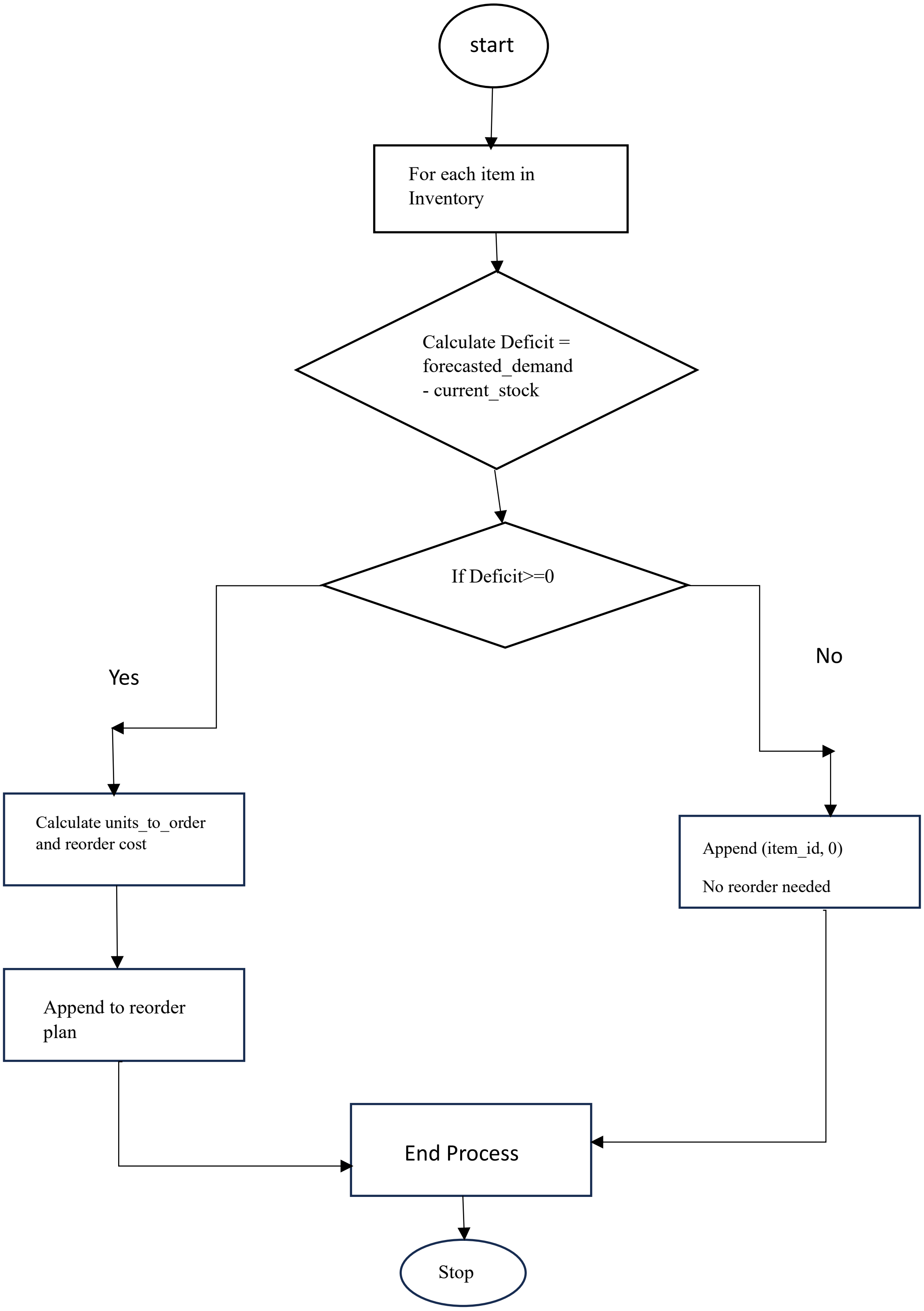
Algorithm Development for Inventory Reordering System

- **Plain Algorithm for Inventory Reordering System**

1. Initialize variables:
    - Create an empty list called reorder_plan to store the reorder details (item_id, units_to_order).
    - Initialize a variable total_reorder_cost to 0, which will accumulate the total reorder cost.
2. Process each item in the inventory:
    - For each item:
        1. Calculate the deficit as the difference between forecasted_demand and current_stock:
        2. If the deficit is greater than 0 (i.e., the item needs to be reordered):
            - Calculate the units to order based on the reorder batch size. The number of units should be the smallest multiple of the batch size that covers the deficit.
            - Calculate the reorder cost for this item by multiplying the units_to_order by the reorder_cost_per_unit:
            - Add the calculated reorder cost to total_reorder_cost.
            - Append the item with item_id and units_to_order to the reorder_plan.
        3. If the deficit is less than or equal to 0 (i.e., no reorder is needed):
            - Append the item with item_id and 0 units to order to the reorder_plan.
3. Return the reorder plan and total reorder cost:
    - After processing all items, return the reorder_plan and the total_reorder**_cost.**

Flow Chart


start
For each item in Inventory
Calculate Deficit = forecasted_demand - current_stock
If Deficit>=0
Yes
No
Calculate units_to_order and reorder cost
Append to reorder plan
Append (item_id, 0)
No reorder needed
End Process
Stop

Sample Run with Test Data

Inventory Data:

item_id	current_stock	forecasted_demand	reorder_cost_per_unit	batch_size
101	50	100	10	20
102	150	100	5	30
103	200	180	7	50
104	30	60	8	10

Execution:

1. **Item 101**:
   - **Deficit** = 100 - 50 = 50
   - Units to order = 60 (rounding up to the nearest multiple of 20)
   - Reorder cost = 60 * 10 = 600
   - Add to reorder plan: (101, 60)
2. **Item 102**:
   - **Deficit** = 100 - 150 = -50 (no reorder needed)
   - Add to reorder plan: (102, 0)
3. **Item 103**:
   - **Deficit** = 180 - 200 = -20 (no reorder needed)
   - Add to reorder plan: (103, 0)
4. **Item 104**:
   - **Deficit** = 60 - 30 = 30
   - Units to order = 30 (no rounding needed as it's a perfect multiple of 10)
   - Reorder cost = 30 * 8 = 240
   - Add to reorder plan: (104, 30)

Programming Task: Employee Payroll System (C# Console Application)

1. Full Code for Payroll System

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

// Interfaces
interface IEmployee
{
    string Name { get; }
    int Id { get; }
    string Role { get; }
    decimal BasicPay { get; }
    decimal Allowances { get; }
    decimal CalculateSalary();
}

// Base Class
class BaseEmployee : IEmployee
{
    public string Name { get; private set; }
    public int Id { get; private set; }
    public string Role { get; private set; }
    public decimal BasicPay { get; private set; }
    public decimal Allowances { get; private set; }

    public BaseEmployee(string name, int id, string role, decimal basicPay, decimal allowances)
    {
        Name = name;
        Id = id;
        Role = role;
        BasicPay = basicPay;
        Allowances = allowances;
    }

    public virtual decimal CalculateSalary()
    {
        decimal deductions = 0.1m * BasicPay; // Example: 10% deductions
        return BasicPay + Allowances - deductions;
    }
}

// Specialized Classes
class Manager : BaseEmployee
{
    public Manager(string name, int id, decimal basicPay, decimal allowances)
        : base(name, id, "Manager", basicPay, allowances) { }
}

class Developer : BaseEmployee
{
    public Developer(string name, int id, decimal basicPay, decimal allowances)
        : base(name, id, "Developer", basicPay, allowances) { }
}

class Intern : BaseEmployee
{
    public Intern(string name, int id, decimal basicPay, decimal allowances, bool isIntern)
        : base(name, id, "Intern", basicPay, allowances) { }
}

// Main Program
class PayrollSystem
{
    private List<IEmployee> employees = new List<IEmployee>();
    private const string filePath = "employees.txt";
```

```
public void Run()
{
    while (true)
    {
        Console.WriteLine("\n1. Add New Employee");
        Console.WriteLine("2. Display All Employees");
        Console.WriteLine("3. Calculate and Display Employee Salary");
        Console.WriteLine("4. Display Total Payroll");
        Console.WriteLine("5. Save Employee Data");
        Console.WriteLine("6. Exit");
        Console.Write("Choose an option: ");

        if (int.TryParse(Console.ReadLine(), out int choice))
        {
            switch (choice)
            {
                case 1:
                    AddEmployee();
                    break;
                case 2:
                    DisplayEmployees();
                    break;
                case 3:
                    CalculateEmployeeSalary();
                    break;
                case 4:
                    DisplayTotalPayroll();
                    break;
                case 5:
                    SaveEmployeesToFile();
                    break;
                case 6:
                    return;
                default:
                    Console.WriteLine("Invalid choice. Please try again.");
                    break;
            }
        }
        else
        {
            Console.WriteLine("Invalid input. Please enter a number.");
        }
    }
}

private void AddEmployee()
{
    Console.Write("Enter Name: ");
    string name = Console.ReadLine();

    Console.Write("Enter ID: ");
    if (!int.TryParse(Console.ReadLine(), out int id) || employees.Any(e => e.Id == id))
    {
        Console.WriteLine("Invalid or duplicate ID.");
        return;
    }

    Console.Write("Enter Role (Manager/Developer/Intern): ");
    string role = Console.ReadLine();

    Console.Write("Enter Basic Pay: ");
    if (!decimal.TryParse(Console.ReadLine(), out decimal basicPay))
    {
        Console.WriteLine("Invalid Basic Pay.");
        return;
    }
```

```
        Console.Write("Enter Allowances: ");
        if (!decimal.TryParse(Console.ReadLine(), out decimal allowances))
        {
            Console.WriteLine("Invalid Allowances.");
            return;
        }

        IEmployee employee = role switch
        {
            "Manager" => new Manager(name, id, basicPay, allowances),
            "Developer" => new Developer(name, id, basicPay, allowances),
            "Intern" => new Intern(name, id, basicPay, allowances, true),
            _ => null
        };

        if (employee == null)
        {
            Console.WriteLine("Invalid role. Employee not added.");
            return;
        }

        employees.Add(employee);
        Console.WriteLine("Employee added successfully.");
    }

    private void DisplayEmployees()
    {
        if (employees.Count == 0)
        {
            Console.WriteLine("No employees to display.");
            return;
        }

        foreach (var employee in employees)
        {
            Console.WriteLine($"Name: {employee.Name}, ID: {employee.Id}, Role: {employee.Role}, Basic Pay: {employee.BasicPay}, Allowances: {employee.Allowances}");
        }
    }

    private void CalculateEmployeeSalary()
    {
        Console.Write("Enter Employee ID: ");
        if (int.TryParse(Console.ReadLine(), out int id))
        {
            var employee = employees.FirstOrDefault(e => e.Id == id);
            if (employee != null)
            {
                Console.WriteLine($"Salary for {employee.Name} ({employee.Role}): {employee.CalculateSalary()}");
            }
            else
            {
                Console.WriteLine("Employee not found.");
            }
        }
        else
        {
            Console.WriteLine("Invalid ID.");
        }
    }

    private void DisplayTotalPayroll()
    {
        if (employees.Count == 0)
        {
            Console.WriteLine("No employees to calculate payroll.");
            return;
        }
        decimal totalPayroll = employees.Sum(e => e.CalculateSalary());
        Console.WriteLine($"Total Payroll: {totalPayroll}");
    }

    private void SaveEmployeesToFile()
    {
        using (StreamWriter writer = new StreamWriter(filePath, false))
        {
            foreach (var employee in employees)
            {
writer.WriteLine($"{employee.Name},{employee.Id},{employee.Role},{employee.BasicPay},{employee.Allowances}");
            }
        }
        Console.WriteLine("Employee data saved to file.");
    }

    private void LoadEmployeesFromFile()
    {
        employees.Clear();

        if (!File.Exists(filePath))
        {
            File.Create(filePath).Close();
            Console.WriteLine("No employee data file found. A new file has been created.");
            return;
        }

        string[] lines = File.ReadAllLines(filePath);
        if (lines.Length == 0)
        {
            Console.WriteLine("Employee data file is empty. Please add employees and save data.");
            return;
```

```
}

foreach (var line in lines)
{
    try
    {
        var parts = line.Split(',');
        if (parts.Length == 5)
        {
            string name = parts[0];
            int id = int.Parse(parts[1]);
            string role = parts[2];
            decimal basicPay = decimal.Parse(parts[3]);
            decimal allowances = decimal.Parse(parts[4]);

            IEmployee employee = role switch
            {
                "Manager" => new Manager(name, id, basicPay, allowances),
                "Developer" => new Developer(name, id, basicPay, allowances),
                "Intern" => new Intern(name, id, basicPay, allowances, true),
                _ => null
            };

            if (employee != null)
            {
                employees.Add(employee);
            }
            else
            {
                Console.WriteLine($"Invalid role in file: {line}");
            }
        }
        else
        {
```

```
                    Console.WriteLine($"Invalid line format: {line}");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error processing line: {line}. Details: {ex.Message}");
            }
        }

        if (employees.Count == 0)
        {
            Console.WriteLine("No valid employee records found in the file.");
        }
        else
        {
            Console.WriteLine("Employee data loaded from file.");
        }
    }

    public static void Main(string[] args)
    {
        PayrollSystem payrollSystem = new PayrollSystem();
        payrollSystem.LoadEmployeesFromFile(); // Automatically load data on startup
        payrollSystem.Run();
    }
}
```

2. Code Explanation

1. IEmployee Interface:

- Defines common properties (Name, ID, Role, etc.) and the CalculateSalary method that all employee types must implement.

2. BaseEmployee Class:

- Implements IEmployee and contains basic employee information and a default salary calculation method. This class can be inherited by specific employee roles.

3. Specialized Employee Classes (Manager, Developer, Intern):

- Inherit from BaseEmployee and override the CalculateSalary method to apply specific salary calculation logic.
- Intern has a custom deduction and a flag (IsPaidIntern) to determine if they receive a salary.

4. PayrollSystem Class:

- Manages a list of employees. Includes methods to:
  - Add new employees.
  - Display employee details.
  - Calculate and display salaries.
  - Calculate and display total payroll.
  - Save and load employee data to/from a file.

5. Program Class (Main):

- Provides a menu-driven interface where the user can perform various operations such as adding employees, calculating salaries, displaying payroll, saving/loading data, etc.

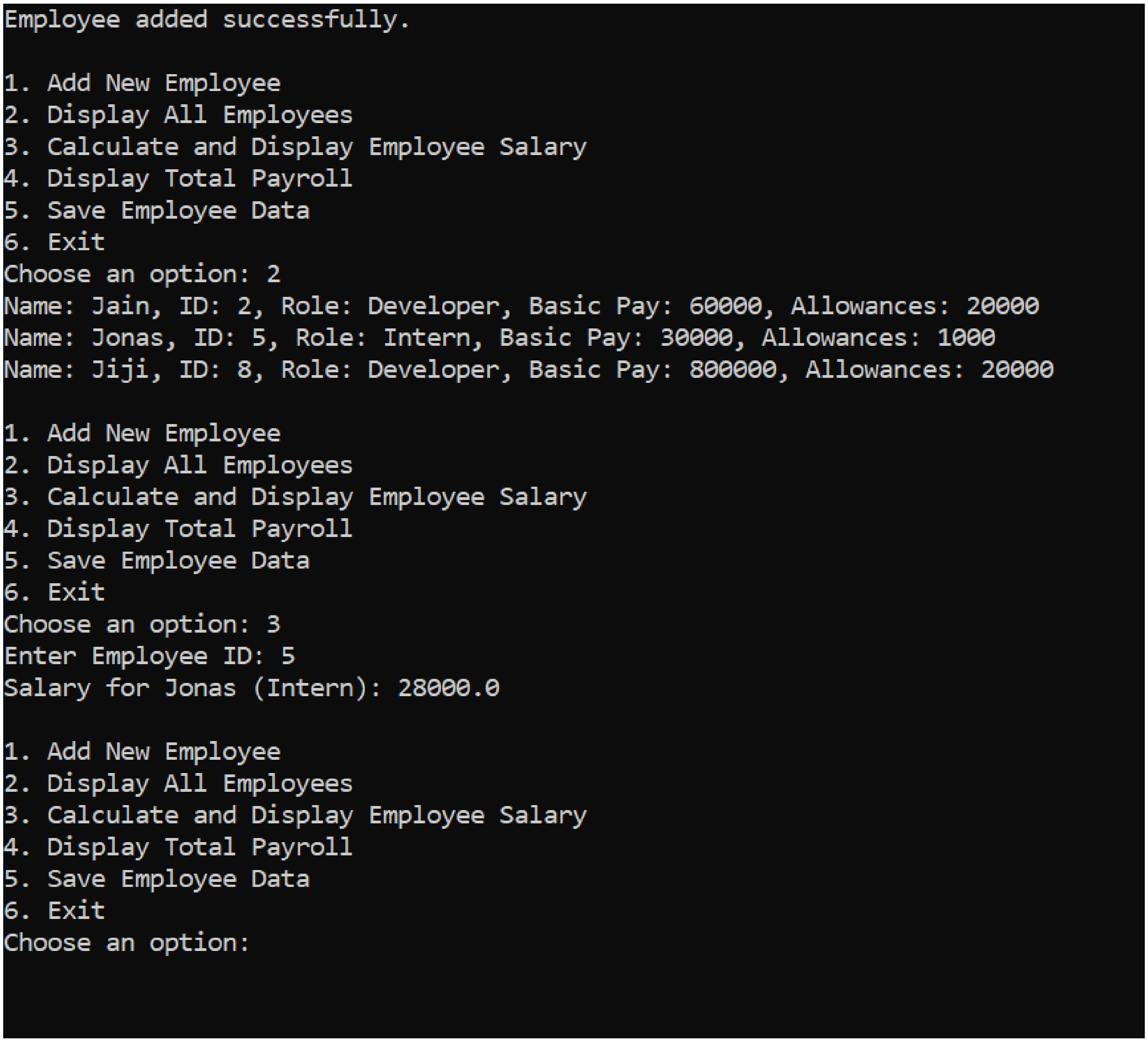
3. How to Run the Application

Step-by-Step Instructions:

1. Open Visual Studio:
   - Open Visual Studio (or any other C# compatible IDE).
2. Create a New Console Application:
   - Go to File -> New -> Project.
   - Select Console Application from the list of templates.
   - Name your project (e.g., PayrollSystemApp).
3. Add the Code:
   - Copy and paste the provided code into the Program.cs file in your project.
4. Build the Solution:
   - Press Ctrl+Shift+B to build the solution.
5. Run the Application:
   - Press Ctrl+F5 to run the application without debugging.
   - Use the menu options to manage employees, calculate salaries, and more.

4. Expected Output

The console will show a menu where you can choose options. For example:



```
Employee added successfully.

1. Add New Employee
2. Display All Employees
3. Calculate and Display Employee Salary
4. Display Total Payroll
5. Save Employee Data
6. Exit
Choose an option: 2
Name: Jain, ID: 2, Role: Developer, Basic Pay: 60000, Allowances: 20000
Name: Jonas, ID: 5, Role: Intern, Basic Pay: 30000, Allowances: 1000
Name: Jiji, ID: 8, Role: Developer, Basic Pay: 800000, Allowances: 20000

1. Add New Employee
2. Display All Employees
3. Calculate and Display Employee Salary
4. Display Total Payroll
5. Save Employee Data
6. Exit
Choose an option: 3
Enter Employee ID: 5
Salary for Jonas (Intern): 28000.0

1. Add New Employee
2. Display All Employees
3. Calculate and Display Employee Salary
4. Display Total Payroll
5. Save Employee Data
6. Exit
Choose an option:
```

5. Features:

1. Add Employee: Allows adding new employees to the payroll system.
2. Display Employee Details: Displays information for all employees.
3. Calculate Salary: Calculates salary for an employee based on the role and their details.
4. Total Payroll: Displays the total payroll for all employees.
5. File Storage: Employee data can be saved to and loaded from a file (employees.txt).

Database Task:

EMPLOYEE MANAGEMENT SYSTEM SQL IMPLEMENTATION

1. Database Schema Creation

```
CREATE TABLE Departments (
    DepartmentID INT IDENTITY (1,1) PRIMARY KEY,
    DepartmentName VARCHAR(100) NOT NULL
);
CREATE TABLE Employees (
    EmployeeID INT IDENTITY(1,1) PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    DepartmentID INT,
    HireDate DATE,
    FOREIGN KEY (DepartmentID) REFERENCES Departments (DepartmentID)
);
CREATE TABLE Salaries (
    EmployeeID INT PRIMARY KEY,
    BaseSalary DECIMAL(10, 2) NOT NULL,
    Bonus DECIMAL(10, 2) DEFAULT 0,
    Deductions DECIMAL(10, 2) DEFAULT 0,
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
);
CREATE TABLE SalaryHistory (
    HistoryID INT IDENTITY(1,1) PRIMARY KEY,
    EmployeeID INT,
    BaseSalary DECIMAL(10, 2),
    Bonus DECIMAL(10, 2),
    Deductions DECIMAL(10, 2),
    ChangeDate TIMESTAMP,
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
);
```

2. SQL Queries

a) List all employees along with their department names

```
SELECT    e. EmployeeID, e. Name, d. DepartmentName FROM Employees e

LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

OUTPUT:-

EmployeeID	Name	DepartmentName
1	John Doe	HR
2	Jane Smith	Finance
3	Mike Johnson	IT
4	Sara Lee	Marketing
5	Tom Harris	Sales
6	Jain	IT

b) Calculate the net salary for each employee

```
SELECT    e. EmployeeID, e. Name, s. BaseSalary, s. Bonus, s. Deductions, (s. BaseSalary + s. Bonus - s.
Deductions) AS NetSalary

FROM Employees e

JOIN Salaries s ON e.EmployeeID = s.EmployeeID;
```

OUTPUT:-

EmployeeID	Name	BaseSalary	Bonus	Deductions	NetSalary
1	John Doe	50000	5000	2000	53000
2	Jane Smith	60000	6000	2500	63500
3	Mike Johnson	70000	7000	3000	74000
4	Sara Lee	80000	8000	3500	84500
5	Tom Harris	90000	9000	4000	95000

c) Identify the department with the highest average salary

```
SELECT TOP 1

   d.DepartmentName,

   AVG (s.BaseSalary + s.Bonus - s.Deductions) AS AvgSalary

FROM Employees e

JOIN Salaries s ON e.EmployeeID = s.EmployeeID

JOIN Departments d ON e.DepartmentID = d.DepartmentID
```

```
GROUP BY d. DepartmentID

ORDER BY AvgSalary DESC
```

OUTPUT:-

DepartmentName	AvgSalary
Sales	95000

3.Stored Procedures

a) Add Employees

```
CREATE PROCEDURE AddEmployee

(@empName VARCHAR (100), @deptID INT, @hireDate DATE)

AS

 BEGIN

IF (SELECT COUNT (*) FROM Departments WHERE DepartmentID = @deptID) > 0

BEGIN

INSERT INTO Employees (Name, DepartmentID, HireDate) VALUES (@empName, @deptID, @hireDate);

 END

 ELSE

BEGIN

PRINT 'Invalid DepartmentID';

END

END;
```

OUTPUT:-

```
EXEC AddEmployee @Name='JONAS', @DepartmentId=2, @hiredate='2025-01-01';
```

EmployeeID	Name	DepartmentID	HireDate
1	John Doe	1	1/1/2025
2	Jane Smith	2	1/2/2025
3	Mike Johnson	3	1/3/2025
4	Sara Lee	4	1/4/2025
5	Tom Harris	5	1/5/2025
6	Jain	3	1/10/2025
7	JONAS	2	1/1/2025

b. Update Employees

```
CREATE PROCEDURE UpdateSalary
  @emp_id INT,
  @new_base_salary DECIMAL (10, 2),
  @new_bonus DECIMAL (10, 2),
  @new_deductions DECIMAL (10, 2)
AS
BEGIN
   DECLARE @current_date DATETIME;
       SET @current_date = GETDATE ();
  UPDATE dbo.Salaries
  SET BaseSalary = @new_base_salary,
    Bonus = @new_bonus,
    Deductions = @new_deductions
  WHERE EmployeeID = @emp_id;
    IF EXISTS (SELECT * FROM dbo.SalaryHistory WHERE EmployeeID = @emp_id)
  BEGIN
    UPDATE dbo.SalaryHistory
    SET BaseSalary = @new_base_salary,
      Bonus = @new_bonus,
      Deductions = @new_deductions,
      Change_Date = @current_date
    WHERE EmployeeID = @emp_id;
  END
  ELSE
  BEGIN
    INSERT INTO dbo.SalaryHistory (EmployeeID, BaseSalary, Bonus, Deductions, Change_Date)
    VALUES (@emp_id, @new_base_salary, @new_bonus, @new_deductions, @current_date);
  END
END;
```

OUTPUT:-

```
EXEC UpdateSalary
    @emp_id=3,
    @new_base_salary=62000.00,
    @new_bonus=8000.00,
    @new_deductions=4000.00;
```

Salary

EmployeeID	BaseSalary	Bonus	Deductions
1	50000	5000	2000
2	60000	6000	2500
3	62000	8000	4000
4	80000	8000	3500
5	90000	9000	4000

SalaryHistory

HistoryID	EmployeeID	BaseSalary	Bonus	Deductions	Change_Date
2	1	99000	5000	1000	7/10/1905 0:00
3	3	62000	8000	4000	1/4/2025 15:17

c.) **Calculate Payroll**

```
CREATE PROCEDURE CalculatePayroll (
    @deptID INT,
    @totalPayroll DECIMAL (10, 2) OUTPUT
)
AS
BEGIN
    IF @deptID IS NULL
    BEGIN
        SELECT @totalPayroll = SUM(BaseSalary + Bonus - Deductions)
        FROM Salaries;
    END
    ELSE
    BEGIN
        SELECT @totalPayroll = SUM(BaseSalary + Bonus - Deductions)
        FROM Salaries s
        JOIN Employees e ON s.EmployeeID = e.EmployeeID
        WHERE e.DepartmentID = @deptID;
    END
END;
```

OUTPUT:-

```
DECLARE @totalPayroll DECIMAL(18, 2);

EXEC CalculatePayroll @DeptID = 2, @totalPayroll = @totalPayroll OUTPUT;

SELECT  @totalPayroll AS TotalPayroll;
```

TotalPayroll
63500

4. Views

a) EmployeeSalaryView

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

OUTPUT:-

EmployeeID	Name	DepartmentName	BaseSalary	Bonus	Deductions	NetSalary
1	John Doe	HR	50000	5000	2000	53000
2	Jane Smith	Finance	60000	6000	2500	63500
3	Mike Johnson	IT	62000	8000	4000	66000
4	Sara Lee	Marketing	80000	8000	3500	84500
5	Tom Harris	Sales	90000	9000	4000	95000

b.) HighEarnerView

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

OUTPUT:-

EmployeeID	Name	DepartmentName	NetSalary
1	John Doe	HR	53000
2	Jane Smith	Finance	63500
3	Mike Johnson	IT	66000
4	Sara Lee	Marketing	84500
5	Tom Harris	Sales	95000

5. Bonus Tasks

a) Add trigger to log salary updates

```
CREATE TRIGGER LogSalaryUpdate
ON Salaries
AFTER UPDATE
AS
BEGIN
    INSERT INTO SalaryHistory (EmployeeID, BaseSalary, Bonus, Deductions, ChangeDate)
    SELECT
        INSERTED.EmployeeID,
        INSERTED.BaseSalary,
        INSERTED.Bonus,
        INSERTED.Deductions,
        GETDATE()
    FROM
        INSERTED;
END;
```

Design Choices

1. Normalization:

Tables are normalized to ensure data integrity and reduce redundancy. For example:
Departments and Employees tables separate department data from employee data.
Salaries is a separate table to manage financial data without cluttering the Employees table.

2. Referential Integrity:

Foreign keys are used to maintain relationships, e.g., DepartmentID in Employees references Departments.

3. Historical Tracking:

SalaryHistory logs salary changes with timestamps, enabling audit trails.

4. Views for Simplification:

EmployeeSalaryView combines tables for a user-friendly salary report.
HighEarnerView isolates high-earning employees for specific insights.

5. Stored Procedures:

Automates and centralizes common operations like adding employees and updating salaries, reducing potential errors.

6. Triggers:

The LogSalaryUpdate trigger ensures salary changes are automatically logged in SalaryHistory.

Indexing and Optimization Strategies

1. Indexes on Primary and Foreign Keys:

Primary keys (e.g., EmployeeID, DepartmentID) are indexed by default.
Foreign keys (e.g., EmployeeID in Salaries) should also have indexes to speed up joins.

2. Additional Indexes:

HireDate in Employees for filtering or sorting employees by their hiring date.
Combined indexes on frequently queried fields, like DepartmentID and EmployeeID, to optimize joins and WHERE clauses.

3. Query Optimization:

Avoid SELECT *, explicitly list columns to minimize data transfer.

4. Triggers and Procedures:

Ensure stored procedures use indexed columns in WHERE clauses for better performance.
Minimize logic within triggers to avoid excessive overhead during DML operations.

5. Threshold for High-Earner View:
If the threshold (e.g., 50,000) is dynamic, consider passing it as a parameter in a stored procedure.