1. Lab Objectives

After completing this lab, students will be able to:

1. Create Views using `CREATE VIEW`.
2. Retrieve records from Views using `SELECT`.
3. Apply filtering and sorting to Views.
4. Create Views using multiple tables and JOIN operations.
5. Create Views using aggregate functions and GROUP BY.
6. Modify Views using `CREATE OR REPLACE VIEW`.
7. Rename and delete Views.
8. Create Materialized Views using `CREATE MATERIALIZED VIEW`.
9. Understand the difference between Views and Materialized Views.
10. Refresh Materialized Views when source data changes.
11. Create indexes on Materialized Views.
12. Use concurrent refresh and manage Materialized Views.

2. Lab Requirements

- PostgreSQL installed on the computer.
- pgAdmin or another PostgreSQL client.
- DVD Rental sample database (`dvdrental`).

Connect to the `dvdrental` database before executing the following queries.

3. Explore the Database

The DVD Rental database contains information about films, customers, actors, categories, rentals, payments, and stores.

Important tables:

Table	Description
`film`	Film information
`actor`	Actor names
`customer`	Customer information
`category`	Film categories
`film_category`	Relationship between films and categories
`inventory`	Film copies available in stores
`rental`	Rental transactions
`payment`	Customer payments
`staff`	Staff information
`store`	Store information

Task 1: Explore the tables

Execute the following queries:

```
SELECT * FROM film LIMIT 10;

SELECT * FROM customer LIMIT 10;

SELECT * FROM actor LIMIT 10;

SELECT * FROM category LIMIT 10;

SELECT * FROM rental LIMIT 10;

SELECT * FROM payment LIMIT 10;
```

Observe the columns and records in each table.

Part A: Views in PostgreSQL

4. Practical Task 1: Create a View of Film Details

Objective: Create a View containing selected columns from the `film` table.

Step 1: Execute the original query

```
SELECT
    film_id,
    title,
    release_year,
    rental_rate,
    length
FROM film;
```

Step 2: Create a View

```
CREATE VIEW film_details AS
SELECT
    film_id,
    title,
    release_year,
    rental_rate,
    length
FROM film;
```

Step 3: Retrieve records

```
SELECT * FROM film_details;
```

Step 4: Apply a condition

```
SELECT *
FROM film_details
WHERE rental_rate = 2.99;
```

Step 5: Sort the records

```
SELECT *
FROM film_details
ORDER BY title;
```

Observation: A View allows us to reuse a saved SELECT query.

5. Practical Task 2: Create a View of Customer Details

Objective: Create a View displaying customer contact information.

Step 1: Create the View

```
CREATE VIEW customer_details AS
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    active
FROM customer;
```

Step 2: Retrieve all customers

```
SELECT *
FROM customer_details;
```

Step 3: Search for a customer

```
SELECT *
FROM customer_details
WHERE first_name = 'MARY';
```

Step 4: Sort customers alphabetically

```
SELECT *
FROM customer_details
ORDER BY first_name;
```

Observation: We can apply WHERE and ORDER BY when retrieving records from a View.

6. Practical Task 3: Create a View Using JOIN

Objective: Combine film and category information using multiple tables.

The relationship involves three tables:

- `film`
- `film_category`
- `category`

Step 1: Execute the JOIN query

```
SELECT
    f.film_id,
    f.title,
    c.name AS category_name
FROM film AS f
INNER JOIN film_category AS fc
    ON f.film_id = fc.film_id
INNER JOIN category AS c
    ON fc.category_id = c.category_id;
```

Step 2: Create the View

```
CREATE VIEW film_categories AS
SELECT
    f.film_id,
    f.title,
    c.name AS category_name
FROM film AS f
INNER JOIN film_category AS fc
    ON f.film_id = fc.film_id
INNER JOIN category AS c
    ON fc.category_id = c.category_id;
```

Step 3: Retrieve records

```
SELECT *
FROM film_categories;
```

Step 4: Display Action films

```sql
SELECT *
FROM film_categories
WHERE category_name = 'Action';
```

Step 5: Sort films by category

```sql
SELECT *
FROM film_categories
ORDER BY category_name, title;
```

Observation: A View can combine information from multiple tables using JOIN.

7. Practical Task 4: Create a View Using CONCAT()

Objective: Combine actor first names and last names into one column.

Step 1: Create the View

```sql
CREATE VIEW actor_details AS
SELECT
    actor_id,
    CONCAT(first_name, ' ', last_name) AS actor_name
FROM actor;
```

Step 2: Retrieve records

```sql
SELECT *
FROM actor_details;
```

Step 3: Search for actors

```sql
SELECT *
FROM actor_details
WHERE actor_name LIKE 'PEN%';
```

Observation: SQL functions can be used inside a View.

8. Practical Task 5: Create a Rental Details View

Objective: Display customer names, film titles, and rental dates in one result.

Required tables:

- `rental`
- `customer`
- `inventory`

- `film`

Step 1: Create the View

```
CREATE VIEW rental_details AS
SELECT
    r.rental_id,
    c.customer_id,
    c.first_name,
    c.last_name,
    f.title,
    r.rental_date,
    r.return_date
FROM rental AS r
INNER JOIN customer AS c
    ON r.customer_id = c.customer_id
INNER JOIN inventory AS i
    ON r.inventory_id = i.inventory_id
INNER JOIN film AS f
    ON i.film_id = f.film_id;
```

Step 2: Retrieve rental records

```
SELECT *
FROM rental_details;
```

Step 3: Display rentals of customer ID 1

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

Step 4: Display recent rentals first

```
SELECT *
FROM rental_details
ORDER BY rental_date DESC;
```

Observation: Multiple JOIN operations can be saved inside a View.

9. Practical Task 6: Create a Customer Payments View

Objective: Display customer information along with payment details.

Step 1: Create the View

```
CREATE VIEW customer_payments AS
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    p.payment_id,
    p.amount,
    p.payment_date
FROM customer AS c
INNER JOIN payment AS p
    ON c.customer_id = p.customer_id;
```

Step 2: Retrieve records

```
SELECT *
FROM customer_payments;
```

Step 3: Display payments greater than 5

```
SELECT *
FROM customer_payments
WHERE amount > 5;
```

Step 4: Sort payments in descending order

```
SELECT *
FROM customer_payments
ORDER BY amount DESC;
```

10. Practical Task 7: Create a View Using Aggregate Functions

Objective: Calculate the total payment made by each customer.

Step 1: Execute the query

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

```
    c.first_name,
    c.last_name;
```

Step 2: Create the View

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

Step 3: Retrieve the summary

```
SELECT *
FROM customer_payment_summary;
```

Step 4: Display customers who paid more than 100

```
SELECT *
FROM customer_payment_summary
WHERE total_payment > 100;
```

Step 5: Display customers with the highest payments first

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

Observation: Views can use aggregate functions such as SUM(), COUNT(), and AVG() with GROUP BY.

11. Practical Task 8: Create a View Using CASE

Objective: Classify films according to their rental rates.

Step 1: Create the View

```
CREATE VIEW film_rental_categories AS
SELECT
    film_id,
```

```
    title,
    rental_rate,
    CASE
        WHEN rental_rate <= 1 THEN 'Low'
        WHEN rental_rate <= 3 THEN 'Medium'
        ELSE 'High'
    END AS rental_category
FROM film;
```

Step 2: Retrieve records

```
SELECT *
FROM film_rental_categories;
```

Step 3: Display high-rental-rate films

```
SELECT *
FROM film_rental_categories
WHERE rental_category = 'High';
```

Observation: A View can contain calculated columns and conditional expressions.

12. Practical Task 9: Modify an Existing View

Objective: Add the film rating column to the existing `film_details` View.

Use `CREATE OR REPLACE VIEW`.

```
CREATE OR REPLACE VIEW film_details AS
SELECT
    film_id,
    title,
    release_year,
    rental_rate,
    length,
    rating
FROM film;
```

Retrieve the updated View:

```
SELECT *
FROM film_details;
```

Observation: The View now includes the additional `rating` column.

Note: When replacing a View, existing output columns must remain compatible in name, order, and data type. New columns can be added at the end.

13. Practical Task 10: Rename a View

Objective: Rename an existing View.

```
ALTER VIEW film_details
RENAME TO movie_details;
```

Retrieve records using the new name:

```
SELECT *
FROM movie_details;
```

14. Practical Task 11: Delete a View

Objective: Remove a View from the database.

```
DROP VIEW movie_details;
```

To avoid an error if the View does not exist:

```
DROP VIEW IF EXISTS movie_details;
```

Observation: Deleting a View does not delete the original table records.

15. Practical Task 12: Explore Existing Views

Objective: Display the Views created in the database.

```
SELECT
    table_schema,
    table_name
FROM information_schema.views
WHERE table_schema = 'public'
ORDER BY table_name;
```

To inspect a View definition:

```
SELECT
    table_name,
    view_definition
FROM information_schema.views
WHERE table_schema = 'public'
AND table_name = 'film_categories';
```

Part B: Materialized Views in PostgreSQL

16. Introduction to Materialized Views

A Materialized View stores the result of an SQL query physically in the database.

Unlike a normal View, which executes its underlying query when accessed, a Materialized View normally retrieves previously stored results.

Example:

A store manager frequently needs a report showing the total payments made by customers.

We can create a Materialized View to store the calculated results and retrieve them without repeating the original aggregation query every time.

Difference Between View and Materialized View

Feature	View	Materialized View
Stores query definition	Yes	Yes
Stores query results physically	No	Yes
Automatically reflects source table changes	Yes	No
Requires refresh	No	Yes
Uses additional storage for results	No	Yes

17. Practical Task 13: Create a Materialized View

Objective: Create a Materialized View showing the total payment made by each customer.

Step 1: Execute the original query

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

Step 2: Create 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 AS c
INNER JOIN payment AS p
    ON c.customer_id = p.customer_id
GROUP BY
    c.customer_id,
    c.first_name,
    c.last_name;
```

Step 3: Retrieve stored results

```
SELECT *
FROM mv_customer_payment_summary;
```

Step 4: Sort the results

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

Step 5: Apply a condition

```
SELECT *
FROM mv_customer_payment_summary
WHERE total_payment > 100;
```

Observation: The Materialized View stores the result of the original query.

18. Practical Task 14: Refresh a Materialized View

Objective: Update stored results after changes in the source tables.

Step 1: Check the current summary

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

Step 2: Insert a payment record

For this demonstration, insert a test payment.

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

Note: Use this test record only in a practice database. It is not a real rental payment.

Step 3: Retrieve the Materialized View again

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

Observe that the newly inserted payment is not yet included in the stored summary.

Step 4: Refresh the Materialized View

```
REFRESH MATERIALIZED VIEW mv_customer_payment_summary;
```

Step 5: Retrieve the results again

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

Observe that the updated payment is now included.

Learning point: Materialized Views do not automatically reflect changes in their source tables. We use `REFRESH MATERIALIZED VIEW` to update stored results.

19. Practical Task 15: Create a Materialized View Using WITH NO DATA

Objective: Create a Materialized View without immediately populating it.

```
CREATE MATERIALIZED VIEW mv_customer_payments AS
SELECT
    customer_id,
```

```
    SUM(amount) AS total_payment
FROM payment
GROUP BY customer_id
WITH NO DATA;
```

Try retrieving records:

```
SELECT *
FROM mv_customer_payments;
```

The Materialized View is not populated yet, so PostgreSQL cannot return its data.

Now refresh it:

```
REFRESH MATERIALIZED VIEW mv_customer_payments;
```

Retrieve the results:

```
SELECT *
FROM mv_customer_payments;
```

Learning point:

- `WITH DATA` is the default and populates the Materialized View during creation.
- `WITH NO DATA` creates it without storing query results.
- A Materialized View created with `WITH NO DATA` must be refreshed before it can be queried.

20. Practical Task 16: Create a Materialized View for Film Categories

Objective: Calculate the number of films and average rental rate for each category.

Step 1: Execute the original query

```
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 AS c
INNER JOIN film_category AS fc
    ON c.category_id = fc.category_id
INNER JOIN film AS f
    ON fc.film_id = f.film_id
GROUP BY
    c.category_id,
    c.name;
```

Step 2: Create the Materialized View

```
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 AS c
INNER JOIN film_category AS fc
    ON c.category_id = fc.category_id
INNER JOIN film AS f
    ON fc.film_id = f.film_id
GROUP BY
    c.category_id,
    c.name;
```

Step 3: Retrieve the results

```
SELECT *
FROM mv_category_film_summary;
```

Step 4: Sort categories by the number of films

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

Step 5: Refresh the Materialized View

```
REFRESH MATERIALIZED VIEW mv_category_film_summary;
```

21. Practical Task 17: Create an Index on a Materialized View

Objective: Create a unique index to improve retrieval performance and enable concurrent refresh.

Create a unique index on `customer_id`:

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

The `customer_id` column uniquely identifies each row in this summary.

Now retrieve records using a condition:

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

Learning point: Indexes can be created on Materialized Views just as they can be created on tables.

22. Practical Task 18: Refresh a Materialized View Concurrently

Objective: Refresh a Materialized View while allowing other sessions to read its existing data.

Execute:

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

Important requirements:

1. The Materialized View must already be populated.
2. It must have a qualifying unique index.
3. The unique index must use ordinary columns and cover all rows.
4. `CONCURRENTLY` cannot be used with `WITH NO DATA`.

Learning point: Concurrent refresh helps avoid blocking other sessions from reading the Materialized View during refresh.

23. Practical Task 19: Rename a Materialized View

Objective: Rename an existing Materialized View.

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

Retrieve records using the new name:

```
SELECT *
FROM mv_customer_payments_report;
```

24. Practical Task 20: Delete a Materialized View

Objective: Delete an existing Materialized View.

```
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;
```

Delete the category summary if it is no longer required:

```
DROP MATERIALIZED VIEW IF EXISTS mv_category_film_summary;
```

Observation: Dropping a Materialized View does not delete the original records from its source tables.

25. Practical Task 21: Explore Existing Materialized Views

Objective: Display Materialized Views in the current database.

PostgreSQL provides the `pg_matviews` system view.

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

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

Part C: Independent Lab Exercises

26. Exercises: Views

Complete the following exercises independently.

Beginner Level

1. Create a View named `film_basic_info` displaying film ID, title, and release year.
2. Create a View named `customer_contact_info` displaying customer ID, first name, last name, and email.
3. Create a View named `actor_names` displaying actor names using CONCAT().
4. Create a View named `expensive_films` displaying films with a rental rate greater than 3.

Intermediate Level

5. Create a View named `category_film_details` displaying film titles and their categories.
6. Create a View named `customer_rental_history` displaying customer names, film titles, and rental dates.
7. Create a View named `payment_details` displaying customer names and payment amounts.
8. Create a View named `long_films` displaying films with a length greater than 120 minutes.

Advanced Level

9. Create a View displaying the total number of rentals for each customer.
10. Create a View displaying the total payment collected for each customer.
11. Create a View displaying the number of films in each category.
12. Create a View displaying the average rental rate for each category.
13. Create a View combining film, category, and rental information.
14. Modify an existing View to include an additional column.

27. Exercises: Materialized Views

Beginner Level

1. Create a Materialized View named `mv_film_details` displaying film ID, title, rental rate, and length.
2. Create a Materialized View named `mv_customer_details` displaying customer ID, first name, last name, and email.
3. Create a Materialized View named `mv_actor_details` displaying actor names using CONCAT().
4. Retrieve all records from each Materialized View.

Intermediate Level

5. Create a Materialized View named `mv_customer_payment_summary` displaying total payments for each customer.
6. Create a Materialized View named `mv_category_summary` displaying the number of films in each category.
7. Create a Materialized View displaying the total number of rentals for each customer.
8. Create a Materialized View displaying average rental rates for each category.
9. Create a Materialized View using multiple JOIN operations.
10. Create a Materialized View using `WITH NO DATA` and populate it using REFRESH.

Advanced Level

11. Insert a test payment record and observe whether the Materialized View updates automatically.
12. Refresh the Materialized View and verify the updated results.
13. Create a unique index on a Materialized View.
14. Perform a concurrent refresh using the unique index.
15. Rename an existing Materialized View.
16. Delete a Materialized View using `DROP MATERIALIZED VIEW`.
17. Display all Materialized Views using `pg_matviews`.

Part D: Challenge Tasks

28. Combined Challenges

Complete these tasks without copying the earlier examples.

1. Create a View displaying each category name and the total number of films in that category.
2. Create a View displaying each customer's name and total number of rentals.

3. Create a View displaying each film title, category, and rental rate.
4. Create a View displaying customers whose total payments exceed the average customer payment.
5. Create a View that classifies customers into payment groups using CASE:
    - High Payment
    - Medium Payment
    - Low Payment
6. Create a Materialized View displaying total payments by customer and staff member.
7. Create a Materialized View displaying the number of rentals by film category.
8. Create a Materialized View displaying total revenue by month.
9. Create a unique index on a suitable Materialized View.
10. Demonstrate the difference between a normal View and a Materialized View after inserting a new payment record.

29. Lab Submission Requirements

Submit a practical lab report containing:

1. Lab title and student information.
2. Database name used.
3. SQL queries executed for each practical task.
4. Screenshots of query execution and output.
5. Solutions to the assigned lab exercises.
6. Solutions to the challenge tasks.
7. A comparison of Views and Materialized Views based on practical observations.
8. Evidence of refreshing a Materialized View after inserting a test record.

Important: Screenshots must clearly show the SQL query and its execution result in pgAdmin or another PostgreSQL client.

30. Lab Assessment Checklist

Assessment Item	Marks
Create and retrieve basic Views	10
Apply WHERE and ORDER BY	5
Create Views using JOIN	10
Create Views using SQL functions and aggregate functions	10
Modify, rename, and delete Views	5
Create and retrieve Materialized Views	10
Refresh Materialized Views	10
Use WITH NO DATA	5
Create indexes and perform concurrent refresh	10

Assessment Item	Marks
Complete independent exercises	10
Challenge tasks and lab report	15
Total	**100**

31. Quick Revision

Task	SQL Command
Create View	`CREATE VIEW`
Retrieve View	`SELECT`
Modify View	`CREATE OR REPLACE VIEW`
Rename View	`ALTER VIEW ... RENAME TO`
Delete View	`DROP VIEW`
Create Materialized View	`CREATE MATERIALIZED VIEW`
Create without data	`WITH NO DATA`
Retrieve Materialized View	`SELECT`
Refresh Materialized View	`REFRESH MATERIALIZED VIEW`
Concurrent refresh	`REFRESH MATERIALIZED VIEW CONCURRENTLY`
Create index	`CREATE UNIQUE INDEX`
Rename Materialized View	`ALTER MATERIALIZED VIEW ... RENAME TO`
Delete Materialized View	`DROP MATERIALIZED VIEW`
List Materialized Views	`SELECT ... FROM pg_matviews`