Beyond Joins and Indexes

Bruce Momjian

EDB

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.

https://momjian.us/presentations

Last updated: May 2025

Further Study

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

This Presentation Covers Everything Else

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

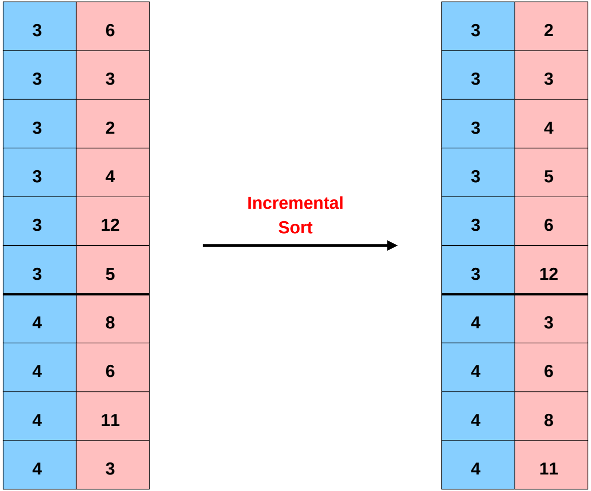
3. Function Scan

```
: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

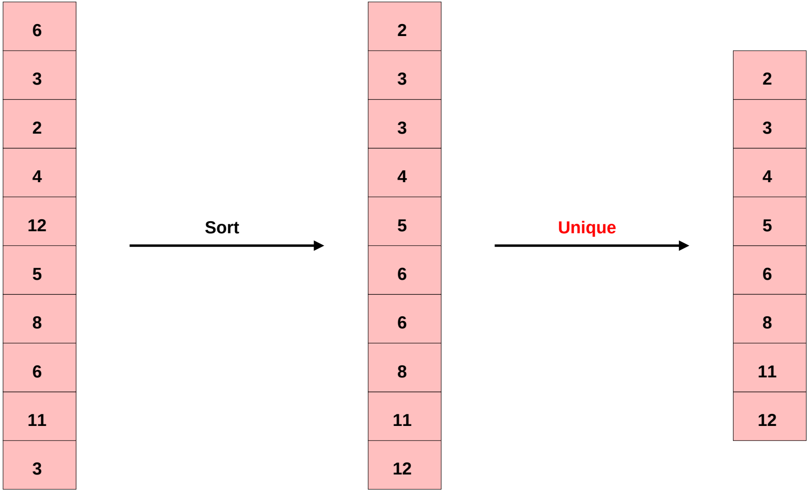
5. Unique, First Example

```
:EXPLAIN SELECT DISTINCT * FROM generate_series(1, 10) ORDER BY 1;
                  QUERY PLAN
----------------------------------------------
 Unique
   ->  Sort
         Sort Key: generate_series
         ->  Function Scan on generate_series
```

Unique, Second Example

```
-- not UNION ALL
:EXPLAIN SELECT 1 UNION SELECT 2;
        QUERY PLAN
--------------------------
 Unique
   ->  Sort
         Sort Key: (1)
         ->  Append
               ->  Result
               ->  Result
```

Unique



6
3
2
4
12
5
8
6
11
3

Sort

2
3
3
4
5
6
6
8
11
12

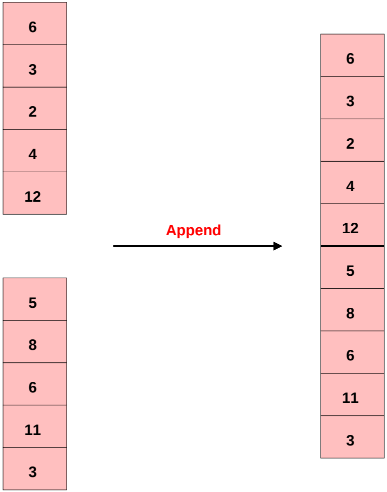
Unique

2
3
4
5
6
8
11
12

6. Append

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

Append



6
3
2
4
12

5
8
6
11
3

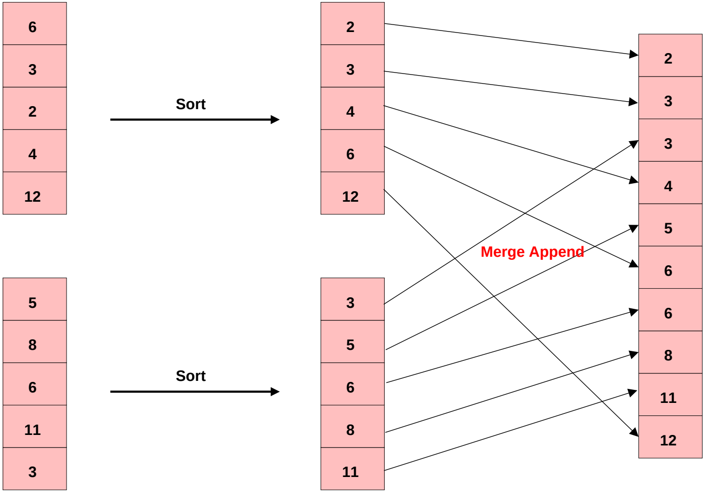
Append

6
3
2
4
12
5
8
6
11
3

7. Merge Append

```
: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

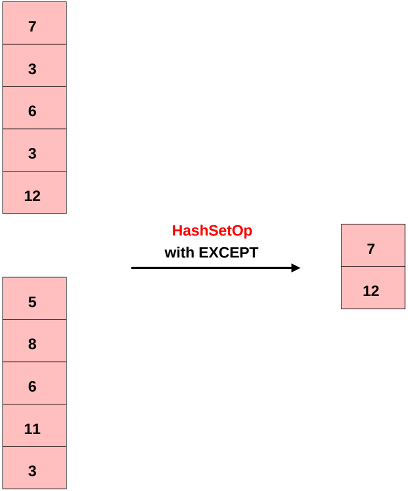


8. HashSetOp

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

:EXPLAIN SELECT * FROM small EXCEPT SELECT * FROM small;
           QUERY PLAN
---------------------------------
 HashSetOp Except
   ->  Seq Scan on small
   ->  Seq Scan on small small_1
```

Improved in Postgres 18

HashSetOp



7
3
6
3
12

5
8
6
11
3

HashSetOp
with EXCEPT

7
12

Use of ALL

While UNION clearly removes duplicates on output, EXCEPT and INSERSECT show removal of duplicates from the queries that are joined:

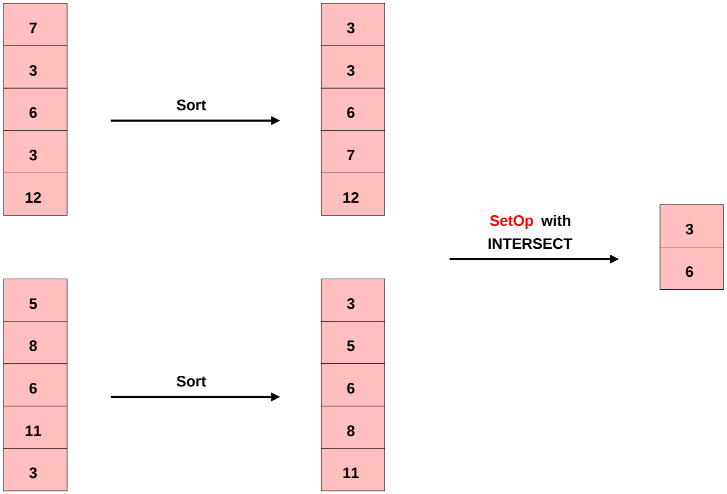
```
VALUES (1), (1), (2), (2) EXCEPT VALUES (1);
 column1
---------
       2

VALUES (1), (1), (2), (2) EXCEPT ALL VALUES (1);
 column1
---------
       1
       2
       2
```

9. SetOp

```
-- table has to be too large to hash
:EXPLAIN SELECT * FROM large INTERSECT SELECT * FROM large;
                      QUERY PLAN
--------------------------------------------------------
 SetOp Intersect
   ->  Incremental Sort
         Sort Key: large.x, large.y
         Presorted Key: large.x
         ->  Index Scan using i_large on large
   ->  Incremental Sort
         Sort Key: large_1.x, large_1.y
         Presorted Key: large_1.x
         ->  Index Scan using i_large on large large_1
```

Improved in Postgres 18

SetOp



7
3
6
3
12

Sort

3
3
6
7
12

5
8
6
11
3

Sort

3
5
6
8
11

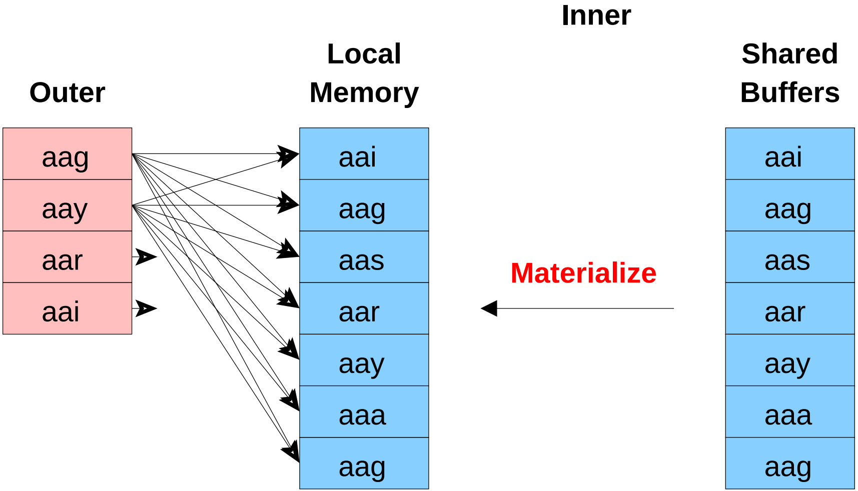
SetOp with INTERSECT

3
6

10. 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



Inner

Outer

aag

aay

aar

aai

Local Memory

aai

aag

aas

aar

aay

aaa

aag

Materialize

Shared Buffers

aai

aag

aas

aar

aay

aaa

aag

11. 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;
```

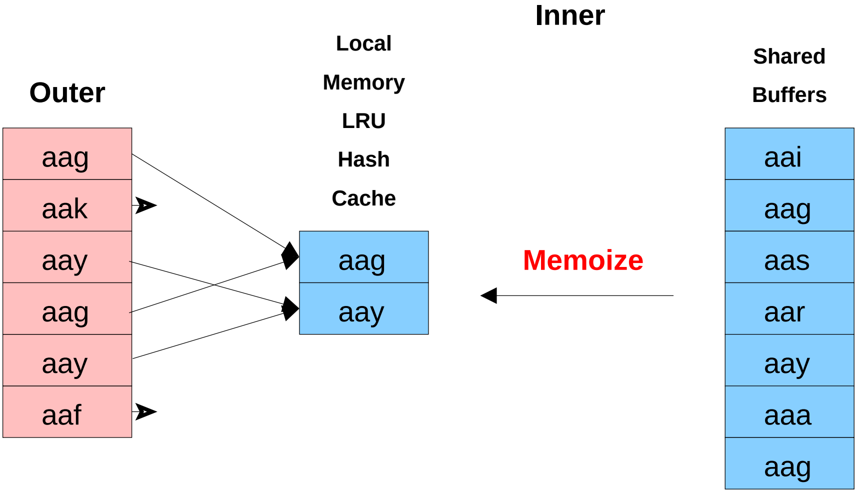
Memoize

```
: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

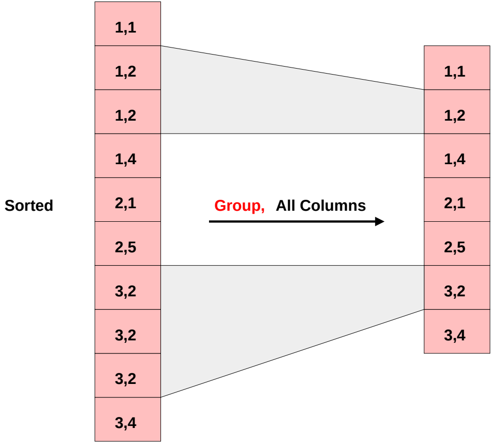


Inner-side lookups that return no rows are also recorded in the cache.

12. 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, All Columns



Sorted

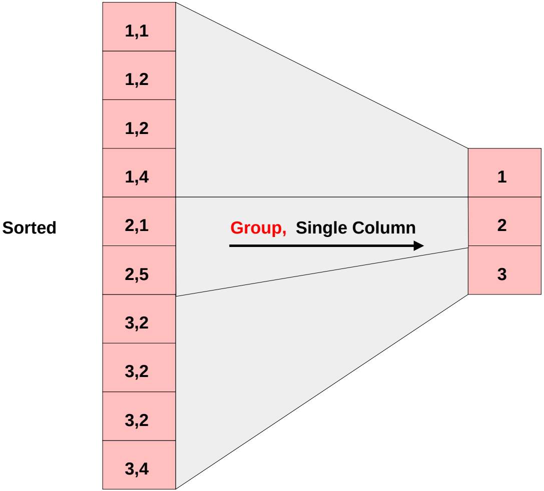
1,1
1,2
1,2
1,4
2,1
2,5
3,2
3,2
3,2
3,4

Group, All Columns

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

GROUP BY without aggregates is similar to SELECT DISTINCT, except duplicate detection can consider more columns than those selected for output.

Group, Single Columns



Sorted

1,1

1,2

1,2

1,4

2,1

2,5

3,2

3,2

3,2

3,4

Group, Single Column

1

2

3

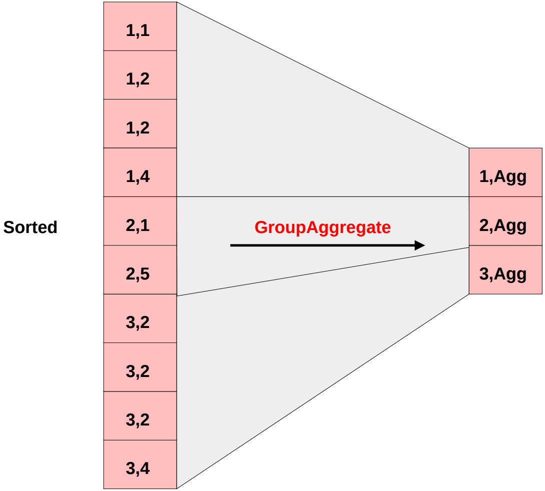
13. Aggregate

```
:EXPLAIN SELECT COUNT(*) FROM medium;
        QUERY PLAN
--------------------------
 Aggregate
   ->  Seq Scan on medium
```

14. GroupAggregate

```
:EXPLAIN SELECT x, COUNT(*) FROM medium GROUP BY x ORDER BY x;
                   QUERY PLAN
------------------------------------------------
 GroupAggregate
   Group Key: x
   ->  Index Only Scan using i_medium on medium
```

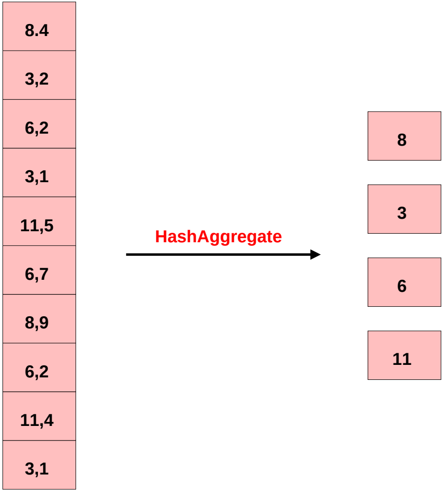
GroupAggregate



15. 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

HashAggregate

8

3

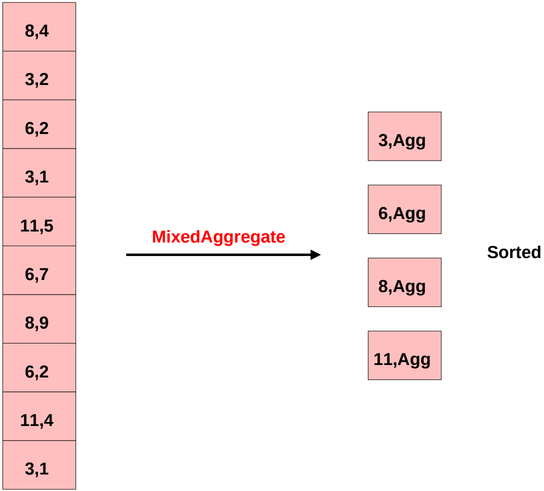
6

11

16. 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

MixedAggregate

3,Agg

6,Agg

8,Agg

11,Agg

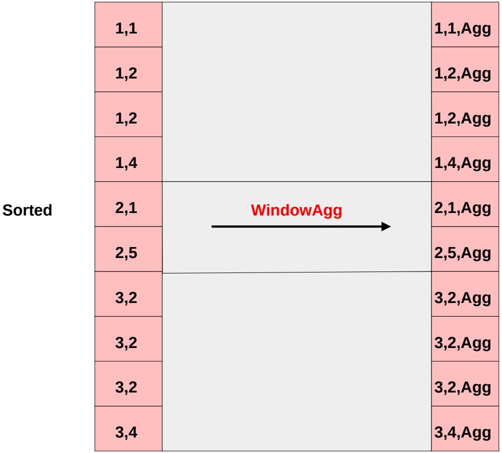
Sorted

17. WindowAgg

```
: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

WindowAgg



Sorted

WindowAgg

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

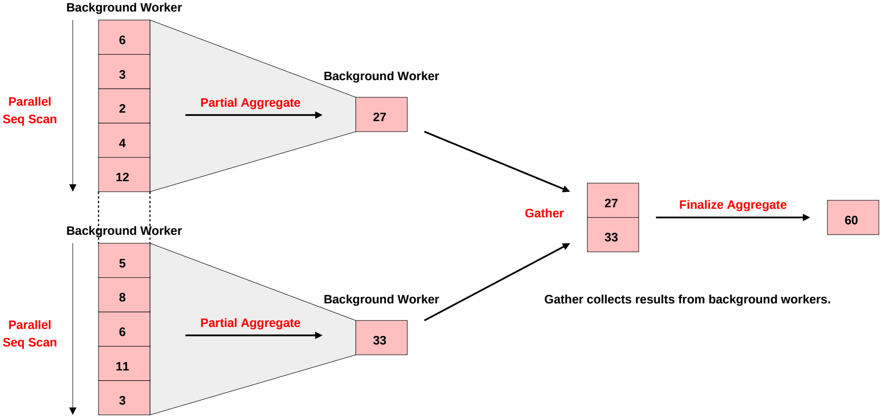
Window functions allow aggregates across rows while the individual rows remain.

18-21. Parallel Seq Scan, Partial Aggregate, Gather, Finalize Aggregate

```
:EXPLAIN SELECT 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, Partial Aggregate, Gather, Finalize Aggregate



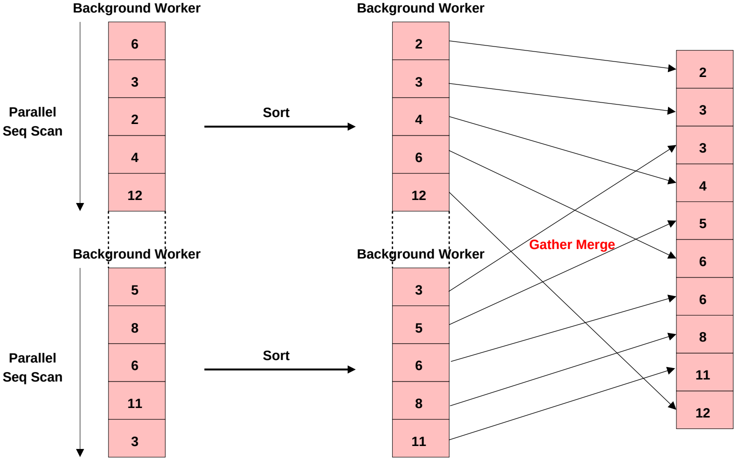
Gather collects results from background workers.

Parallel Seq Scan uses background workers to scan different parts of a table in parallel.

22. 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

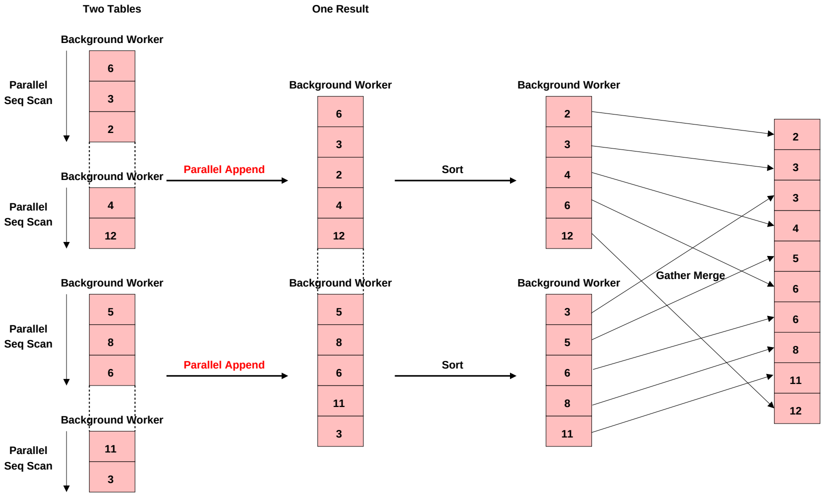


Gather Merge collects ordered results from background workers, retaining their ordering.

23. 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
```

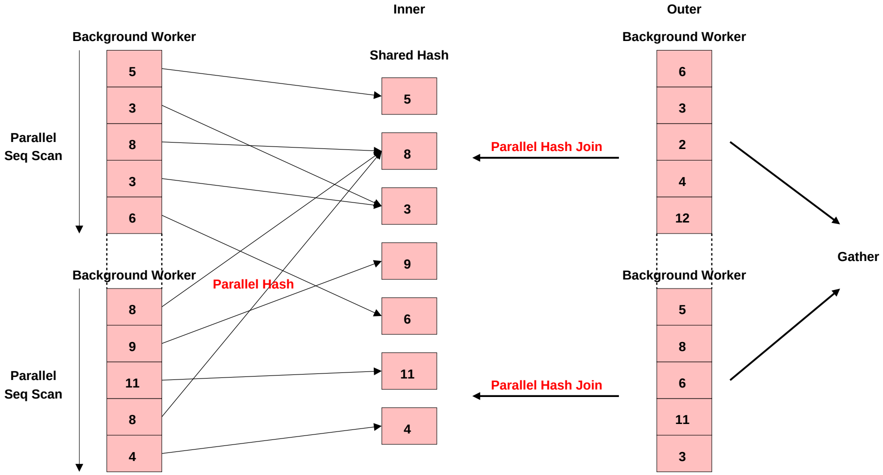
Parallel Append



24, 25. 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

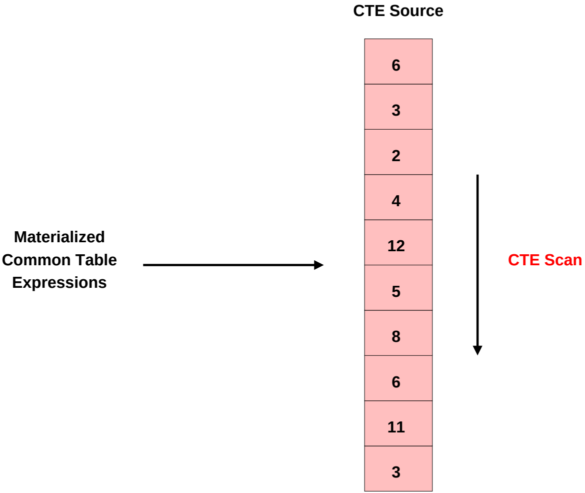


26. CTE Scan

```
: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



CTE Source
6
3
2
4
12
5
8
6
11
3

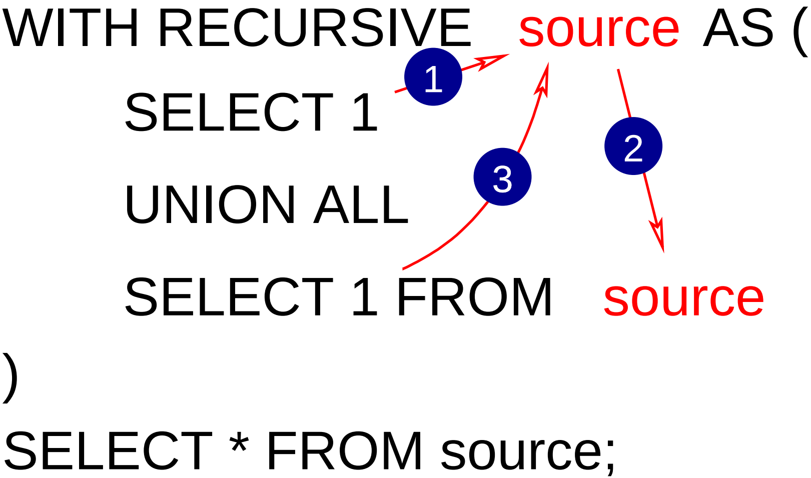
Materialized
Common Table
Expressions

CTE Scan

27, 28. WorkTable Scan, Recursive Union

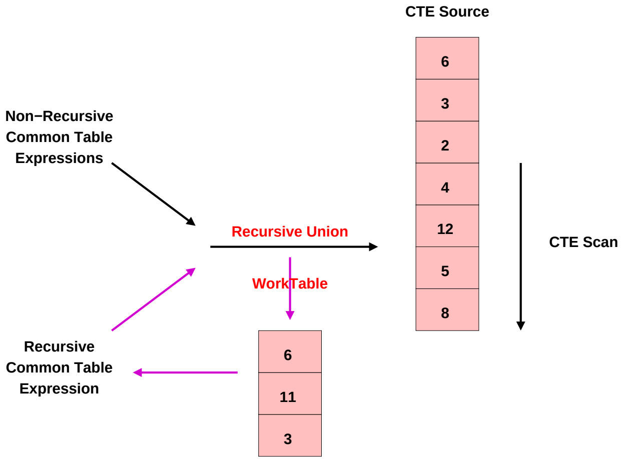
```
:EXPLAIN WITH RECURSIVE source (counter) AS (
    SELECT 1
    UNION ALL
    SELECT counter + 1
    FROM source
    WHERE counter < 10
)
SELECT * FROM source;
                   QUERY PLAN
------------------------------------------------
 CTE Scan on source
   CTE source
     ->  Recursive Union
           ->  Result
           ->  WorkTable Scan on source source_1
                 Filter: (counter < 10)
```

CTE Query Flow



```
WITH RECURSIVE source AS (
        SELECT 1
        UNION ALL
        SELECT 1 FROM source
)
SELECT * FROM source;
```

WorkTable Scan, Recursive Union



WorkTable is cleared before every iteration. Recursion stops when the recursive CTE returns no rows.

29. ProjectSet

```
:EXPLAIN SELECT generate_series(1,4);
  QUERY PLAN
--------------
 ProjectSet
   ->  Result
```

30. Subquery Scan

```
:EXPLAIN SELECT *
FROM (SELECT 1 AS t, generate_series(1,10) AS x) AS ss
WHERE x < 4;
      QUERY PLAN
----------------------
 Subquery Scan on ss
   Filter: (ss.x < 4)
   ->  ProjectSet
         ->  Result
```

31. LockRows

```
: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

```
: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"
```

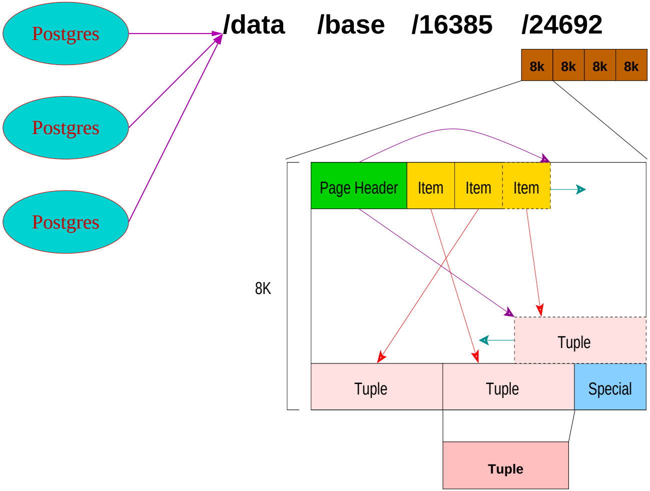
34. Foreign Scan

```
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)
```

Tid Scan


Postgres
Postgres
Postgres
/data
/base
/16385
/24692
8k
8k
8k
8k
8K
Page Header
Item
Item
Item
Tuple
Tuple
Tuple
Special
Tuple

36. Insert

```
: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

```
: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;
                            ^
```

39. Merge

```
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

```
: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

```
: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.

43. Others: Outer Join Removal

```
-- 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
```

Not Covered

- Named Tuplestore Scan: after triggers
- Custom Scan: custom scan providers

Conclusion

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

https://momjian.us/presentations