Task Report

INDEX

Task 1

Algorithm Development Task: Inventory Reordering System

1.1 Objective

The goal of this task is to design an algorithm that ensures uninterrupted stock availability in a warehouse by determining an optimal reordering schedule. The algorithm aims to prevent stockouts by analyzing current inventory levels, forecasted demand, and reordering costs. By doing so, it ensures that inventory levels are sufficient to meet demand while minimizing the total cost associated with reordering, including batch size considerations. This solution is crucial for maintaining operational efficiency and cost-effectiveness in inventory management systems.

1.2 What the Algorithm Must Achieve

The algorithm should:

1. **Maintain Stock Levels:**
   Ensure that the current stock, supplemented by reordered units, is always sufficient to meet forecasted demand. This prevents stockouts that can disrupt operations.
2. **Optimize Costs:**
   Minimize total reordering costs by considering the cost per unit and batch size constraints.
3. **Balance Efficiency and Accuracy:**
   Ensure that stock levels are neither too high (overstocking) nor too low (understocking), reducing storage costs and waste.

1.3 Input Data Details

The algorithm requires the following data for each item in inventory:

1. **Item Attributes:**
   - **item_id**: A unique identifier for the item.
   - **current_stock**: The quantity of the item currently in the warehouse.

- **forecasted_demand**: The expected demand for the item in the upcoming period.
- **reorder_cost_per_unit**: The cost to reorder one unit of the item (used to calculate the total cost of ordering).
- **batch_size**: The minimum quantity that can be ordered for the item, ensuring practical ordering.

Example Input Data

```
[
   { "item_id": 101, "current_stock": 50, "forecasted_demand": 100, "reorder_cost_per_unit":
2, "batch_size": 20 },
   { "item_id": 102, "current_stock": 30, "forecasted_demand": 80, "reorder_cost_per_unit":
1.5, "batch_size": 10 }
]
```

Output Requirements

The algorithm produces a **reordering plan** that includes the following for each item:

1. **Item Identification:**
    - **item_id**: The unique identifier of the item.
2. **Reorder Quantity:**
    - **units_to_order**: The number of units to reorder for each item, ensuring that stock meets forecasted demand.

Example Output

For the input data provided above, the output would look like this:

```
 [
    { "item_id": 101, "units_to_order": 60 },
    { "item_id": 102, "units_to_order": 50 }
]
```

1.4.Algorithm Design and Explanation

Here's how to structure it:

Steps of the Algorithm

1. **Input Validation**
   - Check if the input data is complete and valid (e.g., no missing or invalid values for current_stock, forecasted_demand, etc.).
2. **Reorder Quantity Calculation**
   - For each item in the inventory:
     1. Calculate the required units to meet forecasted demand:

        $$\textbf{Required Units} = \textbf{Forecasted Demand} - \textbf{Current Stock}$$

     2. Determine the number of units to reorder, considering the batch size:

        - If Required Units is greater than zero, round up to the nearest multiple of the batch size:

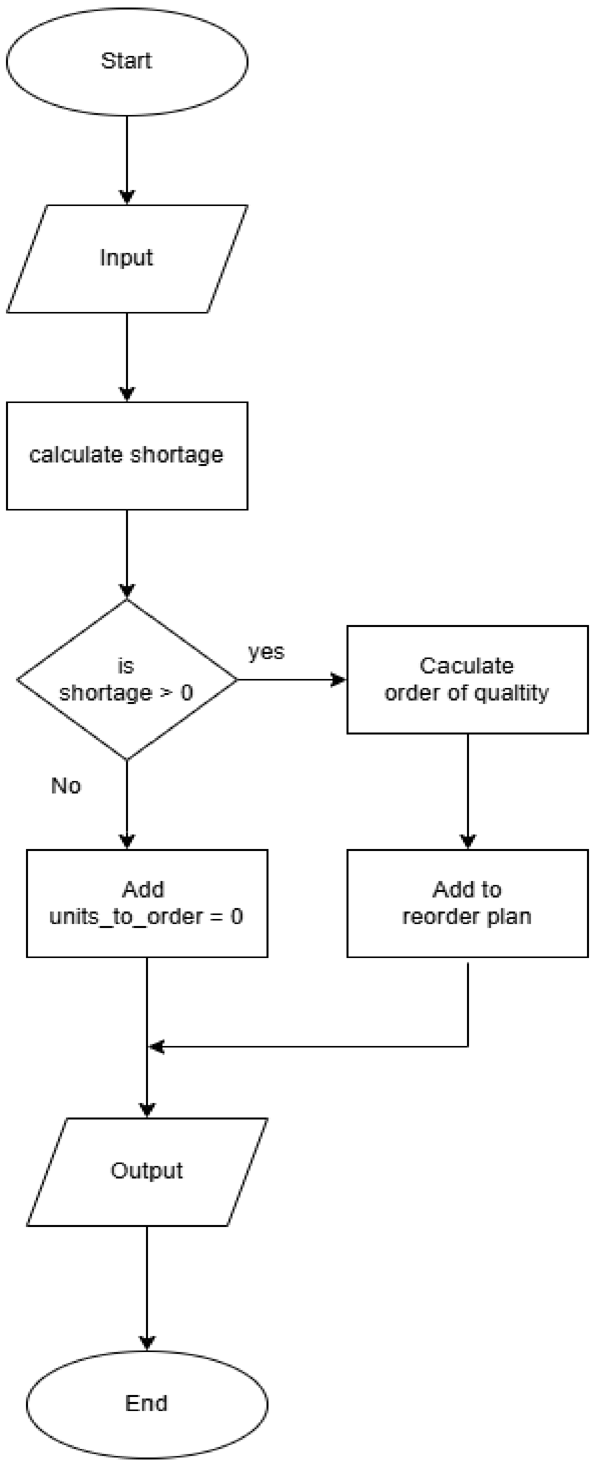
          Units to Order=[Required Units/ Batch Size]* Batch Size

        - Otherwise, set Units to Order to zero.

3. **Generate Output**
    - Compile the reordering plan as a list of objects containing item_id and units_to_order.

4. **Output Validation**
    - Verify that the calculated reorder quantities meet all constraints:
        - Forecasted demand is covered.
        - No unnecessary overstocking occurs.

1.5 Flow chart



1.6.Sample Run with Test Data

1. **Input Data:**

    Item ID: 101

    Current Stock: 50

    Forecasted Demand: 100

    Reorder Cost Per Unit: 2

    Batch Size: 20

    Item ID: 102

    Current Stock: 30

    Forecasted Demand: 80

    Reorder Cost Per Unit: 1.5

    Batch Size: 10

Output

Item ID: 101, Units to Order: 60

Item ID: 102, Units to Order: 50

Task 2

Employee Payroll System

2. Objective

The Employee Payroll System is a console-based application built using C# to manage employee records, calculate salaries, and store data using Object-Oriented Programming (OOP) principles. This application supports role-specific operations for Managers, Developers, and Interns.

2.1 Features

1. Add and manage employee details, including Name, ID, Role, Basic Pay, and Allowances.
2. Calculate and display employee salaries using the formula:

   Salary=Basic Pay + Allowances − Deductions

3. Display all employee details.
4. Save and retrieve employee data using file storage.
5. Calculate and display total payroll for all employees.

2.2 Running from Source Code (Command Line Instructions)

1.**Open a Command Prompt or Terminal:**

Navigate to the folder where the project is located using the cd command.

For example: **cd \path-to-project-folder**

2.**Build the Project:**

Use the dotnet build command to compile the project:

```
dotnet build
```

3. Run the Application:

Run the application using:

```
dotnet run
```

2.3 Application Menu

Upon the application displays a menu-driven interface with the following options:

1. **Add Employee**:
   - Allows adding a new employee by entering details like Name, ID, Role, Basic Pay, and Allowances.
2. **Display All Employees**:
   - Displays a list of all employees along with their details.
3. **Calculate Salary**:
   - Calculates and displays the salary for a specific employee by their ID.
4. **Calculate Total Payroll**:
   - (Optional Feature) Calculates the total salary paid to all employees.
5. **Save/Load Employees**:
   - (Optional Feature) Save employee data to a file or load data from a file.
6. **Exit**:
   - Exits the application.

Task 3

Employee Management System

3.1 Objective

The Employee Management System is a relational database designed to efficiently manage employee records and salaries within an organization. This system aims to streamline various aspects of employee management, such as storing detailed employee information, calculating salaries, and providing insights into departmental performance. By utilizing well-structured tables, stored procedures, and views, the system ensures data integrity, simplifies complex operations like payroll calculation, and enhances reporting capabilities. Its robust design enables organizations to maintain accurate records, automate salary-related processes, and gain valuable insights for better decision-making.

The Employee Management System was developed and tested using **SQLyog**, a graphical interface for MySQL databases. SQLyog was utilized for creating the database schema, writing and executing SQL queries, stored procedures, and views, as well as managing data. Its user-friendly interface and advanced features, such as query building and performance analysis tools, made the development process efficient and streamlined.

3.2 Create Database

- **Creation**

  Creates a new database named EmployeeManagement where all tables and data will be stored.

```
CREATE DATABASE EmployeeManagement;
```

- **Use Database**

Switches the active database to EmployeeManagement for executing further commands.

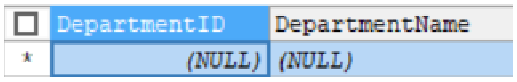
USE EmployeeManagement;

3.3 Tables

- **Departments Table**

Creates the Departments table to store information about departments within the organization.

```
CREATE TABLE Departments (
DepartmentID INT PRIMARY KEY,
DepartmentName VARCHAR(100) NOT NULL
);
```

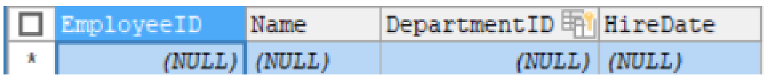


	DepartmentID	DepartmentName
*	(NULL)	(NULL)

- **Employees Table**

Creates the Employees table to store employee details, including a foreign key linking to the Departments table.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    DepartmentID INT NOT NULL,
    HireDate DATE NOT NULL,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID) );
```

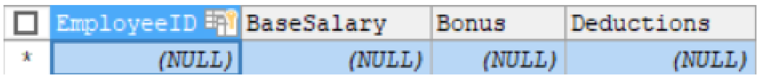


	EmployeeID	Name	DepartmentID	HireDate
*	(NULL)	(NULL)	(NULL)	(NULL)

- **Salaries Table**

Creates the Salaries table to store salary details for each employee, with a foreign key referencing the Employees table.

```
CREATE TABLE Salaries (
    EmployeeID INT PRIMARY KEY,
    BaseSalary DECIMAL(10, 2) NOT NULL,
    Bonus DECIMAL(10, 2),
    Deductions DECIMAL(10, 2),
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID) )
```



EmployeeID	BaseSalary	Bonus	Deductions
(NULL)	(NULL)	(NULL)	(NULL)

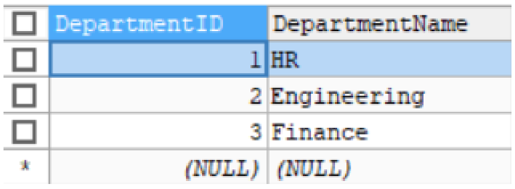
3.4 Data Insertion

- **Insert into Departments Table**

Adds example department records to the Departments table.

INSERT INTO Departments (DepartmentID, DepartmentName)

VALUES (1, 'HR'), (2, 'Engineering'), (3, 'Finance');

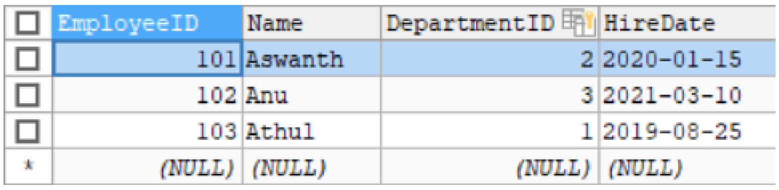


DepartmentID	DepartmentName
1	HR
2	Engineering
3	Finance
(NULL)	(NULL)

- **Insert into Employees Table**

Adds sample employee records to the Employees table.

INSERT INTO Employees (EmployeeID, Name, DepartmentID, HireDate) VALUES (101, 'Aswanth', 2, '2020-01-15'), (102, 'Anu', 3, '2021-03-10'), (103, 'Athul', 1, '2019-08-25');



	EmployeeID	Name	DepartmentID	HireDate
☐	101	Aswanth	2	2020-01-15
☐	102	Anu	3	2021-03-10
☐	103	Athul	1	2019-08-25
*	(NULL)	(NULL)	(NULL)	(NULL)

- **Insert into Salaries Table**

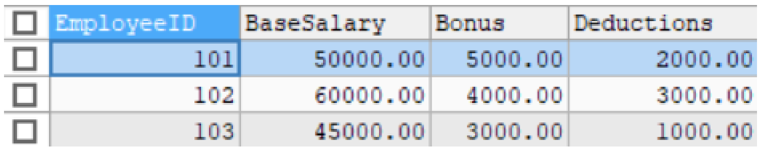
Adds salary details for employees into the Salaries table.

INSERT INTO Salaries (EmployeeID, BaseSalary, Bonus, Deductions)

VALUES (101, 50000, 5000, 2000),

(102, 60000, 4000, 3000),

(103, 45000, 3000, 1000);



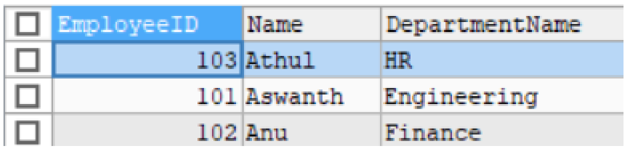
	EmployeeID	BaseSalary	Bonus	Deductions
☐	101	50000.00	5000.00	2000.00
☐	102	60000.00	4000.00	3000.00
☐	103	45000.00	3000.00	1000.00

3.5 Queries

- **List Employees with Department Names**

Fetches employee details along with their department names by joining the Employees and Departments tables.

SELECT e.EmployeeID, e.Name, d.DepartmentName FROM Employees e JOIN Departments d ON e.DepartmentID = d.DepartmentID;



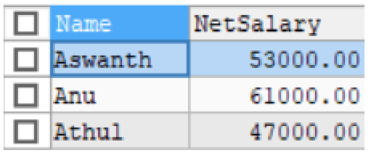
EmployeeID	Name	DepartmentName
103	Athul	HR
101	Aswanth	Engineering
102	Anu	Finance

- **Calculate Net Salary**

  Calculates the net salary for each employee using the formula:

  Net Salary = BaseSalary + Bonus – Deductions

SELECT e.Name, (s.BaseSalary + s.Bonus - s.Deductions) AS NetSalary FROM Employees e JOIN Salaries s ON e.EmployeeID = s.EmployeeID;

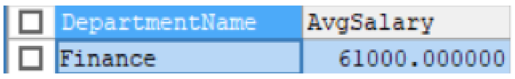


Name	NetSalary
Aswanth	53000.00
Anu	61000.00
Athul	47000.00

- **Find Department with Highest Average Salary**

Finds the department with the highest average salary by aggregating data from the Departments, Employees, and Salaries tables.

SELECT d.DepartmentName, AVG(s.BaseSalary + s.Bonus - s.Deductions) AS AvgSalary FROM Departments d JOIN Employees e ON d.DepartmentID = e.DepartmentID JOIN Salaries s ON e.EmployeeID = s.EmployeeID GROUP BY d.DepartmentName ORDER BY AvgSalary DESC LIMIT 1;



DepartmentName	AvgSalary
Finance	61000.000000

3.6 Stored Procedures

- **Add Employee**

  Creates a stored procedure to insert new employee details into the Employees table.

DELIMITER //

CREATE PROCEDURE AddEmployee (IN empID INT, IN empName VARCHAR(100), IN deptID INT, IN hireDate DATE) BEGIN INSERT INTO Employees (EmployeeID, Name, DepartmentID, HireDate) VALUES (104, fawas, 3, 2020-01-19); END//

DELIMITER ;

- **Update Salary**

Updates salary details for a given employee.

```
DELIMITER //

CREATE PROCEDURE UpdateSalary (
   IN empID INT,
   IN newBaseSalary DECIMAL(10, 2),
   IN newBonus DECIMAL(10, 2),
   IN newDeductions DECIMAL(10, 2)
)
BEGIN
   UPDATE Salaries
   SET BaseSalary = newBaseSalary,
      Bonus = newBonus,
      Deductions = newDeductions
   WHERE EmployeeID = empID;
END //  DELIMITER ;
```

- **Calculate Payroll**

Calculates the total payroll cost for all employees

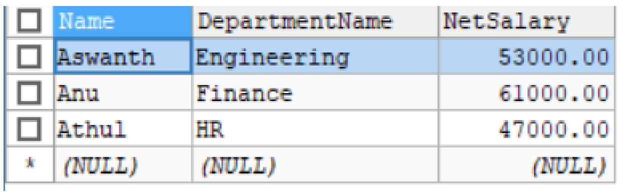
```
DELIMITER //

CREATE PROCEDURE CalculatePayroll ()
BEGIN
   SELECT SUM(BaseSalary + Bonus - Deductions) AS TotalPayroll
   FROM Salaries;
END //

DELIMITER ;
```

3.7 Views

- **EmployeeSalaryView**

Provides a detailed view of employee salaries with department names and net salaries.

CREATE VIEW EmployeeSalaryView AS SELECT e.Name, d.DepartmentName, (s.BaseSalary + s.Bonus - s.Deductions) AS NetSalary FROM Employees e JOIN Salaries s ON e.EmployeeID = s.EmployeeID JOIN Departments d ON e.DepartmentID = d.DepartmentID;

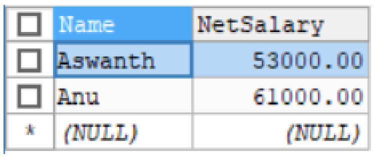


	Name	DepartmentName	NetSalary
☐	Aswanth	Engineering	53000.00
☐	Anu	Finance	61000.00
☐	Athul	HR	47000.00
*	(NULL)	(NULL)	(NULL)

- **HighEarnerView**

Lists employees earning above a specified threshold.

CREATE VIEW HighEarnerView AS SELECT e.Name, (s.BaseSalary + s.Bonus - s.Deductions) AS NetSalary FROM Employees e JOIN Salaries s ON e.EmployeeID = s.EmployeeID WHERE (s.BaseSalary + s.Bonus - s.Deductions) > 50000;



	Name	NetSalary
☐	Aswanth	53000.00
☐	Anu	61000.00
*	(NULL)	(NULL)