University Institute of Engineering

Department of Computer Science & Engineering

EXPERIMENT: 5

NAME: AAYUSH AMAN **UID: 23BCS14046**

BRANCH: BE-CSE **SECTION / GROUP: KRG_3B**

SEMESTER: 5TH **SUBJECT CODE: 23CSP-339**

SUBJECT NAME: ADBMS

1. Aim of the practical:

Performance Benchmarking: Normal View vs. Materialized View [MEDIUM]

- Create a large dataset:
  - Create a table names transaction_data (id , value) with 1 million records.
  - take id 1 and 2, and for each id, generate 1 million records in value column
  - Use Generate_series () and random() to populate the data.
- Create a normal view and materialized view to for sales_summary, which includes total_quantity_sold, total_sales, and total_orders with aggregation.
- Compare the performance and execution time of both.

Views: Securing Data Access with Views and Role-Based Permissions [HARD]

The company TechMart Solutions stores all sales transactions in a central database.
A new reporting team has been formed to analyze sales but they should not have direct access to the base tables for security reasons.
The database administrator has decided to:

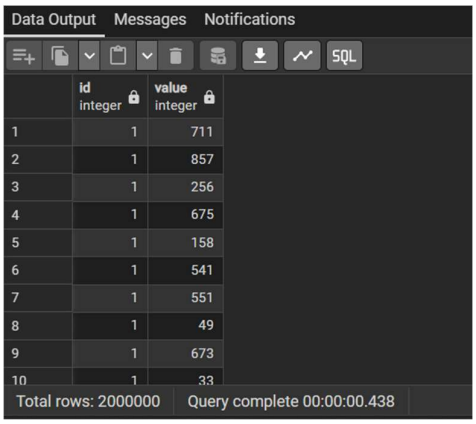
1. Create restricted views to display only summarized, non-sensitive data.
2. Assign access to these views to specific users using DCL commands (GRANT, REVOKE).

2. Tools Used: PgAdmin, PostgresSQL

3. Code:

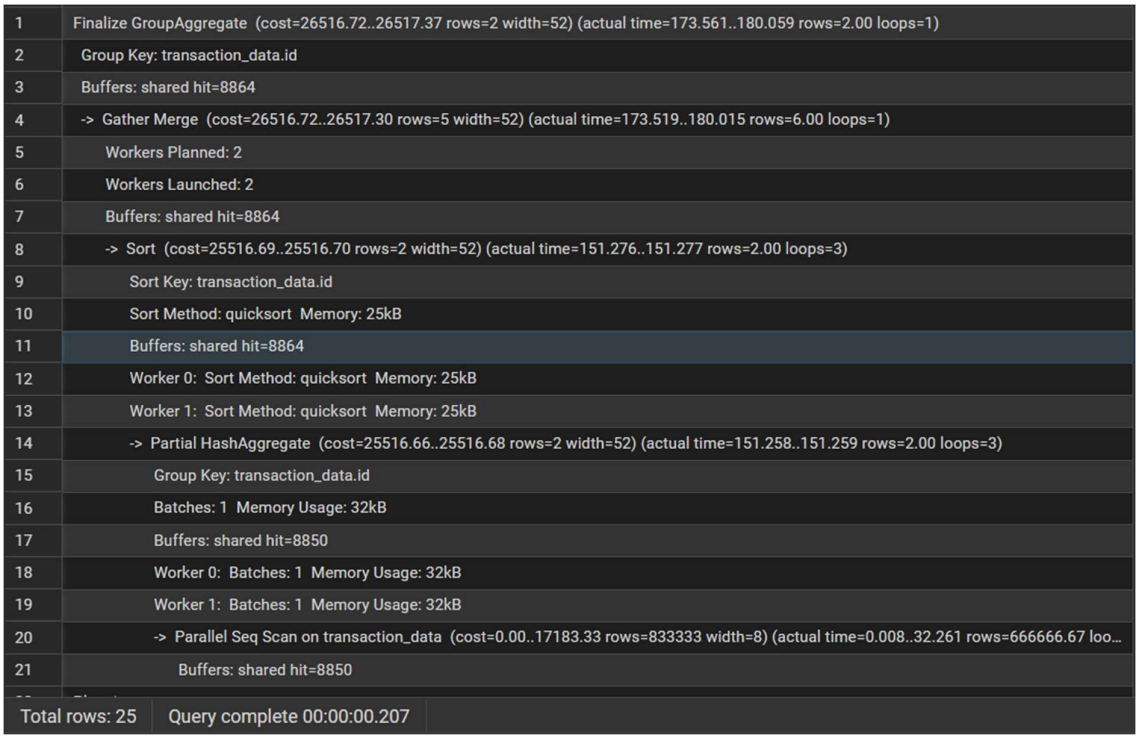
```
-- Database: krg_3b
-- Medium
CREATE TABLE transaction_data (
   id INT,
   value INT
);
```

```
-- For id = 1
INSERT INTO transaction_data (id, value)
SELECT 1, random() * 1000
FROM generate_series(1, 1000000);

-- For id = 2
INSERT INTO transaction_data (id, value)
SELECT 2, random() * 1000
FROM generate_series(1, 1000000);

SELECT *FROM transaction_data
```



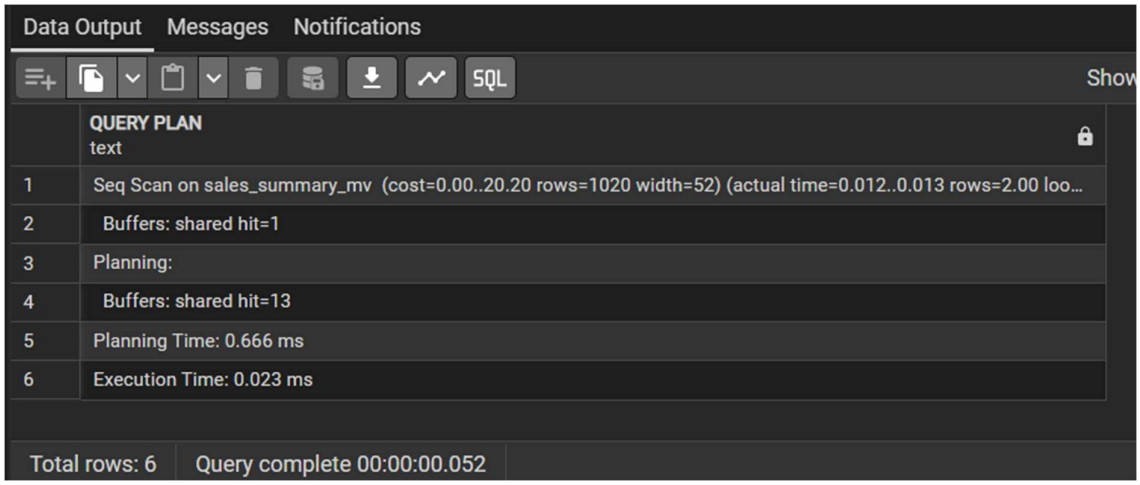
Output: Table created and populated with data

```
--WITH NORMAL VIEW
CREATE OR REPLACE VIEW sales_summary_view AS
SELECT
   id,
   COUNT(*) AS total_orders,
   SUM(value) AS total_sales,
   AVG(value) AS avg_transaction
FROM transaction_data
GROUP BY id;


EXPLAIN ANALYZE
SELECT * FROM sales_summary_view;
```



Output: Result with Normal View

```
--WITH MATERIALIZED VIEW
CREATE MATERIALIZED VIEW sales_summary_mv AS
SELECT
    id,
    COUNT(*) AS total_orders,
    SUM(value) AS total_sales,
    AVG(value) AS avg_transaction
FROM transaction_data
GROUP BY id;



EXPLAIN ANALYZE
SELECT * FROM sales_summary_mv;
```



Output: Results with Materialized View

-- Hard

```
CREATE VIEW vW_ORDER_SUMMARY

AS

SELECT

    O.order_id,

    O.order_date,

    P.product_name,

    C.full_name,

    (P.unit_price * O.quantity) - ((P.unit_price * O.quantity) * O.discount_percent / 100) AS
final_cost

FROM customer_master AS C

JOIN sales_orders AS O

    ON O.customer_id = C.customer_id

JOIN product_catalog AS P

    ON P.product_id = O.product_id;
```

```
SELECT * FROM vW_ORDER_SUMMARY;


--1. CREATE USER

CREATE ROLE Aman

LOGIN

PASSWORD 'Aman123';

GRANT SELECT ON vW_ORDER_SUMMARY TO Aman;

REVOKE SELECT ON vW_ORDER_SUMMARY FROM Aman;
```

4. Learning Outcomes:

1. Materialized views improve query performance by storing precomputed results, especially useful for large datasets.
2. Normal views always reflect live data, while materialized views require manual refresh to stay updated.
3. EXPLAIN ANALYZE helps compare execution efficiency between views and materialized views.
4. Materialized views are ideal for reporting and dashboards where speed matters more than real-time accuracy.
5. PostgreSQL functions like generate_series() and random() are effective for simulating large-scale test data.