

Join Methods



Copyright

© Postgres Professional, 2019–2024

Authors Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Merge Join Algorithm

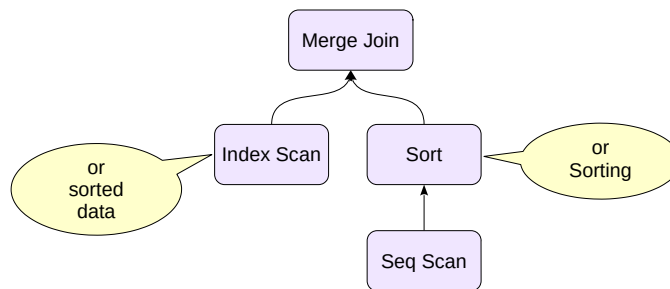
Computational complexity

Merge Join in Parallel Execution Plans

Merge join

Merging Two Sorted Row Sets

The result of the join is automatically sorted



The third and final join method is the merge join.

The idea of this method is that two pre-sorted data sets can be easily merged into a single combined set that is sorted in the same way. The Gather Merge node operates similarly.

Before performing a merge join, both sets of rows need to be sorted.

Сортировка — дорогая операция, она имеет сложность $O(N \log N)$.

But sometimes this phase can be skipped if the rows are already sorted by the required columns, for example, via indexed access to the table.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year	album_id	name
1	Yellow Submarine	1969	1	All Together Now
3	Let It Be	1970	1	All You Need Is Love
4	The Beatles	1968	2	Another Girl
6	Abbey Road	1969	2	Act Naturally
			3	Across the Universe
			5	A Day in the Life

The merge is straightforward. First, we take the first strings from both sets and compare them. In this case, we immediately found a match and can output the first tuple of the result: («Yellow Submarine», «All Together Now»).

The algorithm goes as follows. the general algorithm works by reading the next row from the set with the smaller join field value (one set "catches up" to the other). If the values are equal, as in our example, we proceed to the next row in the second (inner) set.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year
1	Yellow Submarine	1969
3	Let It Be	1970
4	The Beatles	1968
6	Abbey Road	1969


album_id	name
1	All Together Now
1	All You Need Is Love
2	Another Girl
2	Act Naturally
3	Across the Universe
5	A Day in the Life

Once again, the match: ('Yellow Submarine', 'All You Need Is Love')
Once more, read the next row from the second dataset.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year	album_id	name
1	Yellow Submarine	1969	1	All Together Now
3	Let It Be	1970	1	All You Need Is Love
4	The Beatles	1968	2	Another Girl
6	Abbey Road	1969	2	Act Naturally
			3	Across the Universe
			5	A Day in the Life




No match found in this case.

Since $1 < 2$, we proceed to the next string in the first set.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year	album_id	name
1	Yellow Submarine	1969	1	All Together Now
3	Let It Be	1970	1	All You Need Is Love
4	The Beatles	1968	2	Another Girl
6	Abbey Road	1969	2	Act Naturally
			3	Across the Universe
			5	A Day in the Life



No match found.

Since $3 > 2$, we proceed to the next row in the second set.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year	album_id	name
1	Yellow Submarine	1969	1	All Together Now
3	Let It Be	1970	1	All You Need Is Love
4	The Beatles	1968	2	Another Girl
6	Abbey Road	1969	2	Act Naturally
			3	Across the Universe
			5	A Day in the Life

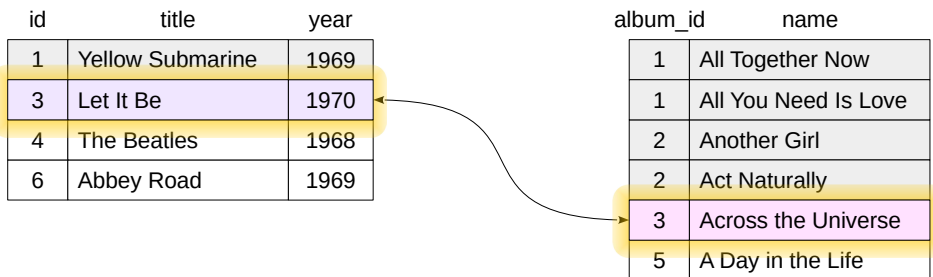
Once more, there's no match, once more $3 > 2$, we read the next row from the second set.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year
1	Yellow Submarine	1969
3	Let It Be	1970
4	The Beatles	1968
6	Abbey Road	1969

album_id	name
1	All Together Now
1	All You Need Is Love
2	Another Girl
2	Act Naturally
3	Across the Universe
5	A Day in the Life



There is a match: 'Let It Be' and 'Across the Universe'

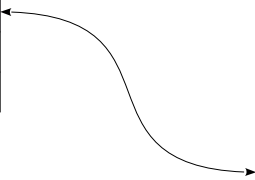
3 = 3, proceed to the next line in the second set.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year
1	Yellow Submarine	1969
3	Let It Be	1970
4	The Beatles	1968
6	Abbey Road	1969

album_id	name
1	All Together Now
1	All You Need Is Love
2	Another Girl
2	Act Naturally
3	Across the Universe
5	A Day in the Life



No match found.

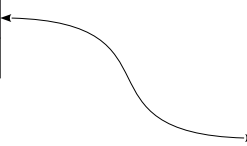
3 < 5, read the row from the first set.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year
1	Yellow Submarine	1969
3	Let It Be	1970
4	The Beatles	1968
6	Abbey Road	1969

album_id	name
1	All Together Now
1	All You Need Is Love
2	Another Girl
2	Act Naturally
3	Across the Universe
5	A Day in the Life



No match found.

$4 < 5$, so read the row from the first set.

Merge

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

id	title	year	album_id	name
1	Yellow Submarine	1969	1	All Together Now
3	Let It Be	1970	1	All You Need Is Love
4	The Beatles	1968	2	Another Girl
6	Abbey Road	1969	2	Act Naturally
			3	Across the Universe
			5	A Day in the Life

And the final step: no match again.

The merge join has concluded.

In reality, the algorithm is more complex — if the first (outer) row set has multiple identical values, it must be able to reread the rows of the second (inner) set using the same join key.

The algorithm's pseudocode can be found in the file `src/backend/executor/nodeMergejoin.c`.

Notably, the merge algorithm returns the join result in a sorted format. In particular, the resulting row set can be used for the next merge join without further sorting.

Merge Join

If the result needs to be sorted, the optimizer might choose a merge join. This is especially the case when the data from the child nodes is already sorted — as shown in this example:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
     JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
```

QUERY PLAN

```
-----
Merge Join
  Merge Cond: (t.ticket_no = tf.ticket_no)
    -> Index Scan using tickets_pkey on tickets t
    -> Index Scan using ticket_flights_pkey on ticket_flights tf
(4 rows)
```

Here's another example featuring two merge joins, where one Merge Join node receives a sorted dataset from another Merge Join node:

```
=> EXPLAIN (costs off) SELECT t.ticket_no, bp.flight_id, bp.seat_no
FROM tickets t
     JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
     JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
     AND bp.flight_id = tf.flight_id
ORDER BY t.ticket_no;
```

QUERY PLAN

```
-----
Merge Join
  Merge Cond: (tf.ticket_no = t.ticket_no)
    -> Merge Join
          Merge Cond: ((tf.ticket_no = bp.ticket_no) AND (tf.flight_id = bp.flight_id))
            -> Index Only Scan using ticket_flights_pkey on ticket_flights tf
            -> Index Scan using boarding_passes_pkey on boarding_passes bp
          -> Index Only Scan using tickets_pkey on tickets t
(7 rows)
```

Here, ticket_flights and boarding_passes are joined, and the tickets (tickets) are joined using a set of rows that is already sorted by ticket numbers.

Merge Join also supports Left, Right, Semi, Anti, and Full variations.

Currently, merge joins support only equijoins. Joins based on 'greater than' or 'less than' operations are not implemented.

Thus, the Full join is available only for hash and merge joins, and both types operate exclusively with equality conditions. If performance isn't a concern, a full join can be implemented by combining the results of a left join and an anti-join—this method would be necessary when a full join with a different condition is required.

$\sim N + M$, где

N and M represent the number of rows in the first and second data sets. no joins required

$\sim N \log N + M \log M$,

if sorting is required

Potential Initial Sorting Costs

Efficient for a large number of rows

In cases where data sorting isn't required, the overall complexity of a merge join is proportional to the total number of rows in both data sets. However, unlike hash joins, there is no overhead involved in building a hash table here.

Therefore, merge joins can be effectively used in both OLTP- and OLAP-queries.

However, if sorting is required, the cost becomes proportional to the number of rows multiplied by the logarithm of that number. On large datasets, this approach is likely to be less efficient than a hash join.

Computational Complexity

Let's examine the cost of a merge join:

```
=> EXPLAIN SELECT *  
FROM tickets t  
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no  
ORDER BY t.ticket_no;
```

QUERY PLAN

```
-----  
Merge Join (cost=0.99..822410.77 rows=8391708 width=136)  
Merge Cond: (t.ticket_no = tf.ticket_no)  
-> Index Scan using tickets_pkey on tickets t (cost=0.43..139115.14 rows=2950178  
width=104)  
-> Index Scan using ticket_flights_pkey on ticket_flights tf (cost=0.56..571023.84  
rows=8391708 width=32)  
(4 rows)
```

Startup cost includes:

- the sum of the startup costs of the child nodes (including the sorting cost if required);
- the cost of retrieving the first pair of matching rows

The total cost includes the initial cost plus

- the total cost of acquiring both data sets
- the cost of comparing rows

The overall conclusion is that the cost of a merge join is proportional to $N + M$ (where N and M represent the number of rows being joined) when no separate sorting is required. Sorting a set of K rows adds at least $K \times \log(K)$ to the estimate.

In contrast to hash joins, merge joins without sorting are well suited for cases where quickly retrieving the initial rows is needed.

```
=> EXPLAIN SELECT *  
FROM tickets t  
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no  
ORDER BY t.ticket_no  
LIMIT 1000;
```

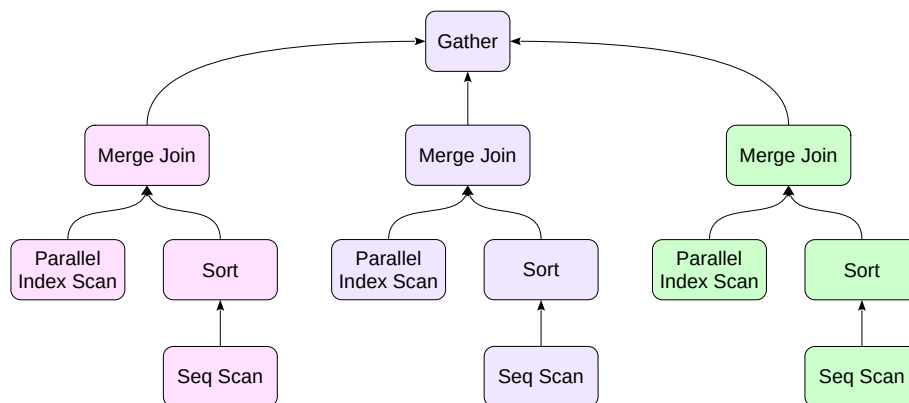
QUERY PLAN

```
-----  
Limit (cost=0.99..98.99 rows=1000 width=136)  
-> Merge Join (cost=0.99..822410.77 rows=8391708 width=136)  
Merge Cond: (t.ticket_no = tf.ticket_no)  
-> Index Scan using tickets_pkey on tickets t (cost=0.43..139115.14  
rows=2950178 width=104)  
-> Index Scan using ticket_flights_pkey on ticket_flights tf  
(cost=0.56..571023.84 rows=8391708 width=32)  
(5 rows)
```

Also, note how the total cost has decreased.

In parallel execution plans

The external dataset is scanned in parallel, while the internal one is processed sequentially by each process.



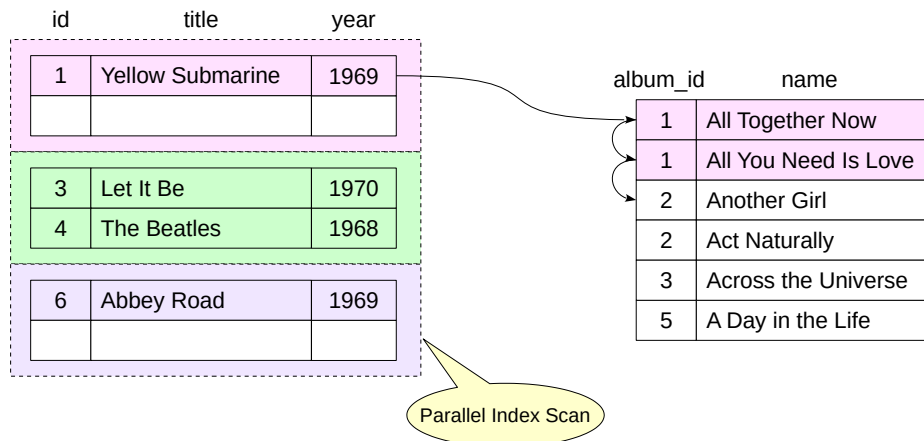
16

The merge join algorithm can be used in a parallelized plan.

Just like with a nested loop join, scanning one set of rows is executed in parallel by worker processes, but the other set of rows is read entirely by each worker process on its own. Therefore, hash joins are far more commonly used in parallel execution plans when dealing with large row sets, due to their efficient parallel algorithm.

In parallel execution plans

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```

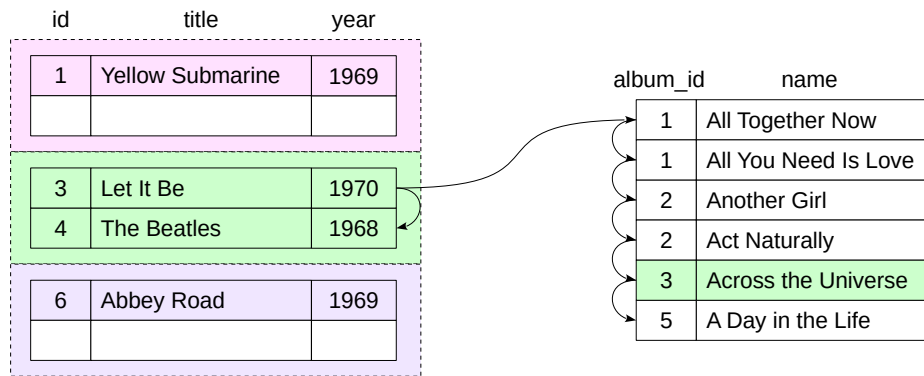


Each worker process will scan the internal dataset from the start until no more matches are found.

This slide shows the rows processed by the first process.

In parallel execution plans

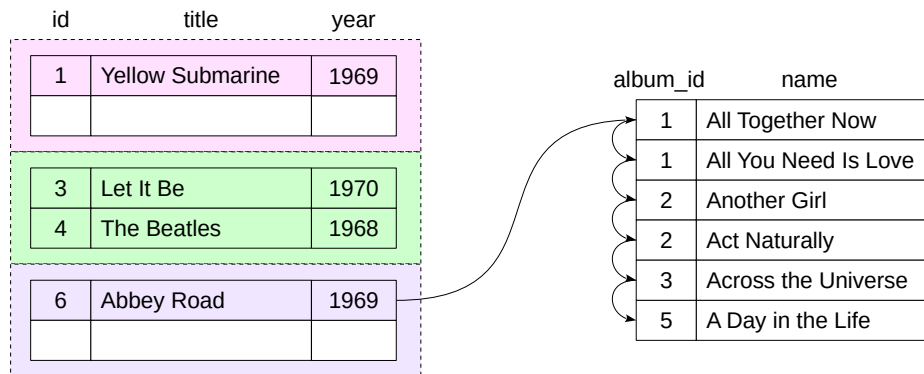
```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```



The second process will scan through the full set and find a match.

In parallel execution plans

```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```



However, the third process will find no matches.

Of course, all three processes examine the internal set at the same time, rather than sequentially.

In parallel execution plans

We'll need an index:

```
=> CREATE INDEX ON tickets(book_ref);
```

CREATE INDEX

Here's an example of a parallel execution plan utilizing a merge join:

```
=> EXPLAIN (costs off)
SELECT count(*)
FROM bookings b
  JOIN (
    SELECT book_ref FROM tickets GROUP BY book_ref
  ) t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
-----
Finalize Aggregate
  -> Gather
      Workers Planned: 2
      -> Partial Aggregate
          -> Merge Join
              Merge Cond: (b.book_ref = tickets.book_ref)
              -> Parallel Index Only Scan using bookings_pkey on bookings b
              -> Group
                  Group Key: tickets.book_ref
                  -> Index Only Scan using tickets_book_ref_idx on tickets

(10 rows)
```

Here, the bookings_pkey index on the bookings table is scanned in parallel, while the aggregation result from subquery t is processed entirely by each process.

Merge join may require some preparation

- The row sets need to be sorted
or have them pre-sorted

Efficient for large samples

- It's beneficial if the row sets are already sorted.
- It's beneficial if a sorted result is required

This method is independent of the join order

Only equijoins are supported.

- Other join types are not implemented, but there are no fundamental restrictions

To perform a merge join, both row sets must be sorted. It's beneficial if the data is already in the correct order; otherwise, sorting is required.

Merge operations are highly efficient, even for large data sets. As a nice bonus, the output is also sorted, making this join method advantageous when higher-level plan nodes need sorting (e.g., a query with an ORDER BY clause or another merge sort).

Thus, the planner has three join methods: nested loop, hashing, and merge (excluding various modifications). Each method has scenarios where it outperforms the others. This allows the planner to select the method that is expected to be the most suitable for each specific scenario.

1. Check the query's execution plan for the list of all seats in the cabins, ordered by aircraft code:

```
SELECT * FROM aircrafts a JOIN seats s ON  
a.aircraft_code = s.aircraft_code ORDER BY  
a.aircraft_code;
```

But present it as a cursor.

Reduce the `cursor_tuple_fraction` parameter value by a factor of ten. How did the execution plan change?

2. An aircraft can be replaced with another if the capacity difference is no more than 20%. Generate a replacement table between Boeing and Airbus models using a full join. How does the query execute?

2. 2. The query to execute:

```
WITH cap AS ( SELECT a.model, COUNT(*)::NUMERIC AS capacity FROM  
aircrafts a JOIN seats s ON a.aircraft_code = s.aircraft_code  
GROUP BY a.model), a AS ( SELECT * FROM cap WHERE model LIKE  
'Airbus%'), b AS ( SELECT * FROM cap WHERE model LIKE 'Boeing  
%') SELECT a.model AS airbus, b.model AS boeing FROM a FULL JOIN b  
ON b.capacity::NUMERIC / a.capacity BETWEEN 0.8 AND 1.2 ORDER BY 1,  
2;
```

1. The cursor_tuple_fraction configuration parameter

Cursor Execution Plan:

```
=> EXPLAIN DECLARE c CURSOR FOR SELECT *
FROM aircrafts a
JOIN seats s ON a.aircraft_code = s.aircraft_code
ORDER BY a.aircraft_code;
```

QUERY PLAN

```
-----
Merge Join (cost=1.51..420.71 rows=1339 width=55)
  Merge Cond: (s.aircraft_code = ml.aircraft_code)
    -> Index Scan using seats_pkey on seats s (cost=0.28..64.60 rows=1339 width=15)
    -> Sort (cost=1.23..1.26 rows=9 width=72)
        Sort Key: ml.aircraft_code
        -> Seq Scan on aircrafts_data ml (cost=0.00..1.09 rows=9 width=72)
(6 rows)
```

Current cursor_tuple_fraction value:

```
=> SHOW cursor_tuple_fraction;
```

```
cursor_tuple_fraction
-----
0.1
(1 row)
```

Lower it:

```
=> SET cursor_tuple_fraction = 0.01;
```

SET

```
=> EXPLAIN DECLARE c CURSOR FOR SELECT *
FROM aircrafts a
JOIN seats s ON a.aircraft_code = s.aircraft_code
ORDER BY a.aircraft_code;
```

QUERY PLAN

```
-----
Merge Join (cost=0.41..431.73 rows=1339 width=55)
  Merge Cond: (ml.aircraft_code = s.aircraft_code)
    -> Index Scan using aircrafts_pkey on aircrafts_data ml (cost=0.14..12.27 rows=9 width=72)
    -> Index Scan using seats_pkey on seats s (cost=0.28..64.60 rows=1339 width=15)
(4 rows)
```

Now the planner chooses a different plan because its initial cost is lower, even though the total cost is higher.

2. Full Join

Let's try executing the query.

```
=> WITH cap AS (
  SELECT a.model, count(*)::numeric capacity
  FROM aircrafts a
  JOIN seats s ON a.aircraft_code = s.aircraft_code
  GROUP BY a.model
), a AS (
  SELECT * FROM cap WHERE model LIKE 'Airbus%'
), b AS (
  SELECT * FROM cap WHERE model LIKE 'Boeing%'
)
SELECT a.model AS airbus, b.model AS boeing
FROM a FULL JOIN b
ON b.capacity::numeric/a.capacity BETWEEN 0.8 AND 1.2
ORDER BY 1,2;
```

ERROR: FULL JOIN is only supported with merge-joinable or hash-joinable join conditions

We encounter an error: full joins are only implemented for equality conditions because only nested loops support joins with

arbitrary conditions, but they don't support full joins.

The restriction can be bypassed by combining the results of a left outer join and an anti-join:

```
=> WITH cap AS (  
    SELECT a.model, count(*)::numeric capacity  
    FROM aircrafts a  
    JOIN seats s ON a.aircraft_code = s.aircraft_code  
    GROUP BY a.model  
) , a AS (  
    SELECT * FROM cap WHERE model LIKE 'Airbus%'  
) , b AS (  
    SELECT * FROM cap WHERE model LIKE 'Boeing%'  
)  
SELECT a.model AS airbus, b.model AS boeing  
FROM a LEFT JOIN b  
    ON b.capacity::numeric/a.capacity BETWEEN 0.8 AND 1.2  
UNION ALL  
SELECT NULL, b.model  
FROM b  
WHERE NOT EXISTS (  
    SELECT 1  
    FROM a  
    WHERE b.capacity::numeric/a.capacity BETWEEN 0.8 AND 1.2  
)  
ORDER BY 1,2;  
  
airbus | boeing  
-----+-----  
(0 rows)
```

The execution plan indicates that joins are implemented using a nested loop:

```
=> EXPLAIN (costs off)  
WITH cap AS (  
    SELECT a.model, count(*)::numeric capacity  
    FROM aircrafts a  
    JOIN seats s ON a.aircraft_code = s.aircraft_code  
    GROUP BY a.model  
) , a AS (  
    SELECT * FROM cap WHERE model LIKE 'Airbus%'  
) , b AS (  
    SELECT * FROM cap WHERE model LIKE 'Boeing%'  
)  
SELECT a.model AS airbus, b.model AS boeing  
FROM a LEFT JOIN b  
    ON b.capacity::numeric/a.capacity BETWEEN 0.8 AND 1.2  
UNION ALL  
SELECT NULL, b.model  
FROM b  
WHERE NOT EXISTS (  
    SELECT 1  
    FROM a  
    WHERE b.capacity::numeric/a.capacity BETWEEN 0.8 AND 1.2  
)  
ORDER BY 1,2;
```


QUERY PLAN

```

-----
Sort
  Sort Key: a.model, b.model
  CTE cap
    -> HashAggregate
      Group Key: (ml.model -> lang())
      -> Hash Join
        Hash Cond: (s.aircraft_code = ml.aircraft_code)
        -> Seq Scan on seats s
        -> Hash
          -> Seq Scan on aircrafts_data ml

  CTE a
    -> CTE Scan on cap
      Filter: (model ~~ 'Airbus% '::text)

  CTE b
    -> CTE Scan on cap cap_1
      Filter: (model ~~ 'Boeing% '::text)

  -> Append
    -> Nested Loop Left Join
      Join Filter: (((b.capacity / a.capacity) >= 0.8) AND ((b.capacity /
a.capacity) <= 1.2))
      -> CTE Scan on a
      -> CTE Scan on b
    -> Nested Loop Anti Join
      Join Filter: (((b_1.capacity / a_1.capacity) >= 0.8) AND ((b_1.capacity /
a_1.capacity) <= 1.2))
      -> CTE Scan on b b_1
      -> CTE Scan on a a_1

(25 rows)

```