**Advanced SQL Lab Manual**

**Lab 1: Window Functions**

**Goal**

Understand and use ROW\_NUMBER, LAG, SUM OVER

**Setup**

sql

CopyEdit

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

name VARCHAR(50),

department VARCHAR(50),

salary INT

);

INSERT INTO employees VALUES

(1, 'Alice', 'HR', 50000),

(2, 'Bob', 'HR', 60000),

(3, 'Charlie', 'IT', 80000),

(4, 'David', 'IT', 75000),

(5, 'Eve', 'HR', 55000);

**Steps**

sql

CopyEdit

-- Rank salaries within each department

SELECT emp\_id, department, salary,

ROW\_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) AS salary\_rank

FROM employees;

-- Compare salary with previous employee

SELECT emp\_id, department, salary,

LAG(salary) OVER(PARTITION BY department ORDER BY salary) AS prev\_salary

FROM employees;

-- Calculate total salary per department

SELECT emp\_id, department, salary,

SUM(salary) OVER(PARTITION BY department) AS dept\_total

FROM employees;

**Lab 2: CTE vs Derived Table**

**Goal**

Compare query readability and reusability between CTEs and derived tables.

sql

CopyEdit

-- Using CTE

WITH dept\_avg AS (

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department

)

SELECT \* FROM dept\_avg WHERE avg\_salary > 55000;

-- Using Derived Table

SELECT \* FROM (

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department

) AS avg\_dept

WHERE avg\_salary > 55000;

**Lab 3: UNION and EXCEPT**

**Goal**

Perform set-based comparisons between two tables.

**Setup**

sql

CopyEdit

CREATE TABLE customers (name VARCHAR(50));

CREATE TABLE suppliers (name VARCHAR(50));

INSERT INTO customers VALUES ('Alice'), ('Bob'), ('Charlie');

INSERT INTO suppliers VALUES ('Bob'), ('David');

**Steps**

sql

CopyEdit

-- Combine distinct names

SELECT name FROM customers

UNION

SELECT name FROM suppliers;

-- Get names in customers but not in suppliers

SELECT name FROM customers

EXCEPT

SELECT name FROM suppliers;

**Lab 4: Materialized View**

**Goal**

Create a summary table with precomputed values.

**Setup**

sql

CopyEdit

CREATE TABLE orders (

order\_id INT,

region VARCHAR(50),

amount INT

);

INSERT INTO orders VALUES

(1, 'North', 200),

(2, 'North', 300),

(3, 'South', 400),

(4, 'South', 100);

**Steps**

sql

CopyEdit

-- Create Materialized View (PostgreSQL syntax)

CREATE MATERIALIZED VIEW sales\_summary AS

SELECT region, SUM(amount) AS total\_sales

FROM orders

GROUP BY region;

-- Query the view

SELECT \* FROM sales\_summary;

-- Refresh the view (if needed)

REFRESH MATERIALIZED VIEW sales\_summary;

**Lab 5: MERGE / UPSERT**

**Goal**

Handle insert/update logic in a single query.

**Setup**

sql

CopyEdit

CREATE TABLE new\_employees (

emp\_id INT,

name VARCHAR(50),

salary INT

);

INSERT INTO new\_employees VALUES

(1, 'Alice', 51000),

(6, 'Frank', 62000);

**Steps**

sql

CopyEdit

-- Merge new data

MERGE INTO employees AS target

USING new\_employees AS source

ON target.emp\_id = source.emp\_id

WHEN MATCHED THEN

UPDATE SET salary = source.salary

WHEN NOT MATCHED THEN

INSERT (emp\_id, name, salary)

VALUES (source.emp\_id, source.name, source.salary);

**Lab 6: Isolation Levels and Deadlocks**

**Goal**

Simulate a deadlock scenario using two sessions.

**Setup**

sql

CopyEdit

CREATE TABLE accounts (

acc\_id INT PRIMARY KEY,

balance INT

);

INSERT INTO accounts VALUES (1, 1000), (2, 1000);

**Steps**

**Session 1**

sql

CopyEdit

BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE acc\_id = 1;

-- wait here

**Session 2**

sql

CopyEdit

BEGIN;

UPDATE accounts SET balance = balance - 200 WHERE acc\_id = 2;

UPDATE accounts SET balance = balance - 100 WHERE acc\_id = 1; -- likely blocked

**Session 1 resumes**

sql

CopyEdit

UPDATE accounts SET balance = balance - 200 WHERE acc\_id = 2; -- Deadlock may occur

**Lab 7: Stored Procedures**

**Goal**

Encapsulate reusable SQL logic for business rules.

**Procedure Definition**

sql

CopyEdit

DELIMITER $$

CREATE PROCEDURE adjust\_salary(IN dept VARCHAR(50), IN pct INT)

BEGIN

UPDATE employees

SET salary = salary \* (1 + pct / 100)

WHERE department = dept;

END$$

DELIMITER ;

-- Execute

CALL adjust\_salary('HR', 10);

**Lab 8: Triggers**

**Goal**

Capture audit trail on salary changes.

**Setup**

sql

CopyEdit

CREATE TABLE audit\_log (

emp\_id INT,

old\_salary INT,

new\_salary INT,

changed\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**Trigger Definition**

sql

CopyEdit

DELIMITER $$

CREATE TRIGGER log\_salary\_change

AFTER UPDATE ON employees

FOR EACH ROW

BEGIN

INSERT INTO audit\_log(emp\_id, old\_salary, new\_salary)

VALUES (OLD.emp\_id, OLD.salary, NEW.salary);

END$$

DELIMITER ;

-- Test

UPDATE employees SET salary = salary + 1000 WHERE emp\_id = 1;

SELECT \* FROM audit\_log;

**Lab 9: Analytic View + Integrity Trigger**

**Goal**

Use a window-based view and enforce integrity with a trigger.

**View Creation**

sql

CopyEdit

CREATE VIEW employee\_analysis AS

SELECT emp\_id, department, salary,

RANK() OVER(PARTITION BY department ORDER BY salary DESC) AS salary\_rank,

SUM(salary) OVER(PARTITION BY department) AS total\_dept\_salary

FROM employees;

**Trigger to Prevent Invalid State**

sql

CopyEdit

DELIMITER $$

CREATE TRIGGER prevent\_negative\_balance

BEFORE UPDATE ON accounts

FOR EACH ROW

BEGIN

IF NEW.balance < 0 THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Insufficient balance!';

END IF;

END$$

DELIMITER ;

-- Test

UPDATE accounts SET balance = -500 WHERE acc\_id = 1;

**Conclusion**

You have completed hands-on practice with:

* Analytical querying using window functions
* Modular query design using CTEs and derived tables
* Efficient data management using materialized views and MERGE
* Ensuring data consistency with isolation levels, triggers, and stored procedures

Let me know if you'd like this turned into a formatted .docx or PDF version for printing or documentation.