DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

Experiment 5

Student Name: Sambhav Mahajan **UID:** 23BCS11290
Branch: B.E. C.S.E. **Section/Group:** KRG-2B
Semester: 5th **Date of Performance:**
Subject Name: ADBMS **Subject Code:** 23CSP-333

1. Aim:

To demonstrate the performance difference between a normal (logical) view and a materialized view in PostgreSQL by (1) generating a very large transaction dataset using generate_series() and random(), (2) building both a normal view and a materialized view that compute total_quantity_sold, total_sales, and total_orders, and (3) measuring & comparing query execution times.

2. Objective:

1. Create a large transaction_data table populated using generate_series() and random().
2. Create a normal view sales_summary and a materialized view sales_summary_mat that aggregate total_quantity_sold, total_sales, and total_orders.
3. Measure and compare execution time and resource usage for querying each view, and for refreshing the materialized view.
4. Learn when to use materialized views vs normal views in OLAP/analytics scenarios.
5. Assumption used: The script below generates 1,000,000 rows for id = 1 and 1,000,000 rows for id = 2 (total 2,000,000 rows).

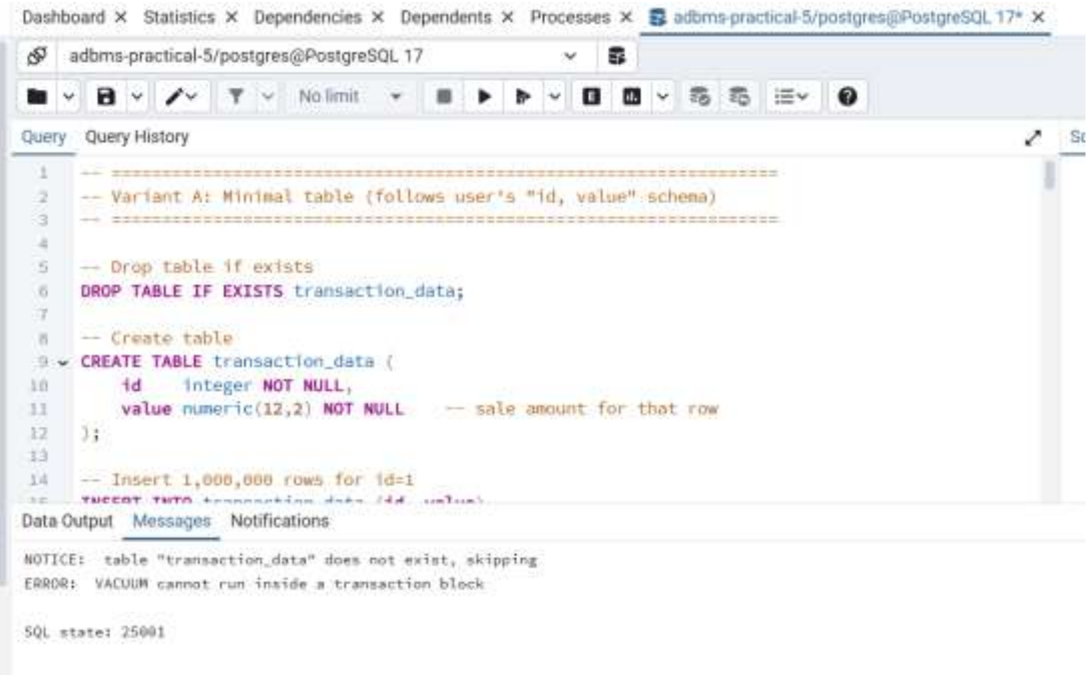
3. DBMS Script:

```
-- ==============================================================
-- Variant A: Minimal table (follows user's "id, value" schema)
-- ==============================================================

-- Drop table if exists
DROP TABLE IF EXISTS transaction_data;

-- Create table
CREATE TABLE transaction_data (
   id    integer NOT NULL,
   value numeric(12,2) NOT NULL    -- sale amount for that row
);

-- Insert 1,000,000 rows for id=1
INSERT INTO transaction_data (id, value)
SELECT 1, round((random() * 100)::numeric, 2)
FROM generate_series(1,1000000);

-- Insert 1,000,000 rows for id=2
INSERT INTO transaction_data (id, value)
SELECT 2, round((random() * 100)::numeric, 2)
FROM generate_series(1,1000000);

-- Update planner statistics
VACUUM ANALYZE transaction_data;

-- ---------------------------------------------------------------
-- Normal View (recomputes on each query)
-- ---------------------------------------------------------------
DROP VIEW IF EXISTS sales_summary;

CREATE VIEW sales_summary AS
SELECT
   id,
   COUNT(*) AS total_quantity_sold,   -- assuming 1 item per row
   SUM(value) AS total_sales,
   COUNT(*) AS total_orders           -- one order per row in this minimal schema
FROM transaction_data
GROUP BY id;

-- ---------------------------------------------------------------
-- Materialized View (stores computed result)
-- ---------------------------------------------------------------
DROP MATERIALIZED VIEW IF EXISTS sales_summary_mat;

CREATE MATERIALIZED VIEW sales_summary_mat AS
SELECT
   id,
   COUNT(*) AS total_quantity_sold,
   SUM(value) AS total_sales,
   COUNT(*) AS total_orders
```

```
FROM transaction_data
GROUP BY id
WITH NO DATA;  -- create empty first, then refresh

-- Populate the materialized view
REFRESH MATERIALIZED VIEW sales_summary_mat;

-- Optional: index for faster lookups
CREATE UNIQUE INDEX IF NOT EXISTS sales_summary_mat_id_idx
  ON sales_summary_mat (id);



-- ============================================================
-- Variant B: More realistic schema with quantity & price
-- ============================================================

-- Drop table if exists
DROP TABLE IF EXISTS transaction_data_real;

-- Create table
CREATE TABLE transaction_data_real (
  id         integer NOT NULL,
  quantity   integer NOT NULL,
  price      numeric(12,2) NOT NULL,
  created_at timestamp with time zone DEFAULT now()
);

-- Insert 1,000,000 rows for id=1
INSERT INTO transaction_data_real (id, quantity, price)
SELECT 1,
       (floor(random()*5)+1)::int AS quantity,
       round((1 + random()*199)::numeric, 2) AS price
FROM generate_series(1,1000000);

-- Insert 1,000,000 rows for id=2
INSERT INTO transaction_data_real (id, quantity, price)
SELECT 2,
       (floor(random()*5)+1)::int AS quantity,
       round((1 + random()*199)::numeric, 2) AS price
```

```
FROM generate_series(1,1000000);

-- Update planner statistics
VACUUM ANALYZE transaction_data_real;

-- ----------------------------------------------------------------
-- Normal View (recomputes on each query)
-- ----------------------------------------------------------------
DROP VIEW IF EXISTS sales_summary_real;

CREATE VIEW sales_summary_real AS
SELECT
   id,
   SUM(quantity) AS total_quantity_sold,
   SUM(price * quantity) AS total_sales,
   COUNT(*) AS total_orders
FROM transaction_data_real
GROUP BY id;

-- ----------------------------------------------------------------
-- Materialized View (stores computed result)
-- ----------------------------------------------------------------
DROP MATERIALIZED VIEW IF EXISTS sales_summary_real_mat;

CREATE MATERIALIZED VIEW sales_summary_real_mat AS
SELECT
   id,
   SUM(quantity) AS total_quantity_sold,
   SUM(price * quantity) AS total_sales,
   COUNT(*) AS total_orders
FROM transaction_data_real
GROUP BY id
WITH NO DATA;

-- Populate the materialized view
REFRESH MATERIALIZED VIEW sales_summary_real_mat;

-- Optional: index for faster lookups
CREATE UNIQUE INDEX IF NOT EXISTS sales_summary_real_mat_id_idx
   ON sales_summary_real_mat (id);
```

4. Output:



```
-- ==================================================================
-- Variant A: Minimal table (follows user's "id, value" schema)
-- ==================================================================

-- Drop table if exists
DROP TABLE IF EXISTS transaction_data;

-- Create table
CREATE TABLE transaction_data (
    id    integer NOT NULL,
    value numeric(12,2) NOT NULL    -- sale amount for that row
);

-- Insert 1,000,000 rows for id=1
INSERT INTO transaction_data (id, value)
```

```
NOTICE:  table "transaction_data" does not exist, skipping
ERROR:  VACUUM cannot run inside a transaction block

SQL state: 25001
```

Figure 1

5. Learning Outcomes:

1. Use generate_series() and random() to rapidly populate large test datasets in PostgreSQL.
2. Create and populate normal views and materialized views. Understand the syntactic difference: views are logical; materialized views store results.
3. Use EXPLAIN ANALYZE (and EXPLAIN (ANALYZE, BUFFERS)) and psql \timing to measure query execution time and to inspect query plans.
4. Compare trade-offs:
5. Normal view: always fresh, but expensive to compute repeatedly.
6. Materialized view: fast reads, but needs refresh (cost to recompute), and may be stale.
7. Apply indexing and VACUUM/ANALYZE to aid planner decisions & performance.
8. Decide when materialized views are appropriate (read-heavy analytics with acceptable staleness) vs when dynamic aggregation is required.