# Lecture 3:

# Activity (Moderate) – Learner – With Solution

## Context

You’re working on a simple e-commerce database. It has four tables:

- Customers: customer basic info

- Products: product catalog with prices

- Orders: each order header

- OrderItems: line-items for each order

Following are the description of the tables:

1. Customers  
- customer\_id (Primary Key)  
- name (VARCHAR(100), NOT NULL)  
- email (VARCHAR(100), UNIQUE, NOT NULL)  
- phone (VARCHAR(15))  
  
2. Products  
- product\_id (INT, Primary Key)  
- name (VARCHAR(100), NOT NULL)  
- description (TEXT)  
- price (DECIMAL(10,2), NOT NULL)  
- stock\_quantity (INT, NOT NULL)  
  
3. Orders  
- order\_id (INT, Primary Key)  
- customer\_id (INT, Foreign Key referencing Customers(customer\_id))  
- order\_date (DATE, NOT NULL)  
- total\_amount (DECIMAL(10,2))  
  
4. OrderItems  
- order\_id (INT, Foreign Key referencing Orders(order\_id))  
- product\_id (INT, Foreign Key referencing Products(product\_id))  
- quantity (INT, NOT NULL)  
- price\_each (DECIMAL(10,2), NOT NULL)  
- PRIMARY KEY (order\_id, product\_id)

## Task

1. Schema Creation (DDL):  
 - Create tables Customers, Products, Orders, OrderItems with appropriate primary keys and foreign keys.

2. Data Manipulation (DML):  
 - Insert at least two sample rows into each table.  
 - Update the price of one product.  
 - Delete any order that has no items.

3. Transaction Control (TCL):  
 - In a single transaction, insert a new order with two items; then roll back.

4. Privilege Control (DCL):  
 - Grant SELECT on Products to a role called analyst\_role.

5. Queries:  
 a) Inner Join: List each order with its customer name and total items.  
 b) Left Join: List all customers and show order count (including zero).  
 c) Full Join: Show all products and any order in which they appear (include products never ordered and order-items for missing products).  
 d) Using your existing database schema (Products, OrderItem) write a query that:

* Calculates the total quantity sold for each product.
* Then returns only those products whose total quantity sold exceeds the average total quantity sold across all products.

6. Views, Indexes, Stored Procedures & Functions:  
 - Create a view MonthlySales showing year, month, and total\_revenue.  
 - Create an index on Orders(order\_date).  
 - Write a stored procedure AddOrder(customerId INT, p1 INT, q1 INT, p2 INT, q2 INT) that inserts one order and two items.  
 - Write a scalar function OrderTotal(oId INT) returning the total amount for order oId.

7. Triggers: Create an AFTER DELETE trigger on Orders that logs deleted orders into an archive table OrderArchive(order\_id, customer\_id, order\_date, deleted\_at).

8. Window Functions: Write a query to list each product with its total\_sales and a rank (descending) over total\_sales within its department.

9. JSON Handling: Create a table ProductDetails(product\_id INT PRIMARY KEY, specs JSON). Insert a sample JSON spec and write a query to extract a specific attribute (e.g., specs->>'weight').

10. Transaction Savepoints: In a single transaction, insert two new products; set a SAVEPOINT after the first insert, then attempt a second insert with a duplicate key to force an error, and ROLLBACK TO SAVEPOINT to undo only the second insert.

11. Aggregate Query:  
- Calculate total sales per month.  
  
12. Complex Query:  
- Retrieve names and emails of customers who purchased products priced above $30.  
  
13. Data Validation:  
- Implement CHECK constraint ensuring stock\_quantity is never negative.  
  
14. Backup and Restore:  
- Briefly describe the database backup and restoration process.

## Solution

-- 1. SCHEMA CREATION (DDL) ------------------------  
CREATE TABLE Customers (  
 customer\_id INT PRIMARY KEY,  
 name VARCHAR(100) NOT NULL,  
 email VARCHAR(100) UNIQUE NOT NULL  
);  
CREATE TABLE Products (  
 product\_id INT PRIMARY KEY,  
 name VARCHAR(100) NOT NULL,  
 price DECIMAL(10,2) NOT NULL  
);  
CREATE TABLE Orders (  
 order\_id INT PRIMARY KEY,  
 customer\_id INT NOT NULL,  
 order\_date DATE NOT NULL,  
 FOREIGN KEY (customer\_id) REFERENCES Customers(customer\_id)  
);  
CREATE TABLE OrderItems (  
 order\_id INT NOT NULL,  
 product\_id INT NOT NULL,  
 quantity INT NOT NULL,  
 price\_each DECIMAL(10,2) NOT NULL,  
 PRIMARY KEY (order\_id, product\_id),  
 FOREIGN KEY (order\_id) REFERENCES Orders(order\_id),  
 FOREIGN KEY (product\_id) REFERENCES Products(product\_id)  
);  
  
-- 2. DATA MANIPULATION (DML) ----------------------  
INSERT INTO Customers VALUES (1, 'Alice Smith', 'alice@example.com'), (2, 'Bob Jones', 'bob@example.com');  
INSERT INTO Products VALUES (10, 'Keyboard', 29.99), (20, 'Webcam', 49.99);  
INSERT INTO Orders VALUES (100, 1, '2025-05-01'), (101, 2, '2025-05-02');  
INSERT INTO OrderItems VALUES (100, 10, 1, 29.99), (100, 20, 2, 49.99), (101, 10, 3, 29.99);  
UPDATE Products SET price = 34.99 WHERE product\_id = 10;  
DELETE FROM Orders o WHERE NOT EXISTS (SELECT 1 FROM OrderItems oi WHERE oi.order\_id = o.order\_id);  
  
-- 3. TRANSACTION CONTROL (TCL) --------------------  
BEGIN TRANSACTION;  
INSERT INTO Orders VALUES (200, 1, '2025-05-10');  
INSERT INTO OrderItems VALUES (200, 10, 1, 34.99), (200, 20, 1, 49.99);  
ROLLBACK;  
  
-- 4. PRIVILEGE CONTROL (DCL) ----------------------  
GRANT SELECT ON Products TO analyst\_role;  
  
-- 5. QUERIES ---------------------------------------  
-- a) Inner Join  
SELECT o.order\_id, c.name AS customer\_name, SUM(oi.quantity) AS total\_items  
FROM Orders o JOIN Customers c ON o.customer\_id = c.customer\_id JOIN OrderItems oi ON o.order\_id = oi.order\_id  
GROUP BY o.order\_id, c.name;  
-- b) Left Join  
SELECT c.customer\_id, c.name, COUNT(o.order\_id) AS num\_orders  
FROM Customers c LEFT JOIN Orders o ON c.customer\_id = o.customer\_id GROUP BY c.customer\_id, c.name;  
-- c) Full Join  
SELECT p.product\_id, p.name AS product\_name, oi.order\_id, oi.quantity  
FROM Products p FULL JOIN OrderItems oi ON p.product\_id = oi.product\_id;  
-- d) -- Using a CTE to calculate total quantity sold for each product

WITH ProductSales AS (SELECT product\_id, SUM(quantity) AS total\_quantity\_sold

FROM OrderItems

GROUP BY product\_id

)

SELECT

p.product\_id,

p.name,

ps.total\_quantity\_sold

FROM

Products p

JOIN

ProductSales ps ON p.product\_id = ps.product\_id

WHERE

ps.total\_quantity\_sold > (

SELECT AVG(total\_quantity\_sold) FROM ProductSales

)

ORDER BY

ps.total\_quantity\_sold DESC;  
-- 6. VIEWS, INDEXES, PROCS & FUNCTIONS ------------  
CREATE VIEW MonthlySales AS SELECT EXTRACT(YEAR FROM o.order\_date) AS sales\_year, EXTRACT(MONTH FROM o.order\_date) AS sales\_month, SUM(oi.quantity \* oi.price\_each) AS total\_revenue FROM Orders o JOIN OrderItems oi ON o.order\_id = oi.order\_id GROUP BY EXTRACT(YEAR FROM o.order\_date), EXTRACT(MONTH FROM o.order\_date);  
CREATE INDEX idx\_orders\_date ON Orders(order\_date);  
CREATE PROCEDURE AddOrder (IN custId INT, IN p1 INT, IN q1 INT, IN p2 INT, IN q2 INT) BEGIN DECLARE newId INT; SELECT COALESCE(MAX(order\_id), 0) + 1 INTO newId FROM Orders; INSERT INTO Orders(order\_id, customer\_id, order\_date) VALUES (newId, custId, CURRENT\_DATE); INSERT INTO OrderItems(order\_id, product\_id, quantity, price\_each) SELECT newId, p1, q1, price FROM Products WHERE product\_id = p1; INSERT INTO OrderItems(order\_id, product\_id, quantity, price\_each) SELECT newId, p2, q2, price FROM Products WHERE product\_id = p2; END;  
CREATE FUNCTION OrderTotal (oId INT) RETURNS DECIMAL(10,2) DETERMINISTIC BEGIN DECLARE tot DECIMAL(10,2); SELECT SUM(quantity \* price\_each) INTO tot FROM OrderItems WHERE order\_id = oId; RETURN tot; END;

-- 7. TRIGGER SOLUTION  
CREATE TABLE OrderArchive (  
 order\_id INT,  
 customer\_id INT,  
 order\_date DATE,  
 deleted\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
CREATE TRIGGER trg\_order\_delete AFTER DELETE ON Orders FOR EACH ROW INSERT INTO OrderArchive(order\_id, customer\_id, order\_date) VALUES (OLD.order\_id, OLD.customer\_id, OLD.order\_date);  
  
-- 8. WINDOW FUNCTION SOLUTION  
SELECT p.product\_id, p.name, SUM(oi.quantity \* oi.price\_each) AS total\_sales, RANK() OVER (PARTITION BY p.department ORDER BY SUM(oi.quantity \* oi.price\_each) DESC) AS sales\_rank FROM Products p JOIN OrderItems oi ON p.product\_id = oi.product\_id GROUP BY p.product\_id, p.name, p.department;  
  
-- 9. JSON HANDLING SOLUTION  
CREATE TABLE ProductDetails (product\_id INT PRIMARY KEY, specs JSON);  
INSERT INTO ProductDetails VALUES (10, '{"weight":"1.2kg","color":"black","warranty":"2 years"}');  
SELECT specs->>'$.weight' AS weight FROM ProductDetails WHERE product\_id = 10;  
  
-- 10. TRANSACTION SAVEPOINT SOLUTION  
BEGIN TRANSACTION;  
INSERT INTO Products(product\_id, name, price) VALUES (30, 'Mouse', 19.99);  
SAVEPOINT before\_duplicate;  
INSERT INTO Products(product\_id, name, price) VALUES (30, 'Mouse Duplicate', 19.99);  
ROLLBACK TO SAVEPOINT before\_duplicate;  
COMMIT;

11. Aggregate Query: Total sales per month  
SELECT EXTRACT(YEAR FROM order\_date) AS Year,   
 EXTRACT(MONTH FROM order\_date) AS Month,   
 SUM(total\_amount) AS TotalSales  
FROM Orders  
GROUP BY EXTRACT(YEAR FROM order\_date), EXTRACT(MONTH FROM order\_date)  
ORDER BY Year, Month;  
  
-- 12. Complex Query: Customers who bought products priced above $30  
SELECT DISTINCT c.name, c.email  
FROM Customers c  
JOIN Orders o ON c.customer\_id = o.customer\_id  
JOIN OrderItems oi ON o.order\_id = oi.order\_id  
JOIN Products p ON oi.product\_id = p.product\_id  
WHERE p.price > 30;  
  
-- 13. Data Validation: CHECK constraint on stock\_quantity  
ALTER TABLE Products   
ADD CONSTRAINT chk\_stock\_quantity CHECK (stock\_quantity >= 0);  
  
-- 14. Backup and Restore (Brief Description):  
To back up the database, use database-specific backup commands (e.g., mysqldump for MySQL or pg\_dump for PostgreSQL) to create a complete database snapshot.   
To restore, use the corresponding restore commands (e.g., mysql command or psql for PostgreSQL) to reload data from backup files.