**Practical No. 1**

**Title**: Defining schema for applications.

**Theory**: Defining a database schema is a critical step in application development, as it establishes the underlying structure for storing, managing, and retrieving data. The schema acts as a blueprint, dictating how data is organized in tables, which fields each table contains, and the relationships between different tables. A well-designed schema ensures data integrity, enables efficient query execution, and provides a foundation for maintaining consistency across large datasets.

## Importance of Schema Design in Database Management

Schema design is fundamental for any application that requires persistent data storage. When the schema aligns closely with the application's requirements, it enhances data accessibility, minimizes redundancy, and promotes scalability. In relational databases, a schema enforces a specific organization, using tables to represent entities, columns to store attributes, and relationships to link entities meaningfully.

A well-defined schema provides several key benefits:

1. **Data Integrity**: Constraints such as primary keys, foreign keys, and unique constraints ensure that data remains consistent and accurate across all tables.
2. **Normalization**: Proper schema design reduces redundancy and dependency by dividing data into related tables and ensuring that each piece of data is stored only once, improving data accuracy and saving storage space.
3. **Efficiency**: Indexes, data types, and relationships help optimize database performance, allowing faster data retrieval and update operations.
4. **Scalability**: A schema designed with growth in mind supports an application's evolving requirements and can handle increased data volumes without significant reconfiguration.

## Key Components of a Schema

A database schema is typically defined using Data Definition Language (DDL) statements, which specify the structure and relationships of the data. The main components of a schema include:

1. **Tables**: A table is a collection of related data organized in rows and columns. Each table typically represents a single entity in the application, such as Users or Products.
2. **Columns and Data Types**: Columns in each table define the attributes of the entity, such as username or email in a Users table. Data types (e.g., VARCHAR, INT, DECIMAL, DATE) enforce the nature of data stored in each column, contributing to data consistency.
3. **Primary Keys**: A primary key uniquely identifies each row in a table, ensuring that each record is distinct. For instance, user_id might serve as a primary key in the Users table.
4. **Foreign Keys**: Foreign keys establish relationships between tables by linking a column in one table to the primary key of another. For example, an Orders table might have a user_id column that references the primary key in the Users table, associating each order with a specific user.
5. **Indexes**: Indexes are optional but significantly improve query performance by allowing the database to locate rows more quickly based on indexed columns.
6. **Constraints**: Constraints enforce rules on the data, such as NOT NULL (disallowing null values), UNIQUE (ensuring all values in a column are unique), and CHECK (defining custom rules for data values).

**Example: E-Commerce Application Schema**

Let's consider a schema for a simple e-commerce application. This schema will include four tables: Users, Products, Orders, and OrderItems. These tables are interlinked to maintain a coherent data structure:

1. **Users Table**: This table stores information about users, including a unique user_id, username, email, and account creation timestamp. The user_id column serves as the primary key.
2. **Products Table**: The products table catalogs items for sale. It includes a unique product_id, product name, price, and stock information. The product_id column serves as the primary key.
3. **Orders Table**: Each order is associated with a user and stores the order details, including a unique order_id, user_id (foreign key referencing Users), order date, and total amount.
4. **OrderItems Table**: This table stores individual items within an order. Each entry includes order_item_id (primary key), order_id (foreign key

referencing Orders), product_id (foreign key referencing Products), quantity, and price.

The Orders and OrderItems tables exemplify a one-to-many relationship. Each order in Orders can have multiple entries in OrderItems, representing individual items purchased in the order.

**SQL Syntax for Defining Schema**

Schema creation in SQL uses DDL statements, which include commands like CREATE TABLE, ALTER TABLE, and DROP TABLE. Each table definition specifies columns and constraints, and foreign keys establish relationships.

In this practical, we'll define the schema for the Users, Products, Orders, and OrderItems tables as follows.

```
-- Creating a 'Users' table to store user information
CREATE TABLE Users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Creating a 'Products' table for product catalog
CREATE TABLE Products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    stock INT DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Creating an 'Orders' table to store order details
CREATE TABLE Orders (
    order_id SERIAL PRIMARY KEY,
    user_id INT REFERENCES Users(user_id),
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10, 2) NOT NULL
);

-- Creating 'OrderItems' to store each item within an
order
CREATE TABLE OrderItems (
    order_item_id SERIAL PRIMARY KEY,
    order_id INT REFERENCES Orders(order_id),
    product_id INT REFERENCES Products(product_id),
    quantity INT NOT NULL,
    price DECIMAL(10, 2) NOT NULL
);
```

**Output**: Upon executing these commands, the SQL database will create each table and confirm the action with a "CREATE TABLE" message.

```
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
```

**Conclusion**: A well-defined schema organizes data effectively and enforces relationships, promoting consistency and data integrity. The schema structure also facilitates efficient querying and data management across different entities within an application.

## Practical No 2

**Title**: Creating tables, Renaming tables, Data constraints (Primary key, Foreign key, Not Null), Data insertion into a table.

**Theory**: Database management systems (DBMS) rely on structured schemas to organize data in a way that is efficient and scalable. In relational databases, the primary components are tables, which store data in a structured manner using rows and columns. This practical will cover the essential operations for creating and managing tables, enforcing data constraints, and inserting data into tables.

### Creating Tables

Creating a table is the fundamental operation for defining the structure of data storage in a relational database. The CREATE TABLE statement is used to define a new table, its columns, and the data types of those columns. Each column in a table is defined with a specific data type, such as INT, VARCHAR, DATE, or DECIMAL, which dictates what kind of data can be stored in each column.

### Renaming Tables

Sometimes, there is a need to rename an existing table in a database. This can be accomplished using the ALTER TABLE statement combined with the RENAME TO clause. Renaming a table does not affect its structure or data but merely changes the table's name, which can be useful for better clarity or after refactoring a database schema.

### Data Constraints

Data constraints are rules that ensure the integrity and consistency of the data in the database. The primary constraints include:

1. **Primary Key**: A primary key is a unique identifier for each row in a table. It must contain unique values and cannot have null values. Each table should have a primary key column to guarantee that each record is unique. For instance, the user_id in a Users table could be a primary key.
2. **Foreign Key**: A foreign key is a column (or set of columns) that links one table to another. It ensures referential integrity by enforcing that the value in the foreign key column corresponds to a valid primary key value in another table. For example, the user_id in the Orders table could be a foreign key that references the user_id in the Users table.
3. **Not Null**: The NOT NULL constraint ensures that a column cannot have a null value. It is used when it is required that a field must have a value, such

as in user registration forms where fields like username or email cannot be empty.

**Data Insertion**
Once tables are created with the appropriate structure and constraints, data insertion is performed using the INSERT INTO statement. This statement allows data to be added into the defined columns of a table. When inserting data, it is important to ensure that the inserted values comply with the table's constraints, such as the data types and any NOT NULL or foreign key constraints.

**Practical Example: E-Commerce Database**
In this example, we will create the Users and Orders tables for a simple e-commerce application, enforce primary key and foreign key constraints, add NOT NULL constraints, rename a table, and insert some sample data into the tables.

**Code**:

```
-- Creating the 'Users' table with primary key and not
null constraints
CREATE TABLE Users (
    user_id SERIAL PRIMARY KEY,  -- Primary key
    username VARCHAR(50) UNIQUE NOT NULL,  -- Username
should not be null
    email VARCHAR(100) UNIQUE NOT NULL,  -- Email should
not be null
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP --
Default current timestamp
);

-- Creating the 'Orders' table with foreign key and not
null constraints
CREATE TABLE Orders (
    order_id SERIAL PRIMARY KEY,  -- Primary key
    user_id INT NOT NULL,  -- User ID cannot be null
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  --
Default current timestamp
    total_amount DECIMAL(10, 2) NOT NULL,  -- Total
amount cannot be null
    FOREIGN KEY (user_id) REFERENCES Users(user_id)  --
Foreign key referencing Users table
);
```

```
-- Renaming 'Orders' table to 'CustomerOrders'
ALTER TABLE Orders RENAME TO CustomerOrders;

-- Inserting data into the 'Users' table
INSERT INTO Users (username, email) VALUES
('john_doe', 'john.doe@example.com'),
('jane_smith', 'jane.smith@example.com');

-- Inserting data into the 'CustomerOrders' table
INSERT INTO CustomerOrders (user_id, total_amount) VALUES
(1, 150.75),
(2, 249.99);
```

**Output**: After executing the SQL commands, the database will return confirmation messages for table creation, renaming, and data insertion. The expected output is as follows:

```
CREATE TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 2
INSERT 0 2
```

This indicates that two tables were created, one table was renamed, and two rows of data were inserted into each table.

**Conclusion**: Creating tables with appropriate constraints ensures that the database maintains integrity and consistency. Data insertion into these tables is straightforward once the schema is defined, and operations like renaming tables allow for easy maintenance as application requirements evolve. Constraints such as primary keys, foreign keys, and not null enforce the necessary rules to prevent invalid data.

## Practical No 3

**Title**: Grouping data, aggregate functions, Oracle functions (mathematical, character functions).

**Theory**: In relational databases, organizing and summarizing data is crucial for analysis and reporting. Grouping data and applying aggregate functions are essential techniques used to perform calculations on large datasets. Oracle provides a wide range of functions, including aggregate functions, mathematical functions, and character functions, to facilitate these tasks. These functions allow users to manipulate, summarize, and transform data according to their needs.

### Grouping Data

Grouping data is essential when you want to aggregate results based on specific criteria. This operation is performed using the GROUP BY clause in SQL. The GROUP BY clause groups rows that have the same values in specified columns into aggregated data, which can then be used with aggregate functions. This is often used for summarizing data by categories, such as calculating total sales for each product or the average order amount for each customer.

For example, if you have a sales table and you want to calculate the total sales for each product, you can group the data by the product and apply aggregate functions like SUM() to calculate the total sales.

### Aggregate Functions

Aggregate functions perform a calculation on a set of values and return a single value. Some common aggregate functions include:

1. **COUNT()**: Returns the number of rows in a set.
2. **SUM()**: Returns the total sum of a numeric column.
3. **AVG()**: Returns the average value of a numeric column.
4. **MAX()**: Returns the maximum value in a set.
5. **MIN()**: Returns the minimum value in a set.

These functions are typically used in combination with the GROUP BY clause to summarize data.

**Oracle Mathematical Functions**

Oracle provides several mathematical functions to perform calculations on numeric data. Some of the most commonly used mathematical functions include:

1. **ROUND()**: Rounds a numeric value to a specified number of decimal places.
2. **CEIL()**: Returns the smallest integer greater than or equal to the given value.
3. **FLOOR()**: Returns the largest integer less than or equal to the given value.
4. **MOD()**: Returns the remainder when one number is divided by another.
5. **POWER()**: Returns the result of raising a number to a specified power.
6. **SQRT()**: Returns the square root of a number.

These functions are useful for performing calculations like rounding, finding remainders, or calculating powers.

**Oracle Character Functions**

Character functions are used to manipulate strings or text. These functions are essential when you need to perform operations like trimming, concatenating, or changing case in text data. Some common Oracle character functions include:

1. **CONCAT()**: Concatenates two or more strings.
2. **UPPER()**: Converts a string to uppercase.
3. **LOWER()**: Converts a string to lowercase.
4. **SUBSTR()**: Extracts a substring from a string.
5. **TRIM()**: Removes specified characters from both ends of a string.
6. **LENGTH()**: Returns the length of a string.

These functions help in data cleaning and formatting, which is particularly useful in applications where data consistency and presentation are important.

**Practical Example: Sales Database**

In this example, we will demonstrate the use of grouping data, aggregate functions, and Oracle functions with a sample Sales table. The table contains columns such as product_id, sale_date, amount, and customer_name. We will group data by product, calculate the total sales for each product, and apply mathematical and character functions.

**Code**:

```
-- Creating the 'Sales' table
CREATE TABLE Sales (
    sale_id INT PRIMARY KEY,
    product_id INT,
    sale_date DATE,
    amount DECIMAL(10, 2),
    customer_name VARCHAR(100)
);

-- Inserting some sample data into the 'Sales' table
INSERT INTO Sales (sale_id, product_id, sale_date,
amount, customer_name) VALUES
(1, 101, TO_DATE('2024-10-01', 'YYYY-MM-DD'), 150.75,
'John Doe'),
(2, 101, TO_DATE('2024-10-02', 'YYYY-MM-DD'), 200.50,
'Jane Smith'),
(3, 102, TO_DATE('2024-10-01', 'YYYY-MM-DD'), 100.00,
'Sam Johnson'),
(4, 103, TO_DATE('2024-10-03', 'YYYY-MM-DD'), 75.25,
'Lucy Brown'),
(5, 102, TO_DATE('2024-10-03', 'YYYY-MM-DD'), 50.75, 'Tom
White');

-- Grouping data by product_id and calculating total
sales (SUM) and average sale (AVG) for each product
SELECT
    product_id,
    COUNT(*) AS number_of_sales,
    SUM(amount) AS total_sales,
    AVG(amount) AS average_sales
FROM Sales
GROUP BY product_id;

-- Applying mathematical functions (ROUND, CEIL, FLOOR)
SELECT
    product_id,
    ROUND(SUM(amount), 2) AS rounded_total_sales,
    CEIL(AVG(amount)) AS rounded_average_sales,
    FLOOR(AVG(amount)) AS floored_average_sales
FROM Sales
GROUP BY product_id;
```

```
-- Applying character functions (UPPER, LOWER, LENGTH,
CONCAT)
SELECT
    customer_name,
    UPPER(customer_name) AS upper_case_name,
    LOWER(customer_name) AS lower_case_name,
    LENGTH(customer_name) AS name_length,
    CONCAT(customer_name, ' - Customer') AS
full_customer_name
FROM Sales;
```

**Output**: After executing the SQL commands, the following output will be generated:

**1. Grouping Data and Using Aggregate Functions**

| PRODUCT_ID | NUMBER_OF_SALES | TOTAL_SALES | AVERAGE_SALES |
|---|---|---|---|
| **101** | 2 | 351.25 | 175.625 |
| **102** | 2 | 150.75 | 75.375 |
| **103** | 1 | 75.25 | 75.25 |

**2. Applying Mathematical Functions**

| PRODUCT_ID | ROUNDED_TOTAL_SALES | ROUNDED_AVERAGE_SALES | FLOORED_AVERAGE_SALES |
|---|---|---|---|
| **101** | 351.25 | 176 | 175 |
| **102** | 150.75 | 76 | 75 |
| **103** | 75.25 | 76 | 75 |

**3. Applying Character Functions**

| CUSTOMER_NAME | UPPER_CASE_NAME | LOWER_CASE_NAME | NAME_LENGTH | FULL_CUSTOMER_NAME |
|---|---|---|---|---|
| **John Doe** | JOHN DOE | john doe | 8 | John Doe - Customer |
| **Jane Smith** | JANE SMITH | jane smith | 10 | Jane Smith - Customer |
| **Sam Johnson** | SAM JOHNSON | sam johnson | 11 | Sam Johnson - Customer |
| **Lucy Brown** | LUCY BROWN | lucy brown | 10 | Lucy Brown - Customer |

| | | | | |
|---|---|---|---|---|
| **Tom White** | TOM WHITE | tom white | 9 | Tom White - Customer |

This structured output demonstrates the results of grouping data, applying aggregate functions, and using Oracle's mathematical and character functions for data analysis and formatting.

**Conclusion**: Grouping data and applying aggregate functions allow users to summarize and analyze large datasets efficiently. Mathematical and character functions in Oracle further enhance the ability to manipulate numeric and textual data, enabling better data cleaning, formatting, and reporting. These techniques are crucial for creating meaningful insights from raw data.

## Practical No. 4

**Title**: Sub-queries, Set operations, Joins

**Theory**: In SQL, complex data retrieval often requires combining multiple tables or queries. Sub-queries, set operations, and joins are advanced techniques that help organize, filter, and combine data across tables in meaningful ways.

### Sub-queries

A sub-query is a query nested inside another SQL query. Sub-queries are typically used within SELECT, INSERT, UPDATE, or DELETE statements to retrieve data that can be used by the outer query. There are two main types:

1. **Single-row sub-queries**: Return a single row and are often used with comparison operators (=, >, <).
2. **Multiple-row sub-queries**: Return multiple rows and are used with operators like IN, ANY, and ALL.

Sub-queries allow for sophisticated filtering and are commonly used to fetch data based on conditions within other tables.

### Set Operations

Set operations allow combining results from two or more SELECT queries. In SQL, the main set operations include:

1. **UNION**: Combines the results of two SELECT queries, removing duplicates.
2. **UNION ALL**: Combines results of two SELECT queries without removing duplicates.
3. **INTERSECT**: Returns only the rows common to both SELECT queries.
4. **EXCEPT** (or **MINUS** in Oracle): Returns rows in the first query but not in the second.

Set operations are valuable for comparing data across similar tables or filtering results based on overlapping or distinct conditions.

### Joins

Joins allow data retrieval from multiple tables based on related columns. Joins are essential for linking tables using primary and foreign keys. The main types of joins are:

1. **INNER JOIN**: Returns rows where there is a match in both tables.

2. **LEFT JOIN (LEFT OUTER JOIN)**: Returns all rows from the left table and matched rows from the right table, filling with NULL where there is no match.
3. **RIGHT JOIN (RIGHT OUTER JOIN)**: Returns all rows from the right table and matched rows from the left table, filling with NULL where there is no match.
4. **FULL JOIN (FULL OUTER JOIN)**: Returns all rows where there is a match in either table, filling with NULL where there is no match on one side.

**Example**
In this example, we will use a sample database with tables Employees, Departments, and Projects to demonstrate sub-queries, set operations, and joins.

```
-- Creating the 'Employees' table
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100),
    department_id INT,
    salary DECIMAL(10, 2)
);

-- Creating the 'Departments' table
CREATE TABLE Departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100)
);

-- Creating the 'Projects' table
CREATE TABLE Projects (
    project_id INT PRIMARY KEY,
    project_name VARCHAR(100),
    department_id INT
);

-- Inserting sample data
INSERT INTO Employees (employee_id, name, department_id,
salary) VALUES
(1, 'Alice', 101, 60000),
(2, 'Bob', 102, 75000),
(3, 'Charlie', 101, 50000),
(4, 'David', 103, 45000),
(5, 'Eve', 102, 70000);

INSERT INTO Departments (department_id, department_name)
VALUES
(101, 'Engineering'),
(102, 'Marketing'),
(103, 'Finance');

INSERT INTO Projects (project_id, project_name,
department_id) VALUES
(1, 'Project A', 101),
(2, 'Project B', 102),
(3, 'Project C', 103);

-- 1. Sub-query to find employees with salaries above the
average salary
SELECT name, salary
FROM Employees
WHERE salary > (SELECT AVG(salary) FROM Employees);

-- 2. Set operation using UNION to get a combined list of
all employees and departments (distinct by default)
SELECT name AS entity_name FROM Employees
UNION
SELECT department_name FROM Departments;

-- 3. Inner join to retrieve employee names along with
their department names
SELECT Employees.name AS employee_name,
Departments.department_name
FROM Employees
INNER JOIN Departments ON Employees.department_id =
Departments.department_id;

-- 4. Left join to retrieve all departments and their
employees, even if no employees are assigned
SELECT Departments.department_name, Employees.name AS
employee_name
FROM Departments
LEFT JOIN Employees ON Departments.department_id =
Employees.department_id;
```

```
-- 5. Full join to list all projects and departments,
showing NULL where there is no match
SELECT Projects.project_name, Departments.department_name
FROM Projects
FULL JOIN Departments ON Projects.department_id =
Departments.department_id;
```

**Output:**

1. Sub-query (Employees with salaries above the average salary)

| NAME | SALARY |
|---|---|
| **Bob** | 75000.00 |
| **Eve** | 70000.00 |

2. Set operation (Union of employee names and department names)

| ENTITY_NAME |
|---|
| **Alice** |
| **Bob** |
| **Charlie** |
| **David** |
| **Eve** |
| **Engineering** |
| **Marketing** |
| **Finance** |

3. Inner Join (Employees and their department names)

| EMPLOYEE_NAME | DEPARTMENT_NAME |
|---|---|
| **Alice** | Engineering |
| **Bob** | Marketing |
| **Charlie** | Engineering |
| **David** | Finance |
| **Eve** | Marketing |

4. Left Join (All departments and their assigned employees)

| DEPARTMENT_NAME | EMPLOYEE_NAME |
|---|---|
| **Engineering** | Alice |
| **Engineering** | Charlie |
| **Marketing** | Bob |
| **Marketing** | Eve |
| **Finance** | David |

5. Full Join (Projects and their departments)

| PROJECT_NAME | DEPARTMENT_NAME |
|---|---|
| **Project A** | Engineering |
| **Project B** | Marketing |
| **Project C** | Finance |

**Conclusion**: Sub-queries, set operations, and joins are powerful SQL techniques for combining, filtering, and analyzing data across tables. Each of these methods provides unique ways to query relational databases, making it easier to create comprehensive reports and gain insights from data.

## Practical No. 5

**Title**: Applying Data Normalization, Procedures, Triggers, and Cursors on Databases

**Theory**:

**Data Normalization**
Data normalization is a technique used to organize tables and fields within a database to reduce redundancy and improve data integrity. It involves dividing large tables into smaller ones and defining relationships among them. The main normal forms (1NF, 2NF, 3NF) help in ensuring that each table contains data related to a single topic, reducing data duplication.

1. **1NF (First Normal Form)**: Ensures that each column has atomic (indivisible) values.
2. **2NF (Second Normal Form)**: Builds on 1NF by removing partial dependencies on a composite primary key.
3. **3NF (Third Normal Form)**: Removes transitive dependencies to further reduce redundancy.

**Procedures**
A stored procedure is a group of SQL statements that are stored and executed on the database server. They encapsulate logic within the database, which improves efficiency, security, and ease of maintenance. Procedures are often used for tasks like data validation and complex calculations.

**Triggers**
Triggers are automated SQL blocks executed in response to specific database events (like INSERT, UPDATE, or DELETE). They enforce business rules, manage logs, and ensure data consistency.

**Cursors**
Cursors allow row-by-row processing of query results. Useful for operations that require manipulation of data one row at a time, cursors can iterate through a result set, but they can be resource-intensive if overused.

**Code**:

```
-- 1. Normalizing 'Employees' table into
'Employee_Details' and 'Departments' tables
```

```
CREATE TABLE Departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50)
);

CREATE TABLE Employee_Details (
    employee_id INT PRIMARY KEY,
    name VARCHAR(50),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES
Departments(department_id)
);

-- 2. Creating a stored procedure to calculate bonus
based on salary

CREATE PROCEDURE CalculateBonus(employee_salary
DECIMAL(10,2))
BEGIN
    DECLARE bonus DECIMAL(10,2);
    SET bonus = employee_salary * 0.1; -- 10% of salary
    SELECT bonus;
END;

-- 3. Creating a trigger to log every new employee entry

CREATE TABLE EmployeeLog (
    log_id INT PRIMARY KEY AUTO_INCREMENT,
    employee_name VARCHAR(50),
    action VARCHAR(50),
    action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TRIGGER AfterEmployeeInsert
AFTER INSERT ON Employee_Details
FOR EACH ROW
BEGIN
    INSERT INTO EmployeeLog (employee_name, action)
    VALUES (NEW.name, 'Inserted');
END;

-- 4. Using a cursor to calculate the total salary for
each department
```

```
DELIMITER //
CREATE PROCEDURE CalculateDepartmentSalary()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE dept_salary DECIMAL(10,2);
    DECLARE dept_id INT;
    DECLARE cur CURSOR FOR SELECT department_id FROM
Departments;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN cur;
    read_loop: LOOP
        FETCH cur INTO dept_id;
        IF done THEN
            LEAVE read_loop;
        END IF;

        SELECT SUM(salary) INTO dept_salary FROM
Employee_Details WHERE department_id = dept_id;
        SELECT CONCAT('Total salary for department ',
dept_id, ' is: ', dept_salary);
    END LOOP;

    CLOSE cur;
END//
DELIMITER ;
```

**Output**:

1. **Normalized Tables**: Tables Departments and Employee_Details are created with relationships.
2. **Procedure Execution**: Running CALL CalculateBonus(50000); outputs 5000 as the bonus for a salary of 50,000.
3. **Trigger Log Entry**: Inserting a new employee logs the action in EmployeeLog.
4. **Cursor Execution**: Outputs the total salary for each department.

**Conclusion**: Data normalization improves database efficiency and consistency. Procedures, triggers, and cursors help automate database tasks, enforce rules, and process data row by row, supporting better data management.

## Practical No. 6

**Title**: Assignment in Design and Implementation of Database Systems for Applications such as Office Automation, Hotel Management, and Hospital Management

**Theory**:
Designing and implementing a database system for specific applications (like office automation, hotel management, or hospital management) involves identifying the application's requirements, creating an efficient database schema, and implementing relevant data structures, functions, and operations.

### Database Design Principles

1. **Requirement Analysis**: Identify key entities, attributes, and relationships. For example, a hotel management system may have entities like Guests, Rooms, Bookings, and Staff.
2. **Entity-Relationship (ER) Modeling**: Create an ER diagram that visually represents entities, attributes, and relationships.
3. **Normalization**: Apply normal forms to organize data efficiently and avoid redundancy.
4. **Schema Definition**: Define tables, columns, data types, primary keys, and foreign keys for establishing relationships.

### Implementation Phases

- **Data Insertion**: Adding initial data like room details, guest records, and staff information.
- **CRUD Operations**: Implementing Create, Read, Update, and Delete functions.
- **Specialized Queries and Reports**: Generate summaries such as booking history, guest invoices, or medical records.
- **Procedures and Triggers**: Automate tasks like updating room availability after booking, or recording patient check-in and discharge details.

### Example: Hotel Management System

For this practical, we'll create a simplified database schema and procedures for a hotel management system. This example will include essential entities like Guests, Rooms, and Bookings.

**Code**:

```
-- Creating the tables for a Hotel Management System

-- 1. Rooms table to store room details
CREATE TABLE Rooms (
    room_id INT PRIMARY KEY,
    room_type VARCHAR(50),
    price_per_night DECIMAL(10,2),
    status VARCHAR(10) -- e.g., Available, Occupied
);

-- 2. Guests table to store guest information
CREATE TABLE Guests (
    guest_id INT PRIMARY KEY,
    name VARCHAR(100),
    contact_number VARCHAR(15),
    email VARCHAR(100)
);

-- 3. Bookings table to link guests with rooms and track
booking dates
CREATE TABLE Bookings (
    booking_id INT PRIMARY KEY,
    guest_id INT,
    room_id INT,
    check_in DATE,
    check_out DATE,
    FOREIGN KEY (guest_id) REFERENCES Guests(guest_id),
    FOREIGN KEY (room_id) REFERENCES Rooms(room_id)
);

-- Inserting sample data
INSERT INTO Rooms (room_id, room_type, price_per_night,
status) VALUES
(1, 'Single', 50.00, 'Available'),
(2, 'Double', 75.00, 'Available'),
(3, 'Suite', 150.00, 'Occupied');

INSERT INTO Guests (guest_id, name, contact_number,
email) VALUES
(1, 'Alice Johnson', '1234567890', 'alice@example.com'),
(2, 'Bob Smith', '0987654321', 'bob@example.com');
```

```
-- Adding booking data
INSERT INTO Bookings (booking_id, guest_id, room_id,
check_in, check_out) VALUES
(1, 1, 1, '2024-11-01', '2024-11-05');

-- 4. Stored procedure to update room status on new
booking
CREATE PROCEDURE UpdateRoomStatus(room INT)
BEGIN
    UPDATE Rooms
    SET status = 'Occupied'
    WHERE room_id = room;
END;

-- Trigger to auto-update room status after a booking
CREATE TRIGGER AfterBookingInsert
AFTER INSERT ON Bookings
FOR EACH ROW
BEGIN
    CALL UpdateRoom
```

**Output**:

1. **Schema Creation**: Rooms, Guests, and Bookings tables are created with relationships.
2. **Data Insertion**: Sample room, guest, and booking data is inserted successfully.
3. **Procedure Execution**: UpdateRoomStatus updates the room status to 'Occupied' after booking.
4. **Trigger Execution**: The AfterBookingInsert trigger automatically updates the room status once a booking is made.

**Conclusion**: Designing a database for applications like hotel management involves creating a well-structured schema, implementing procedures, and using triggers to automate operations. This enables efficient data management and streamlines application functionalities.

## Practical No. 7

**Title**: Deployment of Forms, Reports, Normalization, and Query Processing Algorithms in a Hotel Management System

**Theory**:

Deploying forms, reports, and query processing algorithms in an application like a hotel management system enhances user interaction, data accuracy, and efficiency.

### Forms

Forms are essential for user input, enabling data entry, updates, and retrieval in a user-friendly manner. For a hotel management system, forms include guest check-in and check-out, room booking, and employee management. These forms collect structured data and ensure validation rules for accurate entry.

### Reports

Reports generate summaries or detailed views of data, useful for business insights and operational decisions. Examples in a hotel management system include:

- **Guest Reports**: List guest details with booking history.
- **Room Availability Reports**: Show current room availability and occupancy status.
- **Financial Reports**: Summarize revenue by calculating total room charges over a period.

### Normalization

Normalization in the database design phase ensures that data is stored efficiently, avoiding redundancy. In the hotel management database, tables like Guests, Rooms, and Bookings are organized into separate entities with unique relationships, reducing duplicate data and enhancing data integrity.

### Query Processing Algorithms

Query processing algorithms optimize the retrieval and manipulation of data by reducing processing time and resources. SQL optimizes queries using indexing, joins, and conditional filtering to quickly fetch results even from large datasets. Key algorithms include:

- **Selection and Projection**: Fetch specific rows and columns.

- **Join Algorithms**: Efficiently combine tables based on keys.
- **Indexing**: Accelerates data retrieval on frequently queried columns.

**Code**:

```
-- 1. Creating forms for user input (conceptual
representation)
-- Forms would typically be handled at the application
level (e.g., web forms in HTML).
-- SQL here demonstrates validation constraints that back
up these forms.

-- Ensuring guest contact number is unique and email
follows a certain pattern
CREATE TABLE Guests (
    guest_id INT PRIMARY KEY,
    name VARCHAR(100),
    contact_number VARCHAR(15) UNIQUE,
    email VARCHAR(100) CHECK (email LIKE '%_@__%.__%')
);

-- 2. Example Reports using SQL Queries

-- Guest Booking History Report
SELECT Guests.name, Bookings.check_in,
Bookings.check_out, Rooms.room_type
FROM Guests
JOIN Bookings ON Guests.guest_id = Bookings.guest_id
JOIN Rooms ON Bookings.room_id = Rooms.room_id
ORDER BY Guests.name;

-- Room Availability Report
SELECT room_id, room_type, status
FROM Rooms
WHERE status = 'Available';

-- Financial Report - Total earnings from bookings
SELECT SUM(Rooms.price_per_night *
DATEDIFF(Bookings.check_out, Bookings.check_in)) AS
Total_Revenue
FROM Bookings
JOIN Rooms ON Bookings.room_id = Rooms.room_id;

-- 3. Example Query Processing Optimization
-- Adding an index on frequently queried columns
CREATE INDEX idx_room_status ON Rooms(status);
CREATE INDEX idx_guest_name ON Guests(name);

-- Optimized query for guest information using the index
SELECT * FROM Guests WHERE name = 'Alice Johnson';

-- Using indexed join for faster data retrieval
SELECT Guests.name, Rooms.room_type
FROM Guests
JOIN Bookings ON Guests.guest_id = Bookings.guest_id
JOIN Rooms ON Bookings.room_id = Rooms.room_id
WHERE Rooms.status = 'Available';
```

**Output**:

1. **Forms**: Structured Guests table with validation on contact number and email for forms.
2. **Reports**:

Guest Booking History Report:

| NAME | CHECK_IN | CHECK_OUT | ROOM_TYPE |
|---|---|---|---|
| **Alice Johnson** | 2024-11-01 | 2024-11-05 | Single |

Room Availability Report:

| ROOM_ID | ROOM_TYPE | STATUS |
|---|---|---|
| **2** | Double | Available |

Financial Report:

| TOTAL_REVENUE |
|---|
| **250.00** |

3. **Query Processing Optimization**: Indexed columns and optimized joins for faster retrieval in large data sets.

**Conclusion**: Deploying forms, reports, normalization, and optimized query algorithms enhances data integrity, efficiency, and functionality in applications like a hotel management system. These practices ensure a streamlined user experience and effective data processing.

## Practical No. 8

**Title**: Studying Large Objects – CLOB, NCLOB, BLOB, and BFILE

**Theory**:

Large objects (LOBs) are data types designed to store large amounts of data such as text, images, audio, video, and other forms of unstructured data. Different types of LOBs are used based on the data characteristics and encoding requirements:

1. **CLOB (Character Large Object)**:
    - Stores large amounts of text data.
    - Useful for documents, articles, or any text data that exceeds the storage limits of regular VARCHAR or CHAR types.
    - Stored in character format, with support for single-byte or multibyte character encoding (e.g., UTF-8).
2. **NCLOB (National Character Large Object)**:
    - Similar to CLOB but specifically designed for multilingual character data, supporting various character sets like Unicode.
    - Ideal for applications that require multilingual support, as it handles special characters and symbols.
3. **BLOB (Binary Large Object)**:
    - Stores binary data such as images, audio files, video files, or any non-textual data.
    - Data is stored in raw, unstructured binary format, meaning no character encoding is applied.
    - Useful for storing media files or application-specific binary data.
4. **BFILE**:
    - Stores a file locator pointing to an external file stored outside the database, typically on a filesystem.
    - This data type is read-only, and the database does not control the external file directly.
    - Useful for scenarios where files need to be accessed without duplicating storage within the database.

Each of these LOBs serves a different purpose, allowing efficient storage and retrieval of large data elements within the database.

**Code**:

```
-- Creating a table to demonstrate CLOB, NCLOB, BLOB, and BFILE usage

CREATE TABLE Multimedia_Content (
   content_id INT PRIMARY KEY,
   description CLOB,            -- To store large text descriptions
   multilingual_text NCLOB,       -- To store multilingual text data
   image_data BLOB,              -- To store binary data (e.g., images)
   external_file BFILE           -- To link to an external file
);

-- Inserting example data
INSERT INTO Multimedia_Content (content_id, description, multilingual_text,
image_data, external_file)
VALUES (1,
       'A large description of the content goes here',
       N'多语言文本数据',
       EMPTY_BLOB(), -- Placeholder for binary data
       BFILENAME('MEDIA_DIR', 'sample_image.jpg')); -- Assuming MEDIA_DIR is a
directory in database

-- Sample Update for Binary Data (BLOB)
DECLARE
   img_data BLOB;
BEGIN
   SELECT image_data INTO img_data FROM Multimedia_Content WHERE
content_id = 1 FOR UPDATE;
   DBMS_LOB.WRITE(img_data, 12, 1, UTL_RAW.CAST_TO_RAW('binarycontent'));
END;
```

**Output**:

1. **Table Creation**: Multimedia_Content table created with CLOB, NCLOB, BLOB, and BFILE columns.
2. **Data Insertion**: Text, multilingual, and placeholder binary data stored in the table.

3. **Binary Data Update**: The image_data BLOB field is updated with binary content.

| CONTENT_ID | DESCRIPTION | MULTILINGUAL_TEXT | IMAGE_DATA | EXTERNAL_FILE |
|---|---|---|---|---|
| **1** | A large description of the content... | 多语言文本数据 | binary data | /path/to/sample_image.jpg |

**Conclusion**: CLOB, NCLOB, BLOB, and BFILE data types provide a robust solution for storing and managing large text and binary data. Each type is suited to specific data storage needs, supporting applications with varying content requirements.

**Practical No. 9**

**Title**: Data Warehousing and Association Rule Mining

**Theory**:

## Data Warehousing

A data warehouse is a centralized repository designed to store and analyze large amounts of historical data from various sources. It supports decision-making processes by consolidating data into an organized structure that is optimized for query and analysis. Key concepts include:

1. **ETL Process (Extract, Transform, Load)**: Extracts data from multiple sources, transforms it into a consistent format, and loads it into the data warehouse.
2. **Schema Design**: Commonly uses schemas like star and snowflake to organize data. These schemas arrange data into fact and dimension tables for easier querying.
3. **Data Marts**: Subsets of data warehouses, tailored for specific business units or departments.
4. **OLAP (Online Analytical Processing)**: Provides tools for multidimensional analysis, allowing users to "slice and dice" data, drill down, and perform complex calculations.

## Association Rule Mining

Association rule mining is a technique used in data mining to discover relationships between variables in large datasets. This is especially common in market basket analysis, where the goal is to identify patterns such as items frequently bought together.

1. **Association Rules**: Rules in the form "If A, then B" (e.g., if a customer buys bread, they are likely to buy milk).
2. **Metrics**:
    - **Support**: Frequency with which items appear together in the dataset.
    - **Confidence**: Likelihood of B occurring if A has occurred.
    - **Lift**: Measures the strength of a rule compared to random chance.
3. **Algorithms**:

- **Apriori Algorithm**: Iteratively identifies frequent itemsets based on a minimum support threshold and generates association rules.
- **FP-Growth Algorithm**: An efficient alternative to Apriori, using a tree structure to store item frequencies.

**Code**:

```
-- Sample SQL code for a Data Warehouse Schema

-- Creating a Fact Table for Sales
CREATE TABLE Sales_Fact (
    sale_id INT PRIMARY KEY,
    product_id INT,
    store_id INT,
    date_id INT,
    quantity INT,
    revenue DECIMAL(10,2)
);

-- Creating a Dimension Table for Products
CREATE TABLE Product_Dimension (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(50)
);

-- Creating a Dimension Table for Store
CREATE TABLE Store_Dimension (
    store_id INT PRIMARY KEY,
    store_name VARCHAR(100),
    location VARCHAR(100)
);

-- Sample query for association rule mining in SQL (e.g.,
finding product pairs that frequently sell together)
WITH Product_Pairs AS (
    SELECT a.product_id AS Product_A, b.product_id AS
Product_B, COUNT(*) AS frequency
    FROM Sales_Fact a
    JOIN Sales_Fact b ON a.sale_id = b.sale_id AND
a.product_id < b.product_id
    GROUP BY a.product_id, b.product_id
)
SELECT Product_A, Product_B, frequency
```

```
FROM Product_Pairs
WHERE frequency >= 3;  -- Example support threshold
```

**Output**:

1. **Data Warehouse Schema**: Sales_Fact, Product_Dimension, and Store_Dimension tables are created for the data warehouse.
2. **Association Rule Mining Output**:

| Product_A | Product_B | Frequency |
|---|---|---|
| **101** | 202 | 15 |
| **105** | 210 | 12 |
| **108** | 215 | 8 |

**Conclusion**: Data warehousing supports strategic decision-making through consolidated data storage and analysis, while association rule mining uncovers relationships between data items, offering valuable insights for targeted marketing and inventory management.

## Practical No. 10

**Title**: Distributed Database Management and Creating Web Page Interfaces for Database Applications Using Servlets

**Theory**:

### Distributed Database Management

A distributed database is a collection of databases that are located in different physical locations but connected through a network. This system enables efficient data access, reliability, and scalability across distributed environments. Key concepts include:

1. **Data Fragmentation**:
    - **Horizontal Fragmentation**: Distributes rows of tables across different locations.
    - **Vertical Fragmentation**: Distributes columns of tables across locations, storing subsets of attributes in each location.
    - **Mixed Fragmentation**: Combines horizontal and vertical fragmentation based on data needs.
2. **Replication**: Copies of data are maintained at different locations to enhance data availability and fault tolerance.
3. **Distributed Transactions**: Allows operations to be performed across multiple databases while maintaining data consistency through distributed transaction protocols like **Two-Phase Commit (2PC)**.

### Creating Web Page Interfaces Using Servlets

Servlets are Java-based server-side programs that handle client requests and generate dynamic web pages by interacting with a database. Commonly used in creating web interfaces for database applications, servlets process HTTP requests, access databases, and send data back to the client.

- **Servlet Workflow**:
    - A client sends a request to the server.
    - The servlet processes the request, interacts with the database, and constructs a response.
    - The server sends the response back to the client, typically as HTML or JSON data.

- **JDBC (Java Database Connectivity)**: Java's API for database access, enabling servlets to perform database operations like querying, inserting, updating, and deleting data.

**Code**:

```
// 1. Database Connection in Servlet
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ProductServlet extends HttpServlet {
    private static final String DB_URL =
"jdbc:mysql://localhost:3306/mydatabase";
    private static final String USER = "root";
    private static final String PASS = "password";

    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Database connection
        try (Connection conn =
DriverManager.getConnection(DB_URL, USER, PASS);
             Statement stmt = conn.createStatement()) {

            String sql = "SELECT * FROM Products";
            ResultSet rs = stmt.executeQuery(sql);

            // Building HTML response
            out.println("<html><body><h2>Product
List</h2><table border='1'>");
            out.println("<tr><th>Product
ID</th><th>Product Name</th><th>Price</th></tr>");
            while (rs.next()) {
                out.println("<tr><td>" +
rs.getInt("product_id") + "</td>");
                out.println("<td>" +
rs.getString("product_name") + "</td>");
```

```
                out.println("<td>" +
rs.getDouble("price") + "</td></tr>");
            }
            out.println("</table></body></html>");

        } catch (SQLException e) {
            e.printStackTrace();
            out.println("Database connection error: " +
e.getMessage());
        }
    }
}
```

**Output**:

When accessed via a web browser, the servlet generates an HTML table displaying products from the database:

| **Product ID** | **Product Name** | **Price** |
|---|---|---|
| **1** | Coffee Mug | 12.99 |
| **2** | Notebook | 5.49 |
| **3** | Pen | 1.25 |

**Conclusion**: Distributed database management enhances data access, reliability, and fault tolerance across locations, while servlets enable dynamic web interfaces that connect users to backend databases, offering efficient data-driven web applications.