Engineer

Materialized view

a view that stores the result of its query like a table

What it is

A materialized view is a query whose result the database keeps, so that reading it returns rows computed earlier instead of running the query again. In British spelling it is a materialised view. It sits between two older objects. A plain view stores only its query, and every read runs that query against the tables underneath. A table created with `CREATE TABLE AS` stores a result but forgets the query that produced it. A materialized view keeps both: the rows, which reads come from, and the query, which can be run again to replace them.

PostgreSQL added materialized views in version 9.3, released on September 9, 2013, and its release notes credit the work to Kevin Grittner. The documentation describes them as using the rule system the way ordinary views do while persisting the result in a table-like form. To the query parser a materialized view is a relation like any table or view, so a `SELECT` against it reads the stored rows directly; the stored query matters only when the view is filled. The view cannot be updated directly. To replace its contents with a fresh query result, the stored query must run again.

Two commands do that work. `CREATE MATERIALIZED VIEW` defines the view and, unless it is told `WITH NO DATA`, runs the query at once to fill it. `REFRESH MATERIALIZED VIEW` runs the stored query again and replaces the whole content with the new result. Version 9.4, released on December 18, 2014, added `CONCURRENTLY`, which lets a refresh run without blocking the sessions that read the view. Both commands are PostgreSQL extensions, not part of the SQL standard; other databases offer the idea under their own syntax and rules.

Use the concept in a decision

Put “What is a materialized view?” to work in a real decision. We can help relate the concept to the system you are building or buying. Finding the queries and the missing indexes behind a system that has got heavier every year, and fixing the ones that matter.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. 30 days of electricity data in 180 ms — 180 ms to aggregate a month of meter readings

30 days of electricity data in 180 ms Get in touch →

Why it matters

Some queries are expensive and asked often: a dashboard that sums a year of orders by day, a report that joins six tables, a page that ranks products by sales. Run against the source tables, every request pays for the full scan, join and aggregation again, although the answer barely moves between requests. A materialized view pays that cost once per refresh, and every read after that is a scan of a small table computed in advance. PostgreSQL’s documentation says reading the stored data is “often much faster than accessing the underlying tables directly or through a view”.

The price is freshness. Between refreshes the view shows the data as it stood at the last one, and in PostgreSQL nothing refreshes it unless someone, or something scheduled, asks. Many uses can live with that. Yesterday’s sales do not change today, and a monthly report has no need for this minute’s orders. What the materialized view forces is an explicit decision: how old the answer may be, and whose job it is to keep it from getting older.

It also gives indexes a place to live: the stored result can carry indexes of its own, built for the queries that read it. The PostgreSQL documentation shows a second use for that: data brought in from another system through a foreign data wrapper, which may not support indexes at all, can be copied into a materialized view that does.

For a team that keeps its schema normalized, this is often where the flattened shape belongs. Database normalization stores each fact once and answers most questions through joins. A materialized view can hold the joined, summed result that reporting wants, while the source tables keep their single copy of each fact and every write still goes to one place.

What it looks like in practice

Take a shop's order tables: `orders`, with one row per order and the date it was placed, and `order_lines`, with the SKU, quantity and unit price of each line. A sales dashboard wants units and revenue per product per day. The query is short, but over several years of orders it reads every line:

```
CREATE MATERIALIZED VIEW daily_sales AS
SELECT o.ordered_on                  AS day,
       l.sku,
       sum(l.quantity)               AS units,
       sum(l.quantity * l.unit_price) AS revenue
FROM   orders o
JOIN   order_lines l USING (order_id)
WHERE  o.ordered_on < CURRENT_DATE
GROUP  BY o.ordered_on, l.sku;
```

The `WHERE` clause leaves out today, whose numbers are still moving, so the dashboard shows complete days only; the example in PostgreSQL's own documentation treats a day of unfinished sales the same way. The command runs the query once and stores the result. Next comes an index:

```
CREATE UNIQUE INDEX daily_sales_sku_day ON daily_sales (sku, day);
```

The index does two jobs. It lets the dashboard find one product's days without reading the whole view, and it is what makes a concurrent refresh possible later. The query groups by day and SKU, so each pair appears once and the index can be unique. From now on the dashboard reads the view like a table:

```
SELECT day, units, revenue
FROM   daily_sales
WHERE  sku = 'TEA-250'
ORDER  BY day;
```

Every night, shortly after midnight, a scheduled job brings it up to date:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY daily_sales;
```

Without `CONCURRENTLY`, a refresh that touches many rows tends to use fewer resources and finish sooner, but it could block the dashboard's reads until it is done. With it, reads carry on during the refresh, and the documentation notes it may even be the faster choice when only a few rows change, as here, where one new day arrives each night. The option has two conditions. The view must already hold data, and it must have at least one unique index built from plain column names that covers every row, so no expression index and no `WHERE` clause. The index above meets the second; creating the view with its data met the first.

Other databases draw the line in other places. In BigQuery, materialized views are maintained in the background: changes to the base tables are added incrementally without anyone asking, a query against the view merges in recent changes so the result stays fresh, and smart tuning can reroute a query written against the base tables to the view instead. That holds for incremental views, which accept a limited set of SQL. A view defined with `allow_non_incremental_definition = true` accepts most queries, but every refresh runs the full query and its results are not always fresh.

SQL Server reaches a similar result through indexed views. Adding a clustered index that enforces uniqueness to a view, which must be created `WITH SCHEMABINDING` and have a deterministic definition, stores its result using the same storage arrangement as an indexed table, and the optimizer can use it even for a query that does not name the view. SQL Server updates the indexed view along with every insert, update and delete on the tables it depends on, so there is no refresh to schedule; the cost moves to the writes, and Microsoft's documentation warns that their performance can degrade significantly.

The same trade-off shows up in production work. In an earlier role, the founder built PL/pgSQL ETL and ELT pipelines that used partitioned tables and materialized views to speed up reads on large datasets, alongside primary and foreign key constraints for referential integrity and unique and composite indexes for the joins.

Where it goes wrong

Stale numbers that look current. A refresh job that stopped running a week ago leaves the view answering every query, quickly and wrongly, and nothing in the result says how old it is. Have the job record the time of each successful refresh somewhere the dashboard can show it, and alert when that time is older than the readers agreed to accept.

A refresh that blocks the readers. A plain `REFRESH MATERIALIZED VIEW` over a large view could block every session trying to read it until the refresh finishes. Add a unique index and use `CONCURRENTLY`, or schedule plain refreshes for hours when nobody reads the view. When most rows change on every run, the plain refresh may still be the cheaper one.

`CONCURRENTLY` **refused.** The option needs a unique index on plain columns covering all rows. A unique index on `lower(email)`, or one limited by a `WHERE` clause, does not qualify. It also needs a view that already holds data, so it fails on a view created `WITH NO DATA`, and the two options cannot be combined. Add a qualifying index, and fill the view once with a plain refresh.

An unscannable view. `WITH NO DATA` creates the view without running its query and flags it unscannable, and any query against it fails until the first refresh. That can be the right choice in a migration that should not wait for a long query, but then the first refresh has to be planned, not forgotten.

A changed definition that never arrived. `CREATE MATERIALIZED VIEW IF NOT EXISTS` in a migration does nothing, beyond a notice, when a view of that name already exists, and the documentation gives no guarantee that the existing view resembles the new definition. A migration that changes the query this way leaves the old one in place. Drop the view with `DROP MATERIALIZED VIEW` and create it again in the same migration.

Refresh jobs that overlap. A materialized view can have only one refresh in progress, even with `CONCURRENTLY`, so a schedule tighter than the refresh itself only leaves the view further behind. Measure how long a refresh takes and schedule from that.

Trusting the order of the rows. If the defining query has an `ORDER BY`, the first contents come out in that order, but a refresh does not promise to keep it. Put the `ORDER BY` in the queries that read the view.

Functions that lose their tables. While the defining query runs, at creation or on refresh, PostgreSQL temporarily sets `search_path` to `pg_catalog, pg_temp`, and restricts what the query may do: a function that creates temporary tables fails. A function that finds its tables by unqualified name can pass every test by hand and break only inside the refresh. Schema-qualify the table names in any function the view calls.

The wrong role runs the job. Refreshing requires the `MAINTAIN` privilege on the materialized view. A job running as a reporting role that may read the view but not maintain it fails every night. Give the role that runs the refresh that privilege.

A stored copy where an index would do. If a query is slow because an index is missing, a materialized view hides the problem and adds staleness on top. Read the query plan first. When a query only needs a name, a plain view is enough and is always current; keep materialized views for results that are expensive to compute and safe to read slightly old.

Further reading

- Materialized Views — PostgreSQL's chapter on how materialized views relate to views and tables, with a sales summary refreshed nightly and a foreign data wrapper example.
- CREATE MATERIALIZED VIEW — the command reference, including `WITH NO DATA`, `IF NOT EXISTS` and the restricted `search_path` during the query.
- REFRESH MATERIALIZED VIEW — the refresh command, the conditions for `CONCURRENTLY` and the `MAINTAIN` privilege it requires.
- PostgreSQL 9.3 release notes — the release, in September 2013, that added materialized views.
- Introduction to materialized views — BigQuery's incremental and non-incremental materialized views, automatic refresh and smart tuning.
- Create indexed views — SQL Server's indexed views, their requirements and their cost to writes.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a materialized view?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/materialized-view/