**Week2 PL/SQL Assignment**

-- Schema Creation

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE,

IsVIP VARCHAR2(5) DEFAULT 'FALSE'

);

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Transactions (

TransactionID NUMBER PRIMARY KEY,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10),

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

LoanAmount NUMBER,

InterestRate NUMBER,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

Position VARCHAR2(50),

Salary NUMBER,

Department VARCHAR2(50),

HireDate DATE

);

CREATE TABLE AuditLog (

LogID NUMBER PRIMARY KEY,

TransactionID NUMBER,

LogMessage VARCHAR2(255),

LogDate DATE

);

-- Sample Data Insertion

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)

VALUES (1, 1, 'Savings', 1000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)

VALUES (2, 2, 'Checking', 1500, SYSDATE);

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (1, 1, SYSDATE, 200, 'Deposit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (2, 2, SYSDATE, 300, 'Withdrawal');

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)

VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)

VALUES (2, 2, 7000, 6, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)

VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)

VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));

-- Enable DBMS\_OUTPUT

SET SERVEROUTPUT ON;

**Exercise 1: Control Structures**

-- Scenario 1: Apply a Discount to Loan Interest Rates for Customers Above 60 Years Old

DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM DOB) AS Age, LoanID, InterestRate

FROM Customers C JOIN Loans L ON C.CustomerID = L.CustomerID;

customer\_record customer\_cursor%ROWTYPE;

BEGIN

FOR customer\_record IN customer\_cursor LOOP

IF customer\_record.Age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE LoanID = customer\_record.LoanID;

END IF;

END LOOP;

COMMIT;

END;

/

-- Scenario 2: Promote Customers to VIP Status Based on Their Balance

DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, Balance

FROM Customers;

customer\_record customer\_cursor%ROWTYPE;

BEGIN

FOR customer\_record IN customer\_cursor LOOP

IF customer\_record.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = customer\_record.CustomerID;

END IF;

END LOOP;

COMMIT;

END;

/

-- Scenario 3: Send Reminders for Loans Due Within the Next 30 Days

DECLARE

CURSOR loan\_cursor IS

SELECT CustomerID, LoanID, EndDate

FROM Loans

WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30;

loan\_record loan\_cursor%ROWTYPE;

BEGIN

FOR loan\_record IN loan\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ID ' || loan\_record.CustomerID ||

' has a loan due on ' || TO\_CHAR(loan\_record.EndDate, 'DD-MON-YYYY'));

END LOOP;

END;

/

**Exercise 2: Error Handling**

-- Scenario 1: Handle exceptions during fund transfers between accounts

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

insufficient\_funds EXCEPTION;

PRAGMA EXCEPTION\_INIT(insufficient\_funds, -20001);

BEGIN

-- Check if the from account has sufficient balance

DECLARE

from\_balance NUMBER;

BEGIN

SELECT Balance INTO from\_balance FROM Accounts WHERE AccountID = p\_from\_account\_id;

IF from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

END;

-- Perform the transfer

UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_account\_id;

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to\_account\_id;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in the source account.');

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END SafeTransferFunds;

/

-- Scenario 2: Manage errors when updating employee salaries

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) IS

employee\_not\_found EXCEPTION;

PRAGMA EXCEPTION\_INIT(employee\_not\_found, -20002);

BEGIN

-- Update salary

UPDATE Employees

SET Salary = Salary \* (1 + p\_percentage / 100)

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE employee\_not\_found;

END IF;

COMMIT;

EXCEPTION

WHEN employee\_not\_found THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Employee not found.');

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END UpdateSalary;

/

-- Scenario 3: Ensure data integrity when adding a new customer

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) IS

customer\_exists EXCEPTION;

PRAGMA EXCEPTION\_INIT(customer\_exists, -20003);

BEGIN

-- Insert new customer

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

RAISE customer\_exists;

WHEN customer\_exists THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer with the same ID already exists.');

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END AddNewCustomer;

/

**Exercise 3: Stored Procedures**

-- Scenario 1: Process monthly interest for all savings accounts

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance \* 1.01

WHERE AccountType = 'Savings';

COMMIT;

END ProcessMonthlyInterest;

/

-- Scenario 2: Implement a bonus scheme for employees based on their performance

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department IN VARCHAR2,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Salary = Salary \* (1 + p\_bonus\_percentage / 100)

WHERE Department = p\_department;

COMMIT;

END UpdateEmployeeBonus;

/

-- Scenario 3: Transfer funds between customer accounts

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

insufficient\_funds EXCEPTION;

BEGIN

-- Check if the from account has sufficient balance

DECLARE

from\_balance NUMBER;

BEGIN

SELECT Balance INTO from\_balance FROM Accounts WHERE AccountID = p\_from\_account\_id;

IF from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in the source account.');

END IF;

END;

-- Perform the transfer

UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_account\_id;

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to\_account\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END TransferFunds;

/

**Exercise 4: Functions**

-- Scenario 1: Calculate the age of customers for eligibility checks

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

v\_age := TRUNC((SYSDATE - p\_dob) / 365.25);

RETURN v\_age;

END CalculateAge;

/

-- Scenario 2: Compute the monthly installment for a loan

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_loan\_amount NUMBER,

p\_interest\_rate NUMBER,

p\_duration\_years NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER;

v\_num\_payments NUMBER;

v\_monthly\_payment NUMBER;

BEGIN

v\_monthly\_rate := p\_interest\_rate / 1200;

v\_num\_payments := p\_duration\_years \* 12;

v\_monthly\_payment := (p\_loan\_amount \* v\_monthly\_rate) / (1 - POWER(1 + v\_monthly\_rate, -v\_num\_payments));

RETURN v\_monthly\_payment;

END CalculateMonthlyInstallment;

/

-- Scenario 3: Check if a customer has sufficient balance before making a transaction

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id NUMBER,

p\_amount NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_account\_id;

RETURN v\_balance >= p\_amount;

END HasSufficientBalance;

/

**Exercise 5: Triggers**

-- Scenario 1: Automatically update the last modified date when a customer's record is updated

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END UpdateCustomerLastModified;

/

-- Scenario 2: Maintain an audit log for all transactions

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (LogID, TransactionID, LogMessage, LogDate)

VALUES (AuditLog\_seq.NEXTVAL, :NEW.TransactionID, 'Transaction logged', SYSDATE);

END LogTransaction;

/

-- Scenario 3: Enforce business rules on deposits and withdrawals

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > (SELECT Balance FROM Accounts WHERE AccountID = :NEW.AccountID) THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds for withdrawal.');

ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Deposit amount must be positive.');

END IF;

END CheckTransactionRules;

/

**Exercise 6: Cursors**

-- Scenario 1: Generate monthly statements for all customers

DECLARE

CURSOR transaction\_cursor IS

SELECT CustomerID, TransactionDate, Amount, TransactionType

FROM Transactions

WHERE TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE);

transaction\_record transaction\_cursor%ROWTYPE;

BEGIN

FOR transaction\_record IN transaction\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || transaction\_record.CustomerID ||

', Date: ' || TO\_CHAR(transaction\_record.TransactionDate, 'DD-MON-YYYY') ||

', Amount: ' || transaction\_record.Amount ||

', Type: ' || transaction\_record.TransactionType);

END LOOP;

END;

/

-- Scenario 2: Apply annual fee to all accounts

DECLARE

CURSOR account\_cursor IS

SELECT AccountID, Balance

FROM Accounts;

account\_record account\_cursor%ROWTYPE;

annual\_fee CONSTANT NUMBER := 50;

BEGIN

FOR account\_record IN account\_cursor LOOP

UPDATE Accounts

SET Balance = Balance - annual\_fee

WHERE AccountID = account\_record.AccountID;

END LOOP;

COMMIT;

END;

/

-- Scenario 3: Update the interest rate for all loans based on a new policy

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate

FROM Loans;

loan\_record loan\_cursor%ROWTYPE;

new\_interest\_rate CONSTANT NUMBER := 4.5; -- Example new policy

BEGIN

FOR loan\_record IN loan\_cursor LOOP

UPDATE Loans

SET InterestRate = new\_interest\_rate

WHERE LoanID = loan\_record.LoanID;

END LOOP;

COMMIT;

END;

/

**Exercise 7: Packages**

-- Scenario 1: Group all customer-related procedures and functions into a package

CREATE OR REPLACE PACKAGE CustomerManagement IS

PROCEDURE AddNewCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE, p\_balance IN NUMBER);

PROCEDURE UpdateCustomerDetails(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE);

FUNCTION GetCustomerBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

PROCEDURE AddNewCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE, p\_balance IN NUMBER) IS

BEGIN

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE) IS

BEGIN

UPDATE Customers

SET Name = p\_name, DOB = p\_dob, LastModified = SYSDATE

WHERE CustomerID = p\_customer\_id;

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(p\_customer\_id IN NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_customer\_id;

RETURN v\_balance;

END GetCustomerBalance;

END CustomerManagement;

/

-- Scenario 2: Create a package to manage employee data

CREATE OR REPLACE PACKAGE EmployeeManagement IS

PROCEDURE HireNewEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_position IN VARCHAR2, p\_salary IN NUMBER, p\_department IN VARCHAR2, p\_hire\_date IN DATE);

PROCEDURE UpdateEmployeeDetails(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_position IN VARCHAR2, p\_salary IN NUMBER, p\_department IN VARCHAR2);

FUNCTION CalculateAnnualSalary(p\_employee\_id IN NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

PROCEDURE HireNewEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_position IN VARCHAR2, p\_salary IN NUMBER, p\_department IN VARCHAR2, p\_hire\_date IN DATE) IS

BEGIN

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)

VALUES (p\_employee\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);

END HireNewEmployee;

PROCEDURE UpdateEmployeeDetails(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_position IN VARCHAR2, p\_salary IN NUMBER, p\_department IN VARCHAR2) IS

BEGIN

UPDATE Employees

SET Name = p\_name, Position = p\_position, Salary = p\_salary, Department = p\_department

WHERE EmployeeID = p\_employee\_id;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(p\_employee\_id IN NUMBER) RETURN NUMBER IS

v\_annual\_salary NUMBER;

BEGIN

SELECT Salary \* 12 INTO v\_annual\_salary FROM Employees WHERE EmployeeID = p\_employee\_id;

RETURN v\_annual\_salary;

END CalculateAnnualSalary;

END EmployeeManagement;

/

-- Scenario 3: Group all account-related operations into a package

CREATE OR REPLACE PACKAGE AccountOperations IS

PROCEDURE OpenNewAccount(p\_account\_id IN NUMBER, p\_customer\_id IN NUMBER, p\_account\_type IN VARCHAR2, p\_balance IN NUMBER);

PROCEDURE CloseAccount(p\_account\_id IN NUMBER);

FUNCTION GetTotalBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations IS

PROCEDURE OpenNewAccount(p\_account\_id IN NUMBER, p\_customer\_id IN NUMBER, p\_account\_type IN VARCHAR2, p\_balance IN NUMBER) IS

BEGIN

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)

VALUES (p\_account\_id, p\_customer\_id, p\_account\_type, p\_balance, SYSDATE);

END OpenNewAccount;

PROCEDURE CloseAccount(p\_account\_id IN NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_account\_id;

END CloseAccount;

FUNCTION GetTotalBalance(p\_customer\_id IN NUMBER) RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total\_balance FROM Accounts WHERE CustomerID = p\_customer\_id;

RETURN v\_total\_balance;

END GetTotalBalance;

END AccountOperations;

/

Submitted by:

Name: POOJA NAYAK

SuperSet ID: 5007013

Email: poojanayak1001@gmail.com