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.
```