**PL/SQL HANDS ON:**

**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.

**COMMANDS:**

SET SERVEROUTPUT ON;

DELETE FROM Customers WHERE CustomerID = 999;

DELETE FROM Loans WHERE CustomerID = 999;

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

VALUES (999, 'Senior Citizen', TO\_DATE('1950-01-01', 'YYYY-MM-DD'), 5000, SYSDATE);

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

VALUES (999, 999, 10000, 7, SYSDATE, ADD\_MONTHS(SYSDATE, 24));

BEGIN

  FOR rec IN (

    SELECT c.CustomerID, c.Name, c.DOB, l.LoanID, l.InterestRate

    FROM Customers c

    JOIN Loans l ON c.CustomerID = l.CustomerID

  ) LOOP

    IF TRUNC(MONTHS\_BETWEEN(SYSDATE, rec.DOB)/12) > 60 THEN

      UPDATE Loans

      SET InterestRate = InterestRate - 1

      WHERE LoanID = rec.LoanID;

      DBMS\_OUTPUT.PUT\_LINE('Discount applied to: ' || rec.Name ||

                           ' | New Interest Rate Applied.');

    ELSE

      DBMS\_OUTPUT.PUT\_LINE('No discount: ' || rec.Name);

    END IF;

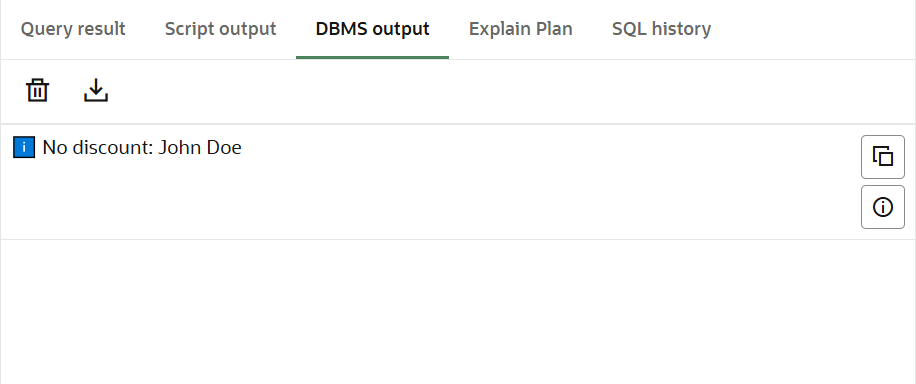
  END LOOP;

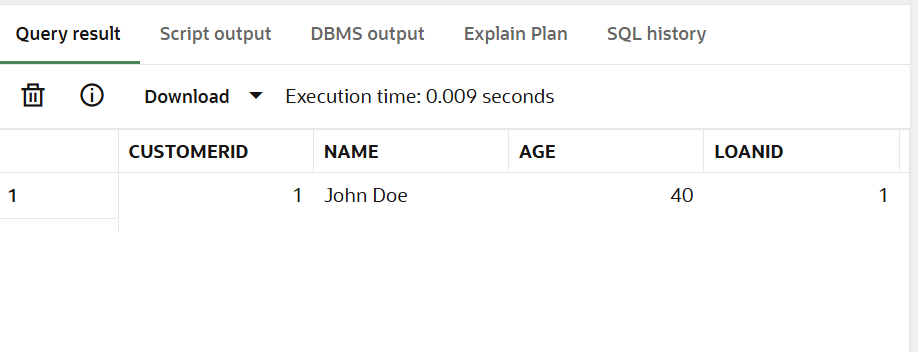
  COMMIT;

END;

/

**OUTPUT:**





**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.

**COMMANDS:**

SET SERVEROUTPUT ON;

BEGIN

  FOR rec IN (

    SELECT CustomerID, Name, Balance, IsVIP FROM Customers

  ) LOOP

    IF rec.Balance > 10000 THEN

      UPDATE Customers

      SET IsVIP = 'TRUE'

      WHERE CustomerID = rec.CustomerID;

      DBMS\_OUTPUT.PUT\_LINE(' VIP status granted to: ' || rec.Name ||

                           ' | Balance: $' || rec.Balance);

    ELSE

      DBMS\_OUTPUT.PUT\_LINE(' Not eligible for VIP: ' || rec.Name ||

                           ' | Balance: $' || rec.Balance);

    END IF;

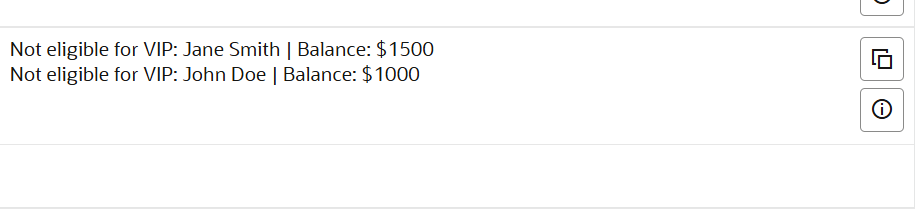
  END LOOP;

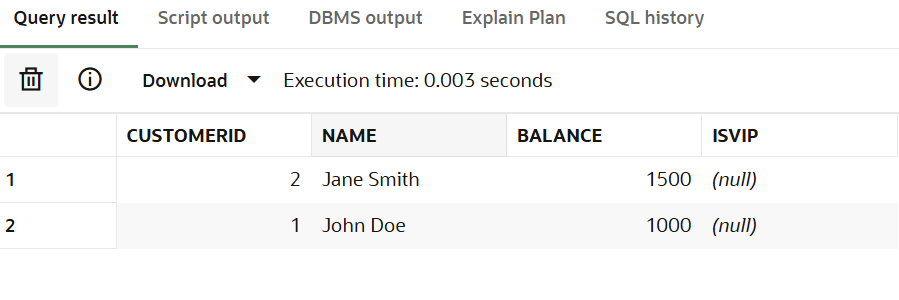
  COMMIT;

END;

/

**OUTPUT:**





**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.

**COMMANDS:**

BEGIN

  FOR loan\_rec IN (

    SELECT l.LoanID, c.Name, c.CustomerID

    FROM Loans l

    JOIN Customers c ON l.CustomerID = c.CustomerID

    WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

  )

  LOOP

    DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ' || loan\_rec.Name ||

                         ' (ID: ' || loan\_rec.CustomerID ||

                         ') has loan ' || loan\_rec.LoanID ||

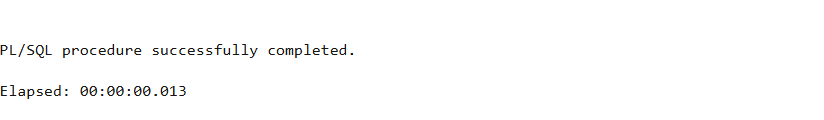
                         ' due within 30 days.');

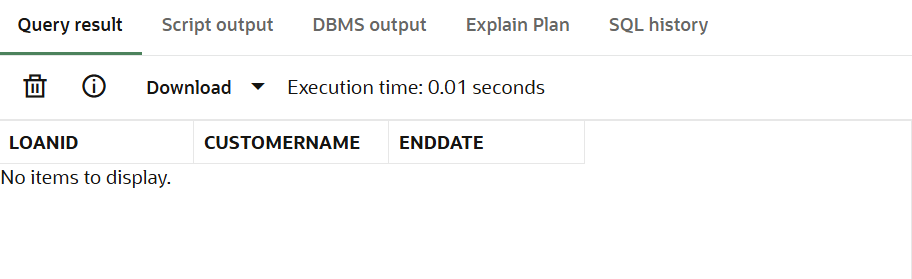
  END LOOP;

END;

/

**OUTPUT:**





**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.

**COMMANDS:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

  p\_from\_acc NUMBER,

  p\_to\_acc NUMBER,

  p\_amount NUMBER

) AS

  v\_balance NUMBER;

BEGIN

  SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from\_acc;

  IF v\_balance < p\_amount THEN

    DBMS\_OUTPUT.PUT\_LINE(' Insufficient funds in Account ' || p\_from\_acc);

  ELSE

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

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

    DBMS\_OUTPUT.PUT\_LINE(' Transferred ' || p\_amount || ' from Account ' || p\_from\_acc || ' to Account ' || p\_to\_acc);

    COMMIT;

  END IF;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

    DBMS\_OUTPUT.PUT\_LINE(' Account not found.');

  WHEN OTHERS THEN

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

    ROLLBACK;

END;

/

SET SERVEROUTPUT ON;

BEGIN

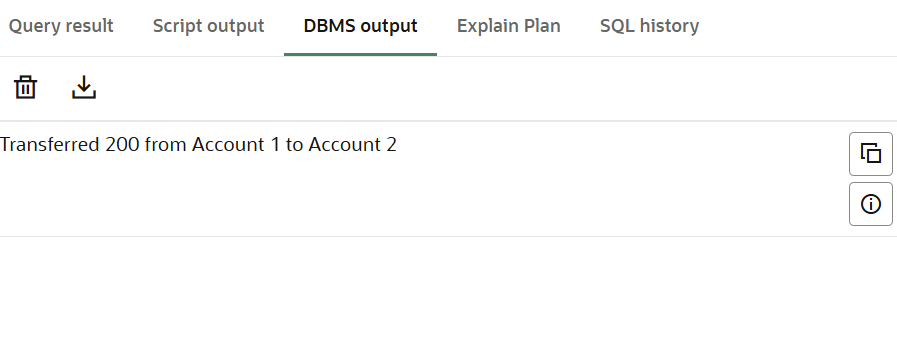
  SafeTransferFunds(1, 2, 200);

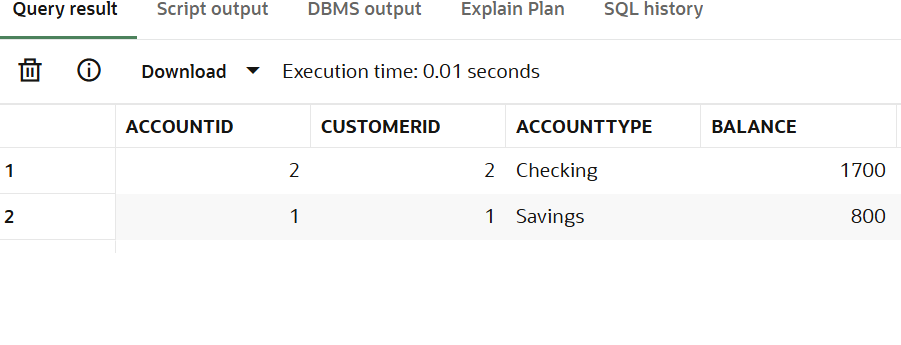
END;

/

SELECT \* FROM Accounts;

**OUTPUT :**





**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.

**COMMANDS:**

CREATE OR REPLACE PROCEDURE UpdateSalary(

  p\_emp\_id NUMBER,

  p\_percent NUMBER

) AS

BEGIN

  UPDATE Employees

  SET Salary = Salary + (Salary \* p\_percent / 100)

  WHERE EmployeeID = p\_emp\_id;

  IF SQL%ROWCOUNT = 0 THEN

    DBMS\_OUTPUT.PUT\_LINE(' Employee ID ' || p\_emp\_id || ' not found.');

  ELSE

    DBMS\_OUTPUT.PUT\_LINE(' Salary updated for Employee ID ' || p\_emp\_id);

  END IF;

  COMMIT;

EXCEPTION

  WHEN OTHERS THEN

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

    ROLLBACK;

END;

/

SET SERVEROUTPUT ON;

BEGIN

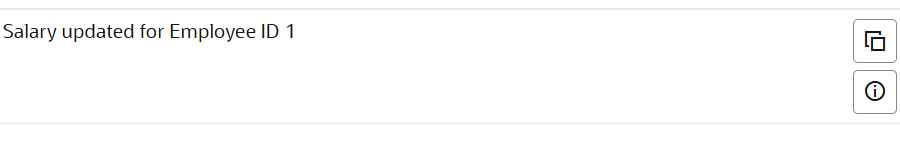
 UpdateSalary(1, 10);

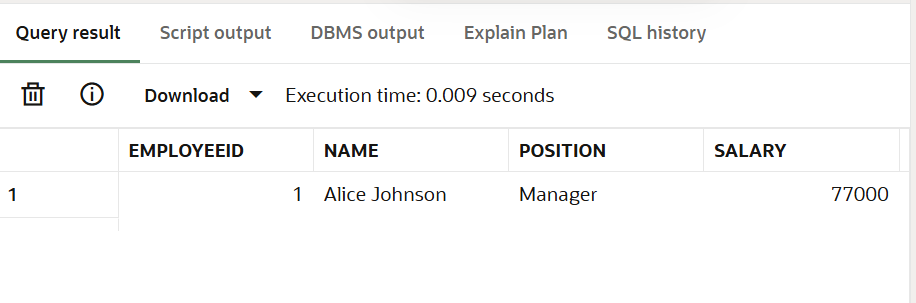
END;

/

SELECT \* FROM Employees WHERE EmployeeID = 1;

**OUTPUT :**





**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.

**COMMANDS:**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

  p\_id NUMBER,

  p\_name VARCHAR2,

  p\_dob DATE,

  p\_balance NUMBER

) AS

BEGIN

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

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

  DBMS\_OUTPUT.PUT\_LINE(' Customer added: ' || p\_name);

  COMMIT;

EXCEPTION

  WHEN DUP\_VAL\_ON\_INDEX THEN

    DBMS\_OUTPUT.PUT\_LINE(' Customer ID ' || p\_id || ' already exists.');

    ROLLBACK;

  WHEN OTHERS THEN

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

    ROLLBACK;

END;

/

SET SERVEROUTPUT ON;

BEGIN

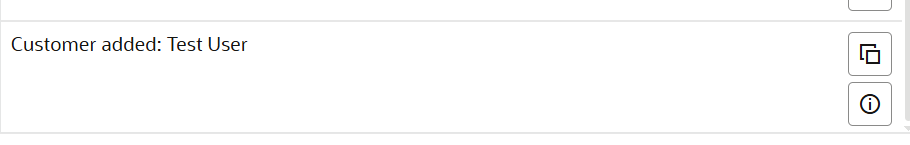
  AddNewCustomer(999, 'Test User', TO\_DATE('1990-01-01', 'YYYY-MM-DD'), 5000);

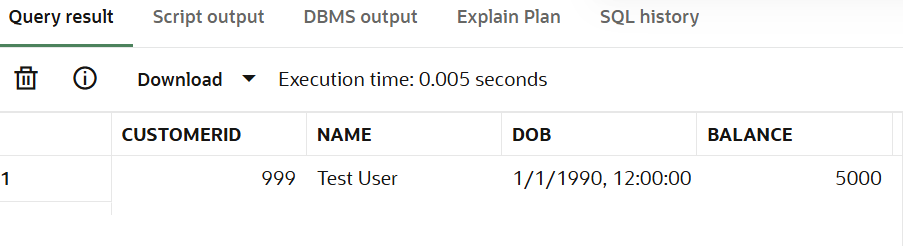
END;

/

SELECT \* FROM Customers WHERE CustomerID = 999;

**OUTPUT:**





**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.

**COMMANDS:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

  FOR acc\_rec IN (

    SELECT AccountID, Balance

    FROM Accounts

    WHERE AccountType = 'Savings'

  ) LOOP

    UPDATE Accounts

    SET Balance = Balance + (acc\_rec.Balance \* 0.01)

    WHERE AccountID = acc\_rec.AccountID;

    DBMS\_OUTPUT.PUT\_LINE(' Interest applied to Account ID: ' || acc\_rec.AccountID);

  END LOOP;

  COMMIT;

END;

/

SET SERVEROUTPUT ON;

BEGIN

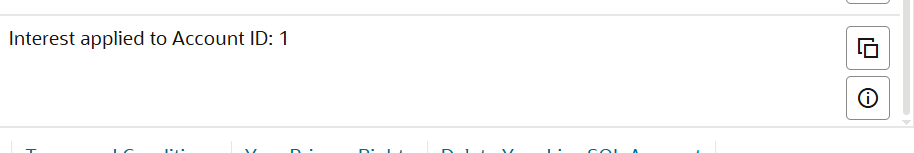
  ProcessMonthlyInterest;

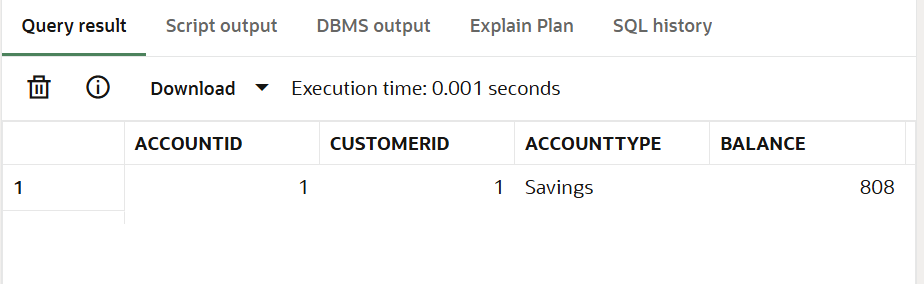
END;

/

SELECT \* FROM Accounts WHERE AccountType = 'Savings';

**OUTPUT**





**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.

**COMMANDS:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

  p\_dept VARCHAR2,

  p\_bonus\_percent NUMBER

) IS

BEGIN

  FOR emp\_rec IN (

    SELECT EmployeeID, Salary FROM Employees WHERE Department = p\_dept

  ) LOOP

    UPDATE Employees

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

    WHERE EmployeeID = emp\_rec.EmployeeID;

    DBMS\_OUTPUT.PUT\_LINE(' Bonus added to Employee ID: ' || emp\_rec.EmployeeID);

  END LOOP;

  COMMIT;

END;

/

SET SERVEROUTPUT ON;

BEGIN

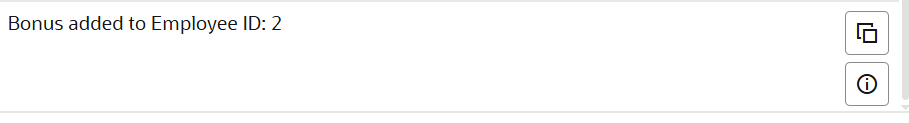
  UpdateEmployeeBonus('IT', 10);

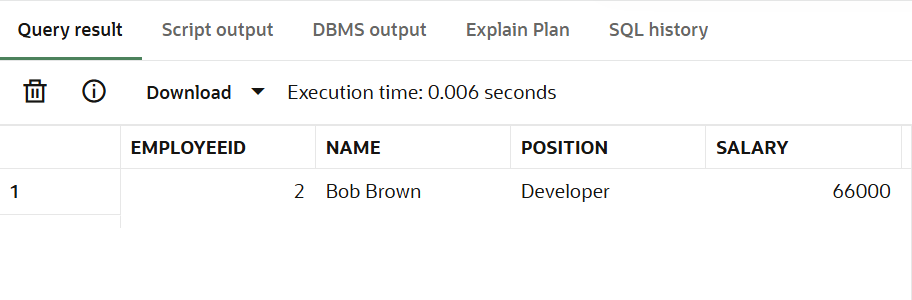
END;

/

SELECT \* FROM Employees WHERE Department = 'IT';

OUTPUT:





**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.

**COMMANDS:**

CREATE OR REPLACE PROCEDURE TransferFunds(

  p\_from\_acc NUMBER,

  p\_to\_acc NUMBER,

  p\_amount NUMBER

) IS

  v\_balance NUMBER;

BEGIN

  SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from\_acc;

  IF v\_balance < p\_amount THEN

    DBMS\_OUTPUT.PUT\_LINE(' Insufficient balance in Account ID: ' || p\_from\_acc);

  ELSE

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

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

    DBMS\_OUTPUT.PUT\_LINE(' Transferred ' || p\_amount || ' from Account ' || p\_from\_acc || ' to Account ' || p\_to\_acc);

    COMMIT;

  END IF;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

    DBMS\_OUTPUT.PUT\_LINE(' One of the accounts was not found.');

  WHEN OTHERS THEN

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

    ROLLBACK;

END;

/

SET SERVEROUTPUT ON;

BEGIN

  TransferFunds(1, 2, 100);

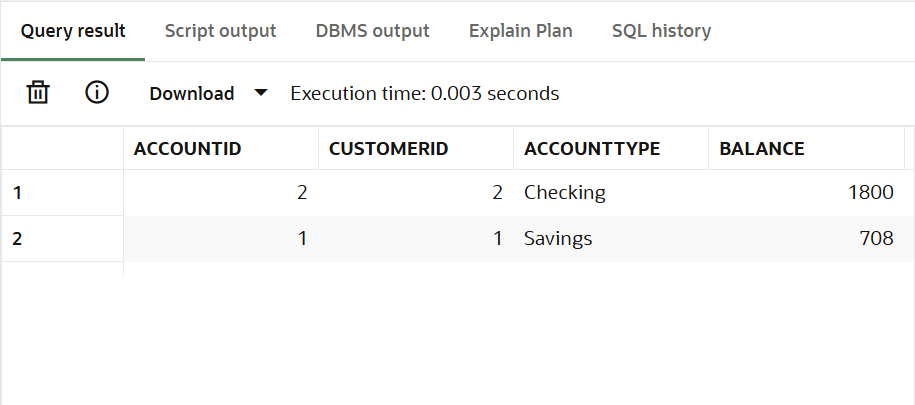
END;

/

SELECT \* FROM Accounts;

**OUTPUT:**





**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.

**COMMANDS:**

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE)

RETURN NUMBER IS

  v\_age NUMBER;

BEGIN

  v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

  RETURN v\_age;

END;

/

SET SERVEROUTPUT ON;

DECLARE

  v\_age NUMBER;

BEGIN

  SELECT CalculateAge(DOB) INTO v\_age

  FROM Customers

  WHERE CustomerID = 999; -- Change to an existing ID

  DBMS\_OUTPUT.PUT\_LINE(' Age of customer 999: ' || v\_age || ' years');

END;

/

SELECT

  CustomerID,

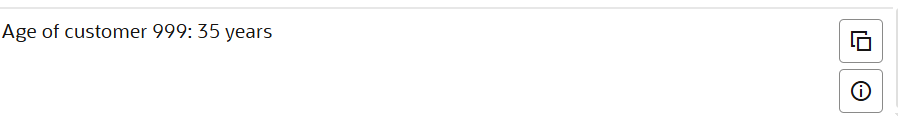
  Name,

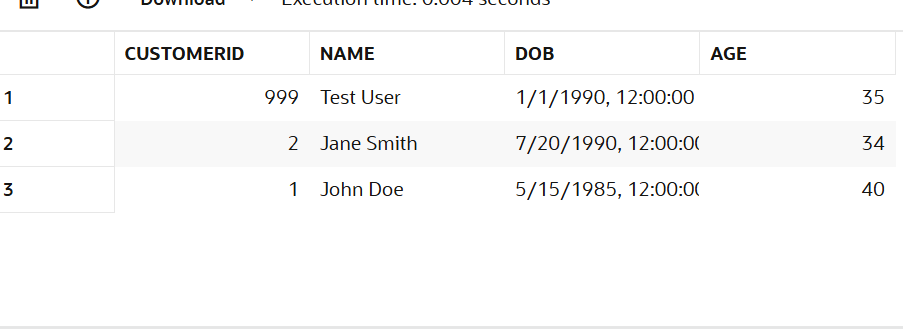
  DOB,

  TRUNC(MONTHS\_BETWEEN(SYSDATE, DOB) / 12) AS Age

FROM Customers;

OUTPUT:





**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.

**COMMANDS:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

  p\_amount NUMBER,

  p\_rate   NUMBER,

  p\_years  NUMBER

) RETURN NUMBER IS

  v\_monthly NUMBER;

  r NUMBER := p\_rate / 12 / 100;

  n NUMBER := p\_years \* 12;

BEGIN

  v\_monthly := (p\_amount \* r \* POWER(1 + r, n)) /

               (POWER(1 + r, n) - 1);

  RETURN ROUND(v\_monthly, 2);

END;

/

SET SERVEROUTPUT ON;

DECLARE

  v\_emi NUMBER;

BEGIN

  v\_emi := CalculateMonthlyInstallment(10000, 10, 2); -- 10k, 10% interest, 2 years

  DBMS\_OUTPUT.PUT\_LINE(' Monthly EMI: $' || v\_emi);

END;

/

SELECT

  LoanID,

  CustomerID,

  LoanAmount,

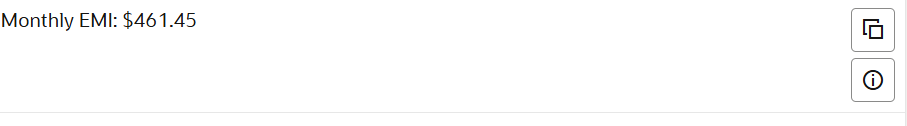
  InterestRate,

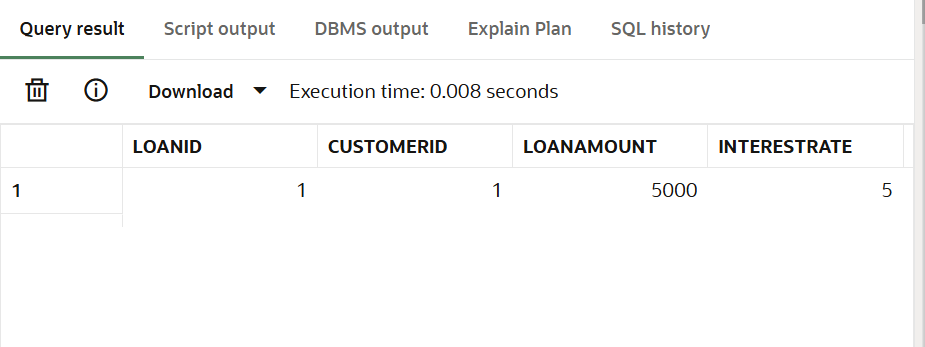
  StartDate,

  EndDate

FROM Loans;

OUTPUT:





**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.

**COMMANDS:**

CREATE OR REPLACE FUNCTION HasSufficientBalance(

  p\_acc\_id NUMBER,

  p\_amount NUMBER

) RETURN BOOLEAN IS

  v\_balance NUMBER;

BEGIN

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

  RETURN v\_balance >= p\_amount;

EXCEPTION

  WHEN NO\_DATA\_FOUND THEN

    RETURN FALSE;

END;

/

SET SERVEROUTPUT ON;

DECLARE

  result BOOLEAN;

BEGIN

  result := HasSufficientBalance(1, 1000);

  IF result THEN

    DBMS\_OUTPUT.PUT\_LINE(' Account has sufficient balance.');

  ELSE

    DBMS\_OUTPUT.PUT\_LINE(' Account does NOT have sufficient balance.');

  END IF;

END;

/

SELECT

  AccountID,

  CustomerID,

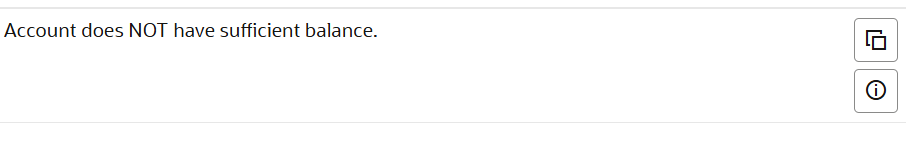
  Balance,

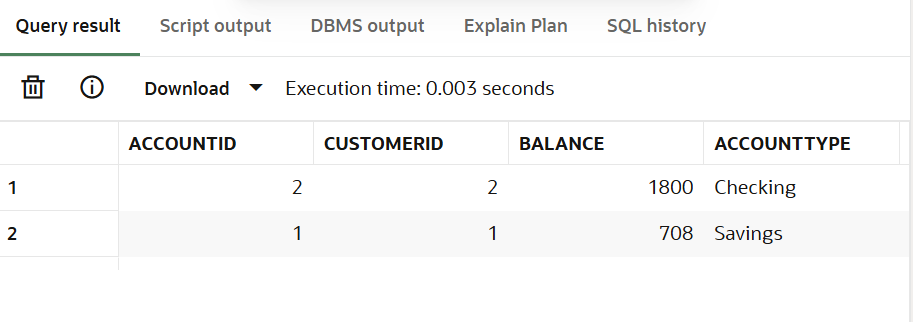
  AccountType,

  LastModified

FROM Accounts;

**OUTPUT:**





**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.

**COMMANDS:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

  :NEW.LastModified := SYSDATE;

  DBMS\_OUTPUT.PUT\_LINE(' LastModified updated for Customer ID: ' || :NEW.CustomerID);

END;

/

SET SERVEROUTPUT ON;

UPDATE Customers

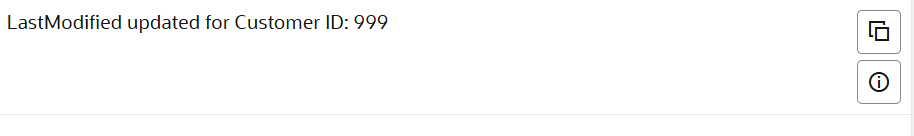
SET Name = 'Updated Name'

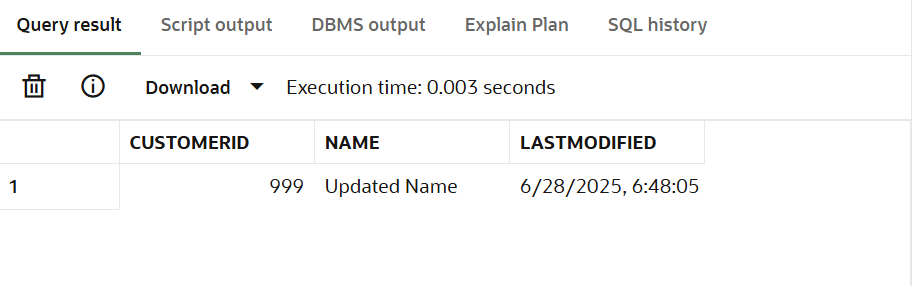
WHERE CustomerID = 999;

COMMIT;

SELECT CustomerID, Name, LastModified FROM Customers WHERE CustomerID = 999;

**OUTPUT:**





**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.

**COMMANDS:**

CREATE TABLE AuditLog (

  AuditID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

  AccountID NUMBER,

  Amount NUMBER,

  TransactionType VARCHAR2(10),

  LoggedAt DATE DEFAULT SYSDATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

  INSERT INTO AuditLog (AccountID, Amount, TransactionType)

  VALUES (:NEW.AccountID, :NEW.Amount, :NEW.TransactionType);

  DBMS\_OUTPUT.PUT\_LINE(' Logged transaction for Account ' || :NEW.AccountID);

END;

/

SET SERVEROUTPUT ON;

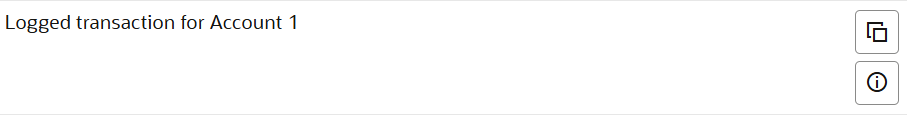
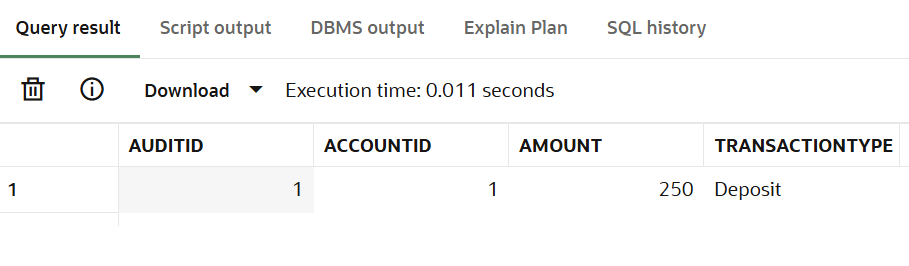
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (99, 1, SYSDATE, 250, 'Deposit');

COMMIT;

SELECT \* FROM AuditLog ORDER BY AuditID DESC;

**OUTPUT:**

**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.

**COMMANDS:**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

  v\_balance NUMBER;

BEGIN

  -- Get current account balance

  SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;

  IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v\_balance THEN

    RAISE\_APPLICATION\_ERROR(-20001, ' Withdrawal exceeds balance');

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

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

  END IF;

END;

/

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

VALUES (100, 1, SYSDATE, 999999, 'Withdrawal');

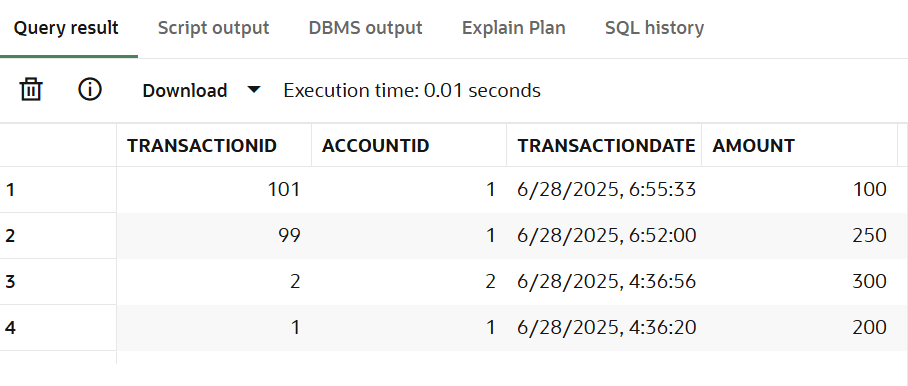
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (101, 1, SYSDATE, 100, 'Deposit');

COMMIT;

SELECT \* FROM Transactions ORDER BY TransactionID DESC;

**OUTPUT:**



**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.

**COMMANDS:**

SET SERVEROUTPUT ON;

DECLARE

  CURSOR cur\_transactions IS

    SELECT t.TransactionID, c.Name AS CustomerName, t.Amount, t.TransactionDate, t.TransactionType

    FROM Transactions t

    JOIN Accounts a ON t.AccountID = a.AccountID

    JOIN Customers c ON a.CustomerID = c.CustomerID

    WHERE TO\_CHAR(t.TransactionDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY');

  rec cur\_transactions%ROWTYPE;

BEGIN

  OPEN cur\_transactions;

  LOOP

    FETCH cur\_transactions INTO rec;

    EXIT WHEN cur\_transactions%NOTFOUND;

    DBMS\_OUTPUT.PUT\_LINE(' Statement - ' || rec.CustomerName ||

                         ': ' || rec.TransactionType || ' of $' || rec.Amount ||

                         ' on ' || TO\_CHAR(rec.TransactionDate, 'DD-Mon-YYYY'));

  END LOOP;

  CLOSE cur\_transactions;

END;

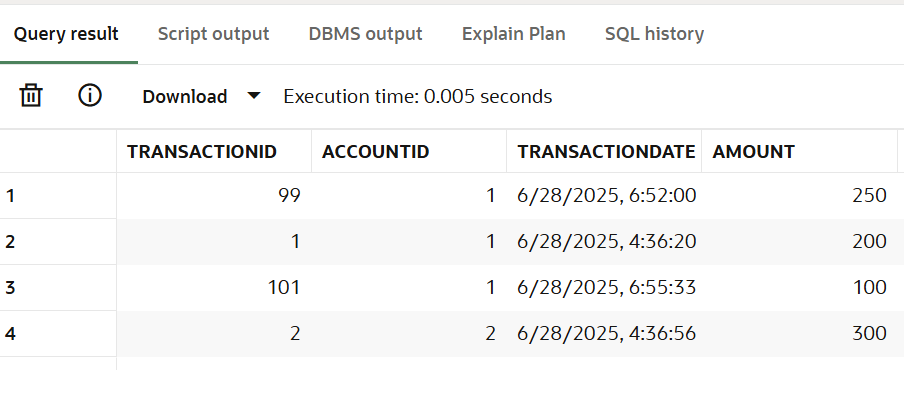
/

SELECT \* FROM Transactions

WHERE TO\_CHAR(TransactionDate, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY');

**OUTPUT:**





**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.

**COMMANDS:**

SET SERVEROUTPUT ON;

DECLARE

  CURSOR cur\_accounts IS

    SELECT AccountID, Balance FROM Accounts;

  rec cur\_accounts%ROWTYPE;

  fee NUMBER := 50;

BEGIN

  OPEN cur\_accounts;

  LOOP

    FETCH cur\_accounts INTO rec;

    EXIT WHEN cur\_accounts%NOTFOUND;

    UPDATE Accounts

    SET Balance = Balance - fee

    WHERE AccountID = rec.AccountID;

    DBMS\_OUTPUT.PUT\_LINE('$' || fee || ' fee applied to Account ID: ' || rec.AccountID);

  END LOOP;

  CLOSE cur\_accounts;

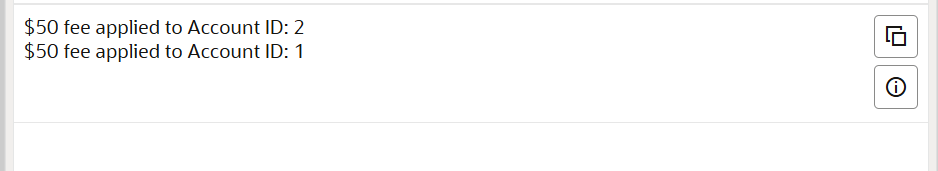
  COMMIT;

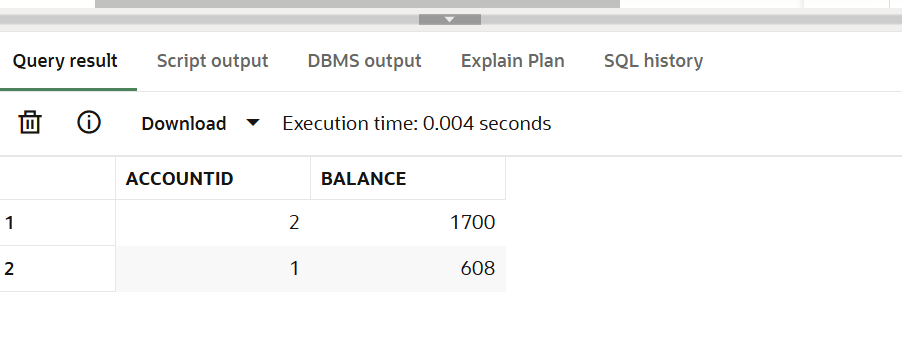
END;

/

SELECT AccountID, Balance FROM Accounts;

**OUTPUT:**

****

****

**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.

**COMMANDS:**

SET SERVEROUTPUT ON;

DECLARE

  CURSOR cur\_loans IS

    SELECT LoanID, LoanAmount, InterestRate FROM Loans;

  rec cur\_loans%ROWTYPE;

BEGIN

  OPEN cur\_loans;

  LOOP

    FETCH cur\_loans INTO rec;

    EXIT WHEN cur\_loans%NOTFOUND;

    IF rec.LoanAmount > 10000 THEN

      UPDATE Loans

      SET InterestRate = InterestRate - 0.5

      WHERE LoanID = rec.LoanID;

      DBMS\_OUTPUT.PUT\_LINE('Interest reduced for Loan ' || rec.LoanID);

    ELSE

      UPDATE Loans

      SET InterestRate = InterestRate + 0.5

      WHERE LoanID = rec.LoanID;

      DBMS\_OUTPUT.PUT\_LINE('Interest increased for Loan ' || rec.LoanID);

    END IF;

  END LOOP;

  CLOSE cur\_loans;

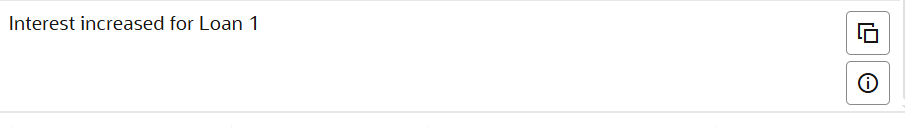
  COMMIT;

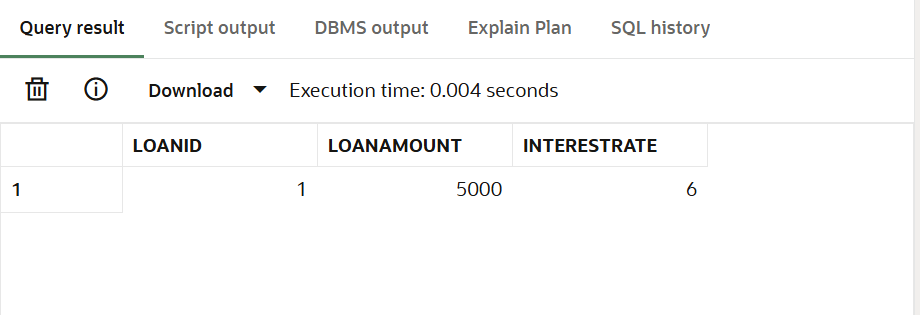
END;

/

SELECT LoanID, LoanAmount, InterestRate FROM Loans;

**OUTPUT:**

****



**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.

**COMMANDS:**

CREATE OR REPLACE PACKAGE CustomerManagement AS

  PROCEDURE AddCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER);

  PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2);

  FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

  PROCEDURE AddCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER) IS

  BEGIN

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

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

    DBMS\_OUTPUT.PUT\_LINE(' Customer added: ' || p\_name);

    COMMIT;

  EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

      DBMS\_OUTPUT.PUT\_LINE(' Customer ID already exists.');

  END;

  PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2) IS

  BEGIN

    UPDATE Customers SET Name = p\_name, LastModified = SYSDATE

    WHERE CustomerID = p\_id;

    DBMS\_OUTPUT.PUT\_LINE(' Updated customer: ' || p\_id);

    COMMIT;

  END;

  FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS

    v\_balance NUMBER;

  BEGIN

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

    RETURN v\_balance;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

      RETURN NULL;

  END;

END CustomerManagement;

/

SET SERVEROUTPUT ON;

BEGIN

  CustomerManagement.AddCustomer(1111, 'PackageUser', TO\_DATE('1995-01-01','YYYY-MM-DD'), 7000);

  CustomerManagement.UpdateCustomer(1111, 'UpdatedPackageUser');

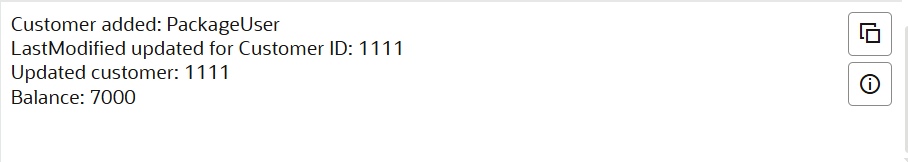
  DBMS\_OUTPUT.PUT\_LINE('Balance: ' || CustomerManagement.GetCustomerBalance(1111));

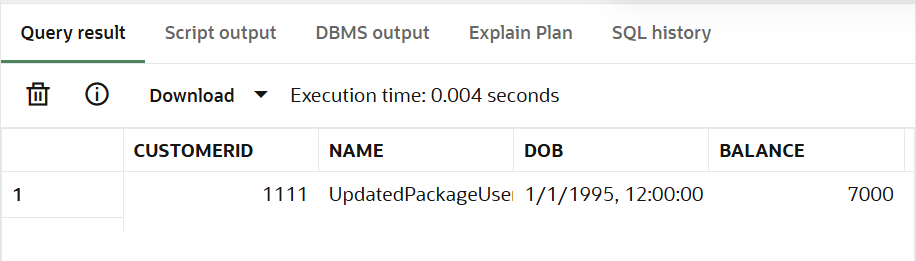
END;

/

SELECT \* FROM Customers WHERE CustomerID = 1111;

**OUTPUT:**





**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.

**COMMANDS:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

  PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2, p\_hire DATE);

  PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER);

  FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

  PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2, p\_hire DATE) IS

  BEGIN

    INSERT INTO Employees VALUES(p\_id, p\_name, p\_position, p\_salary, p\_dept, p\_hire);

    DBMS\_OUTPUT.PUT\_LINE(' Hired: ' || p\_name);

    COMMIT;

  END;

  PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER) IS

  BEGIN

    UPDATE Employees SET Salary = p\_salary WHERE EmployeeID = p\_id;

    DBMS\_OUTPUT.PUT\_LINE(' Salary updated for Employee ID: ' || p\_id);

    COMMIT;

  END;

  FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

    v\_salary NUMBER;

  BEGIN

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

    RETURN v\_salary;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

      RETURN NULL;

  END;

END EmployeeManagement;

/

SET SERVEROUTPUT ON;

BEGIN

  EmployeeManagement.HireEmployee(5555, 'PackEmployee', 'Analyst', 5000, 'R&D', SYSDATE);

  EmployeeManagement.UpdateEmployee(5555, 6000);

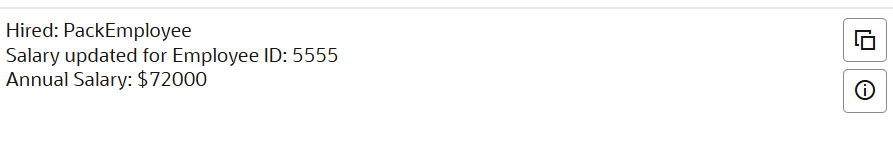
  DBMS\_OUTPUT.PUT\_LINE(' Annual Salary: $' || EmployeeManagement.GetAnnualSalary(5555));

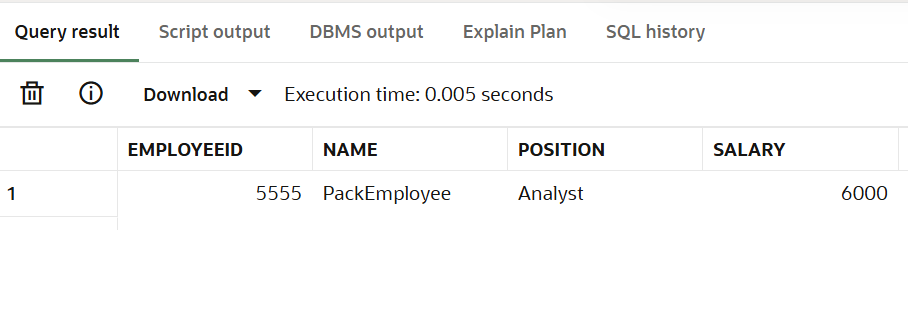
END;

/

SELECT \* FROM Employees WHERE EmployeeID = 5555;

**OUTPUT:**

****



**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.

**COMMANDS:**

CREATE OR REPLACE PACKAGE AccountOperations AS

  PROCEDURE OpenAccount(p\_acc\_id NUMBER, p\_cust\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

  PROCEDURE CloseAccount(p\_acc\_id NUMBER);

  FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

  PROCEDURE OpenAccount(p\_acc\_id NUMBER, p\_cust\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

  BEGIN

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

    VALUES (p\_acc\_id, p\_cust\_id, p\_type, p\_balance, SYSDATE);

    DBMS\_OUTPUT.PUT\_LINE(' Account opened: ' || p\_acc\_id);

    COMMIT;

  END;

  PROCEDURE CloseAccount(p\_acc\_id NUMBER) IS

  BEGIN

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

    DBMS\_OUTPUT.PUT\_LINE(' Account closed: ' || p\_acc\_id);

    COMMIT;

  END;

  FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER IS

    v\_total NUMBER;

  BEGIN

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

    RETURN v\_total;

  END;

END AccountOperations;

/

SET SERVEROUTPUT ON;

BEGIN

  AccountOperations.OpenAccount(2022, 999, 'Savings', 9000);

  DBMS\_OUTPUT.PUT\_LINE(' Total Balance for Customer 999: ' || AccountOperations.GetTotalBalance(999));

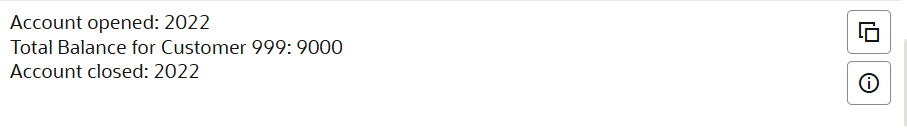
  AccountOperations.CloseAccount(2022);

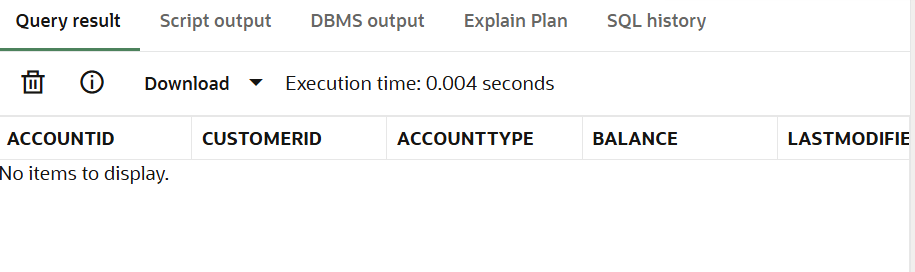
END;

/

SELECT \* FROM Accounts WHERE CustomerID = 999;

**OUTPUT:**



****