

# Query Optimization: Materialization



## 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](mailto: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.

Query Materialization

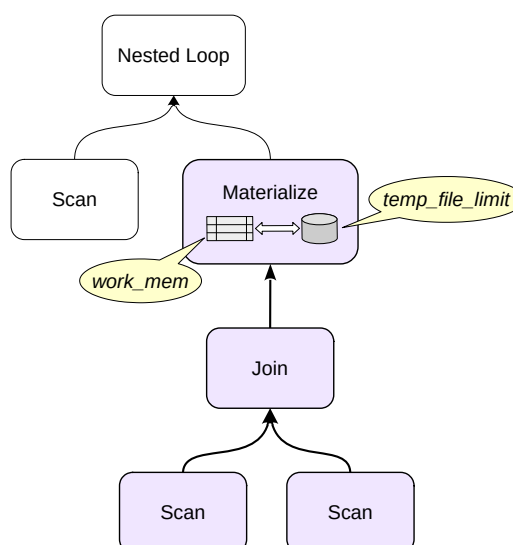
Temporary tables

Managing Connection Order

Materialized Views

# Materialize Node

Материализация —  
сохранение промежуточного  
набора строк  
для последующего  
многократного  
использования



3

Materialization refers to storing an intermediate set of rows for later reuse, often multiple times. The set can be stored at various levels: for a particular query, at the session level, or at the database level.

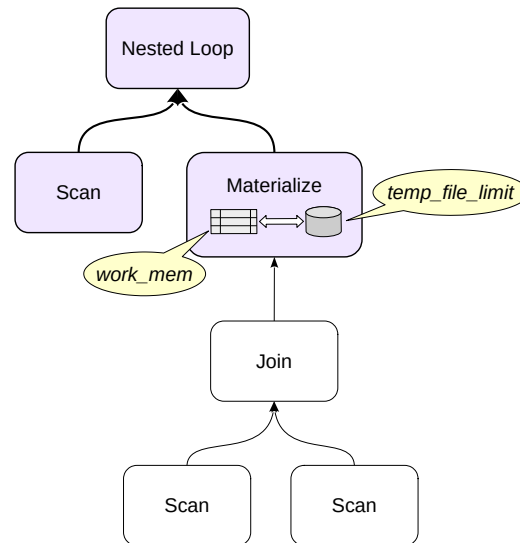
Query nodes typically exchange data using a pipeline model: when the algorithm in a plan node requires the next row from the dataset, the node requests the next batch from one of its child nodes. However, in some cases, it's beneficial (and sometimes necessary) for the query executor to immediately retrieve all rows, store them, and have the ability to access the stored result again. This materialization is handled by the Materialize node.

Rows are stored in main memory as long as their size does not exceed the `work_mem` limit. When the limit is exceeded, all rows are written to a temporary file and read from it as needed. The total size of all temporary files per session is constrained by the `temp_file_limit` parameter value.

For example, the top Nested Loop node in the example executes a nested loop join, but there's no efficient way to access the inner data set—it's computed using another join. To avoid repeatedly executing the nested join, its result can be materialized.

# Materialize Node

Материализация —  
сохранение промежуточного  
набора строк  
для последующего  
многократного  
использования



Once the nested join's result is materialized, the Nested Loop node accesses the already prepared row set.

## Materialize Node

If an operation requires significant resources and the result is accessed multiple times, the planner may opt for a plan with a Materialize node, which stores the retrieved rows for reuse:

```
=> EXPLAIN (costs off)
SELECT a1.city, a2.city
FROM airports a1, airports a2
WHERE a1.timezone = 'Europe/Moscow'
      AND abs(a2.coordinates[1]) > 66.652; -- за полярным кругом
```

### QUERY PLAN

```
-----
Nested Loop
  -> Seq Scan on airports_data ml
      Filter: (timezone = 'Europe/Moscow'::text)
  -> Materialize
      -> Seq Scan on airports_data ml_1
          Filter: (abs(coordinates[1]) > '66.652'::double precision)
(6 rows)
```

Since there's no suitable index for the inner row set's predicate, the materialization plan becomes advantageous.

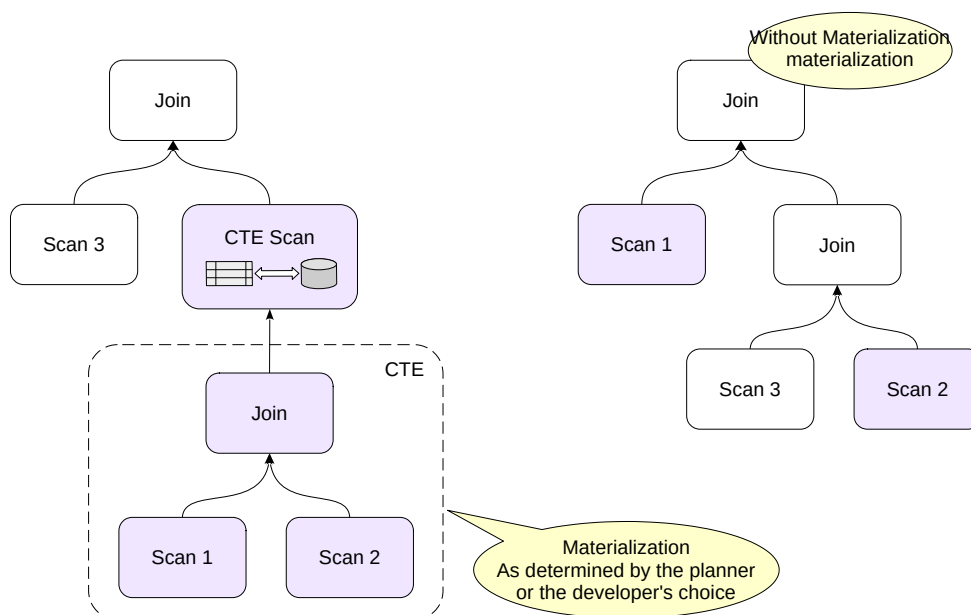
In some cases, the planner employs materialization of the inner data set during a merge join to enable re-reading parts of the data when the outer set contains repeated values:

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

### QUERY PLAN

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

# CTE Materialization



6

Common Table Expressions (CTE), also known as WITH subqueries, provide an effective way to structure a query and enhance its clarity. Unlike regular subqueries, CTEs avoid deep nesting.

The planner expands CTE subqueries (without materializing them) when possible. This allows it to select the optimal join order. By default, the subquery is materialized in the following cases:

- The main query accesses the subquery multiple times to avoid redundant calculations.

In such cases, materialization can be disabled by specifying the AS NOT MATERIALIZED clause.

- The subquery has side effects (modifies data) —so that the change occurs exactly once. (Side effects also include calls to volatile functions; see the «Functions» section.) (К побочным эффектам относится также обращение к изменчивым функциям, см. тему «Функции».)

Materialization cannot be disabled in this case.

You can always force materialization by specifying the AS MATERIALIZED clause.

<https://postgrespro.ru/docs/postgresql/16/queries-with#QUERIES-WITH-CTE-MATERIALIZATION>

## CTE Materialization

The optimizer avoids materializing subqueries in WITH unless necessary:

```
=> EXPLAIN (costs off)
WITH q AS (
  SELECT f.flight_id, a.aircraft_code
  FROM flights f
  JOIN aircrafts a ON a.aircraft_code = f.aircraft_code
)
SELECT *
FROM q
JOIN seats s ON s.aircraft_code = q.aircraft_code
WHERE s.seat_no = '1A';
```

### QUERY PLAN

```
-----
Hash Join
  Hash Cond: (f.aircraft_code = ml.aircraft_code)
    -> Hash Join
      Hash Cond: (f.aircraft_code = s.aircraft_code)
      -> Seq Scan on flights f
      -> Hash
        -> Seq Scan on seats s
        Filter: ((seat_no)::text = '1A'::text)
    -> Hash
      -> Seq Scan on aircrafts_data ml
(10 rows)
```

However, an explicit directive causes the optimizer to plan the subquery separately from the main query:

```
=> EXPLAIN (costs off)
WITH q AS MATERIALIZED (
  SELECT f.flight_id, a.aircraft_code
  FROM flights f
  JOIN aircrafts a ON a.aircraft_code = f.aircraft_code
)
SELECT *
FROM q
JOIN seats s ON s.aircraft_code = q.aircraft_code
WHERE s.seat_no = '1A';
```

### QUERY PLAN

```
-----
Hash Join
  Hash Cond: (q.aircraft_code = s.aircraft_code)
  CTE q
    -> Hash Join
      Hash Cond: (f.aircraft_code = ml.aircraft_code)
      -> Seq Scan on flights f
      -> Hash
        -> Seq Scan on aircrafts_data ml
    -> CTE Scan on q
  -> Hash
    -> Seq Scan on seats s
    Filter: ((seat_no)::text = '1A'::text)
(12 rows)
```

When a subquery is used multiple times in a query, the planner may choose to materialize it to avoid repeatedly executing the same operations:

```
=> EXPLAIN (analyze, costs off, buffers)
WITH b AS (
  SELECT * FROM bookings
)
SELECT *
FROM b AS b1
JOIN b AS b2 ON b1.book_ref = b2.book_ref
WHERE b2.book_ref = '000112';
```

# QUERY PLAN

```
-----
Nested Loop (actual time=0.054..1535.026 rows=1 loops=1)
  Buffers: shared read=13447, temp read=8483 written=8483
  CTE b
    -> Seq Scan on bookings (actual time=0.031..321.335 rows=2111110 loops=1)
        Buffers: shared read=13447
    -> CTE Scan on b b1 (actual time=0.046..440.102 rows=1 loops=1)
        Filter: (book_ref = '000112'::bpchar)
        Rows Removed by Filter: 2111109
        Buffers: shared read=1, temp read=8483 written=1
    -> CTE Scan on b b2 (actual time=0.004..1094.917 rows=1 loops=1)
        Filter: (book_ref = '000112'::bpchar)
        Rows Removed by Filter: 2111109
        Buffers: shared read=13446, temp written=8482
Planning:
  Buffers: shared hit=29 read=2 dirtied=1
Planning Time: 0.726 ms
Execution Time: 1553.675 ms
(17 rows)
```

This is typically justified, especially when the subquery involves costly operations. However, in some cases (such as this example), a materialization-free plan might be more efficient, so it can be avoided. Compare the buffers value to the previous version:

```
=> EXPLAIN (analyze, costs off, buffers)
WITH b AS NOT MATERIALIZED (
  SELECT * FROM bookings
)
SELECT *
FROM b AS b1
JOIN b AS b2 ON b1.book_ref = b2.book_ref
WHERE b2.book_ref = '000112';
```

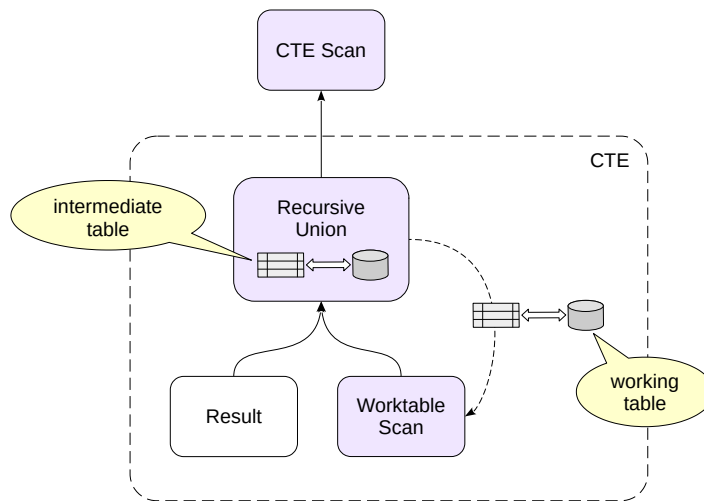
# QUERY PLAN

```
-----
Nested Loop (actual time=0.583..0.585 rows=1 loops=1)
  Buffers: shared hit=4 read=4
  -> Index Scan using bookings_pkey on bookings (actual time=0.564..0.564 rows=1
loops=1)
      Index Cond: (book_ref = '000112'::bpchar)
      Buffers: shared read=4
  -> Index Scan using bookings_pkey on bookings bookings_1 (actual time=0.014..0.014
rows=1 loops=1)
      Index Cond: (book_ref = '000112'::bpchar)
      Buffers: shared hit=4
Planning:
  Buffers: shared hit=2
Planning Time: 0.125 ms
Execution Time: 0.601 ms
(12 rows)
```

However, if the subquery modifies data, materialization will still be performed: the modification must occur only once.



# Recursive queries



Recursive queries are based on common table expressions.

The Recursive Union node relies on a working table and an intermediate table: the working table stores the rows generated during the current iteration, while the intermediate table builds up the result. The Worktable Scan node reads the content of the working table during the recursive part of the query.

Each of these two tables is materialized following the same rules: while the table's content fits into `work_mem`, it is stored in memory, and then all rows are written to disk.

Once the recursive query completes, the intermediate table's accumulated data is passed to the CTE Scan node, while the working table is discarded.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

## Recursive Queries

Let's confirm that the recursive query materializes intermediate data. To achieve this, we'll intentionally structure it so that the working and intermediate tables don't fit in memory:

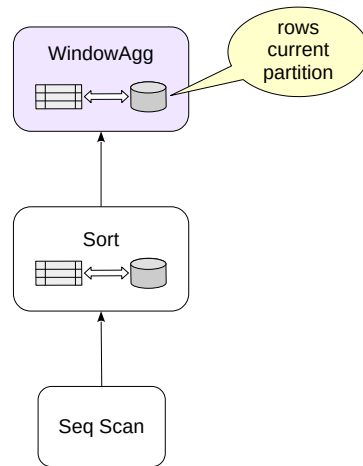
```
=> EXPLAIN (analyze, buffers, costs off, timing off)
WITH RECURSIVE r(n, airport_code) AS (
  SELECT 1, a.airport_code
  FROM airports a
  UNION ALL
  SELECT r.n+1, f.arrival_airport
  FROM r
  JOIN flights f ON f.departure_airport = r.airport_code
  WHERE r.n < 2
)
SELECT * FROM r;
```

### QUERY PLAN

```
-----
CTE Scan on r (actual rows=214971 loops=1)
  Buffers: shared read=2627, temp read=473 written=945
  CTE r
    -> Recursive Union (actual rows=214971 loops=1)
      Buffers: shared read=2627, temp read=473 written=473
      -> Seq Scan on airports_data ml (actual rows=104 loops=1)
        Buffers: shared read=3
      -> Hash Join (actual rows=107434 loops=2)
        Hash Cond: (f.departure_airport = r_1.airport_code)
        Buffers: shared read=2624, temp read=473 written=1
        -> Seq Scan on flights f (actual rows=214867 loops=1)
          Buffers: shared read=2624
        -> Hash (actual rows=52 loops=2)
          Buckets: 1024 Batches: 1 Memory Usage: 13kB
          Buffers: temp read=473 written=1
          -> WorkTable Scan on r r_1 (actual rows=52 loops=2)
            Filter: (n < 2)
            Rows Removed by Filter: 107434
            Buffers: temp read=473 written=1

Planning:
  Buffers: shared hit=9
Planning Time: 0.176 ms
Execution Time: 331.740 ms
(23 rows)
```

Pay attention to the temp read/write entries in the WorkTable Scan and Recursive Union nodes, which are specific to recursive queries.



When evaluating window functions in the WindowAgg node (see the "Sorting" section), materialization is also used: rows from the current partition (PARTITION BY) may be included multiple times in the window, and thus get materialized.

In this case, only the WindowAgg node uses the materialized rows; the parent node receives the computed result, not these intermediate data.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

# Temporary tables

## A table accessible during a single session

- is stored in the system catalog
- is not written to the log
- Files are written to disk
- Session-local memory caching (temp\_buffers)

## Vacuum and analysis

- only performed manually

Intermediate data can be obtained using a complex algorithm, such as one written in a procedural language. In such a scenario, the data cannot be processed in a CTE but can be stored in a temporary table and reused across multiple queries.

Temporary tables are more suitable for intermediate data than regular tables because they exist only within a session or transaction (depending on the mode specified during creation) and are automatically removed along with the data and dependent objects like views and indexes. Moreover, temporary tables are not logged and are cached in the local memory of the backend process handling the session, making them more efficient to use.

The process's local memory is not accessible to autovacuum processes, so vacuuming and analyzing must be done manually. The cache memory is allocated on demand and restricted by the temp\_buffer parameter for the session (once the temporary table is accessed for the first time, the limit cannot be modified).

<https://postgrespro.com/docs/postgresql/16/sql-copy>

However, temporary tables generate entries in the system catalog and are stored as disk files. As a result, bulk processing of temporary tables—such as in the 1C system—can lead to the system catalog growing and increased strain on the file system. Therefore, 1C uses specific patches to mitigate these unwanted side effects.

## Temporary Tables Temporary Tables

We'll create a temporary table:

```
=> CREATE TEMP TABLE airports_msk
ON COMMIT PRESERVE ROWS -- по умолчанию
AS SELECT *
FROM airports
WHERE timezone = 'Europe/Moscow';

SELECT 44
```

The temporary table is created in the pg\_temp schema.

By default, the table persists until the end of the session; you can choose to have the rows deleted (ON COMMIT DELETE ROWS) or the table dropped (ON COMMIT DROP) when the transaction completes.

It's often useful to analyze a recently filled temporary table before incorporating it into queries. Compare the cardinality estimates when querying a regular table versus a temporary one:

```
=> EXPLAIN
SELECT *
FROM airports
WHERE timezone = 'Europe/Moscow';

QUERY PLAN
-----
Seq Scan on airports_data ml (cost=0.00..26.52 rows=44 width=99)
  Filter: (timezone = 'Europe/Moscow'::text)
(2 rows)
```

```
=> EXPLAIN
SELECT *
FROM airports_msk;

QUERY PLAN
-----
Seq Scan on airports_msk (cost=0.00..15.20 rows=520 width=128)
(1 row)
```

In the latter case, without statistics, the optimizer estimates the table occupies 10 pages containing 520 rows, resulting in an overestimated cost:

```
=> SELECT relpages, reltuples,
10 * current_setting('seq_page_cost')::float +
520 * current_setting('cpu_tuple_cost')::float AS cost
FROM pg_class
WHERE relname = 'airports_msk'
AND relpersistence = 't'; -- временная

relpages | reltuples | cost
-----+-----+-----
0 | -1 | 15.2
(1 row)
```

If the table is analyzed, the estimate becomes accurate:

```
=> ANALYZE airports_msk;

ANALYZE

=> EXPLAIN
SELECT *
FROM airports_msk;

QUERY PLAN
-----
Seq Scan on airports_msk (cost=0.00..1.44 rows=44 width=67)
(1 row)
```

The number of join combinations increases exponentially as the number of tables in the query increases.

## JOIN Operation

Full enumeration when the number of joins does not exceed

`join_collapse_limit = 8`

Next, in groups of `join_collapse_limit` tables

## Comma-separated join

Full enumeration when the number of joins is no more than `geqo_threshold = 12`

Next, using a genetic algorithm

## Materialization – Planner Hint

13

As the number of tables in a query increases, the number of join combinations—and thus the cost of choosing a plan—grows exponentially.

If the number of joins (using the JOIN syntax) exceeds `join_collapse_limit` (default 8), the planner evaluates potential join combinations in groups of `join_collapse_limit` tables and then merges these groups.

There's a similar parameter for subqueries in the FROM clause — `from_collapse_limit`.

If tables are joined with commas (without the JOIN keyword), the planner evaluates all join combinations but switches to a genetic algorithm when the number of joins exceeds `geqo_threshold` (default 12).

This can result in suboptimal execution plans. Details can be found in Pavel Tolmachev's article: <https://habr.com/ru/company/postgrespro/blog/662021/>

By manually managing materialization—either through CTE or by breaking down the query into parts and utilizing temporary tables—developers can group tables, allowing the optimizer to plan each separately. Typically, CTE is a more straightforward and efficient way, but the second option enables analyzing a temporary table, thereby giving the planner more information about the data.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

## Managing Join Order

The default value of the `join_collapse_limit` parameter is chosen to ensure that join planning for that many tables doesn't require excessive resources:

```
=> SHOW join_collapse_limit;
```

```
join_collapse_limit
-----
8
(1 row)
```

Let's execute a query that explicitly joins tables using the JOIN keyword:

```
=> EXPLAIN (costs on)
SELECT *
FROM tickets t
JOIN ticket_flights tf ON (tf.ticket_no = t.ticket_no)
JOIN flights f ON (f.flight_id = tf.flight_id);
```

### QUERY PLAN

```
-----
Hash Join (cost=171633.93..778520.95 rows=8391708 width=199)
  Hash Cond: (tf.ticket_no = t.ticket_no)
    -> Hash Join (cost=9767.51..302687.25 rows=8391708 width=95)
      Hash Cond: (tf.flight_id = f.flight_id)
        -> Seq Scan on ticket_flights tf (cost=0.00..153850.08 rows=8391708 width=32)
        -> Hash (cost=4772.67..4772.67 rows=214867 width=63)
          -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 width=63)
      -> Hash (cost=78910.30..78910.30 rows=2949530 width=104)
        -> Seq Scan on tickets t (cost=0.00..78910.30 rows=2949530 width=104)
(9 rows)
```

In the selected plan, the flights (flights) and ticket\_flights tables are joined first, and the resulting set is then joined with the tickets (tickets) table.

By setting the `join_collapse_limit` parameter to 1, you can enforce a specific join order:

```
=> SET join_collapse_limit = 1;
```

```
SET
```

```
=> EXPLAIN (costs on)
SELECT *
FROM tickets t
JOIN ticket_flights tf ON (tf.ticket_no = t.ticket_no)
JOIN flights f ON (f.flight_id = tf.flight_id);
```

### QUERY PLAN

```
-----
Hash Join (cost=171633.93..860470.95 rows=8391708 width=199)
  Hash Cond: (tf.flight_id = f.flight_id)
    -> Hash Join (cost=161866.42..498563.77 rows=8391708 width=136)
      Hash Cond: (tf.ticket_no = t.ticket_no)
        -> Seq Scan on ticket_flights tf (cost=0.00..153850.08 rows=8391708 width=32)
        -> Hash (cost=78910.30..78910.30 rows=2949530 width=104)
          -> Seq Scan on tickets t (cost=0.00..78910.30 rows=2949530 width=104)
      -> Hash (cost=4772.67..4772.67 rows=214867 width=63)
        -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 width=63)
(9 rows)
```

Now, the tables are joined in the order they appear in the query, even though the query cost is higher.

However, this approach places too much responsibility on the query author. The planner can only choose which set of rows to use as the outer or inner join.

```
=> RESET join_collapse_limit;
```

```
RESET
```

It is recommended to familiarize yourself with the similar parameter `from_collapse_limit` in practice.

## Materializing the Query Result

- read-only

- It is possible to create indexes

## Data Update

- Manual synchronization

- Incremental Update: pg\_ivm Extension

## Vacuum and analysis

- manually and automatically

A materialized view is a named query result stored at the database level. A materialized view can be treated like a regular read-only table.

You can create indexes on a materialized view (but you can't add integrity constraints — these must be enforced in the base tables). A materialized view collects the same statistics as regular tables.

Unlike regular views, the rows of a materialized view remain unchanged when the base tables are modified. Synchronization is done manually.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

Full synchronization of a materialized view may be too expensive. Built-in incremental update (individual rows as base tables change) is not natively supported, but the pg\_ivm extension enables this feature (author – Suguru Nagata, [https://github.com/sraoss/pg\\_ivm](https://github.com/sraoss/pg_ivm))



## Materialized Views Materialized Views

Create a materialized view:

```
=> CREATE MATERIALIZED VIEW airports_msk AS
SELECT *
FROM airports
WHERE timezone = 'Europe/Moscow';
```

```
SELECT 44
```

```
=> \dt airports_msk
```

```
          List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
pg_temp_3 | airports_msk | table | postgres
(1 row)
```

The pg\_temp schema is prioritized in the search order, so you'll need to access the materialized view using its full name. This is inconvenient; it's better to remove the temporary table:

```
=> DROP TABLE pg_temp.airports_msk;
```

```
DROP TABLE
```

Materialized views support indexing:

```
=> CREATE UNIQUE INDEX ON airports_msk (airport_code);
```

```
CREATE INDEX
```

```
=> EXPLAIN (costs off) SELECT *
FROM airports_msk
ORDER BY airport_code
LIMIT 3;
```

```
          QUERY PLAN
-----
Limit
  -> Index Scan using airports_msk_airport_code_idx on airports_msk
(2 rows)
```

During analysis and auto-analysis, materialized views gather the same statistics as tables.

When the content of underlying tables changes, the rows in the materialized view remain unchanged:

```
=> INSERT INTO airports_data (airport_code, airport_name, city, coordinates, timezone)
VALUES ('ZIA', '{"en": "Zhukovsky"}', '{}', point(38.1517, 55.5533), 'Europe/Moscow');
```

```
INSERT 0 1
```

```
=> SELECT count(*)
FROM airports_msk
WHERE airport_code = 'ZIA';
```

```
count
-----
0
(1 row)
```

Synchronization must be performed explicitly. The REFRESH command fully locks the materialized view during rebuilding, which may be undesirable. In this case, you can avoid this by using the CONCURRENTLY option, as a unique index exists on the materialized view:

```
=> REFRESH MATERIALIZED VIEW CONCURRENTLY airports_msk;
```

```
REFRESH MATERIALIZED VIEW
```

```
=> SELECT count(*)
FROM airports_msk
WHERE airport_code = 'ZIA';
```

```
count
-----
      1
(1 row)
```

The optimizer can materialize rows from the plan nodes for reuse

You can control materialization using CTEs, temporary tables, and materialized views.

Materialization enables control over the join order

1. At the start of the demo, examples of two queries with a Materialize node were shown. Instruct the planner not to use this node and check whether it can avoid materialization in some cases and not in others.
2. Check the query execution plan with the `from_collapse_limit` parameter set to 8 (default) and 1:

```
SELECT *FROM ( SELECT * FROM ticket_flights tf, tickets
t WHERE tf.ticket_no = t.ticket_no ) ttf, flights fWHERE
f.flight_id = ttf.flight_id;
```

1. 1. Use the `enable_material` parameter `enable_material`:  
<https://postgrespro.ru/docs/postgresql/16/runtime-config-query#GUC-ENABLE-MATERIAL>

2. 2. See the documentation for details on the parameter:  
<https://postgrespro.ru/docs/postgresql/16/explicit-joins>

## 1. Disabling the Materialize Node

Let's examine the execution plan for the first query in the demonstration:

```
=> EXPLAIN SELECT a1.city, a2.city
FROM airports a1, airports a2
WHERE a1.timezone = 'Europe/Moscow'
AND abs(a2.coordinates[1]) > 66.652;
```

### QUERY PLAN

```
-----
Nested Loop (cost=0.00..805.90 rows=1540 width=64)
-> Seq Scan on airports_data ml (cost=0.00..4.30 rows=44 width=49)
    Filter: (timezone = 'Europe/Moscow'::text)
-> Materialize (cost=0.00..4.74 rows=35 width=49)
    -> Seq Scan on airports_data ml_1 (cost=0.00..4.56 rows=35 width=49)
        Filter: (abs(coordinates[1]) > '66.652'::double precision)
(6 rows)
```

Let's request the planner to avoid using materialization:

```
=> SET enable_material = off;
```

SET

```
=> EXPLAIN SELECT a1.city, a2.city
FROM airports a1, airports a2
WHERE a1.timezone = 'Europe/Moscow'
AND abs(a2.coordinates[1]) > 66.652;
```

### QUERY PLAN

```
-----
Nested Loop (cost=0.00..948.16 rows=1540 width=64)
-> Seq Scan on airports_data ml_1 (cost=0.00..4.56 rows=35 width=49)
    Filter: (abs(coordinates[1]) > '66.652'::double precision)
-> Seq Scan on airports_data ml (cost=0.00..4.30 rows=44 width=49)
    Filter: (timezone = 'Europe/Moscow'::text)
(5 rows)
```

The planner avoided materialization, but the plan's cost increased.

Let's check the second query: Let's look at the second query:

```
=> RESET enable_material;
```

RESET

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

### QUERY PLAN

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

```
=> SET enable_material = off;
```

SET

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

## QUERY PLAN

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

In this case, the planner must use materialization because a merge join requires not just moving forward through the dataset but also moving backward.

```
=> RESET enable_material;
```

```
RESET
```

## 2. The from\_collapse\_limit Parameter

First, let's see how the query behaves with the default from\_collapse\_limit setting:

```
=> SHOW from_collapse_limit;
```

```
from_collapse_limit
-----
8
(1 row)
```

```
=> EXPLAIN (costs off, summary off, settings)
```

```
qSELECT *
```

```
FROM
```

```
(
  SELECT *
  FROM ticket_flights tf, tickets t
  WHERE tf.ticket_no = t.ticket_no
) ttf,
flights f
WHERE f.flight_id = ttf.flight_id;
```

```
ERROR:  syntax error at or near "qSELECT"
```

```
LINE 2: qSELECT *
       ^
```

The subquery joins two tables, but the optimizer unfolds it and determines the join order for all three tables. The initial joins here involve the ticket\_flights and flights tables.

Now let's set from\_collapse\_limit to 1 and examine the new query plan for the same query:

```
=> SET from_collapse_limit = 1;
```

```
SET
```

```
=> EXPLAIN (costs off, summary off, settings)
```

```
SELECT *
```

```
FROM
```

```
(
  SELECT *
  FROM ticket_flights tf, tickets t
  WHERE tf.ticket_no = t.ticket_no
) ttf,
flights f
WHERE f.flight_id = ttf.flight_id;
```

## QUERY PLAN

```
-----
Hash Join
  Hash Cond: (tf.flight_id = f.flight_id)
    -> Hash Join
          Hash Cond: (tf.ticket_no = t.ticket_no)
            -> Seq Scan on ticket_flights tf
            -> Hash
                  -> Seq Scan on tickets t
    -> Hash
          -> Seq Scan on flights f
Settings: search_path = 'bookings, public', jit = 'off', from_collapse_limit = '1'
(10 rows)
```

The query plan has changed — the subquery is no longer unfolded, and the first join involves two tables from it.

If a JOIN construct is used in FROM instead of a list of tables, the optimizer first converts each such construct into a table list

(processing tables in groups of up to `join_collapse_limit`), then unfolds subqueries (to a depth of up to `from_collapse_limit`). Therefore, it's advisable to set both parameters to the same value.