Filtering and Organizing Data

1. Filtering with WHERE Clause

The `WHERE` clause is used to filter records based on specific conditions.

Example:

```
SELECT * FROM employees
WHERE department = 'Sales' AND salary > 50000;
```

This retrieves employees in the Sales department earning more than 50,000.

2. Sorting and Limiting Query Results

- Use `ORDER BY` to sort results in ascending (`ASC`) or descending (`DESC`) order.
- Use `LIMIT` to restrict the number of rows.

Example:

```
SELECT * FROM employees
ORDER BY salary DESC
LIMIT 5;
```

This retrieves the top 5 highest-paid employees.

3. Expressions and Comparison Operators

supports comparison operators such as `=`, `!=`, `<`, `>`, `<=`, `>=`, `BETWEEN`, `LIKE`, `IN`.

Example:

```
SELECT * FROM products
WHERE price BETWEEN 100 AND 500;
```

This retrieves products priced between 100 and 500.

Advanced Query Techniques

4. Nested Select Statements and Subqueries

A subquery is a query within another query.

Example:

```
SELECT name, salary FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

This retrieves employees earning more than the average salary.

5. Correlated Subqueries

A correlated subquery depends on the outer query.

Example:

```
SELECT name, department, salary
FROM employees e1
WHERE salary > (SELECT AVG(salary) FROM employees e2 WHERE e1.department =
e2.department);
```

This retrieves employees earning above the average salary in their department.

6. Grouping Rows and HAVING Clause

- Use `GROUP BY` to aggregate data.
- Use `HAVING` to filter grouped data.

Example:

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 50000;
```

This retrieves departments where the average salary is greater than 50,000.

7. Using Regular Expressions

supports pattern matching using `REGEXP` in databases like My.

Example:

```
SELECT * FROM customers
WHERE name REGEXP '^A|B';
```

This retrieves customers whose names start with "A" or "B".

8. Common Table Expressions (CTEs)

CTEs improve readability and are useful for recursive queries.

Example:

```
WITH HighSalary AS (
  SELECT name, salary FROM employees WHERE salary > 70000
)
SELECT * FROM HighSalary;
```

This creates a temporary table `HighSalary` and selects from it.

9. Hierarchical Queries

Used for working with hierarchical data (like org charts).

Example (for Oracle using `CONNECT BY`)

```
SELECT employee_id, name, manager_id
FROM employees
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id;
```

This retrieves employees in a hierarchical structure.

Difference Between Joins and Set Operators

Feature	Joins	Set Operators
Definition	Combines columns from multiple tables	Combines rows from multiple queries
Used When	You need related data from different tables	You need to merge results from different queries
Result Type	Adds more **columns**	Adds more **rows**
Example	`INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`	`UNION`, `INTERSECT`, `EXCEPT`

Example: Join (Combining Columns)

Suppose we have two tables:

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    total_amount DECIMAL(10,2),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

INSERT INTO customers VALUES (1, 'Alice'), (2, 'Bob');
INSERT INTO orders VALUES (101, 1, 200.00), (102, 2, 300.00);
```

Query Using JOIN:

```
SELECT customers.name, orders.total_amount
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id;
```

Result:

name	total_amount
Alice	200.00
Bob	300.00

Example: Set Operator (Combining Rows)

Suppose we have two different queries:

```
CREATE TABLE old_customers (
    name VARCHAR(100)
);

CREATE TABLE new_customers (
    name VARCHAR(100)
);

INSERT INTO old_customers VALUES ('Alice'), ('Charlie');
INSERT INTO new_customers VALUES ('Bob'), ('Charlie');
```

Query Using `UNION`:

```
SELECT name FROM old_customers
UNION
SELECT name FROM new_customers;
```

Result (Merged Unique Rows):

name
Alice
Bob
Charlie

UNION, INTERSECT, and EXCEPT

1. UNION (Combining Unique Rows)

```
SELECT name FROM old_customers
UNION
SELECT name FROM new_customers;
```

Removes duplicates (Charlie appears only once).

2. UNION ALL (Combining All Rows)

```
SELECT name FROM old_customers
UNION ALL
```

```
SELECT name FROM new_customers;
```

Keeps duplicates (Charlie appears twice).

3. INTERSECT (Common Rows)

```
SELECT name FROM old_customers
INTERSECT
SELECT name FROM new_customers;
```

Returns only **Charlie** because it's in both tables.

4. EXCEPT (Find Differences)

```
SELECT name FROM old_customers
EXCEPT
SELECT name FROM new_customers;
```

Returns **Alice** (present in `old_customers` but not in `new_customers`).

Creating Views and Procedures

Views store **predefined queries** as virtual tables.

Example: Create a View for High-Value Orders

```
CREATE VIEW high_value_orders AS
SELECT order_id, customer_id, total_amount
FROM orders
WHERE total_amount > 250;
```

Usage:

```
SELECT * FROM high_value_orders;
```

Stored Procedure for Frequent Orders

```
DELIMITER $$

CREATE PROCEDURE frequent_customers(IN min_orders INT)
BEGIN
    SELECT customer_id, COUNT(order_id) AS order_count
    FROM orders
    GROUP BY customer_id
    HAVING order_count >= min_orders;
END $$
```

```
DELIMITER ;
```

Usage:

```
CALL frequent_customers(2);
```

Query Performance Analysis & Execution Plans

EXPLAIN helps analyze query execution plans.

Example: Checking Execution Plan

```
EXPLAIN SELECT * FROM orders WHERE total_amount > 250;
```

Key Output Columns in `EXPLAIN`

- `type` → **Index**, **ALL**, **range** (better = index)
- `possible_keys` → Suggested indexes
- `rows` → Estimated rows scanned (lower = better)

Indexes and Query Optimization

Indexes speed up searches but slow down inserts/updates.

Creating Index on Orders

```
CREATE INDEX idx_total_amount ON orders(total_amount);
```

Query Performance Before & After Index

```
SELECT * FROM orders WHERE total_amount > 250;
```

- **Without Index:** Full table scan
- **With Index:** Uses `idx_total_amount`

Checking Index Usage

```
SHOW INDEXES FROM orders;
```

Summary of Best Practices

Use `JOIN` when combining columns, `UNION` when combining rows
Index frequently searched columns (`WHERE`, `ORDER BY`)
Use `EXPLAIN` to check performance issues
Create views for complex queries to improve readability
Use stored procedures for reusable logic

Window (Analytic) Functions in My

Window functions allow row-by-row calculations over a defined partition of the dataset without collapsing it.

Sample Table: `employees`

```
CREATE TABLE employees (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    department VARCHAR(50),
    salary INT
);

INSERT INTO employees (name, department, salary) VALUES
('Alice', 'HR', 60000),
('Bob', 'HR', 55000),
('Charlie', 'HR', 75000),
('David', 'IT', 90000),
('Eve', 'IT', 85000),
('Frank', 'IT', 95000),
('Grace', 'Sales', 70000),
('Hank', 'Sales', 72000),
('Ivy', 'Sales', 68000);
```

1.1 Ranking Employees by Salary (RANK, DENSE_RANK, ROW_NUMBER)

```
SELECT name, department, salary,
       RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank,
       DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
dense_rank,
       ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS
row_num
FROM employees;
```

Output:

name	department	salary	rank	dense_rank	row_num
Charlie	HR	75000	1	1	1
Alice	HR	60000	2	2	2
Bob	HR	55000	3	3	3
Frank	IT	95000	1	1	1
David	IT	90000	2	2	2
Eve	IT	85000	3	3	3
Hank	Sales	72000	1	1	1
Grace	Sales	70000	2	2	2
Ivy	Sales	68000	3	3	3

Differences Between Ranking Functions:

- `RANK()`: Skips numbers if ties exist.
- `DENSE_RANK()`: Does not skip numbers for ties.
- `ROW_NUMBER()`: Assigns unique row numbers.

---

2. Common Table Expressions (CTEs)

CTEs provide a **temporary result set** that can be referenced within a query.

Example: Get Employees Who Earn Above Department Average

```
WITH DeptSalary AS (
    SELECT department, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department
)
SELECT e.name, e.department, e.salary, ds.avg_salary
FROM employees e
JOIN DeptSalary ds ON e.department = ds.department
WHERE e.salary > ds.avg_salary;
```

Output:

name	department	salary	avg_salary
Charlie	HR	75000	63333.33
Frank	IT	95000	90000.00
Hank	Sales	72000	70000.00

---

3. Recursive CTE (Hierarchy Queries)

Used for self-referencing relationships (e.g., **employee-manager hierarchy**).

Sample Table: `employees_hierarchy`

```
CREATE TABLE employees_hierarchy (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    manager_id INT NULL
);

INSERT INTO employees_hierarchy (id, name, manager_id) VALUES
(1, 'CEO', NULL),
(2, 'Manager A', 1),
(3, 'Manager B', 1),
(4, 'Employee 1', 2),
(5, 'Employee 2', 2),
(6, 'Employee 3', 3);
```

Recursive CTE to Find Hierarchy

```
WITH RECURSIVE EmployeeTree AS (
    SELECT id, name, manager_id, 1 AS level
    FROM employees_hierarchy
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.id, e.name, e.manager_id, et.level + 1
    FROM employees_hierarchy e
    JOIN EmployeeTree et ON e.manager_id = et.id
)
SELECT * FROM EmployeeTree;
```

Output:

id	name	manager_id	level
1	CEO	NULL	1
2	Manager A	1	2
3	Manager B	1	2
4	Employee 1	2	3
5	Employee 2	2	3
6	Employee 3	3	3

Use Case: Find **all employees reporting to a given manager**.

---

4. JSON Data Handling (My 5.7+)

My allows storing **JSON** data and querying it efficiently.

Sample Table: `customers`

```
CREATE TABLE customers (
    id INT PRIMARY KEY AUTO_INCREMENT,
```

```
    name VARCHAR(50),
    details JSON
);

INSERT INTO customers (name, details) VALUES
('Alice', '{"age": 30, "city": "New York", "orders": [{"product": "Laptop",
"price": 1200}]}'),
('Bob', '{"age": 25, "city": "Los Angeles", "orders": [{"product": "Phone",
"price": 800}]}');
```

Querying JSON Fields

```
SELECT name, details->>'$.city' AS city, details->>'$.age' AS age
FROM customers;
```

Output:

name	city	age
Alice	New York	30
Bob	Los Angeles	25

---

5. Full-Text Search (My 5.6+)

Allows **fast keyword searches** on large text fields.

Create Table with Full-Text Index

```
CREATE TABLE articles (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255),
    content TEXT,
    FULLTEXT(title, content)
);

INSERT INTO articles (title, content) VALUES
('My Performance Tips', 'Learn how to optimize My queries.'),
('Full-Text Search in My', 'Using MATCH() AGAINST() for text search.');
```

Searching for Keywords

```
SELECT * FROM articles
WHERE MATCH(title, content) AGAINST('My' IN NATURAL LANGUAGE MODE);
```

Output:

id	title	content
1	My Performance Tips	Learn how to optimize My queries.
2	Full-Text Search in My	Using MATCH() AGAINST() for text search.

---

6. Indexing for Performance

Indexes speed up queries by **avoiding full table scans**.

Creating an Index on `employees.salary`

```
CREATE INDEX idx_salary ON employees(salary);
```

Benefits:

- Queries like `WHERE salary > 70000` will **run faster**.

---

7. Partitioning Large Tables

Partitioning helps in **query optimization**.

Example: Range Partitioning (By Year)

```
CREATE TABLE sales (
    id INT NOT NULL,
    sale_date DATE NOT NULL,
    amount DECIMAL(10,2) NOT NULL,
    PRIMARY KEY (id, sale_date)
) PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION p2024 VALUES LESS THAN (2025)
);
```

Create Sample Database and Table

```
CREATE DATABASE BankDB;
USE BankDB;

CREATE TABLE accounts (
    account_id INT PRIMARY KEY AUTO_INCREMENT,
    account_name VARCHAR(50),
    balance DECIMAL(10,2) NOT NULL
);
```

---

2. Insert Sample Data

```
INSERT INTO accounts (account_name, balance) VALUES
('Alice', 5000.00),
```

```
('Bob', 3000.00),
('Charlie', 7000.00);
```

---

3. Using `ROLLBACK` (Undo All Changes)

Scenario: Transfer Money from Alice to Bob, but an Error Occurs

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 1000 WHERE account_name = 'Alice';
UPDATE accounts SET balance = balance + 1000 WHERE account_name = 'Bob';

-- Oops! Something went wrong. Rollback everything.
ROLLBACK;

-- Check the balances after rollback
SELECT * FROM accounts;
```

Effect: Since we rolled back, Alice's and Bob's balances remain unchanged.

---

4. Using `SAVEPOINT` (Partial Rollback)

Scenario: Transfer Money in Two Steps, but Only Undo One Step

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE account_name = 'Alice';
SAVEPOINT step1;

UPDATE accounts SET balance = balance + 500 WHERE account_name = 'Bob';
SAVEPOINT step2;

UPDATE accounts SET balance = balance + 300 WHERE account_name = 'Charlie';

-- Something went wrong after step 2, rollback only the last step
ROLLBACK TO step2;

-- Commit the remaining transactions
COMMIT;

-- Check the balances after partial rollback
SELECT * FROM accounts;
```

Effect: The last step (adding 300 to Charlie) is undone, but Alice's and Bob's transactions are committed.

---

5. Using `ROLLBACK` in Stored Procedure

Scenario: Automatic Rollback on Error

```
DELIMITER $$

CREATE PROCEDURE transfer_money(IN from_acc INT, IN to_acc INT, IN amount
DECIMAL(10,2))
BEGIN
    DECLARE EXIT HANDLER FOR EXCEPTION
    BEGIN
        ROLLBACK;
        SIGNAL STATE '45000' SET MESSAGE_TEXT = 'Transaction Failed';
    END;

    START TRANSACTION;

    -- Deduct from sender
    UPDATE accounts SET balance = balance - amount WHERE account_id = from_acc;

    -- Intentional error: transferring to a non-existent account
    UPDATE accounts SET balance = balance + amount WHERE account_id = to_acc;

    COMMIT;
END $$

DELIMITER ;
```

---

6. Test the Stored Procedure

Scenario: Transfer Money from Alice to a Non-Existent Account

```
CALL transfer_money(1, 99, 500);
```

Effect: The transaction fails due to an invalid account ID, and `ROLLBACK` is triggered, keeping balances unchanged.

---

7. Verify Final Balances

```
SELECT * FROM accounts;
```

Create Sample Table

```
CREATE DATABASE CompanyDB;
USE CompanyDB;

CREATE TABLE employees (
```

```
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    manager_id INT NULL
);
```

---

2. Insert Sample Data (Employee-Manager Relationship)

```
INSERT INTO employees (emp_id, emp_name, manager_id) VALUES
(1, 'Alice', NULL),    -- Alice is the CEO (no manager)
(2, 'Bob', 1),         -- Bob reports to Alice
(3, 'Charlie', 1),     -- Charlie reports to Alice
(4, 'David', 2),       -- David reports to Bob
(5, 'Eve', 2),         -- Eve reports to Bob
(6, 'Frank', 3),       -- Frank reports to Charlie
(7, 'Grace', 3);       -- Grace reports to Charlie
```

---

3. Recursive Query to Find Employee Hierarchy

Let's find all employees reporting to **Alice (CEO - emp_id = 1)**.

```
WITH RECURSIVE employee_hierarchy AS (
    -- Base case: Select the CEO (Alice)
    SELECT emp_id, emp_name, manager_id, 1 AS level
    FROM employees
    WHERE manager_id IS NULL  -- Alice has no manager

    UNION ALL

    -- Recursive case: Find employees reporting to those in the previous level
    SELECT e.emp_id, e.emp_name, e.manager_id, eh.level + 1
    FROM employees e
    INNER JOIN employee_hierarchy eh ON e.manager_id = eh.emp_id
)

SELECT * FROM employee_hierarchy;
```

---

4. Output Explanation

emp_id	emp_name	manager_id	level
1	Alice	NULL	1
2	Bob	1	2
3	Charlie	1	2
4	David	2	3
5	Eve	2	3
6	Frank	3	3
7	Grace	3	3

- **Alice (Level 1)** is the CEO.

- **Bob & Charlie (Level 2)** report to Alice.
- **David & Eve (Level 3)** report to Bob.
- **Frank & Grace (Level 3)** report to Charlie.

---

5. Recursive Query to Find an Employee’s Manager Chain

To find **who manages "David" (emp_id = 4)** up to the CEO:

```
WITH RECURSIVE manager_chain AS (
    -- Base case: Select David
    SELECT emp_id, emp_name, manager_id
    FROM employees
    WHERE emp_name = 'David'

    UNION ALL

    -- Recursive case: Find David's manager, then manager's manager, and so on
    SELECT e.emp_id, e.emp_name, e.manager_id
    FROM employees e
    INNER JOIN manager_chain mc ON e.emp_id = mc.manager_id
)

SELECT * FROM manager_chain;
```

---

6. Output (Manager Chain for David)

emp_id	emp_name	manager_id
4	David	2
2	Bob	1
1	Alice	NULL

- **David is managed by Bob**
- **Bob is managed by Alice (CEO)**

---

7. When to Use Recursive Queries?

Good for:

- Employee **hierarchies**
- **Category trees** (e.g., parent-child relationships in e-commerce)
- **Graph traversal** (e.g., social networks, routes, dependencies)

ATM Database Schema

A basic ATM database will have the following tables:

A. Customers Table (Stores user details)

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(20) UNIQUE
);
```

B. Accounts Table (Each customer has a bank account)

```
CREATE TABLE accounts (
    account_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    balance DECIMAL(10,2) NOT NULL DEFAULT 0.00,
    account_type ENUM('savings', 'checking'),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

C. Transactions Table (Records all deposits, withdrawals, and transfers)

```
CREATE TABLE transactions (
    transaction_id INT PRIMARY KEY AUTO_INCREMENT,
    account_id INT,
    transaction_type ENUM('deposit', 'withdrawal', 'transfer'),
    amount DECIMAL(10,2) NOT NULL,
    transaction_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (account_id) REFERENCES accounts(account_id)
);
```

---

2. ATM Transactions with My

To ensure **secure transactions**, we use `START TRANSACTION`, `COMMIT`, and `ROLLBACK`.

A. Withdraw Money (Using Transaction & Rollback)

> **Scenario: A user tries to withdraw money, but the balance should not go negative.**

```
DELIMITER $$

CREATE PROCEDURE withdraw_money(IN account_num INT, IN withdraw_amount
DECIMAL(10,2))
BEGIN
```

```
    DECLARE current_balance DECIMAL(10,2);

    -- Start transaction
    START TRANSACTION;

    -- Get current balance
    SELECT balance INTO current_balance FROM accounts WHERE account_id =
account_num FOR UPDATE;

    -- Check if enough balance is available
    IF current_balance >= withdraw_amount THEN
        -- Deduct amount
        UPDATE accounts SET balance = balance - withdraw_amount WHERE account_id
= account_num;

        -- Insert transaction record
        INSERT INTO transactions (account_id, transaction_type, amount) VALUES
(account_num, 'withdrawal', withdraw_amount);

        -- Commit the transaction
        COMMIT;
    ELSE
        -- Not enough balance, rollback
        ROLLBACK;
        SIGNAL STATE '45000' SET MESSAGE_TEXT = 'Insufficient funds';
    END IF;
END $$

DELIMITER ;
```

Test Withdrawal

```
CALL withdraw_money(1, 500);
```

- If the user has **enough balance**, the transaction **commits**.
- If the user has **insufficient balance**, the transaction **rolls back** and **prevents overdrawing**.

B. Transfer Money (With ACID Compliance)

Scenario: A user transfers money from account A to account B. If any step fails, the transaction must roll back.

```
DELIMITER $$

CREATE PROCEDURE transfer_money(IN from_acc INT, IN to_acc INT, IN
transfer_amount DECIMAL(10,2))
BEGIN
    DECLARE from_balance DECIMAL(10,2);

    -- Start transaction
    START TRANSACTION;

    -- Get balance of sender
    SELECT balance INTO from_balance FROM accounts WHERE account_id = from_acc
FOR UPDATE;
```

```
    -- Check if the sender has enough balance
    IF from_balance >= transfer_amount THEN
        -- Deduct from sender
        UPDATE accounts SET balance = balance - transfer_amount WHERE account_id
= from_acc;

        -- Add to receiver
        UPDATE accounts SET balance = balance + transfer_amount WHERE account_id
= to_acc;

        -- Insert transaction records
        INSERT INTO transactions (account_id, transaction_type, amount) VALUES
        (from_acc, 'transfer', -transfer_amount),
        (to_acc, 'transfer', transfer_amount);

        -- Commit transaction
        COMMIT;
    ELSE
        -- Rollback transaction if balance is insufficient
        ROLLBACK;
        SIGNAL STATE '45000' SET MESSAGE_TEXT = 'Insufficient funds for
transfer';
    END IF;
END $$

DELIMITER ;
```

Test Money Transfer

```
CALL transfer_money(1, 2, 200);
```

- Ensures that **both debit and credit operations** succeed before committing.
- If an error occurs, **ROLLBACK** prevents partial transactions.

---

3. Handling ATM Card & PIN Authentication

To simulate an **ATM PIN system**, add an extra column in the `customers` table:

```
ALTER TABLE customers ADD COLUMN pin VARCHAR(6) NOT NULL;
```

Then, create a **stored procedure** for authentication:

```
DELIMITER $$

CREATE PROCEDURE validate_atm_card(IN customer_id INT, IN entered_pin
VARCHAR(6))
BEGIN
    DECLARE stored_pin VARCHAR(6);

    -- Get stored PIN
    SELECT pin INTO stored_pin FROM customers WHERE customer_id = customer_id;

    -- Check PIN
```

```
    IF stored_pin = entered_pin THEN
        SELECT 'Access Granted' AS message;
    ELSE
        SIGNAL STATE '45000' SET MESSAGE_TEXT = 'Invalid PIN';
    END IF;
END $$

DELIMITER ;
```

Test ATM Authentication

```
CALL validate_atm_card(1, '123456');
```

If the PIN matches, **access is granted**; otherwise, an error occurs.

---

4. Scaling ATM System with Sharding

If the ATM system **grows**, consider **sharding strategies**:

Range-Based Sharding (Accounts 1-1M in `bank_shard_1`, 1M+ in `bank_shard_2`).
Hash-Based Sharding (Use `MOD(account_id, num_shards)`).
Geo-Based Sharding (USA, Europe, Asia databases).

Example **Hash-Based Sharding Logic**:

```
python

def get_shard(account_id):
    return account_id % 3  # Assume 3 shards

# Route queries to the correct shard
def get_balance(account_id):
    shard_id = get_shard(account_id)
    db = shards[shard_id]  # Connect to correct database
    cursor = db.cursor()
    cursor.execute("SELECT balance FROM accounts WHERE account_id = %s",
(account_id,))
    return cursor.fetchone()[0]
```

---

5. Security Considerations

Use Encryption for PIN Storage (`SHA2`, `AES_ENCRYPT`)

```
UPDATE customers SET pin = SHA2('123456', 256) WHERE customer_id = 1;
```

Enable Two-Factor Authentication (OTP via SMS/Email)
Use Indexing for Faster Queries (`INDEX(account_id)`)
Audit Log for All Transactions (Monitor Fraud)
Implement Rate Limiting (Prevent brute-force attacks)

Final Thoughts

- My supports **ACID transactions** to ensure **secure ATM operations**.
- `COMMIT` and `ROLLBACK` **prevent data corruption** in case of failures.
- **Stored procedures** handle **withdrawals, deposits, and transfers** securely.
- **Sharding** improves **scalability** for large banking systems.

Stored Procedure for Employee Bonus Calculation

Scenario: Calculate and update bonuses based on salary.
Condition: If salary is **above $5000**, bonus is **10%**, otherwise **5%**.

Step 1: Create Employee Table & Insert Data

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    salary DECIMAL(10,2),
    bonus DECIMAL(10,2) DEFAULT 0.00
);

INSERT INTO employees (name, salary) VALUES
('Alice', 7000),
('Bob', 4000),
('Charlie', 5500);
```

Step 2: Create Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE calculate_bonus(IN emp_id_input INT)
BEGIN
    DECLARE emp_salary DECIMAL(10,2);
    DECLARE emp_bonus DECIMAL(10,2);

    -- Get employee's salary
    SELECT salary INTO emp_salary FROM employees WHERE emp_id = emp_id_input;

    -- Calculate bonus based on salary
    IF emp_salary > 5000 THEN
        SET emp_bonus = emp_salary * 0.10;
    ELSE
        SET emp_bonus = emp_salary * 0.05;
    END IF;

    -- Update bonus column
    UPDATE employees SET bonus = emp_bonus WHERE emp_id = emp_id_input;

    SELECT name, salary, bonus FROM employees WHERE emp_id = emp_id_input;
END $$
```

```
DELIMITER ;
```

Step 3: Execute the Procedure

```
CALL calculate_bonus(1);  -- Alice (7000) → Bonus 10% → 700
CALL calculate_bonus(2);  -- Bob (4000) → Bonus 5% → 200
CALL calculate_bonus(3);  -- Charlie (5500) → Bonus 10% → 550
```

---

2. Stored Procedure to Count Orders in a Date Range

Scenario: Get the number of orders within a date range.

Step 1: Create Orders Table & Insert Data

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_name VARCHAR(100),
    order_date DATE,
    total_amount DECIMAL(10,2)
);

INSERT INTO orders (customer_name, order_date, total_amount) VALUES
('Alice', '2024-02-01', 150.00),
('Bob', '2024-02-03', 200.50),
('Charlie', '2024-02-05', 320.75),
('David', '2024-02-10', 400.00);
```

Step 2: Create Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE count_orders(IN start_date DATE, IN end_date DATE)
BEGIN
    SELECT COUNT(*) AS total_orders
    FROM orders
    WHERE order_date BETWEEN start_date AND end_date;
END $$

DELIMITER ;
```

Step 3: Execute the Procedure

```
CALL count_orders('2024-02-01', '2024-02-05');  -- Returns 3 orders
CALL count_orders('2024-02-01', '2024-02-10');  -- Returns 4 orders
```

---

3. Stored Procedure to Update Product Stock After Sale

Scenario: When a customer purchases a product, reduce stock.

Condition: Ensure stock does not go negative.

Step 1: Create Products Table & Insert Data

```
CREATE TABLE products (
    product_id INT PRIMARY KEY AUTO_INCREMENT,
    product_name VARCHAR(100),
    stock INT NOT NULL
);

INSERT INTO products (product_name, stock) VALUES
('Laptop', 10),
('Phone', 20),
('Headphones', 15);
```

Step 2: Create Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE sell_product(IN prod_id INT, IN quantity INT)
BEGIN
    DECLARE current_stock INT;

    -- Get current stock
    SELECT stock INTO current_stock FROM products WHERE product_id = prod_id;

    -- Check if stock is sufficient
    IF current_stock >= quantity THEN
        -- Deduct quantity from stock
        UPDATE products SET stock = stock - quantity WHERE product_id = prod_id;
        SELECT 'Purchase Successful' AS message;
    ELSE
        SELECT 'Not Enough Stock' AS message;
    END IF;
END $$

DELIMITER ;
```

Step 3: Execute the Procedure

```
CALL sell_product(1, 5);  -- Laptop stock reduced from 10 to 5
CALL sell_product(2, 25); -- Phone stock is 20, so purchase fails
```

---

4. Stored Procedure to Find Employees with High Salaries

Scenario: Get all employees who earn more than a given salary.

Step 1: Create Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE high_salary_employees(IN min_salary DECIMAL(10,2))
BEGIN
    SELECT name, salary FROM employees WHERE salary > min_salary;
END $$

DELIMITER ;
```

Step 2: Execute the Procedure

```
CALL high_salary_employees(5000);  -- Lists employees earning more than $5000
```

---

5. Stored Procedure to Add a New Customer

Scenario: Insert a new customer into the customers table.

Step 1: Create Customers Table

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
);
```

Step 2: Create Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE add_customer(IN cust_name VARCHAR(100), IN cust_email
VARCHAR(100))
BEGIN
    INSERT INTO customers (name, email) VALUES (cust_name, cust_email);
    SELECT 'Customer Added Successfully' AS message;
END $$

DELIMITER ;
```

Step 3: Execute the Procedure

```
CALL add_customer('Eve', 'eve@example.com');
CALL add_customer('John', 'john@example.com');
```

Conclusion

Stored procedures help in: **Automating tasks** like salary calculation, order counting, and stock updates.

Improving security by restricting direct table access.

Boosting performance by reducing query execution time.