1. Introduction

- A Materialized View is a database object that stores the result of an SQL query physically in the database.
- It is useful when we have complex queries involving multiple tables, joins, and calculations that take more time to execute.

For example, in the DVD Rental database, we may want to calculate the total payments made by every customer.

Instead of executing the same calculation repeatedly, we can create a Materialized View that stores the calculated results.

2. View vs. Materialized View

Both Views and Materialized Views are based on SQL queries, but they work differently.

Feature	View	Materialized View
Stores query definition	Yes	Yes
Stores query results physically	No	Yes
Executes the query when accessed	Yes	No, normally reads stored results
Automatically reflects table changes	Yes	No
Requires refresh	No	Yes
Uses additional storage	Normally no result storage	Yes

Simple example:

- A View is like a live report that runs its query whenever you open it.
- A Materialized View is like a saved report. It displays previously calculated results until you refresh it.

3. Create Your First Materialized View

We will use the PostgreSQL DVD Rental database (`dvdrental`).

Task: Create a report showing the total payment made by each customer.

Step 1: Understand the tables

We need two tables:

- `customer`: Contains customer information.
- `payment`: Contains payment records.

The `customer_id` column connects these tables.

Step 2: Write the SQL query

First, execute the query normally:

```
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    SUM(p.amount) AS total_payment
FROM customer c
JOIN payment p
    ON c.customer_id = p.customer_id
GROUP BY
    c.customer_id,
    c.first_name,
    c.last_name
ORDER BY total_payment DESC;
```

This query calculates the total payment made by each customer.

Step 3: Create a Materialized View

Now save the query results as a Materialized View.

```
CREATE MATERIALIZED VIEW mv_customer_payment_summary AS
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    SUM(p.amount) AS total_payment
FROM customer c
JOIN payment p
    ON c.customer_id = p.customer_id
GROUP BY
    c.customer_id,
    c.first_name,
    c.last_name;
```

Created

The Materialized View is now created and populated with the query results.

Step 4: Retrieve data

We can retrieve the stored results just like a table:

```
SELECT *
FROM mv_customer_payment_summary;
```

We can also use filtering and sorting:

```
SELECT *
FROM mv_customer_payment_summary
WHERE total_payment > 150;
```

```
SELECT *
FROM mv_customer_payment_summary
ORDER BY total_payment DESC;
```

4. Refresh a Materialized View

A Materialized View stores results at the time it is created or refreshed.

Suppose a new payment is added to the `payment` table:

```
INSERT INTO payment (
    customer_id,
    staff_id,
    rental_id,
    amount,
    payment_date
)
VALUES (
    1,
    1,
    1,
    10.00,
    CURRENT_TIMESTAMP
);
```

The stored results in `mv_customer_payment_summary` will not automatically include this new payment.

To update the stored results, execute:

```
REFRESH MATERIALIZED VIEW mv_customer_payment_summary;
```

Now retrieve the updated results:

```
SELECT *
FROM mv_customer_payment_summary
WHERE customer_id = 1;
```

Important: Refreshing the Materialized View executes its original query again and replaces the stored results with updated results.

5. Create a Materialized View Without Data

PostgreSQL provides the `WITH NO DATA` option.

It creates the Materialized View without immediately storing the query results.

```
CREATE MATERIALIZED VIEW mv_customer_payments AS
SELECT
    customer_id,
    SUM(amount) AS total_payment
FROM payment
GROUP BY customer_id
WITH NO DATA;
```

You cannot retrieve data from this Materialized View until it is populated.

To populate it:

```
REFRESH MATERIALIZED VIEW mv_customer_payments;
```

By default, PostgreSQL uses `WITH DATA` when creating a Materialized View.

6. Create a Unique Index on a Materialized View

Indexes can improve data retrieval performance.

We can create a unique index on `customer_id`:

```
CREATE UNIQUE INDEX idx_mv_customer_payment_customer
ON mv_customer_payment_summary (customer_id);
```

This index also enables us to use concurrent refresh, provided the Materialized View is already populated and the unique index meets PostgreSQL's requirements.

7. Refresh a Materialized View Concurrently

Normally, refreshing a Materialized View can block queries that are trying to read it.

PostgreSQL provides:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY
mv_customer_payment_summary;
```

This allows other sessions to continue reading the Materialized View during the refresh.

Important requirements:

- The Materialized View must already be populated.
- It must have a qualifying unique index.
- The unique index must use ordinary columns and cover all rows.

- `CONCURRENTLY` cannot be used with `WITH NO DATA`.

8. Create Another Practical Materialized View

Task: Create a report showing the number of films and average rental rate for each category.

```
CREATE MATERIALIZED VIEW mv_category_film_summary AS
SELECT
    c.category_id,
    c.name AS category_name,
    COUNT(fc.film_id) AS total_films,
    AVG(f.rental_rate) AS average_rental_rate
FROM category c
JOIN film_category fc
    ON c.category_id = fc.category_id
JOIN film f
    ON fc.film_id = f.film_id
GROUP BY
    c.category_id,
    c.name;
```

Retrieve the results:

```
SELECT *
FROM mv_category_film_summary
ORDER BY total_films DESC;
```

Refresh the results:

```
REFRESH MATERIALIZED VIEW mv_category_film_summary;
```

9. Rename a Materialized View

Use `ALTER MATERIALIZED VIEW` to rename it.

```
ALTER MATERIALIZED VIEW mv_customer_payment_summary
RENAME TO mv_customer_payments_report;
```

10. Delete a Materialized View

Use `DROP MATERIALIZED VIEW` to remove it.

```
DROP MATERIALIZED VIEW mv_customer_payments_report;
```

To avoid an error if it does not exist:

```
DROP MATERIALIZED VIEW IF EXISTS mv_customer_payments_report;
```

11. View All Materialized Views

PostgreSQL provides a system catalog view named `pg_matviews`.

Use the following query:

```
SELECT
    schemaname,
    matviewname,
    ispopulated
FROM pg_matviews
WHERE schemaname = 'public';
```

The `ispopulated` column indicates whether the Materialized View currently contains stored results.

For more details about `pg_matviews`, See PostgreSQL pg_matviews: List Materialized Views

12. When Should We Use Materialized Views?

Materialized Views are useful when:

1. A query involves multiple table joins.
2. A report contains complex calculations.
3. The same report is accessed frequently.
4. Query performance is important.
5. Data does not need to be updated immediately after every table change.

Limitations:

- They require additional disk storage.
- Their results can become outdated.
- They must be refreshed to reflect changes in source tables.
- Refreshing large Materialized Views can take time.

13. Quick Revision

Task	SQL Command
Create	`CREATE MATERIALIZED VIEW`
Retrieve	`SELECT`
Refresh	`REFRESH MATERIALIZED VIEW`
Concurrent refresh	`REFRESH MATERIALIZED VIEW CONCURRENTLY`
Create index	`CREATE UNIQUE INDEX`

Task	SQL Command
Rename	`ALTER MATERIALIZED VIEW ... RENAME TO`
Delete	`DROP MATERIALIZED VIEW`
List	`SELECT ... FROM pg_matviews`