
-- not UNION ALL
:EXPLAIN SELECT 1 UNION SELECT 2;

QUERY PLAN

--------------------------
| Unique                 |
| -> Sort                |
|   -> Sort Key: (1)    |
|   -> Append           |
|     -> Result         |
|     -> Result         |

11 / 68
6. Append

```sql
:EXPLAIN SELECT 1 UNION ALL SELECT 2;
QUERY PLAN
--------------
Append
  -> Result
  -> Result
```
7. Merge Append

```sql
:EXPLAIN (VALUES (1), (2) ORDER BY 1)
UNION ALL
  (VALUES (3), (4) ORDER BY 1)
ORDER BY 1;
```

QUERY PLAN

```
-----------------------------------------
Merge Append
  Sort Key: "VALUES".column1
    -> Sort
      Sort Key: "VALUES".column1
        -> Values Scan on "VALUES"
    -> Sort
      Sort Key: "VALUES_1".column1
        -> Values Scan on "VALUES_1"
```

15 / 68
Merge Append

```
6
3
2
4
12

5
8
6
11
3

2
3
4
6
12

2
3
4
5
6
7
8
11
12

Sort

Merge Append
```
8, 9. Subquery Scan, HashSetOp

CREATE TABLE small (x) AS
SELECT generate_series(1, 1000);
ANALYZE small;

:EXPLAIN SELECT * FROM small EXCEPT SELECT * FROM small;
QUERY PLAN

---------------------------------------------
HashSetOp Except
  -> Append
    -> Subquery Scan on "*SELECT* 1"
      -> Seq Scan on small
    -> Subquery Scan on "*SELECT* 2"
      -> Seq Scan on small small_1
HashSetOp

Logically UNION, EXCEPT, and INTERSECT without ALL remove duplicates in joined queries and results. For example, the query VALUES (1), (1), (2), (2) EXCEPT VALUES (1) returning 2 shows joined query removal, while UNION shows result removal.

The count is useful for efficiently processing duplicates when ALL is used.
10. SetOp

```
-- table has to be too large to hash
:EXPLAIN SELECT * FROM large INTERSECT SELECT * FROM large;

QUERY PLAN

---------------------------------------------------
SetOp Intersect
  -> Sort
     Sort Key: "*SELECT* 1".x, "*SELECT* 1".y
  -> Append
     -> Subquery Scan on "*SELECT* 1"
        -> Seq Scan on large
     -> Subquery Scan on "*SELECT* 2"
        -> Seq Scan on large large_1
```

19 / 68
SetOp

1. Append with "1" Label
   - 7
   - 3
   - 6
   - 12

2. Append with "2" Label
   - 5
   - 8
   - 6
   - 11
   - 3

3. Sort
   - 3,1
   - 6,1
   - 3,1
   - 12,1
   - 5,2
   - 8,2
   - 6,2
   - 11,2
   - 3,2

4. Count
   - 3,1: 2
   - 3,2: 1
   - 5,2: 1
   - 6,1: 1
   - 6,2: 1
   - 7,1: 1
   - 8,2: 1
   - 11,2: 1
   - 12,1: 1

5. SetOp with INTERSECT
   - 3
   - 6
11. Materialize

:EXPLAIN SELECT * FROM small s1, small s2 WHERE s1.x != s2.x;

QUERY PLAN
----------------------------------
Nested Loop
  Join Filter: (s1.x <> s2.x)
  -> Seq Scan on small s1
  -> Materialize
    -> Seq Scan on small s2
Materialize

Outer

- aag
- aay
- aar
- aai

Local Memory

- aai
- aag
- aas
- aar
- aay
- aaa
- aag

Inner

Shared Buffers

- aai
- aag
- aas
- aar
- aay
- aaa
- aag
12. Memoize, Setup

-- needs duplicates and too small for a hash join
CREATE TABLE small_with_dups (x) AS
SELECT generate_series(1, 1000)
FROM generate_series(1, 10);

-- unique and too big for a hash join
CREATE TABLE medium (x) AS
SELECT generate_series(1, 100000);

-- index required for this memoize example
CREATE INDEX i_medium ON medium (x);
ANALYZE;
Memoize

:EXPLAIN SELECT * FROM small_with_dups JOIN medium USING (x);

QUERY PLAN

---------------------------------------------
Nested Loop
  -> Seq Scan on small_with_dups
  -> Memoize
    Cache Key: small_with_dups.x
    Cache Mode: logical
    -> Index Only Scan using i_medium on medium
      Index Cond: (x = small_with_dups.x)

Only happens in nested loops; supported in Postgres 14 and later.

https://blog.jooq.org/postgresql-14s-enable.memoize-for-improved-performance-of-nested-loop-joins/
Memoize

Inner-side lookups that return no rows are also recorded in the cache.
13. Group

```
-- must be small enough not to trigger HashAggregate
-- removing WHERE and adding ORDER BY x does the same
:EXPLAIN SELECT x FROM large WHERE x < 0 GROUP BY x;

QUERY PLAN

----------------------------------------------
Group
  Group Key: x
    -> Index Only Scan using i_large on large
       Index Cond: (x < 0)
```
GROUP BY without aggregates is similar to SELECT DISTINCT, except duplicate detection can consider more columns than those selected for output.
14. Aggregate

```
:EXPLAIN SELECT COUNT(*) FROM medium;

QUERY PLAN
--------------------------
Aggregate
  -> Seq Scan on medium
```
15. GroupAggregate

```sql
:EXPLAIN SELECT x, COUNT(*) FROM medium GROUP BY x ORDER BY x;
```
GroupAggregate

Sorted

GroupAggregate

1,1
1,2
1,2
1,4
2,1
2,5
3,2
3,2
3,2
3,2
3,4

1,Agg
2,Agg
3,Agg
16. HashAggregate

```
:EXPLAIN SELECT DISTINCT x FROM medium;
QUERY PLAN
--------------------------
HashAggregate
   Group Key: x
   -> Seq Scan on medium
```
HashAggregate

HashAggregate

HashAggregate

8.4
3.2
6.2
3.1
11.5
6.7
8.9
6.2
11.4
3.1
:EXPLAIN SELECT x FROM medium GROUP BY ROLLUP(x);

QUERY PLAN

--------------

MixedAggregate
  Hash Key: x
  Group Key: ()
  -> Seq Scan on medium
MixedAggregate

MixedAggregate

Sorted

35 / 68
EXPLAIN SELECT x, SUM(x) OVER ()
FROM generate_series(1, 10) AS f(x);

QUERY PLAN
------------------------------------------
WindowAgg
  -> Function Scan on generate_series f

https://momjian.us/main/presentations/sql.html#window
Window functions allow aggregates across rows while the individual rows remain.
:EXPLAIN SELECT \texttt{SUM(x)} FROM \texttt{large};

\texttt{QUERY PLAN}

\begin{verbatim}
Finalize Aggregate
  \rightarrow Gather
    Workers Planned: 2
      \rightarrow Partial Aggregate
        \rightarrow Parallel Seq Scan on \texttt{large}
\end{verbatim}

https://www.postgresql.org/docs/current/parallel-plans.html
Parallel Seq Scan uses background workers to scan different parts of a table in parallel. Gather collects results from background workers. Finally, the results are aggregated to produce the final result.
23. Gather Merge

CREATE TABLE huge (x) AS SELECT generate_series(1, 100000000);
ANALYZE huge;

:EXPLAIN SELECT * FROM huge ORDER BY 1;

QUERY PLAN
---------------------------------------

Gather Merge
  Workers Planned: 2
  -> Sort
    Sort Key: x
    -> Parallel Seq Scan on huge
Gather Merge collects ordered results from background workers, retaining their ordering.
24. Parallel Append

```sql
EXPLAIN SELECT * FROM huge UNION ALL SELECT * FROM huge ORDER BY 1;
```

QUERY PLAN
```
Gather Merge
  Workers Planned: 2
  --> Sort
    Sort Key: huge.x
    --> Parallel Append
      --> Parallel Seq Scan on huge
      --> Parallel Seq Scan on huge huge_1
```
Parallel Append

Two Tables

<table>
<thead>
<tr>
<th>Parallel Seq Scan</th>
<th>Background Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel Seq Scan</th>
<th>Background Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>

One Result

<table>
<thead>
<tr>
<th>Background Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

| Sort | 2 |
| 3    |
| 4    |
| 6    |
| 12   |

| Gather Merge | 2 |
| 3            |
| 3            |
| 4            |
| 5            |
| 6            |
| 6            |
| 8            |
| 11           |
| 12           |
25, 26. Parallel Hash, Parallel Hash Join

:EXPLAIN SELECT * FROM huge h1 JOIN huge h2 USING (x);

QUERY PLAN

-----------------------------------
Gather
Workers Planned: 2
  -> Parallel Hash Join
    Hash Cond: (h1.x = h2.x)
      -> Parallel Seq Scan on huge h1
      -> Parallel Hash
        -> Parallel Seq Scan on huge h2
Parallel Hash, Parallel Hash Join

- **Parallel Seq Scan**
  - Background Worker: 5, 3, 8, 3, 6
  - Parallel Hash
    - Background Worker: 8, 9, 11, 8, 4

- **Shared Hash**
  - Background Worker: 5, 8, 3, 6, 9

- **Parallel Hash Join**
  - Inner: 5, 8, 3
  - Outer: 6, 3, 2, 4, 12

- **Gather**
  - Background Worker: 5, 8, 6, 11, 3
27. CTE Scan

```
:EXPLAIN WITH source AS MATERIALIZED (  
    SELECT 1  
 )  
SELECT * FROM source;  
    QUERY PLAN
-----------------------------------
CTE Scan on source  
    CTE source  
        -> Result
```

https://momjian.us/main/presentations/sql.html#cte
28, 29. WorkTable Scan, Recursive Union

```sql
:EXPLAIN WITH RECURSIVE source (counter) AS (  
SELECT 1  
UNION ALL  
SELECT counter + 1  
FROM source  
WHERE counter < 10
)
SELECT * FROM source;
```

**QUERY PLAN**

```
CTE Scan on source
CTE source
  -> Recursive Union
  --> Result
  --> WorkTable Scan on source source_1
      Filter: (counter < 10)
```
WITH RECURSIVE source AS ( 
    SELECT 1
    UNION ALL
    SELECT 1 FROM source
) 
SELECT * FROM source;
WorkTable is cleared before every iteration. Recursion stops when the recursive CTE returns no rows.
:EXPLAIN SELECT generate_series(1,4);

QUERY PLAN

-------------

ProjectSet

  -> Result
31. LockRows

```sql
:EXPLAIN SELECT * FROM small FOR UPDATE;

QUERY PLAN

-------------------------
LockRows
-> Seq Scan on small
```
32. Sample Scan

:EXPLAIN SELECT * FROM small TABLESAMPLE SYSTEM(50);

QUERY PLAN
---------------------------------
Sample Scan on small
  Sampling: system ('50'::real)
33. Table Function Scan

```sql
:EXPLAIN SELECT *
FROM XMLTABLE('/ROWS/ROW'
PASSING $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>US</COUNTRY_ID>
  </ROW>
</ROWS>
$$
COLUMNS id int PATH '@id',
_id FOR ORDINALITY);

-----------------------------------
Table Function Scan on "xmltable"
```
CREATE EXTENSION postgres_fdw;

CREATE SERVER postgres_fdw_test
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', dbname 'fdw_test');

CREATE USER MAPPING FOR PUBLIC
SERVER postgres_fdw_test
OPTIONS (password '');

CREATE FOREIGN TABLE other_world (greeting TEXT)
SERVER postgres_fdw_test
OPTIONS (table_name 'world');

:EXPLAIN SELECT * FROM other_world;

-----------------------------
QUERY PLAN

Foreign Scan on other_world
35. Tid Scan

```sql
:EXPLAIN SELECT * FROM small WHERE ctid = '(0,1)';
  QUERY PLAN

-----------------------------
 Tid Scan on small
  TID Cond: (ctid = '(0,1)::tid)```
36. Insert

```sql
:EXPLAIN INSERT INTO small VALUES (0);

QUERY PLAN

-------------

Insert on small
  -> Result
```
37. Update

:EXPLAIN UPDATE small SET x = 1 WHERE x = 0;
QUERY PLAN

-------------
Update on small
  -> Seq Scan on small
    Filter: (x = 0)
38. Delete

```sql
:EXPLAIN DELETE FROM small;

QUERY PLAN
-------------------------
  Delete on small
     -> Seq Scan on small

-- You cannot run EXPLAIN on utility commands like TRUNCATE.
:EXPLAIN TRUNCATE small;
ERROR:  syntax error at or near "TRUNCATE"
LINE 1: EXPLAIN (COSTS OFF) TRUNCATE small;
^
CREATE TABLE mergetest (x, y) AS VALUES (1, NULL), (3, NULL), (5, NULL);

:EXPLAIN MERGE INTO mergetest
USING (VALUES (1), (2), (3), (4), (5), (6)) m (x)
ON mergetest.x = m.x
WHEN NOT MATCHED THEN
    INSERT (x) VALUES (m.x)
WHEN MATCHED THEN
    UPDATE SET y = TRUE;

QUERY PLAN
-------------------------------------------------------
Merge on mergetest
  -> Hash Right Join
    Hash Cond: (mergetest.x = "VALUES".column1)
    -> Seq Scan on mergetest
    -> Hash
      -> Values Scan on "VALUES"
40. Semi Join, First Example

:EXPLAIN SELECT *
FROM small
WHERE EXISTS (SELECT * FROM medium WHERE medium.x = small.x);

QUERY PLAN

-----------------------------------
Hash Semi Join
  Hash Cond: (small.x = medium.x)
  -> Seq Scan on small
  -> Hash
    -> Seq Scan on medium

Stop scan after first inner match.
Semi Join, Second Example

```sql
:EXPLAIN SELECT *
FROM small
WHERE small.x IN (SELECT medium.x FROM medium);
```

**QUERY PLAN**

```
-----------------------------------
| Hash Semi Join                  |
| Hash Cond: (small.x = medium.x) |
| -> Seq Scan on small            |
| -> Hash                        |
|    -> Seq Scan on medium        |
```

**EXISTS and IN are equivalent in handling of NULLs because EXISTS only checks for row existence while IN logically does OR comparisons that can ignore non-true results from NULL comparisons.**
:EXPLAIN SELECT *
FROM medium
WHERE NOT EXISTS (SELECT * FROM small WHERE small.x = medium.x);

QUERY PLAN

-----------------------------------
Hash Anti Join
  Hash Cond: (medium.x = small.x)
    -> Seq Scan on medium
    -> Hash
      -> Seq Scan on small

Stop scan after first inner match; negate result.
42. SubPlan

```
:EXPLAIN SELECT *
FROM small
WHERE small.x NOT IN (SELECT medium.x FROM medium);
```

QUERY PLAN

```
------------------------------------
Seq Scan on small
  Filter: (NOT (hashed SubPlan 1))
  SubPlan 1
    -> Seq Scan on medium
```

`NOT IN` and `NOT EXISTS` are not equivalent for NULLs because `NOT IN` logically does repeated not-equal `AND` comparisons which must all be true to return true; NULL affects this.
-- UNIQUE index guarantees at most one right row match
CREATE UNIQUE INDEX i_small ON small (x);

-- LEFT JOIN guarantees every left row is returned
EXPLAIN SELECT medium.x FROM medium LEFT JOIN small USING (x);
QUERY PLAN
--------------------
Seq Scan on medium
• Named Tuplestore Scan: after triggers
• Custom Scan: custom scan providers
Conclusion

https://momjian.us/presentations

https://www.flickr.com/photos/glassholic/