Unit - 4

Comparison of DAC, MAC, and Data Encryption:

1. Discretionary Access Control (DAC)

In **Discretionary Access Control**, the database owner can grant or revoke permissions to other users. This example demonstrates granting and revoking access permissions to users on a table in a database.

Query: Granting and Revoking Permissions

-- Granting SELECT permission on 'employees' table to user 'john_doe'

GRANT SELECT ON employees TO john_doe;

-- Revoking SELECT permission on 'employees' table from user 'john_doe'

REVOKE SELECT ON employees FROM john_doe;

Explanation:

- **GRANT SELECT:** The GRANT statement is used to allow a user (john_doe) the permission to perform a specific action (in this case, SELECT) on the employees table.
- **REVOKE SELECT:** The REVOKE statement removes the permission previously granted to the user on the table.
- **DAC:** This type of control allows the owner of the database object (in this case, the employees table) to grant or revoke access to other users. The decision to grant access is left to the discretion of the object owner.

2. Mandatory Access Control (MAC)

In **Mandatory Access Control**, access to resources is determined by the system based on predefined policies and classifications. This example demonstrates how a user's access might be determined by security labels (such as sensitivity levels). In most DBMSs, MAC is implemented using security models, but for the sake of simplicity, let's assume we are dealing with a table and we have a column indicating the classification level of data.

Query: Using a Classification Column to Implement MAC

-- Granting access to sensitive data based on user clearance level

```
SELECT first_name, last_name, email
FROM employees
WHERE clearance_level >= (SELECT clearance_level FROM users WHERE username = 'admin')
  AND data_classification <= 'Confidential';
```

Explanation:

- **clearance_level and data_classification:** The employees table is assumed to have a clearance_level column and a data_classification column. These columns control access based on the user's security clearance and the sensitivity of the data.
- **WHERE clause:** This condition ensures that users can only access employees' data if their clearance_level is greater than or equal to that of the admin user, and the data_classification level is classified as 'Confidential' or lower.
- **MAC:** In this case, the access control system is enforcing mandatory policies. The access rights are determined by predefined security clearance levels and the classification of data, not by the discretion of the users.

3. Data Encryption

Data encryption ensures that sensitive data is securely stored and transmitted. Here, we will demonstrate how to encrypt and decrypt sensitive data using a **column-level encryption** example in SQL (Note: actual encryption requires database-specific tools, so the syntax may vary depending on your DBMS).

Query: Encrypting and Decrypting Data

-- Inserting an encrypted credit card number into the 'transactions' table

```
INSERT INTO transactions (transaction_id, user_id, encrypted_credit_card)
VALUES (1, 101, ENCRYPT('1234-5678-9876-5432', 'encryption_key'));
```

-- Selecting and decrypting the encrypted credit card number from the 'transactions' table

```
SELECT transaction_id, user_id, DECRYPT(encrypted_credit_card, 'encryption_key') AS decrypted_credit_card
FROM transactions
WHERE user_id = 101;
```

Explanation:

- **ENCRYPT() and DECRYPT():** These are hypothetical functions that represent the encryption and decryption of sensitive data. The actual functions used to encrypt or decrypt data vary depending on the DBMS (e.g., AES_ENCRYPT and AES_DECRYPT in MySQL).
- **INSERT:** The ENCRYPT function is used to encrypt the credit card number before storing it in the transactions table.
- **SELECT:** The DECRYPT function is used to decrypt the encrypted data when retrieved. The encrypted credit card number is stored in the database, and only authorized users with the correct decryption key can access the plain text data.
- **Data Encryption:** Encryption ensures that the sensitive data is stored in a secure, unreadable format, protecting it from unauthorized access, even if the database is compromised.

Summary of Concepts:

- **DAC:** Allows the resource owner to decide who has access to the data (e.g., granting or revoking permissions).
- **MAC:** Access to resources is enforced by the system based on predefined policies (e.g., based on user security clearance and data classification).
- **Data Encryption:** Protects sensitive data by converting it into an unreadable format that can only be decrypted by authorized parties with the appropriate key.

1. Sub-queries

Definition: A sub-query is a query that is embedded within another query. It is used to perform an operation that requires more than one step. Sub-queries are enclosed in parentheses and can be used in SELECT, INSERT, UPDATE, and DELETE operations.

Types of Sub-queries:

- **Single-row Sub-query:** Returns a single row as a result.
- **Multi-row Sub-query:** Returns multiple rows as a result.
- **Multi-column Sub-query:** Returns multiple columns as a result.

Example:

```
SELECT * FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

2. Correlated Sub-queries

Definition: A correlated sub-query is a sub-query that refers to the outer query. It is executed for each row of the outer query. Unlike a normal sub-query, which can be executed independently, a correlated sub-query depends on the outer query's values.

Example:

```
SELECT employee_id, name, salary

FROM employees e

WHERE salary > (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id);
```

3. Use of GROUP BY, HAVING, and ORDER BY

- **GROUP BY:**
    - Used to group rows that have the same values into summary rows (such as "total salary" per department).
    - Often used with aggregate functions (e.g., COUNT, SUM, AVG).
    - Syntax: GROUP BY column_name;

Example:

```
SELECT department_id, COUNT(*)
FROM employees
GROUP BY department_id;
```

- **HAVING:**
    - Used to filter groups after the GROUP BY operation. It is similar to the WHERE clause, but it is applied to groups rather than rows.
    - Syntax: HAVING condition;

Example:

```
SELECT department_id, COUNT(*)
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5;
```

- **ORDER BY:**
    - Used to sort the result set in ascending or descending order.
    - Syntax: ORDER BY column_name [ASC|DESC];

Example:

```
SELECT employee_id, name, salary
FROM employees
ORDER BY salary DESC;
```

4. Joins and Its Types

Definition: A JOIN is used to combine rows from two or more tables based on a related column between them. Joins are crucial in relational databases to retrieve data from multiple tables in a single query.

Types of Joins:

- **INNER JOIN:**
    - Retrieves records that have matching values in both tables.
    - If there is no match, the row will not appear in the result.

Example:

SELECT employees.name, departments.department_name

FROM employees

INNER JOIN departments ON employees.department_id = departments.department_id;

- **LEFT JOIN (or LEFT OUTER JOIN):**
    - Retrieves all records from the left table and the matching records from the right table. If there is no match, NULL values are returned for the right table columns.

Example:

SELECT employees.name, departments.department_name

FROM employees

LEFT JOIN departments ON employees.department_id = departments.department_id;

- **RIGHT JOIN (or RIGHT OUTER JOIN):**
    - Retrieves all records from the right table and the matching records from the left table. If there is no match, NULL values are returned for the left table columns.

Example:

SELECT employees.name, departments.department_name

FROM employees

RIGHT JOIN departments ON employees.department_id = departments.department_id;

- **FULL JOIN (or FULL OUTER JOIN):**

- Retrieves records when there is a match in either left (table1) or right (table2). It returns NULL for non-matching rows from both tables.

Example:

```
SELECT employees.name, departments.department_name
FROM employees
FULL OUTER JOIN departments ON employees.department_id = departments.department_id;
```

- **SELF JOIN:**
  - Joins a table with itself. It is useful for hierarchical or recursive data relationships.

Example:

```
SELECT e1.name AS Employee, e2.name AS Manager
FROM employees e1
LEFT JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

5. EXISTS, ANY, and ALL

- **EXISTS:**
    - The EXISTS operator checks whether the sub-query returns any rows. It returns TRUE if the sub-query returns one or more rows, and FALSE otherwise.

Example:

```
SELECT employee_id, name
FROM employees
WHERE EXISTS (SELECT * FROM projects WHERE projects.employee_id = employees.employee_id);
```

- **ANY:**
    - The ANY operator compares a value to any value in a set of values returned by a sub-query. It returns TRUE if the condition is true for any value in the set.

Example:

```
SELECT employee_id, name
FROM employees
WHERE salary > ANY (SELECT salary FROM employees WHERE department_id = 3);
```

- **ALL:**
    - The ALL operator compares a value to all values in a set returned by a sub-query. It returns TRUE if the condition is true for all values in the set.

Example:

```
SELECT employee_id, name
FROM employees
WHERE salary > ALL (SELECT salary FROM employees WHERE department_id = 3);
```

6. Views and Its Types

Definition: A view is a virtual table based on the result set of an SQL query. It doesn't store data physically but provides a way to access data in a more manageable form. Views can simplify complex queries and provide an abstraction layer.

Types of Views:

- **Simple View:**
    - A simple view is based on a single table and does not contain any complex SQL operations like joins, groupings, or sub-queries.

Example:

```
CREATE VIEW employee_view AS

SELECT employee_id, name, salary

FROM employees;
```

- **Complex View:**
    - A complex view involves multiple tables and includes joins, aggregations, or sub-queries. It may also involve grouping data or performing calculations.

Example:

```
CREATE VIEW department_summary AS

SELECT department_id, COUNT(*) AS total_employees, AVG(salary) AS avg_salary

FROM employees

GROUP BY department_id;
```

- **Updatable View:**
    - Some views allow you to perform operations like INSERT, UPDATE, and DELETE directly on the view, but only under certain conditions (e.g., no aggregations or joins).
- **Non-updatable View:**
    - Views that include complex joins, groupings, or aggregation functions are typically non-updatable. These views are used primarily for data retrieval.

Advantages of Views:

- Simplifies complex queries.
- Provides data security (only specific columns can be viewed).
- Reduces data redundancy by storing query logic instead of data.

Example of using a View:

SELECT * FROM employee_view;

Summary of Key Concepts:

- **Sub-queries** help in fetching data based on another query.
- **Correlated sub-queries** depend on the outer query.
- **GROUP BY, HAVING, ORDER BY** help in grouping, filtering, and sorting data.
- **Joins** are used to combine tables. Types include INNER, LEFT, RIGHT, and FULL OUTER joins.
- **EXISTS, ANY, ALL** are used for comparison operations involving sub-queries.
- **Views** provide a simplified representation of data and can be categorized into simple and complex views.

📌 1. Cursors

🔹 **Definition:**

A **cursor** is a database object used to retrieve, manipulate, and navigate through a result set row by row.

🔹 **Types of Cursors:**

- **Implicit Cursor**: Automatically created by DBMS when a query is executed.
- **Explicit Cursor**: Defined and controlled by the programmer for more complex row-by-row operations.

🔹 **Syntax (PL/SQL):**

```
DECLARE
  CURSOR emp_cursor IS
    SELECT emp_id, emp_name FROM employees;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('ID: ' || emp_record.emp_id || ', Name: ' || emp_record.emp_name);
  END LOOP;
  CLOSE emp_cursor;
END;
```

🔹 **Use Cases:**

- Row-by-row processing.
- Complex logic on each row.

📌 2. Stored Procedures

🔹 Definition:

A **stored procedure** is a precompiled collection of SQL statements and control-flow logic, stored in the database and executed on demand.

🔹 Syntax (MySQL example):

```
DELIMITER //

CREATE PROCEDURE GetEmployeeDetails()

BEGIN

   SELECT * FROM employees;

END;

//

DELIMITER ;
```

🔹 Call the Procedure:

```
CALL GetEmployeeDetails();
```

🔹 Features:

- Reusability
- Improved performance (precompiled)
- Security (control access)

📌 3. Stored Functions

🔹 Definition:

A **stored function** is similar to a procedure but **returns a value**. It can be used in SQL statements like SELECT.

🔹 Syntax (MySQL example):

```
DELIMITER //

CREATE FUNCTION GetTotalSalary(dept_id INT) RETURNS DECIMAL(10,2)

BEGIN

    DECLARE total_salary DECIMAL(10,2);

    SELECT SUM(salary) INTO total_salary FROM employees WHERE department_id = dept_id;

    RETURN total_salary;

END;

//

DELIMITER ;
```

🔹 Use the Function:

```
SELECT GetTotalSalary(101);
```

🔹 Differences Between Procedure and Function:

Feature	Stored Procedure	Stored Function
Returns a value	Optional	Mandatory
Used in SQL	Cannot be used in SQL	Can be used in SQL
Invocation	CALL statement	SELECT or within query

📌 4. Database Triggers

🔹 Definition:

A **trigger** is a stored program that is automatically executed or fired when a specific event occurs on a table (INSERT, UPDATE, DELETE).

🔹 Types:

- **BEFORE Trigger**
- **AFTER Trigger**
- **INSTEAD OF Trigger** (used in views)

🔹 Syntax (MySQL example):

```
CREATE TRIGGER before_employee_insert
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
  SET NEW.hire_date = CURDATE();
END;
```

🔹 Use Cases:

- Audit logging
- Enforcing business rules
- Automatically updating values

🧠 Summary Table:

Concept	Key Purpose	Example Usage
Cursor	Row-by-row processing	Processing employee records
Stored Procedure	Encapsulate logic, reusable actions	Get list of employees
Stored Function	Compute and return a single value	Calculate total salary in department
Trigger	React to table events automatically	Log insert, validate before insert