CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Balance INT,

LastModified DATE

);

CREATE TABLE Accounts (

AccountID INT PRIMARY KEY,

CustomerID INT,

AccountType VARCHAR(20),

Balance INT,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Transactions (

TransactionID INT PRIMARY KEY,

AccountID INT,

TransactionDate DATE,

Amount INT,

TransactionType VARCHAR(10),

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

CREATE TABLE Loans (

LoanID INT PRIMARY KEY,

CustomerID INT,

LoanAmoun INT,

InterestRate INT,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100),

Position VARCHAR(50),

Salary INT,

Department VARCHAR(50),

HireDate DATE

);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  
VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, CURRENT\_DATE);  
  
INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  
VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, CURRENT\_DATE);  
  
INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)  
VALUES (1, 1, 'Savings', 1000, CURRENT\_DATE);  
  
INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)  
VALUES (2, 2, 'Checking', 1500, CURRENT\_DATE);  
  
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
VALUES (1, 1, CURRENT\_TIMESTAMP, 200, 'Deposit');  
  
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
VALUES (2, 2, CURRENT\_TIMESTAMP, 300, 'Withdrawal');  
  
INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)  
VALUES (1, 1, 5000, 5, CURRENT\_DATE, (CURRENT\_DATE + INTERVAL '60 months'));  
  
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')); >

Exercise 1: Control Structures:

Scenario 1: The bank wants to apply a discount to loan interest rates for customers above 60 years old.

Question: Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

CREATE OR REPLACE FUNCTION ApplyDiscount() RETURNS VOID AS $$  
DECLARE  
cust RECORD;  
BEGIN  
FOR cust IN  
SELECT Customers.CustomerID, Loans.LoanID, Loans.InterestRate  
FROM Loans  
JOIN Customers ON Loans.CustomerID = Customers.CustomerID  
WHERE DATE\_PART('year', AGE(Customers.DOB)) > 60  
LOOP  
UPDATE Loans  
SET InterestRate = InterestRate \* 0.99  
WHERE LoanID = cust.LoanID;  
END LOOP;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT ApplyDiscount();

Scenario 2: A customer can be promoted to VIP status based on their balance.

Question: Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.

ALTER TABLE Customers ADD COLUMN IsVIP BOOLEAN DEFAULT FALSE;  
CREATE OR REPLACE FUNCTION PromoteVIP() RETURNS VOID AS $$  
DECLARE  
    cust RECORD;  
BEGIN  
    FOR cust IN  
        SELECT CustomerID  
        FROM Customers  
        WHERE Balance > 10000  
    LOOP  
        UPDATE Customers  
        SET IsVIP = TRUE  
        WHERE CustomerID = cust.CustomerID;  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT PromoteVIP();

Scenario 3: The bank wants to send reminders to customers whose loans are due within the next 30 days.

Question: Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

CREATE OR REPLACE FUNCTION SendReminders() RETURNS VOID AS $$  
DECLARE  
    loan RECORD;  
BEGIN  
    FOR loan IN  
        SELECT LoanID, CustomerID, EndDate  
        FROM Loans  
        WHERE EndDate BETWEEN CURRENT\_DATE AND (CURRENT\_DATE + INTERVAL '30 days')  
    LOOP  
        RAISE NOTICE 'Reminder: Loan % for customer % is due on %', loan.LoanID, loan.CustomerID, loan.EndDate;  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;  
  
  
SELECT SendReminders();

Exercise 2: Error Handling

Scenario 1: Handle exceptions during fund transfers between accounts.

Question: Write a stored procedure SafeTransferFunds that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.

CREATE OR REPLACE FUNCTION SafeTransferFunds(  
    p\_from\_account INT,  
    p\_to\_account INT,  
    p\_amount DECIMAL  
) RETURNS VOID AS $$  
DECLARE  
    v\_balance DECIMAL;  
BEGIN  
    BEGIN  
        SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from\_account;  
        IF v\_balance < p\_amount THEN  
            RAISE EXCEPTION 'Error: Insufficient funds.';  
        END IF;  
  
        UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_account;  
        UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to\_account;  
    EXCEPTION  
        WHEN OTHERS THEN  
            RAISE NOTICE 'Error occurred during fund transfer.';  
    END;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT SafeTransferFunds(1, 2, 100.00);

Scenario 2: Manage errors when updating employee salaries.

Question: Write a stored procedure UpdateSalary that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

CREATE OR REPLACE FUNCTION UpdateSalary(  
    p\_employee\_id INT,  
    p\_percentage DECIMAL  
) RETURNS VOID AS $$  
DECLARE  
    v\_count INT;  
BEGIN  
    BEGIN  
        SELECT COUNT(\*) INTO v\_count FROM Employees WHERE EmployeeID = p\_employee\_id;  
        IF v\_count = 0 THEN  
            RAISE EXCEPTION 'Error: Employee not found.';  
        END IF;  
  
        UPDATE Employees  
        SET Salary = Salary + (Salary \* p\_percentage / 100)  
        WHERE EmployeeID = p\_employee\_id;  
    EXCEPTION  
        WHEN OTHERS THEN  
            RAISE NOTICE 'Error occurred while updating salary.';  
    END;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT UpdateSalary(1, 10.00);

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

Question: Write a stored procedure AddNewCustomer that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

CREATE OR REPLACE FUNCTION AddNewCustomer(  
    p\_customer\_id INT,  
    p\_name VARCHAR,  
    p\_dob DATE,  
    p\_balance DECIMAL  
) RETURNS VOID AS $$  
DECLARE  
    v\_count INT;  
BEGIN  
    BEGIN  
        SELECT COUNT(\*) INTO v\_count FROM Customers WHERE CustomerID = p\_customer\_id;  
        IF v\_count > 0 THEN  
            RAISE EXCEPTION 'Error: Customer already exists.';  
        END IF;  
  
        INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  
        VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, CURRENT\_DATE);  
    EXCEPTION  
        WHEN OTHERS THEN  
            RAISE NOTICE 'Error occurred while adding new customer.';  
    END;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT AddNewCustomer(3, 'New Customer', '1990-01-01', 2000.00);

Exercise 3: Stored Procedures

Scenario 1: The bank needs to process monthly interest for all savings accounts.

Question: Write a stored procedure ProcessMonthlyInterest that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

CREATE OR REPLACE FUNCTION ProcessMonthlyInterest() RETURNS VOID AS $$  
BEGIN  
    UPDATE Accounts  
    SET Balance = Balance + (Balance \* 0.01)  
    WHERE AccountType = 'Savings';  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT ProcessMonthlyInterest();

Scenario 2: The bank wants to implement a bonus scheme for employees based on their performance.

Question: Write a stored procedure UpdateEmployeeBonus that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

CREATE OR REPLACE FUNCTION UpdateEmployeeBonus(  
    p\_department VARCHAR,  
    p\_bonus\_percentage DECIMAL  
) RETURNS VOID AS $$  
BEGIN  
    UPDATE Employees  
    SET Salary = Salary + (Salary \* p\_bonus\_percentage / 100)  
    WHERE Department = p\_department;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT UpdateEmployeeBonus('IT', 5.00);

Scenario 3: Customers should be able to transfer funds between their accounts

.Question: Write a stored procedure TransferFunds that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

CREATE OR REPLACE FUNCTION TransferFunds(  
    p\_from\_account INT,  
    p\_to\_account INT,  
    p\_amount DECIMAL  
) RETURNS VOID AS $$  
DECLARE  
    v\_balance DECIMAL;  
BEGIN  
    BEGIN  
        SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from\_account;  
        IF v\_balance < p\_amount THEN  
            RAISE EXCEPTION 'Error: Insufficient funds.';  
        END IF;  
  
        UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from\_account;  
        UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to\_account;  
    EXCEPTION  
        WHEN OTHERS THEN  
            RAISE NOTICE 'Error occurred during fund transfer.';  
    END;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT TransferFunds(1, 2, 100.00);

Exercise 4: Functions

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

Question: Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURNS INT AS $$  
DECLARE  
    v\_age INT;  
BEGIN  
    SELECT DATE\_PART('year', AGE(p\_dob)) INTO v\_age;  
    RETURN v\_age;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT CalculateAge('1985-05-15');

Scenario 2: The bank needs to compute the monthly installment for a loan.

Question: Write a function CalculateMonthlyInstallment that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(  
    p\_loan\_amount DECIMAL,  
    p\_interest\_rate DECIMAL,  
    p\_duration\_years INT  
) RETURNS DECIMAL AS $$  
DECLARE  
    v\_monthly\_installment DECIMAL;  
BEGIN  
    v\_monthly\_installment = (p\_loan\_amount \* (1 + (p\_interest\_rate / 100) \* p\_duration\_years)) / (p\_duration\_years \* 12);  
    RETURN v\_monthly\_installment;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT CalculateMonthlyInstallment(5000, 5, 5);

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

Question: Write a function HasSufficientBalance that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

CREATE OR REPLACE FUNCTION HasSufficientBalance(  
    p\_account\_id INT,  
    p\_amount DECIMAL  
) RETURNS BOOLEAN AS $$  
DECLARE  
    v\_balance DECIMAL;  
BEGIN  
    SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_account\_id;  
    RETURN v\_balance >= p\_amount;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT HasSufficientBalance(1, 1000.00);

Exercise 5: Triggers

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

Question: Write a trigger UpdateCustomerLastModified that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

CREATE OR REPLACE FUNCTION UpdateCustomerLastModified() RETURNS TRIGGER AS $$  
BEGIN  
    NEW.LastModified = CURRENT\_DATE;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER UpdateCustomerLastModified  
BEFORE UPDATE ON Customers  
FOR EACH ROW  
EXECUTE FUNCTION UpdateCustomerLastModified();

Scenario 2: Maintain an audit log for all transactions.

Question: Write a trigger LogTransaction that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

CREATE TABLE AuditLog (  
AuditID SERIAL PRIMARY KEY,  
TransactionID INT,  
AccountID INT,  
TransactionDate DATE,  
Amount DECIMAL,  
TransactionType VARCHAR(10),  
LogDate TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

CREATE OR REPLACE FUNCTION LogTransaction() RETURNS TRIGGER AS $$  
BEGIN  
INSERT INTO AuditLog (TransactionID, AccountID, TransactionDate, Amount, TransactionType)  
VALUES (NEW.TransactionID, NEW.AccountID, NEW.TransactionDate, NEW.Amount, NEW.TransactionType);  
RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER LogTransaction  
AFTER INSERT ON Transactions  
FOR EACH ROW  
EXECUTE FUNCTION LogTransaction();

Scenario 3: Enforce business rules on deposits and withdrawals.

Question: Write a trigger CheckTransactionRules that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

CREATE OR REPLACE FUNCTION CheckTransactionRules() RETURNS TRIGGER AS $$  
DECLARE  
    v\_balance DECIMAL;  
BEGIN  
    IF NEW.TransactionType = 'Withdrawal' THEN  
        SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = NEW.AccountID;  
        IF v\_balance < NEW.Amount THEN  
            RAISE EXCEPTION 'Error: Insufficient balance for withdrawal.';  
        END IF;  
    END IF;  
  
    IF NEW.Amount <= 0 THEN  
        RAISE EXCEPTION 'Error: Transaction amount must be positive.';  
    END IF;  
  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER CheckTransactionRules  
BEFORE INSERT ON Transactions  
FOR EACH ROW  
EXECUTE FUNCTION CheckTransactionRules();

Exercise 6: Cursors

Scenario 1: Generate monthly statements for all customers.

Question: Write a PL/SQL block using an explicit cursor GenerateMonthlyStatements that retrieves all transactions for the current month and prints a statement for each customer.

SELECT column\_name  
FROM information\_schema.columns  
WHERE table\_name = 'Transactions';  
  
  
  
CREATE OR REPLACE FUNCTION GenerateMonthlyStatements() RETURNS VOID AS $$  
DECLARE  
    cur\_transactions CURSOR FOR  
        SELECT c.CustomerID, t.TransactionType, t.Amount, t.TransactionDate  
        FROM Transactions t  
        JOIN Customers c ON t.AccountID = c.CustomerID  
        WHERE t.TransactionDate BETWEEN DATE\_TRUNC('month', CURRENT\_DATE) AND DATE\_TRUNC('month', CURRENT\_DATE) + INTERVAL '1 month' - INTERVAL '1 day';  
    v\_customer\_id INT;  
    v\_transaction\_type VARCHAR;  
    v\_amount DECIMAL;  
    v\_transaction\_date DATE;  
BEGIN  
    OPEN cur\_transactions;  
    LOOP  
        FETCH cur\_transactions INTO v\_customer\_id, v\_transaction\_type, v\_amount, v\_transaction\_date;  
        EXIT WHEN NOT FOUND;  
        RAISE NOTICE 'Customer % had a % of % on %', v\_customer\_id, v\_transaction\_type, v\_amount, v\_transaction\_date;  
    END LOOP;  
    CLOSE cur\_transactions;  
END;  
$$ LANGUAGE plpgsql;  
  
  
SELECT GenerateMonthlyStatements();

Scenario 2: Apply annual fee to all accounts.

Question: Write a PL/SQL block using an explicit cursor ApplyAnnualFee that deducts an annual maintenance fee from the balance of all accounts.

CREATE OR REPLACE FUNCTION ApplyAnnualFee() RETURNS VOID AS $$  
DECLARE  
    cur\_accounts CURSOR FOR  
        SELECT AccountID, Balance  
        FROM Accounts;  
    v\_account\_id INT;  
    v\_balance DECIMAL;  
    v\_annual\_fee CONSTANT DECIMAL := 50.00;  
BEGIN  
    OPEN cur\_accounts;  
    LOOP  
        FETCH cur\_accounts INTO v\_account\_id, v\_balance;  
        EXIT WHEN NOT FOUND;  
        UPDATE Accounts  
        SET Balance = Balance - v\_annual\_fee  
        WHERE AccountID = v\_account\_id;  
    END LOOP;  
    CLOSE cur\_accounts;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT ApplyAnnualFee();

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

Question: Write a PL/SQL block using an explicit cursor UpdateLoanInterestRates that fetches all loans and updates their interest rates based on the new policy.

CREATE OR REPLACE FUNCTION UpdateLoanInterestRates() RETURNS VOID AS $$  
DECLARE  
    cur\_loans CURSOR FOR  
        SELECT LoanID, InterestRate  
        FROM Loans;  
    v\_loan\_id INT;  
    v\_interest\_rate DECIMAL;  
    v\_new\_interest\_rate CONSTANT DECIMAL := 3.5;  
BEGIN  
    OPEN cur\_loans;  
    LOOP  
        FETCH cur\_loans INTO v\_loan\_id, v\_interest\_rate;  
        EXIT WHEN NOT FOUND;  
        UPDATE Loans  
        SET InterestRate = v\_new\_interest\_rate  
        WHERE LoanID = v\_loan\_id;  
    END LOOP;  
    CLOSE cur\_loans;  
END;  
$$ LANGUAGE plpgsql;  
  
SELECT UpdateLoanInterestRates();

Exercise 7: Packages

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

Question: Create a package CustomerManagement with procedures for adding a new customer, updating customer details, and a function to get customer balance.

CREATE OR REPLACE FUNCTION AddCustomer(  
    p\_customer\_id INT,  
    p\_name VARCHAR,  
    p\_dob DATE,  
    p\_balance DECIMAL  
) RETURNS VOID AS $$  
BEGIN  
    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  
    VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, CURRENT\_DATE);  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE OR REPLACE FUNCTION UpdateCustomerDetails(  
    p\_customer\_id INT,  
    p\_name VARCHAR,  
    p\_dob DATE,  
    p\_balance DECIMAL  
) RETURNS VOID AS $$  
BEGIN  
    UPDATE Customers  
    SET Name = p\_name, DOB = p\_dob, Balance = p\_balance, LastModified = CURRENT\_DATE  
    WHERE CustomerID = p\_customer\_id;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE OR REPLACE FUNCTION GetCustomerBalance(p\_customer\_id INT) RETURNS DECIMAL AS $$  
DECLARE  
    v\_balance DECIMAL;  
BEGIN  
    SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_customer\_id;  
    RETURN v\_balance;  
END;  
$$ LANGUAGE plpgsql;

Scenario 2: Create a package to manage employee data.

Question: Write a package EmployeeManagement with procedures to hire new employees, update employee details, and a function to calculate annual salary.

CREATE OR REPLACE FUNCTION HireEmployee(  
    p\_employee\_id INT,  
    p\_name VARCHAR,  
    p\_position VARCHAR,  
    p\_salary DECIMAL,  
    p\_department VARCHAR,  
    p\_hire\_date DATE  
) RETURNS VOID AS $$  
BEGIN  
    INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)  
    VALUES (p\_employee\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE OR REPLACE FUNCTION UpdateEmployeeDetails(  
    p\_employee\_id INT,  
    p\_name VARCHAR,  
    p\_position VARCHAR,  
    p\_salary DECIMAL,  
    p\_department VARCHAR  
) RETURNS VOID AS $$  
BEGIN  
    UPDATE Employees  
    SET Name = p\_name, Position = p\_position, Salary = p\_salary, Department = p\_department  
    WHERE EmployeeID = p\_employee\_id;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE OR REPLACE FUNCTION CalculateAnnualSalary(p\_employee\_id INT) RETURNS DECIMAL AS $$  
DECLARE  
    v\_salary DECIMAL;  
BEGIN  
    SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_employee\_id;  
    RETURN v\_salary \* 12;  
END;  
$$ LANGUAGE plpgsql;

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

Question: Create a package AccountOperations with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

CREATE OR REPLACE FUNCTION OpenAccount(  
    p\_account\_id INT,  
    p\_customer\_id INT,  
    p\_account\_type VARCHAR,  
    p\_balance DECIMAL  
) RETURNS VOID AS $$  
BEGIN  
    INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)  
    VALUES (p\_account\_id, p\_customer\_id, p\_account\_type, p\_balance, CURRENT\_DATE);  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE OR REPLACE FUNCTION CloseAccount(p\_account\_id INT) RETURNS VOID AS $$  
BEGIN  
    DELETE FROM Accounts WHERE AccountID = p\_account\_id;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE OR REPLACE FUNCTION GetTotalBalance(p\_customer\_id INT) RETURNS DECIMAL AS $$  
DECLARE  
    v\_total\_balance DECIMAL;  
BEGIN  
    SELECT SUM(Balance) INTO v\_total\_balance FROM Accounts WHERE CustomerID = p\_customer\_id;  
    RETURN v\_total\_balance;  
END;  
$$ LANGUAGE plpgsql;