

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.com

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.

General Considerations on Joins

Nested loop join

Variations: left, semi-, and anti-joins

Computational complexity

Parallel Execution Plans with Nested Loops

Join types are not SQL joins

inner, left, right, full, and cross joins, along with 'in' and 'exists' — logical operations

Join types are the implementation mechanism

It's not tables that are joined, but sets of rows.

may originate from any node in the execution plan tree

Row sets are joined in pairs

The order of joins is crucial for performance.

The order within the pair is typically important.

Simply retrieving data using the discussed access methods isn't enough; you also need to know how to join them. PostgreSQL offers several methods for this.

Join methods are algorithms designed to combine two row sets. These methods also implement other SQL constructs, such as EXISTS. Avoid confusing the two: SQL joins are logical operations on two sets, whereas PostgreSQL's join methods are the actual implementations that consider performance.

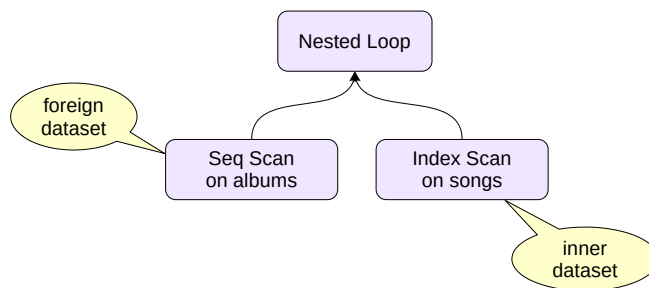
It's common to hear that tables are joined. This is a convenient simplification, but in reality, row sets are what get joined. These sets can be directly retrieved from the table (via one of the access methods), but they can also, for instance, result from joining other row sets.

Finally, row sets are always joined pairwise. The order in which tables are joined typically doesn't affect the query result (such as a join with b or b join with c), but it can greatly impact performance. As we'll see later, the order in which two row sets are joined matters (e.g., a join b versus b join a).

Nested Loop

For each row in one set, we look for matching rows in the other set

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



Let's begin with the nested loop join, the simplest approach. The algorithm works by iterating through each row in one set and returning the corresponding rows from the second set. In essence, this is two nested loops, which is why it's called the nested loop method.

Note that the second (inner) set is accessed as many times as there are rows in the first (outer) set. If there's no efficient way to find the corresponding rows in the second set (i.e., an index on the table), you'll have to scan many non-matching rows repeatedly. It's clear that this isn't the optimal choice, even for small datasets, where the algorithm can still be quite effective.

In the query plan, you'll see a Nested Loop node with two child nodes (which can represent not just access methods, but also other operations like joins or aggregations).

Nested Loop

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

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

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

The figures illustrate this connection method. In the figures:

- Rows that had already been accessed are shown in gray.
- Rows currently being accessed are highlighted in color;
- Rows forming a pair that match the join condition are highlighted with an orange border (in this case, by equality of numeric identifiers).

First, we read the first row of the first set and find its corresponding row in the second set. A match was found, and the first result row is ready to be returned to the parent plan node: ('Let It Be', 'Across the Universe').

Nested Loop

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

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

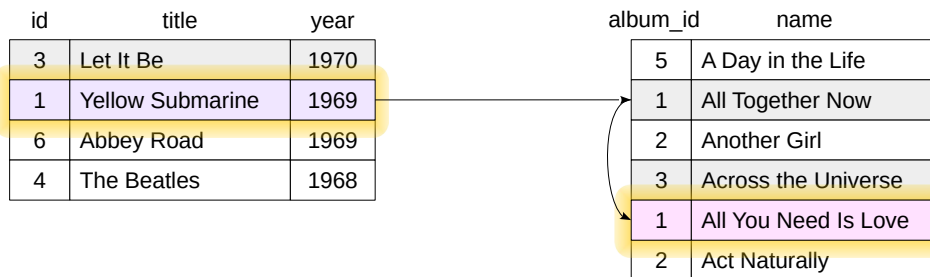
Read the second string from the first set.

We also go through the pairs from the second set for her. First, return ("Yellow Submarine", "All Together Now")...

Nested Loop

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

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



...then the second set ("Yellow Submarine", "All You Need Is Love")

Nested Loop

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

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

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

Proceed to the third row of the first set. No matches found for her.

Nested Loop

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

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

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

Not all of them
internal set
were read

The fourth row also has no matches. The connection ends here.

Some rows from the second set were not considered at all — as shown in the figure, they remained white.

(The algorithm's source code is available in the file
src/backend/executor/nodeNestloop.c.)

Nested Loop Join

This is the execution plan for a nested loop join, which the optimizer typically prefers for small samples (looking at flights included in two tickets):

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no IN ('0005432312163', '0005432312164');
```

QUERY PLAN

```
-----
Nested Loop
-> Index Scan using tickets_pkey on tickets t
    Index Cond: (ticket_no = ANY ('{0005432312163,0005432312164}'::bpchar[]))
-> Index Scan using ticket_flights_pkey on ticket_flights tf
    Index Cond: (ticket_no = t.ticket_no)
(5 rows)
```

The planner employs a parametrized join. For each row in the outer set (Index Scan node on the tickets table), rows from the inner set (Index Scan node on the flights table) that satisfy the join condition are retrieved. During each iteration of the outer loop, the index access condition for the inner set is `ticket_no = constant`.

The process repeats until the outer set runs out of rows.

The EXPLAIN ANALYZE command provides details on how many times the nested loop ran (loops), the average number of rows selected (rows), and the time spent per iteration (time). It's clear that the planner made a slight error — 8 rows were ultimately retrieved instead of 6:

```
=> EXPLAIN (analyze, summary off)
SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no IN ('0005432312163', '0005432312164');
```

QUERY PLAN

```
-----
Nested Loop (cost=0.99..46.10 rows=6 width=136) (actual time=31.725..36.647 rows=8
loops=1)
-> Index Scan using tickets_pkey on tickets t (cost=0.43..12.90 rows=2 width=104)
(actual time=30.119..30.161 rows=2 loops=1)
    Index Cond: (ticket_no = ANY ('{0005432312163,0005432312164}'::bpchar[]))
-> Index Scan using ticket_flights_pkey on ticket_flights tf (cost=0.56..16.57
rows=3 width=32) (actual time=1.257..3.189 rows=4 loops=2)
    Index Cond: (ticket_no = t.ticket_no)
(5 rows)
```

Here's another example of a nested loop join. In this case, we retrieve aircraft that can handle flights of a given distance:

```
=> EXPLAIN (costs off) SELECT *
FROM ( VALUES (1000), (10000) ) d(range)
JOIN aircrafts a ON a.range >= d.range;
```

QUERY PLAN

```
-----
Nested Loop
Join Filter: (ml.range >= "VALUES".column1)
-> Seq Scan on aircrafts_data ml
-> Materialize
    -> Values Scan on "VALUES*"
(5 rows)
```

This is a non-parametrized join: the entire inner set is scanned for each row in the outer set (the Materialize node will be discussed later; in this case, it simply refers to values from VALUES), and the Join Filter condition is evaluated for each resulting row pair. In essence, this represents a filtered Cartesian product of two row sets.

Note that a nested loop join can join rows using any condition, not just value equality.

Variants

There are several algorithm variants. For the left join:

```
=> EXPLAIN (costs off) SELECT *
FROM aircrafts a
LEFT JOIN seats s ON (a.aircraft_code = s.aircraft_code)
WHERE a.model LIKE 'Airbus%';
```

QUERY PLAN

```
-----
Nested Loop Left Join
-> Seq Scan on aircrafts_data ml
    Filter: ((model ->> lang()) ~~ 'Airbus% '::text)
-> Bitmap Heap Scan on seats s
    Recheck Cond: (ml.aircraft_code = aircraft_code)
    -> Bitmap Index Scan on seats_pkey
        Index Cond: (aircraft_code = ml.aircraft_code)
(7 rows)
```

This variant returns rows even if the left (a) set has no corresponding matches in the right (s) set.

An antijoin returns the rows from one set that have no corresponding matches in the other set. This variant can be used to handle the NOT EXISTS predicate:

```
=> EXPLAIN (costs off) SELECT *
FROM aircrafts a
WHERE a.model LIKE 'Airbus%'
AND NOT EXISTS (
    SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
```

QUERY PLAN

```
-----
Nested Loop Anti Join
-> Seq Scan on aircrafts_data ml
    Filter: ((model ->> lang()) ~~ 'Airbus% '::text)
-> Index Only Scan using seats_pkey on seats s
    Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

The same antijoin operation is used for a similar query expressed differently:

```
=> EXPLAIN (costs off) SELECT *
FROM aircrafts a
LEFT JOIN seats s ON (a.aircraft_code = s.aircraft_code)
WHERE a.model LIKE 'Airbus%'
AND s.aircraft_code IS NULL;
```

QUERY PLAN

```
-----
Nested Loop Anti Join
-> Seq Scan on aircrafts_data ml
    Filter: ((model ->> lang()) ~~ 'Airbus% '::text)
-> Index Scan using seats_pkey on seats s
    Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

A semijoin can be used for the EXISTS predicate, returning the rows from one set that have at least one matching row in the other set:

```
=> EXPLAIN SELECT *
FROM aircrafts a
WHERE a.model LIKE 'Airbus%'
AND EXISTS (
    SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
```

QUERY PLAN

```
-----
Nested Loop Semi Join (cost=0.28..4.02 rows=1 width=40)
-> Seq Scan on aircrafts_data ml (cost=0.00..3.39 rows=1 width=72)
    Filter: ((model ->> lang()) ~~ 'Airbus% '::text)
-> Index Only Scan using seats_pkey on seats s (cost=0.28..6.88 rows=149 width=4)
    Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

Keep in mind that even though the query plan for table s indicates 149 rows, actually, retrieving a single row suffices to evaluate

the EXISTS predicate.

PostgreSQL does just that (actual rows=1):

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM aircrafts a
WHERE a.model LIKE 'Airbus%'
AND EXISTS (
  SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
```

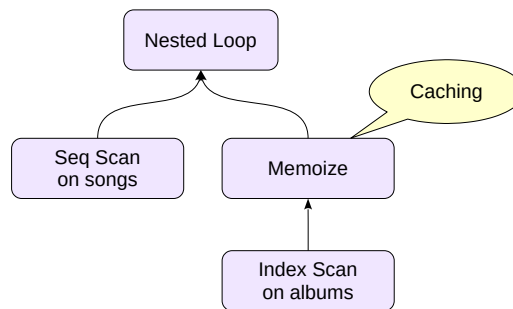
QUERY PLAN

```
Nested Loop Semi Join (actual rows=0 loops=1)
  -> Seq Scan on aircrafts_data ml (actual rows=0 loops=1)
      Filter: ((model ->> lang()) ~~ 'Airbus% '::text)
      Rows Removed by Filter: 9
  -> Index Only Scan using seats_pkey on seats s (never executed)
      Index Cond: (aircraft_code = ml.aircraft_code)
      Heap Fetches: 0
(7 rows)
```

The nested loop algorithm does not have modifications for right (RIGHT) and full (FULL) joins. This is because a full scan of the second row set may not be executed.

Caching repeated data in the inner relation

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



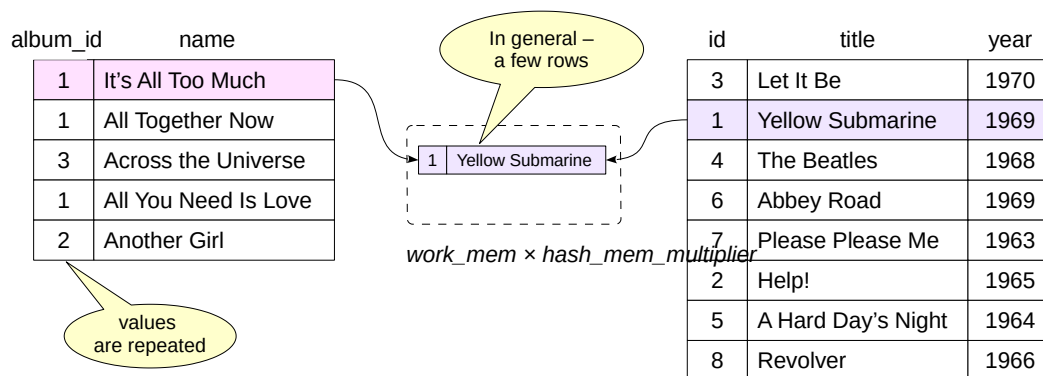
If the internal set is scanned multiple times with repeated parameter values, caching the result can help avoid repeatedly reading the same data. This operation is known as memoization . It is performed in the Memoize node, which is placed between the Nested Loop and the data-providing node.

(If the planner's calculations are incorrect, you can disable memoization by setting the `enable_memoize` parameter to off.)

Caching is implemented using a hash table. The hash key is a parameter or parameters used to access the internal set.

Memoization

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



12

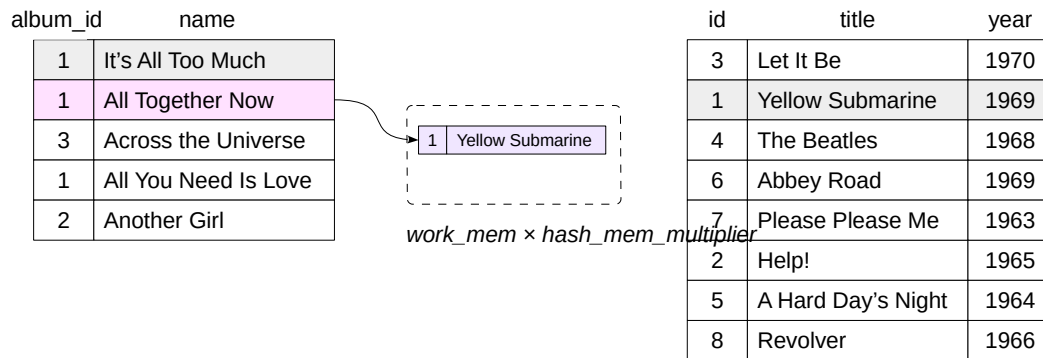
Consider the example in the demo. In contrast to the previous case, there are fewer songs than albums here; additionally, nearly all of them are from the same album.

If the required row isn't in the hash table, the Memoize node fetches it from the inner dataset, caches it, and passes it to the parent node Nested Loop.

In general, a parameter value may match multiple rows from the inner dataset — all of these rows are cached. If all of them don't fit into memory (limited to $work_mem \times hash_mem_multiplier$), the parameter value is ignored because caching only part of the rows is pointless. In the query execution plan, the number of such situations is reported as overflow.

Memoization

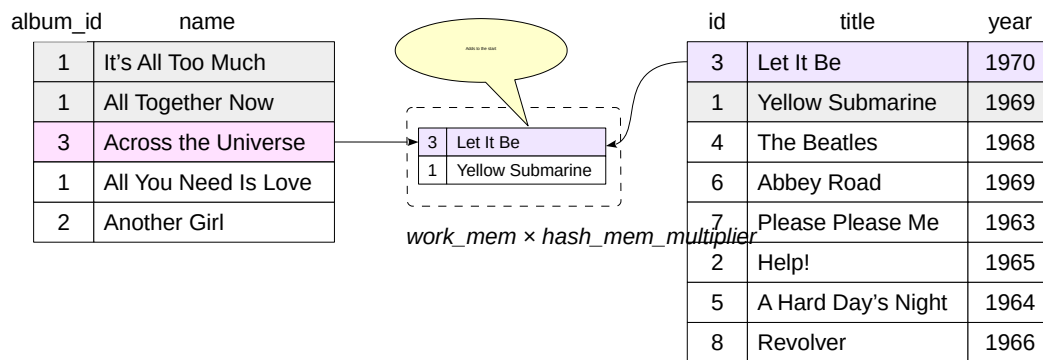
```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```



If the required row is already in the cache, the Memoize node immediately returns it to the Nested Loop node. The inner relation is not accessed.

Memoization

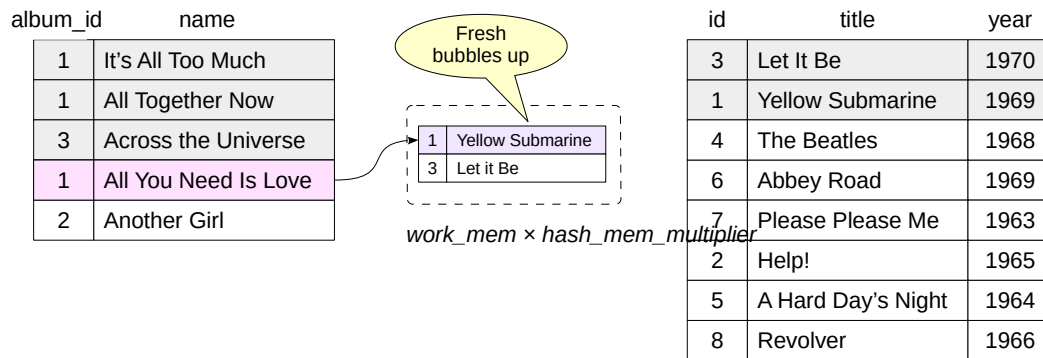
```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```



As long as there's space in the cache, new values keep getting cached. Meanwhile, the new value is added to the front of the cache.

Memoization

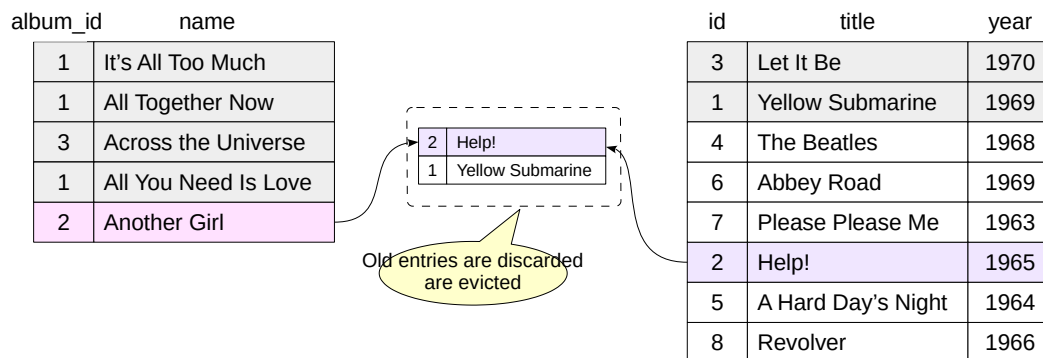
```
SELECT a.title, s.name FROM albums a JOIN songs s ON  
a.id = s.album_id;
```



The already cached value rises to the top when accessed, while the others sink down accordingly.

Memoization

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



When memory runs out, the least recently used entries are evicted from the cache. Thus, the LRU replacement algorithm is implemented.

Memoization

Let's examine the query plan that joins the flights and aircraft models tables.

The example is designed so that only one row is retrieved from the inner table (aircrafts_data) based on the key f.aircraft_code (Cache Key), and this row will be cached in the Memoize node:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM flights f
  JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
 WHERE f.flight_no = 'PG0003';
```

QUERY PLAN

```
-----
Nested Loop (actual rows=113 loops=1)
  -> Bitmap Heap Scan on flights f (actual rows=113 loops=1)
      Recheck Cond: (flight_no = 'PG0003'::bpchar)
      Heap Blocks: exact=2
      -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key (actual
rows=113 loops=1)
          Index Cond: (flight_no = 'PG0003'::bpchar)
  -> Memoize (actual rows=1 loops=113)
      Cache Key: f.aircraft_code
      Cache Mode: logical
      Hits: 112 Misses: 1 Evictions: 0 Overflows: 0 Memory Usage: 1kB
      -> Index Scan using aircrafts_pkey on aircrafts_data a (actual rows=1 loops=1)
          Index Cond: (aircraft_code = f.aircraft_code)
(12 rows)
```

Note that

- For the first time, a lookup is required to retrieve the desired row (Misses: 1);
- All repeated accesses are served by the cache (Hits: 112); one kilobyte of memory was sufficient for this.
- No cache evictions (Evictions: 0);
- The rows selected from the inner set consistently fit within the allocated memory (Overflows: 0).

$\sim N \times M$, где

If N represents the number of rows in the external dataset and M represents the average number of rows in the internal dataset per iteration, the overall join complexity is proportional to the product of N and M .

A join is effective only when dealing with a small number of rows.

If N represents the number of rows in the external dataset and M represents the average number of rows in the internal dataset per iteration, the overall join complexity is proportional to the product of N and M .

In a non-parameterized join, M is exactly equal to the number of rows in the internal dataset; in a parameterized join, M can be much smaller.

The nested loop join is only effective when dealing with a small number of rows. In particular, this method (combined with index access) is typical for OLTP queries that require quickly returning a small number of rows.

Nested Loop Join Cost

Let's examine the cost of the following execution plan:

```
=> EXPLAIN SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no IN ('0005432312163','0005432312164');
```

QUERY PLAN

```
-----
Nested Loop (cost=0.99..46.10 rows=6 width=136)
-> Index Scan using tickets_pkey on tickets t (cost=0.43..12.90 rows=2 width=104)
    Index Cond: (ticket_no = ANY ('{0005432312163,0005432312164}'::bpchar[]))
-> Index Scan using ticket_flights_pkey on ticket_flights tf (cost=0.56..16.57
rows=3 width=32)
    Index Cond: (ticket_no = t.ticket_no)
(5 rows)
```

The first result is delivered immediately without any prior steps, which means the initial cost of the Nested Loop node equals the sum of the initial costs of its child Index Scan nodes.

The total cost of the Nested Loop node comprises:

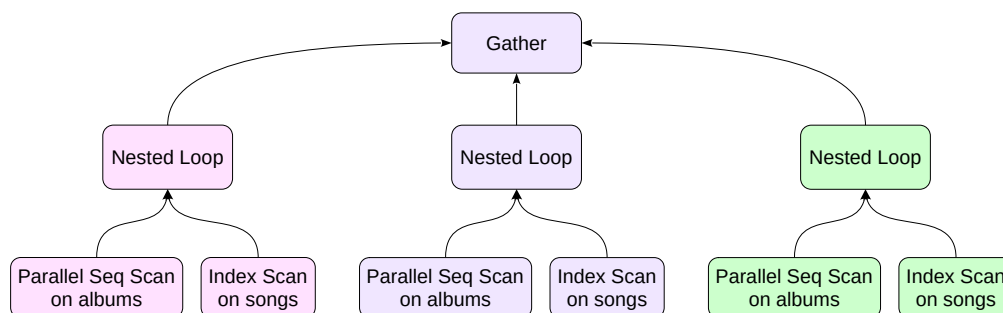
- the cost of accessing data from the outer relation (total cost of the Index Scan on tickets);
- data retrieval cost from the inner set (full cost of the Index Scan on flights), multiplied by the estimated number of rows in the outer set (2);
- CPU processing cost for rows

In general, the formula is more complex, but the main conclusion is that the cost is proportional to $N \times M$, where N is the number of rows in the outer dataset and M is the average number of rows from the inner set processed per iteration. In the worst case, this leads to quadratic cost.

In parallel execution plans

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

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



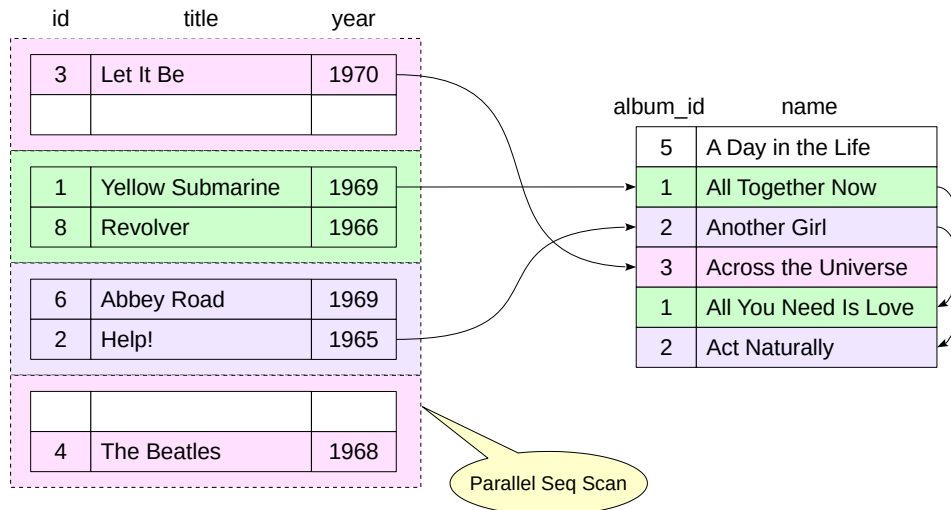
20

Nested loop join merge joins can be used in parallel execution plans.

The external row set is scanned in parallel by multiple worker processes. After obtaining a row from the external set, the process then sequentially iterates through the corresponding rows in the internal set.

In parallel execution plans

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



To retrieve the next page from the outer set, processes synchronize. Each process accesses the inner data set independently.

Nested Loop in Parallel Execution Plans

Find all passengers who purchased tickets for a specific flight:

```
=> EXPLAIN (costs off)
SELECT t.passenger_name
FROM tickets t
      JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
      JOIN flights f ON f.flight_id = tf.flight_id
WHERE f.flight_id = 12345;
```

QUERY PLAN

```
-----
Nested Loop
->  Index Only Scan using flights_pkey on flights f
    Index Cond: (flight_id = 12345)
->  Gather
    Workers Planned: 2
    -> Nested Loop
        -> Parallel Seq Scan on ticket_flights tf
            Filter: (flight_id = 12345)
        -> Index Scan using tickets_pkey on tickets t
            Index Cond: (ticket_no = tf.ticket_no)

(10 rows)
```

At the highest level, a nested loop join is employed. The outer data set is comprised of a single row retrieved from the flights table using a unique index.

A parallel execution plan is used to obtain the inner dataset. Each process processes its portion of the flights table (ticket_flights) and joins it with the tickets (tickets) table using another nested loop.

A nested loop requires no preparatory actions

Can deliver the join result without delay

Effective for small samples

The outer row set isn't very large.

The inner table can be accessed efficiently (typically through an index).

It depends on the join order

It's usually better if the outer row set is smaller than the inner one.

Supports joins on any condition

Supports equijoins as well as any other

The main advantage of the nested loop join is its simplicity: it requires no preparation, allowing results to be returned almost immediately.

The downside is that this approach is highly inefficient with large datasets. The same applies to indexes: the larger the data volume, the higher the overhead.

Therefore, a nested loop join is worth using when:

- one of the row sets is small
- Efficient access to the other dataset is available via the join condition.
- The result set contains a small number of rows.

This is a typical case for OLTP queries (e.g., user interface queries where a web page or form must load quickly without handling large data volumes).

Another important point to note is that nested loop joins can handle any join condition. It works for equijoins (such as the example given) as well as any other join condition.

1. Create an index on the `departure_airport` column of the `flights` table. Find all flights departing from Ulyanovsk and examine the query's execution plan.
2. Create a distance table between all airports (with each pair appearing only once).

2. Use the `<@>` operator from the `earthdistance` extension.

1. Flights from Ulyanovsk

Index on the flights table:

```
=> CREATE INDEX ON flights(departure_airport);
```

CREATE INDEX

Execution Plan:

```
=> EXPLAIN SELECT *
    FROM flights f JOIN airports a ON a.airport_code = f.departure_airport
    WHERE a.city = 'Ulyanovsk';
```

QUERY PLAN

```
-----
Nested Loop (cost=24.31..3732.03 rows=2066 width=162)
  -> Seq Scan on airports_data ml (cost=0.00..30.56 rows=1 width=145)
      Filter: ((city ->> lang()) = 'Ulyanovsk'::text)
  -> Bitmap Heap Scan on flights f (cost=24.31..2637.48 rows=2066 width=63)
      Recheck Cond: (departure_airport = ml.airport_code)
      -> Bitmap Index Scan on flights_departure_airport_idx (cost=0.00..23.79
rows=2066 width=0)
          Index Cond: (departure_airport = ml.airport_code)
(7 rows)
```

The planner utilized a nested loop join.

Note that a join is necessary here because Ulyanovsk has two airports:

```
=> SELECT airport_code, airport_name FROM airports WHERE city = 'Ulyanovsk';
```

```
airport_code | airport_name
-----+-----
(0 rows)
```

2. Distance Table Between Airports

```
=> CREATE EXTENSION earthdistance CASCADE;
```

NOTICE: installing required extension "cube"

CREATE EXTENSION

To eliminate duplicate pairs, you can join the tables using the "greater than" condition:

```
=> EXPLAIN SELECT a1.airport_code "from",
    a2.airport_code "to",
    a1.coordinates <@> a2.coordinates "distance, miles"
    FROM airports a1 JOIN airports a2 ON a1.airport_code > a2.airport_code;
```

QUERY PLAN

```
-----
Nested Loop (cost=0.14..147.97 rows=3605 width=16)
  -> Seq Scan on airports_data ml (cost=0.00..4.04 rows=104 width=20)
  -> Index Scan using airports_data_pkey on airports_data ml_1 (cost=0.14..0.95
rows=35 width=20)
      Index Cond: (airport_code < ml.airport_code)
(4 rows)
```

A nested loop is the only method of joining in this case.