Day8\_PavithraMani\_SDET177

-- 1. Create view vw\_updatable\_products (use same query whatever I used in the training)

-- Try updating view with below query and see if the product table also gets updated.

-- Update query:

-- UPDATE updatable\_products SET unit\_price = unit\_price \* 1.1 WHERE units\_in\_stock < 10;

--create view

CREATE OR REPLACE VIEW VW\_UPDATABLE\_PRODUCTS AS

SELECT

PRODUCT\_ID,

PRODUCT\_NAME,

UNIT\_PRICE,

UNITS\_IN\_STOCK

FROM

PRODUCTS

WHERE

DISCONTINUED = 0;

--update

UPDATE VW\_UPDATABLE\_PRODUCTS

SET

UNIT\_PRICE = UNIT\_PRICE \* 1.1

WHERE

UNITS\_IN\_STOCK < 10;

--verify

SELECT

PRODUCT\_ID,

PRODUCT\_NAME,

UNIT\_PRICE,

UNITS\_IN\_STOCK

FROM

PRODUCTS

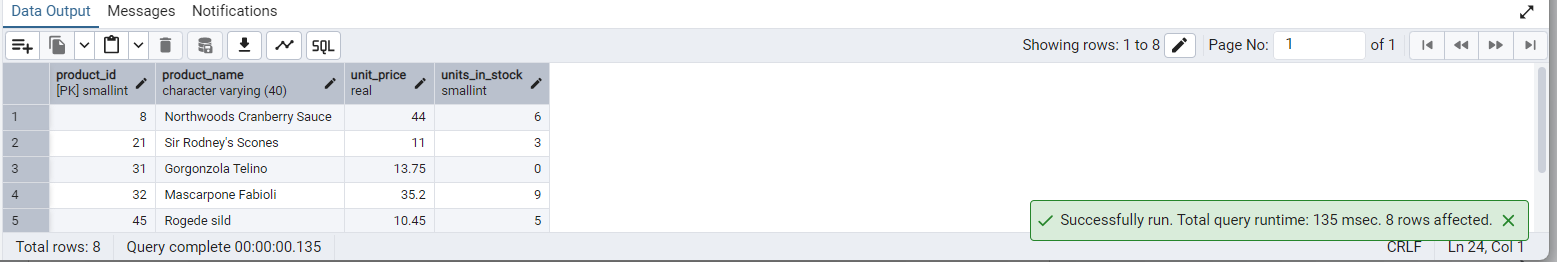
WHERE

UNITS\_IN\_STOCK < 10

AND DISCONTINUED = 0;

### 

### OUTPUT



-- 2. Transaction:

-- Update the product price for products by 10% in category id=1

-- Try COMMIT and ROLLBACK and observe what happens.

-- Transaction 1: COMMIT

BEGIN;

-- Update prices by 10% where category\_id = 1

UPDATE PRODUCTS

SET

UNIT\_PRICE = UNIT\_PRICE \* 1.1

WHERE

CATEGORY\_ID = 1;

-- View changes (this will show updated prices)

SELECT

PRODUCT\_ID,

PRODUCT\_NAME,

UNIT\_PRICE

FROM

PRODUCTS

WHERE

CATEGORY\_ID = 1;

-- Commit the changes

COMMIT;

-- Check again to confirm changes are saved

SELECT

PRODUCT\_ID,

PRODUCT\_NAME,

UNIT\_PRICE

FROM

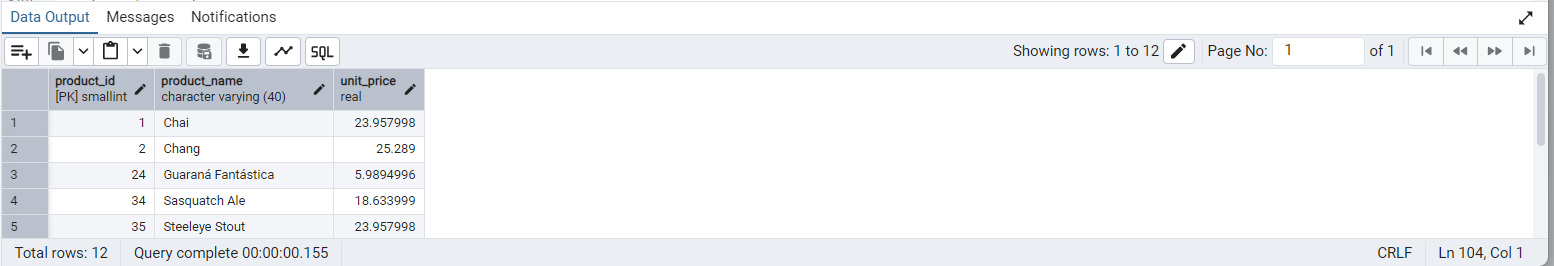
PRODUCTS

WHERE

CATEGORY\_ID = 1;

This output is after updating and commiting

### OUTPUT



-- Transaction 2: ROLLBACK

BEGIN;

-- Update again by another 10%

UPDATE PRODUCTS

SET

UNIT\_PRICE = UNIT\_PRICE \* 1.1

WHERE

CATEGORY\_ID = 1;

-- View changes after second update

SELECT

PRODUCT\_ID,

PRODUCT\_NAME,

UNIT\_PRICE

FROM

PRODUCTS

WHERE

CATEGORY\_ID = 1;

-- Rollback the second update

ROLLBACK;

-- Final check: should show committed values only (first 10% increase)

SELECT

PRODUCT\_ID,

PRODUCT\_NAME,

UNIT\_PRICE

FROM

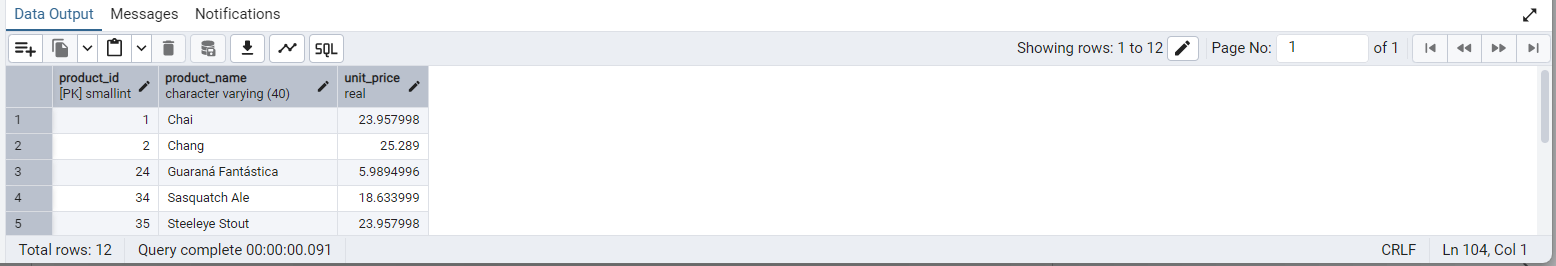
PRODUCTS

WHERE

CATEGORY\_ID = 1;

This output is after again updating and rollback

### OUTPUT



-- 3. Create a regular view which will have below details (Need to do joins):

-- Employee\_id,

-- Employee\_full\_name,

-- Title,

-- Territory\_id,

-- territory\_description,

-- region\_description

CREATE OR REPLACE VIEW VW\_EMPLOYEE\_TERRITORIES AS

SELECT

E.EMPLOYEE\_ID,

E.FIRST\_NAME || ' ' || E.LAST\_NAME AS EMPLOYEE\_FULL\_NAME,

E.TITLE,

T.TERRITORY\_ID,

T.TERRITORY\_DESCRIPTION,

R.REGION\_DESCRIPTION

FROM

EMPLOYEES E

JOIN EMPLOYEE\_TERRITORIES ET ON E.EMPLOYEE\_ID = ET.EMPLOYEE\_ID

JOIN TERRITORIES T ON ET.TERRITORY\_ID = T.TERRITORY\_ID

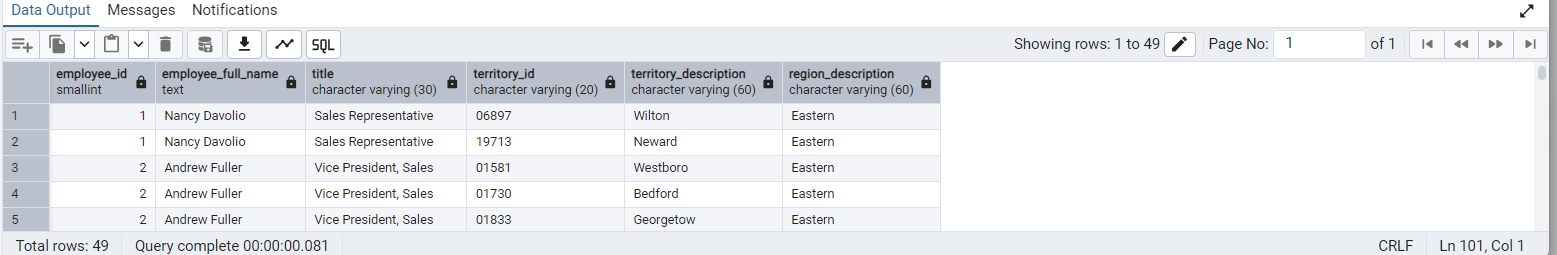
JOIN REGION R ON T.REGION\_ID = R.REGION\_ID;

### 

### 

### 

### OUTPUT



-- 4. Create a recursive CTE based on Employee Hierarchy

WITH RECURSIVE

EMPLOYEE\_HIERARCHY AS (

-- Anchor: Top-level employee(s) with no manager

SELECT

EMPLOYEE\_ID,

FIRST\_NAME || ' ' || LAST\_NAME AS FULL\_NAME,

TITLE,

REPORTS\_TO,

1 AS LEVEL

FROM

EMPLOYEES

WHERE

REPORTS\_TO IS NULL

UNION ALL

-- Recursive: employees who report to others

SELECT

E.EMPLOYEE\_ID,

E.FIRST\_NAME || ' ' || E.LAST\_NAME AS FULL\_NAME,

E.TITLE,

E.REPORTS\_TO,

EH.LEVEL + 1 AS LEVEL

FROM

EMPLOYEES E

INNER JOIN EMPLOYEE\_HIERARCHY EH ON E.REPORTS\_TO = EH.EMPLOYEE\_ID

)

SELECT

\*

FROM

EMPLOYEE\_HIERARCHY

ORDER BY

LEVEL,

REPORTS\_TO,

EMPLOYEE\_ID;

### OUTPUT

