Production Postgres | Analytics

Indexing Materialized Views in Postgres

Elizabeth Christensen

Feb 3, 2025 • 5 min read • More by this author

Materialized views are widely used in Postgres today. Many of us are working with using connected systems through foreign data wrappers, separate analytics systems like **data warehouses**, and merging data from different locations with Postgres queries. Materialized views let you precompile a query or partial table, for both local and remote data. Materialized views are static and have to be refreshed.

One of the things that can be really important for using materialized views efficiently is indexing.

Adding indexes to Postgres in general is critical for operation and query performance. Adding indexes for materialized views is also generally recommended for a few different reasons.

- Materialized views are typically used for larger data sets where queries or large joins benefit from being precompiled and indexes add an additional performance boost.
- Even if an underlying table data has indexes, those indexes will not be used in a materialized view. The materialized view is saved on disk separately, so it needs a stand alone index.
- When querying a materialized view, Postgres treats it like a regular table. There's no special query planner for materialized views. If it is the type of data that would benefit from an index if it was table, the materialized view would benefit from an index.

Views vs Materialized views

In case you have not thought about Postgres views and Postgres materialized views recently, let's just do a quick refresher.

- A view is a saved query. It is not stored on the disk. It dynamically fetches data from the underlying tables whenever queried. Since views do not have their own storage, **views cannot have indexes**.
- Materialized views do not dynamically fetch data from underlying tables- they are stored on disk - and must be explicitly refreshed to update the contents. This makes them ideal for scenarios involving complex queries

or frequent access to relatively static datasets. Because they can be stored on disk, **materialized views can be indexed**.

Building a materialized view with indexes

I have an example in a **materialized view tutorial** on the **Postgres developer tutorials site**. There's three tables from a demo ecommerce site - products, orders, product_orders. We'd like to show on our demo site how often this product has been purchased. This is helpful for marketing the product but we don't need to recalculate this from the database every time a product is displayed. So static information in a materialized view is perfect for this use case. Pre-joining tables and having this ready to go will make queries to the sku really easy.

Here's a sample materialized view that shows recent product sales by sku.

```
CREATE MATERIALIZED VIEW recent_product_sales AS
SELECT
    p.sku,
    SUM(po.qty) AS total_quantity
FROM
    products p
    JOIN product_orders po ON p.sku = po.sku
    JOIN orders o ON po.order_id = o.order_id
WHERE
    o.status = 'Shipped'
GROUP BY
    p.sku
ORDER BY
    2 DESC;
```

We will likely be looking this up by sku, so we can add a simple b-tree index, calling out the materialized view like we would a table.

```
CREATE INDEX sku_index ON recent_product_sales (sku);
```

Creating indexes for materialized views works exactly like it does with tables. Postgres supports all the major index types, B-tree, hast, GiST, GIN, BRIN, and others on materialized views. If you need a basic intro to indexing I have a **blog on Postgres Indexing types**.

Refreshing our materialized view and indexes

Materialized views are static, so to add new data, we have to refresh it. There's two ways Postgres can refresh a materialized view. A regular refresh and one done concurrently.

Non-Concurrent (locking) refresh

This refresh completely replaces the content of the materialized view. The index you built prior to this remain and Postgres will recreate the index with the refreshed data.

Postgres acquires an exclusive lock on the materialized view during this refresh, preventing any reads or writes. This is the fastest option but it often won't work for production systems with live reads coming in.

```
REFRESH MATERIALIZED VIEW recent_product_sales;
```

In addition to building the materialized view, Postgres will have to rebuild the index from scratch. Depending on data size, this can be a pretty long operation.

Concurrent (non-locking) refresh

This refresh will update the materialized view without locking the table, letting you read currently while the refresh is happening. This utilizes the Postgres reindex concurrently too if you're familiar with that feature. Postgres will reindex everything as the data is refreshed. This is normally slower than a regular refresh due to the incremental approach but allowing reads during the process makes it the favorable option for production databases.

Concurrent refresh **requires a unique index** on the materialized view to function. The unique index ensures that each row in the materialized view can be uniquely identified. The b-tree index we added earlier has not been explicitly declared as unique, so we can add a new unique index and drop the old one.

```
CREATE UNIQUE INDEX unique_idx_recent_product_sales ON recent_product_sales(sku);

DROP INDEX sku_index ON recent_product_sales(sku);
```

Now we can do our concurrent refresh:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY recent_product_sales;
```

Materialized views that generate columns with non-unique values cannot use unique indexes - and cannot use the concurrent refresh option. In that case, you'll have to work around it with the regular refresh.

Summary

There's always additional considerations with indexing. Unique planning for each project is needed to review index usage based on query patterns, refresh frequency, and the materialized view's size to maximize efficiency. Indexes are stored on disk so they require their own additional storage. They can have a performance impact, especially with large or complex views and indexes. You can monitor the time index refreshes are taking \timing or logs.

Summary notes:

- Even if an underlying table data has indexes, they have to be recreated with materialized views.

- Materialized views often benefit from indexing.
- Materialized views are static and need to be refreshed. A regular refresh will lock the view from reads, a concurrent refresh will not.
- When using the refresh concurrently option for materialized views, the data has to have a UNIQUE index.

Migrate from RDS

Migrate from Heroku

Resources

Customer Portal

Documentation

Postgres Tutorials

Crunchy Bridge Walkthrough

Postgres Operator Walkthrough

Blog

Events

Company

About

Team

News

Contact Us

Newsletter

Branding

Security

Subscribe to the Crunchy Data Newsletter
and receive Postgres content every month.

Enter your email

Subscribe