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.
Explaining the Postgres Query Optimizer

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
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | Result | 12 | Memoize | 23 | Gather Merge |
| 2 | Values Scan | 13 | Group | 24 | Parallel Append |
| 3 | Function Scan | 14 | Aggregate | 25 | Parallel Hash |
| 4 | Incremental Sort | 15 | GroupAggregate | 26 | Parallel Hash Join |
| 5 | Unique | 16 | HashAggregate | 27 | CTE Scan |
| 6 | Append | 17 | MixedAggregate | 28 | WorkTable Scan |
| 7 | Merge Append | 18 | WindowAgg | 29 | Recursive Union |
| 8 | Subquery Scan | 19 | Parallel Seq Scan | 30 | ProjectSet |
| 9 | HashSetOp | 20 | Partial Aggregate | 31 | LockRows |
| 10 | SetOp | 21 | Gather | 32 | Sample Scan |
| 11 | Materialize | 22 | Finalize Aggregate | 33 | Table Function Scan |
| 12 |   |   |   |   | 34 | Foreign Scan |
| 13 |   |   |   |   | 35 | Tid Scan |
| 14 |   |   |   |   | 36 | Insert |
| 15 |   |   |   |   | 37 | Update |
| 16 |   |   |   |   | 38 | Delete |
| 17 |   |   |   |   | 39 | Merge |
| 18 |   |   |   |   | 40 | Semi Join |
| 19 |   |   |   |   | 41 | Anti Join |
| 20 |   |   |   |   | 42 | SubPlan |
| 21 |   |   |   |   | 43 | Others |

https://www.pgmustard.com/docs/explain
 Controls

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

-- 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.
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

```sql
EXPLAIN SELECT * FROM generate_series(1,4);

QUERY PLAN

----------------------------------------------------------------------------------
Function Scan on generate_series
```
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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Incremental Sort

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
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
6. Append

:EXPLAIN SELECT 1 UNION ALL SELECT 2;
QUERY PLAN
-------------
  Append
    -> Result
    -> Result
Append

Append

6
3
2
4
12

5
8
6
11
3

6
3
2
4
12
5
8
6
11
3
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"
```
Merge Append

Sort

Merge Append
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
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.
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
SetOp

Append with "1" Label

Append with "2" Label

Sort

SetOp with INTERSECT

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

3,1
3,2
5,2
6,1
6,2
7,1
8,2
11,2
12,1

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
-- 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.
Group, Single Columns
14. Aggregate

:EXPLAIN SELECT COUNT(*) FROM medium;

QUERY PLAN

--------------------------
Aggregate
  ->  Seq Scan on medium
:EXPLAIN SELECT x, \texttt{COUNT(*)} FROM medium \texttt{GROUP BY} x ORDER BY x;

QUERY PLAN

| GroupAggregate
| Group Key: x
| -> Index Only Scan using i_medium on medium |
GroupAggregate

Sorted

GroupAggregate

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

8.4
3.2
6.2
3.1
11.5
6.7
8.9
6.2
11.4
3.1

8
3
6
11
17. MixedAggregate

:EXPLAIN SELECT x FROM medium GROUP BY ROLLUP(x);

QUERY PLAN

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

```
8,4
3,2
6,2
3,1
11,5
6,7
8,9
6,2
11,4
3,1
```

Sorted

```
3,Agg
6,Agg
8,Agg
11,Agg
```
18. WindowAgg

```sql
: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 \( \text{SUM}(x) \) FROM large;

QUERY PLAN

Finalize Aggregate
  -> Gather
    Workers Planned: 2
    -> Partial Aggregate
      -> Parallel Seq Scan on large

https://www.postgresql.org/docs/current/parallel-plans.html
Parallel Seq Scan uses background workers to scan different parts of a table in parallel.

Parallel Seq Scan, Partial Aggregate, Gather, Finalize Aggregate

Parallel Seq Scan

Partial Aggregate

Background Worker

Gather

Finalize Aggregate

Gather collects results from background workers.
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

```
: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

Parallel Seq Scan

Background Worker

Parallel Append

Background Worker

Parallel Append

Background Worker

Parallel Append

Background Worker

Parallel Append

Background Worker

Parallel Seq Scan

Parallel Seq Scan

Parallel Seq Scan

Parallel Seq Scan

One Result

Background Worker

Parallel Append

Background Worker

Parallel Append

Background Worker

Parallel Append

Background Worker

Two Tables One Result
: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
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
CTE Scan

Materialized Common Table Expressions

CTE Source

6
3
2
4
12
5
8
6
11
3

CTE Scan
: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.
30. ProjectSet

:EXPLAIN SELECT generate_series(1,4);

QUERY PLAN

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

ProjectSet
  -> Result
31. LockRows

:EXPLAIN SELECT * FROM small FOR UPDATE;
QUERY PLAN
--------------------
LockRows
   -> Seq Scan on small
32. Sample Scan

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

QUERY PLAN

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

: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);

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

```
: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

```sql
:EXPLAIN UPDATE small SET x = 1 WHERE x = 0;

QUERY PLAN

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

: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

```sql
: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

```
: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.
41. Anti Join

`: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

```sql
: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/