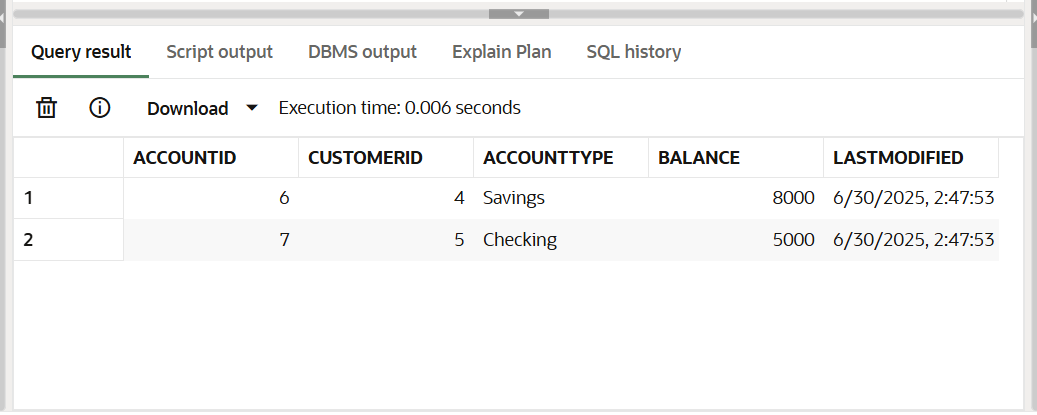
**Exercise 2: Error Handling**

**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.

**CODE:**

SELECT \* FROM Accounts WHERE AccountID IN (6, 7);



**PL SQL CODE:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

sourceID IN NUMBER,

targetID IN NUMBER,

amount IN NUMBER

) AS

src\_balance NUMBER;

BEGIN

SELECT Balance INTO src\_balance FROM Accounts WHERE AccountID = sourceID;

IF src\_balance < amount THEN

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

END IF;

UPDATE Accounts

SET Balance = Balance - amount, LastModified = SYSDATE

WHERE AccountID = sourceID;

UPDATE Accounts

SET Balance = Balance + amount, LastModified = SYSDATE

WHERE AccountID = targetID;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

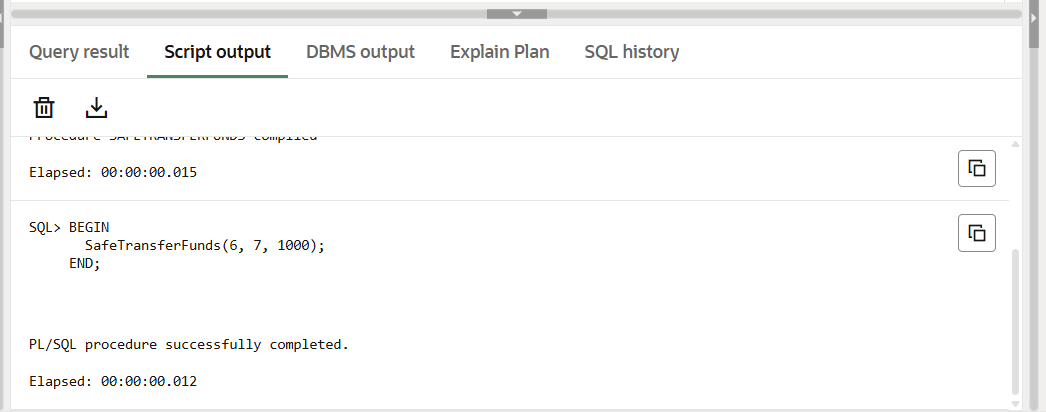
BEGIN

SafeTransferFunds(6, 7, 1000);

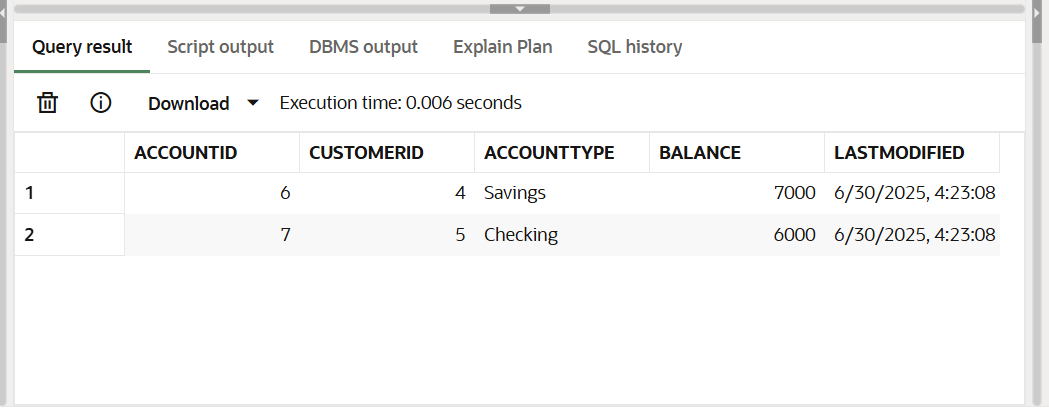
END;

/

**OUTPUT:**



SELECT \* FROM Accounts WHERE AccountID IN (6, 7);

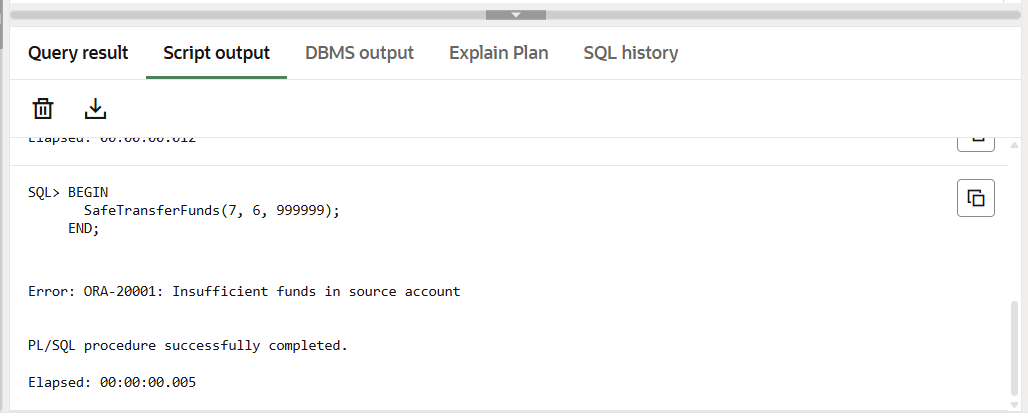


BEGIN

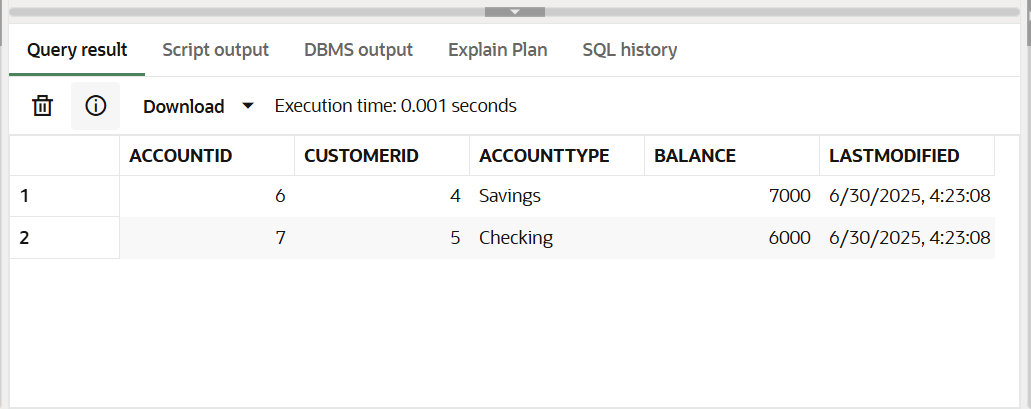
SafeTransferFunds(7, 6, 999999);

END;

/

**OUTPUT:**

SELECT \* FROM Accounts WHERE AccountID IN (6, 7);

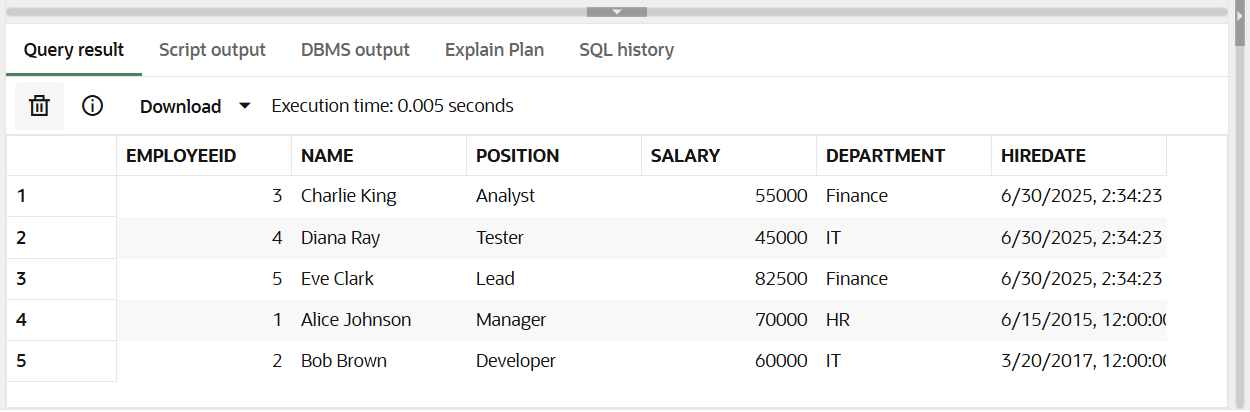


**Scenario 2:** Manage errors when updating employee salaries.

**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.

**CODE:**

SELECT \* FROM Employees;



**PL SQL CODE:**

CREATE OR REPLACE PROCEDURE UpdateSalary (

empID IN NUMBER,

percent IN NUMBER

) AS

current\_salary NUMBER;

BEGIN

-- Get current salary

SELECT Salary INTO current\_salary FROM Employees WHERE EmployeeID = empID;

-- Update salary

UPDATE Employees

SET Salary = current\_salary + (current\_salary \* percent / 100)

WHERE EmployeeID = empID;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID ' || empID || ' does not exist.');

WHEN OTHERS THEN

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

END;

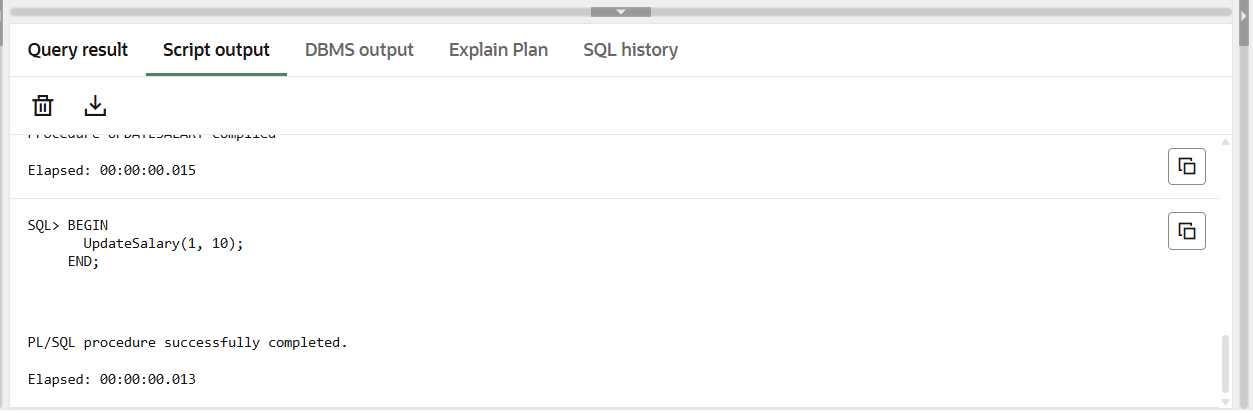
/

BEGIN

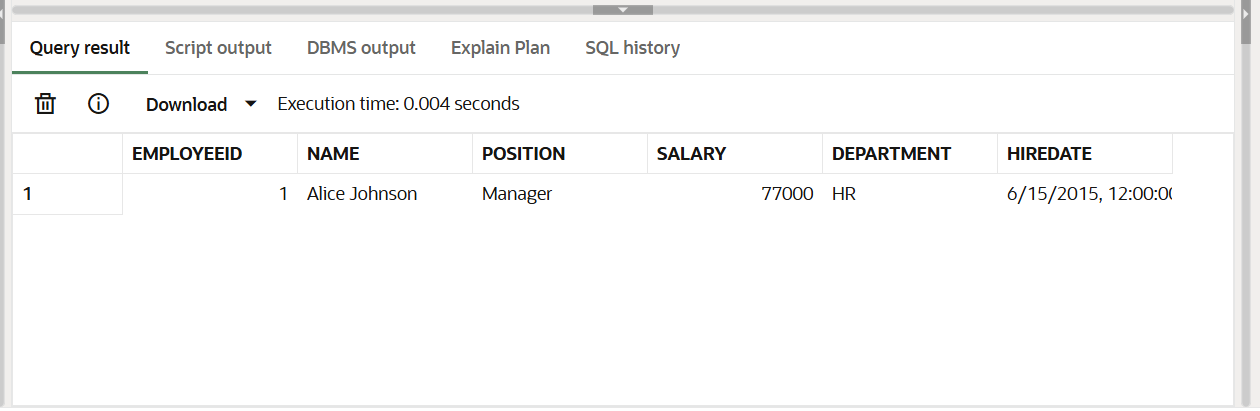
UpdateSalary(1, 10);

END;

/



SELECT \* FROM Employees WHERE EmployeeID = 1;



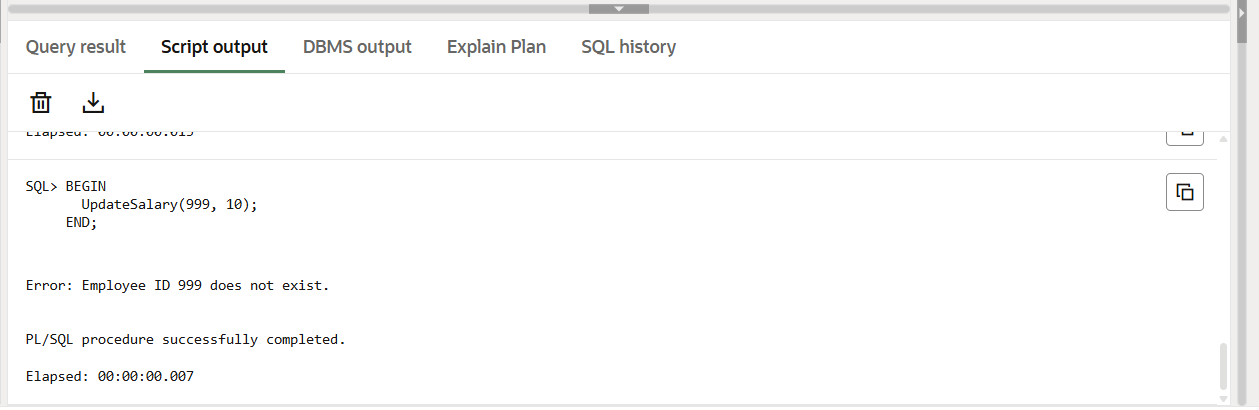
BEGIN

UpdateSalary(999, 10);

END;

/

**OUTPUT:**

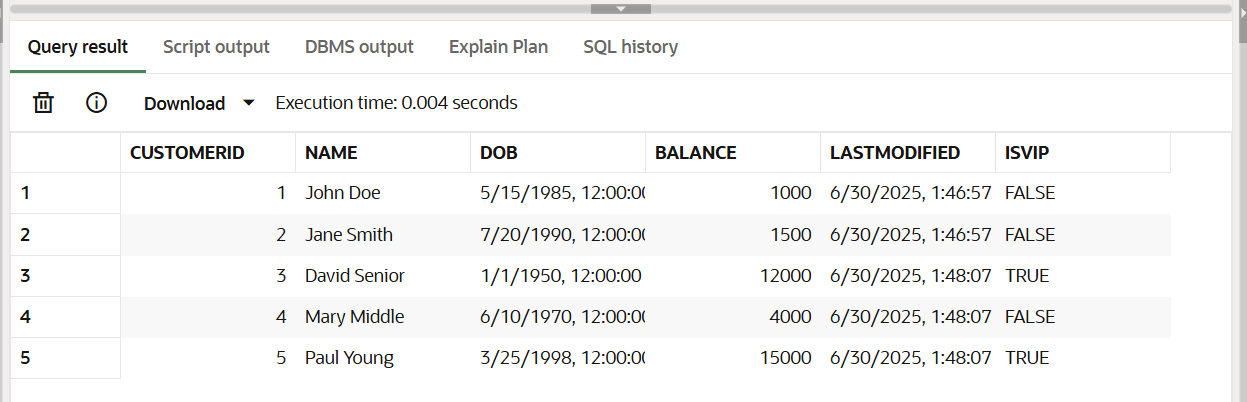


**Scenario 3:** Ensure data integrity when adding a new customer.

**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.

**CODE:**

SELECT \* FROM Customers;



**PL SQL CODE:**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

custID IN NUMBER,

custName IN VARCHAR2,

dob IN DATE,

balance IN NUMBER

) AS

BEGIN

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

VALUES (custID, custName, dob, balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || custID || ' already exists.');

WHEN OTHERS THEN

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

END;

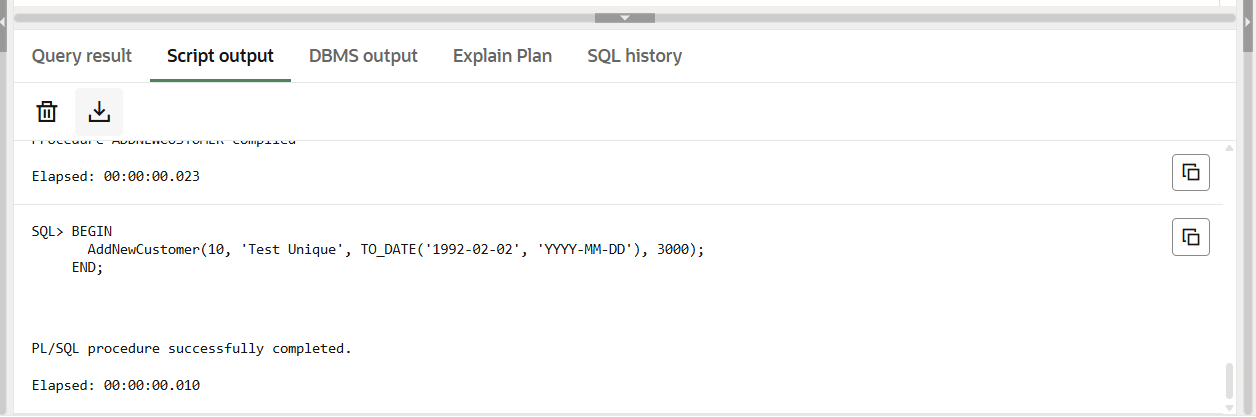
/

BEGIN

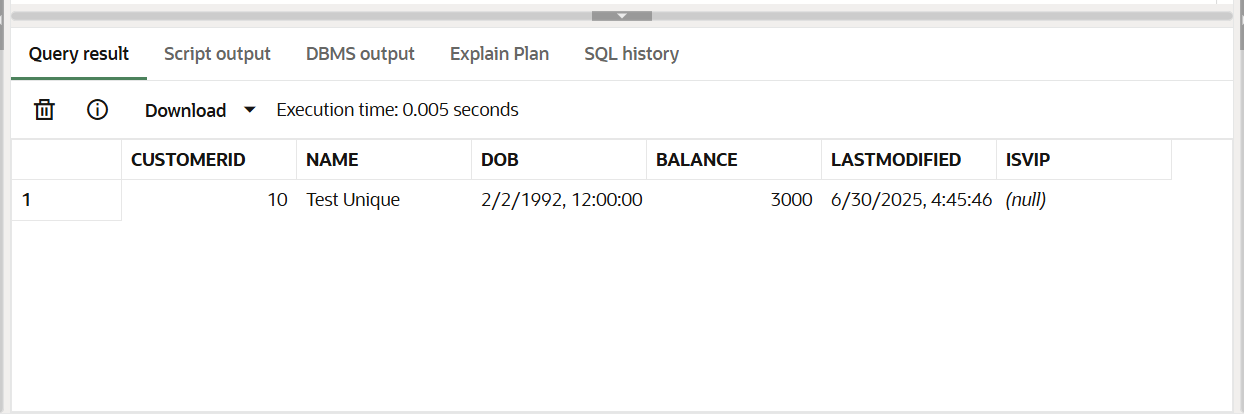
AddNewCustomer(10, 'Test Unique', TO\_DATE('1992-02-02', 'YYYY-MM-DD'), 3000);

END;

/



SELECT \* FROM Customers WHERE CustomerID = 10;



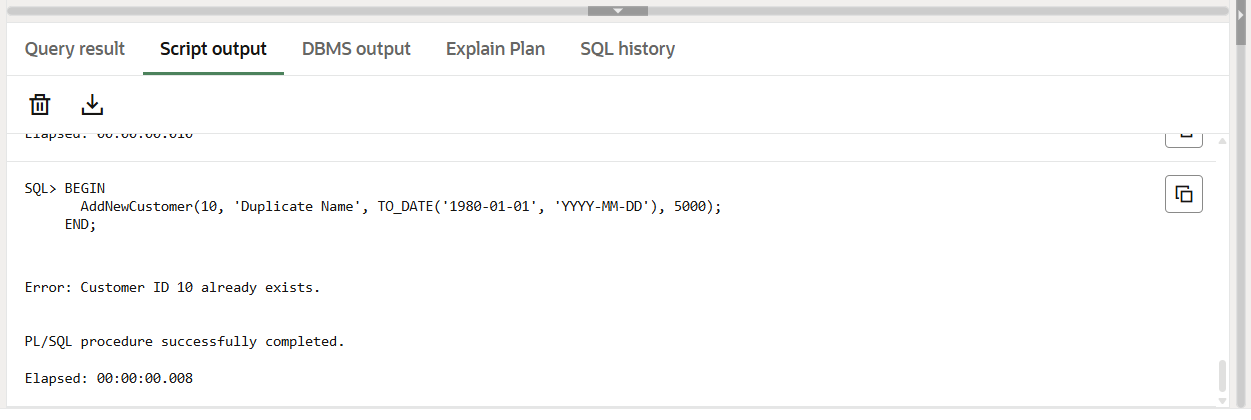
BEGIN

AddNewCustomer(10, 'Duplicate Name', TO\_DATE('1980-01-01', 'YYYY-MM-DD'), 5000);

END;

/

**OUTPUT:**



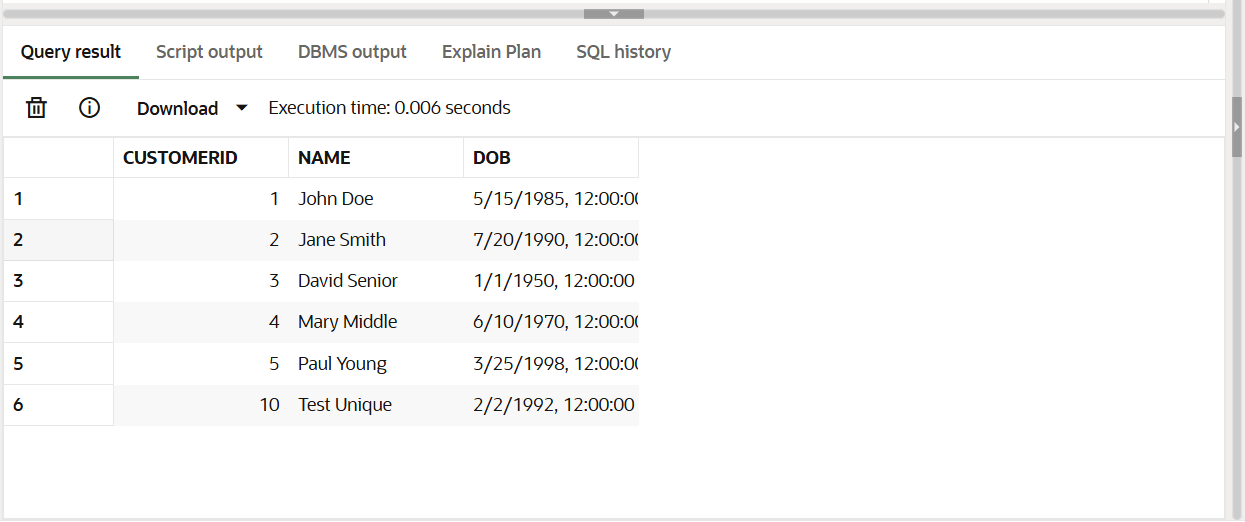
**Exercise 4: Functions**

**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.

**CODE:**

SELECT CustomerID, Name, DOB FROM Customers;

****

**PL SQL CODE:**

CREATE OR REPLACE FUNCTION CalculateAge(dob DATE)

RETURN NUMBER

IS

age NUMBER;

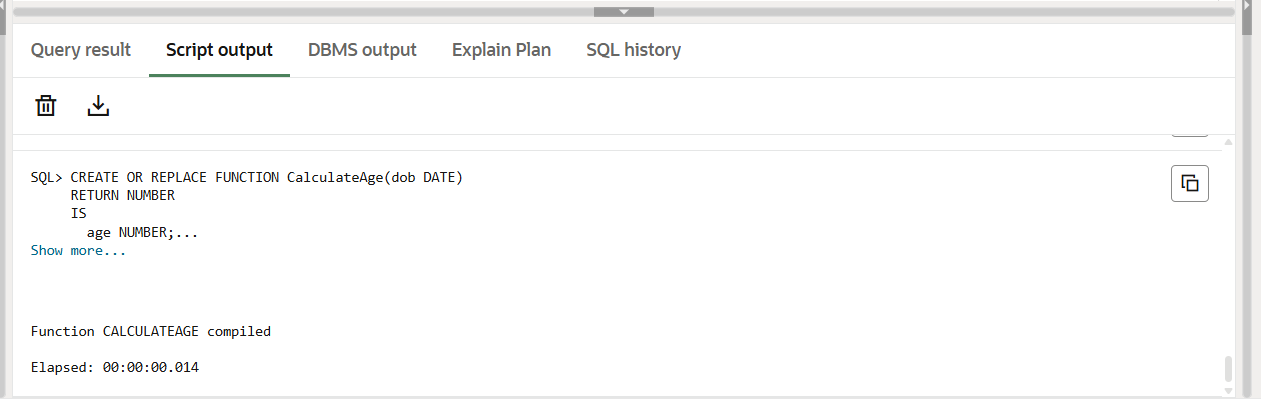
BEGIN

age := FLOOR(MONTHS\_BETWEEN(SYSDATE, dob) / 12);

RETURN age;

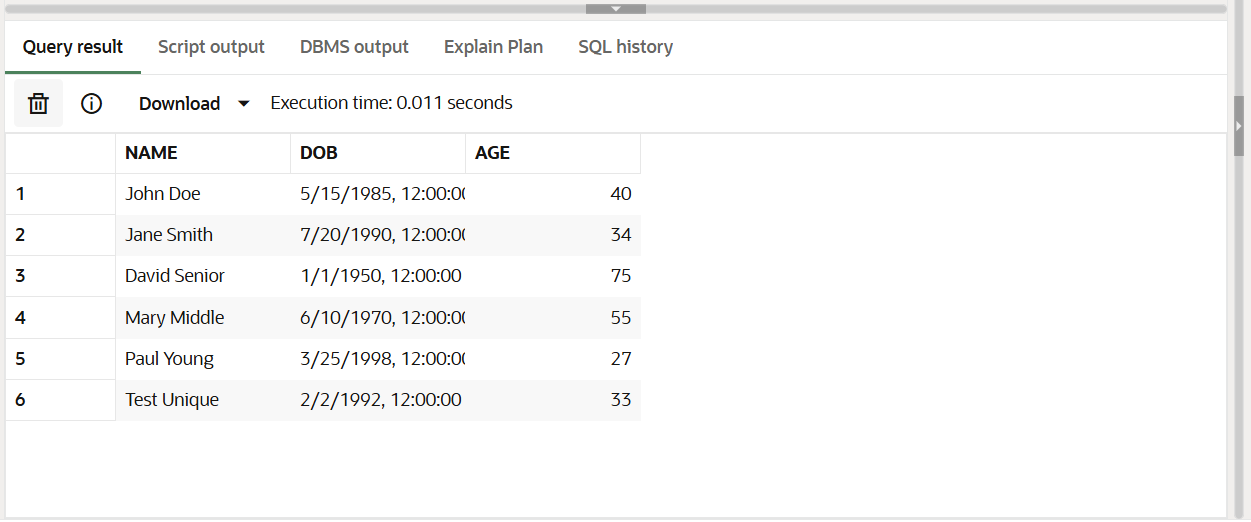
END;

/



SELECT Name, DOB, CalculateAge(DOB) AS Age

FROM Customers;



**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.

**PL SQL CODE:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

loan\_amount NUMBER,

annual\_rate NUMBER,

duration\_years NUMBER

) RETURN NUMBER

IS

r NUMBER;

n NUMBER;

emi NUMBER;

BEGIN

r := annual\_rate / 12 / 100;

n := duration\_years \* 12;

IF r = 0 THEN

emi := loan\_amount / n;

ELSE

emi := loan\_amount \* r \* POWER(1 + r, n) / (POWER(1 + r, n) - 1);

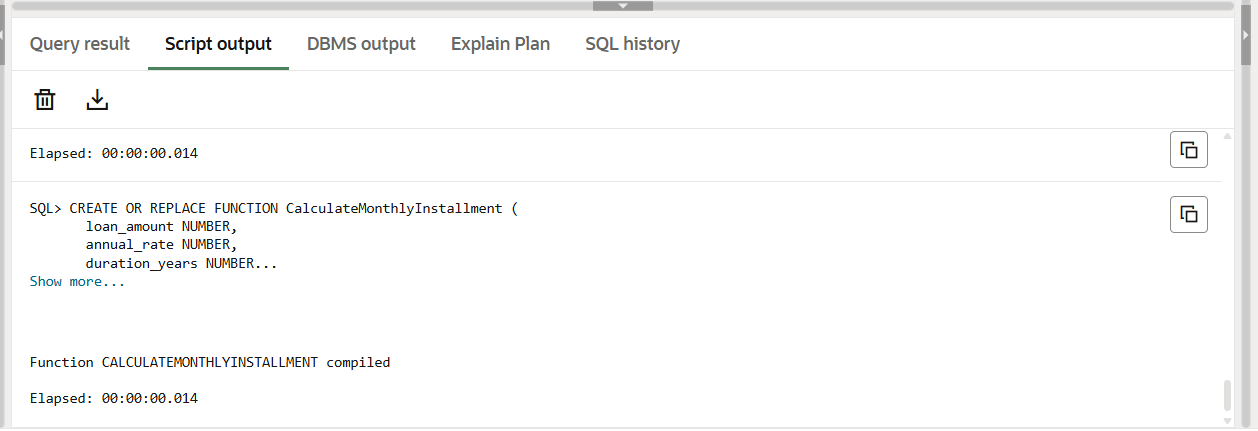
END IF;

RETURN ROUND(emi, 2);

END;

/

**OUTPUT:**



SELECT

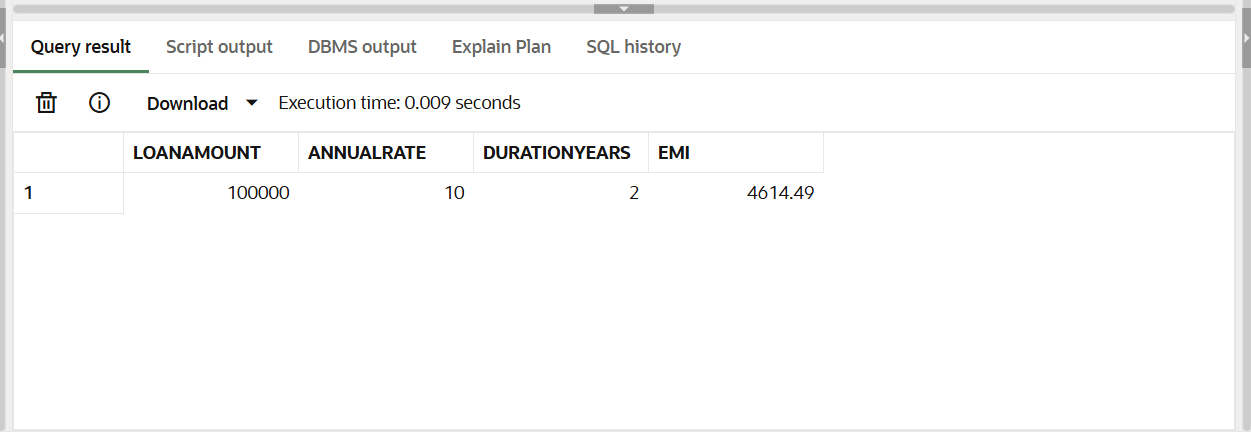
100000 AS LoanAmount,

10 AS AnnualRate,

2 AS DurationYears,

CalculateMonthlyInstallment(100000, 10, 2) AS EMI

FROM dual;

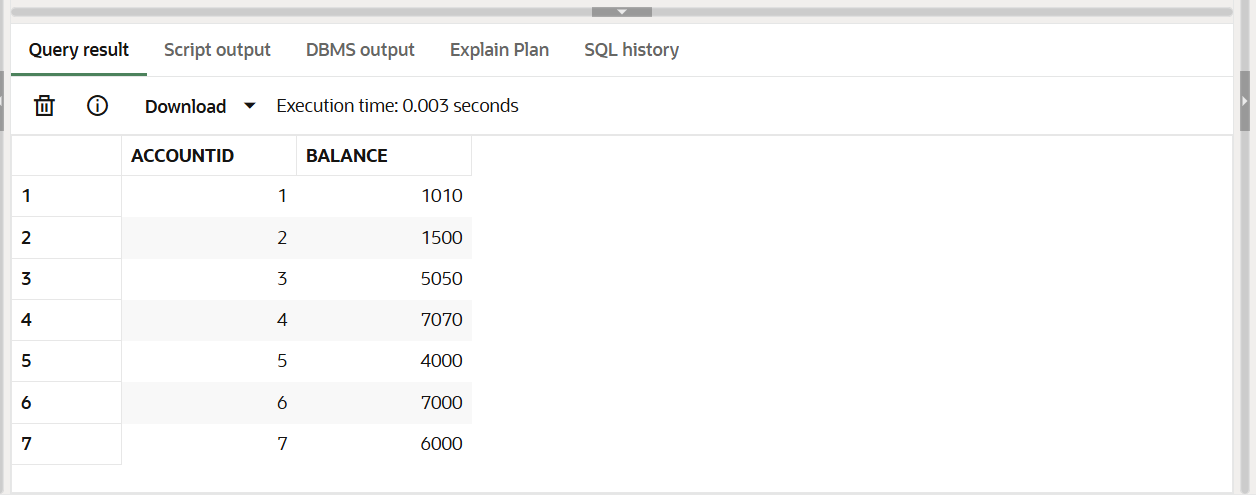


**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

**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.

**CODE:**

SELECT AccountID, Balance FROM Accounts;



**PL SQL CODE:**

CREATE OR REPLACE FUNCTION HasSufficientBalance (

accID IN NUMBER,

amount IN NUMBER

) RETURN BOOLEAN

IS

acc\_balance NUMBER;

BEGIN

SELECT Balance INTO acc\_balance FROM Accounts WHERE AccountID = accID;

IF acc\_balance >= amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

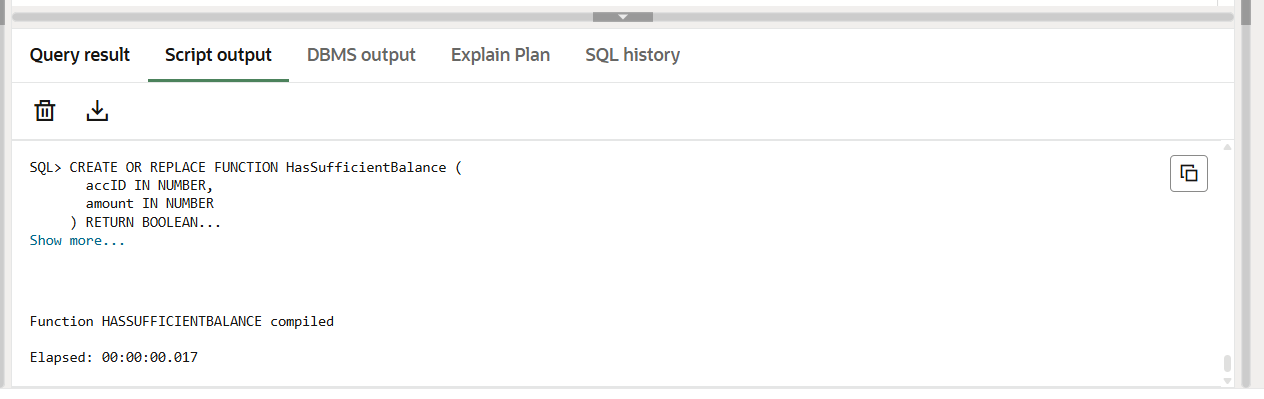
WHEN NO\_DATA\_FOUND THEN

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

RETURN FALSE;

END;

/



DECLARE

result BOOLEAN;

BEGIN

result := HasSufficientBalance(6, 2000);

IF result THEN

DBMS\_OUTPUT.PUT\_LINE('Sufficient balance.');

ELSE

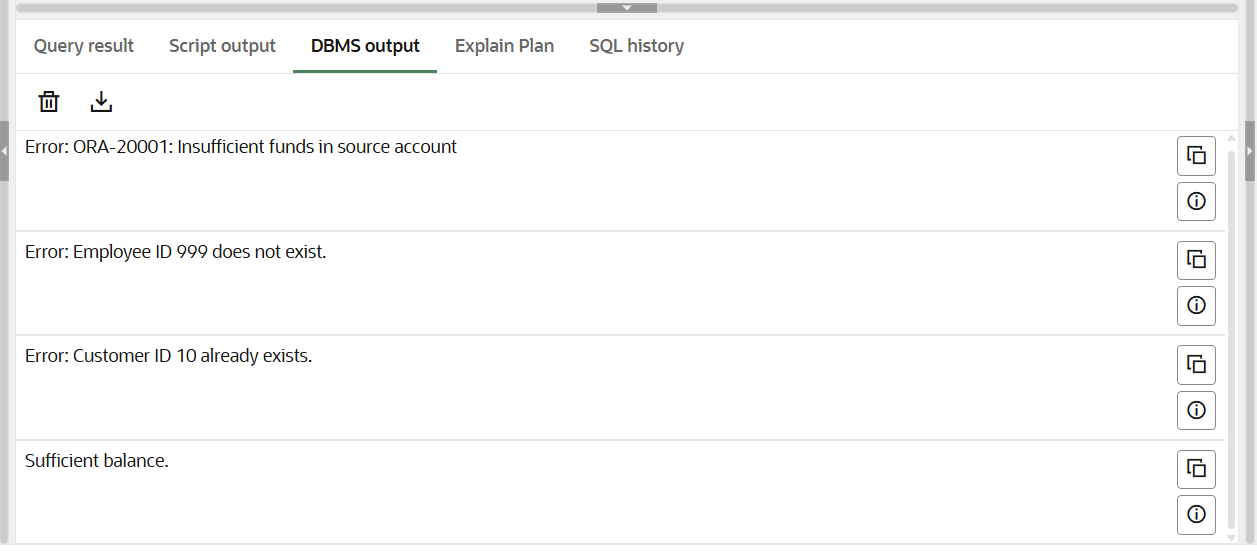
DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

END IF;

END;

/

**OUTPUT:**



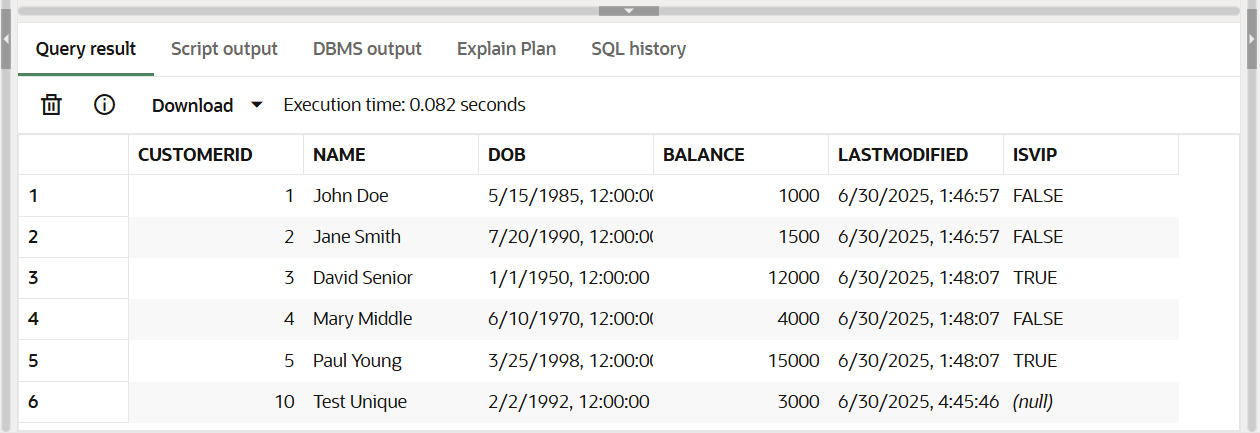
**Exercise 5: Triggers**

**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.

**CODE:**

SELECT \* FROM Customers;

****

**PL SQL CODE:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

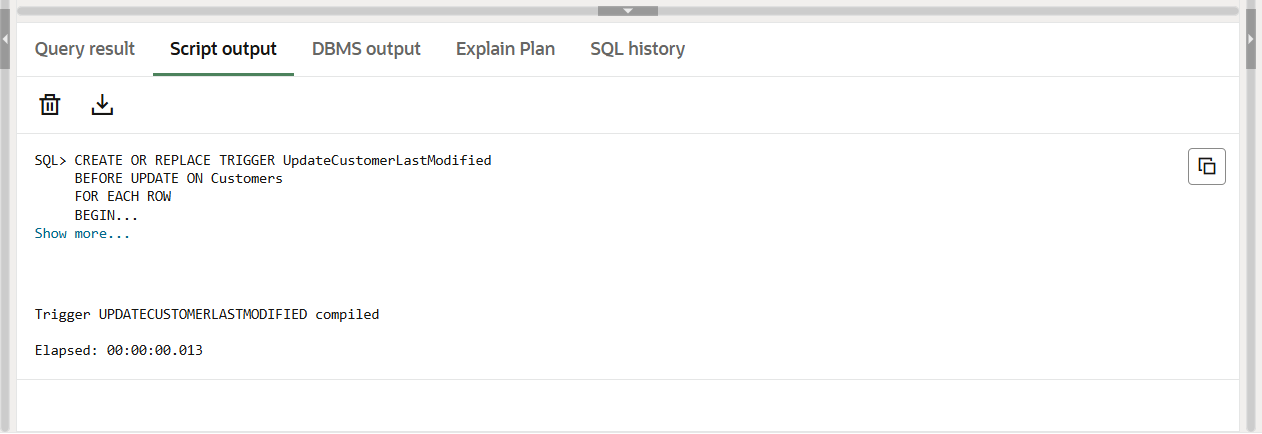
BEGIN

:NEW.LastModified := SYSDATE;

END;

/

**OUTPUT:**



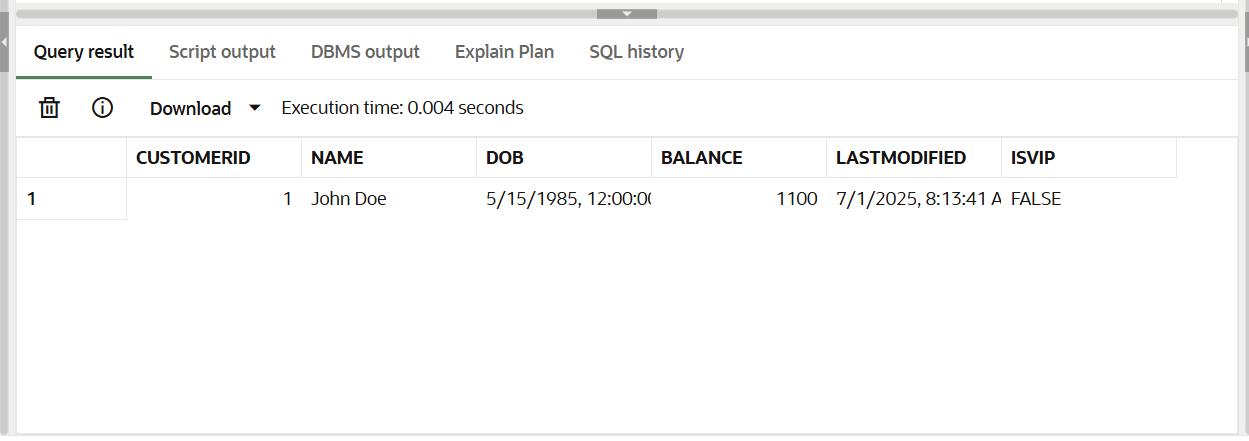
UPDATE Customers

SET Balance = Balance + 100

WHERE CustomerID = 1;

COMMIT;

SELECT \* FROM Customers WHERE CustomerID = 1;

****

**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.

**CODE:**

CREATE TABLE AuditLog (

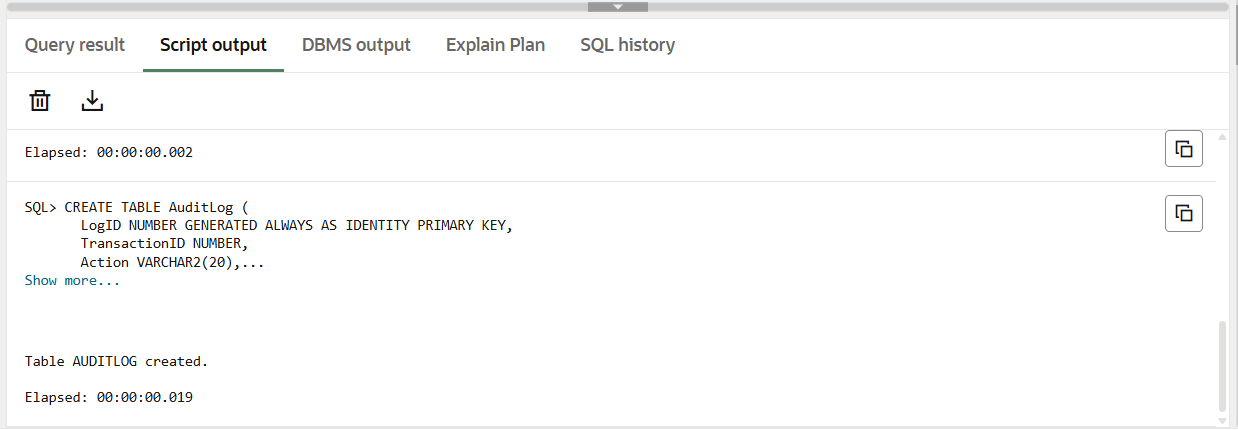
LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

Action VARCHAR2(20),

LogDate DATE

);



**PL SQL CODE:**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

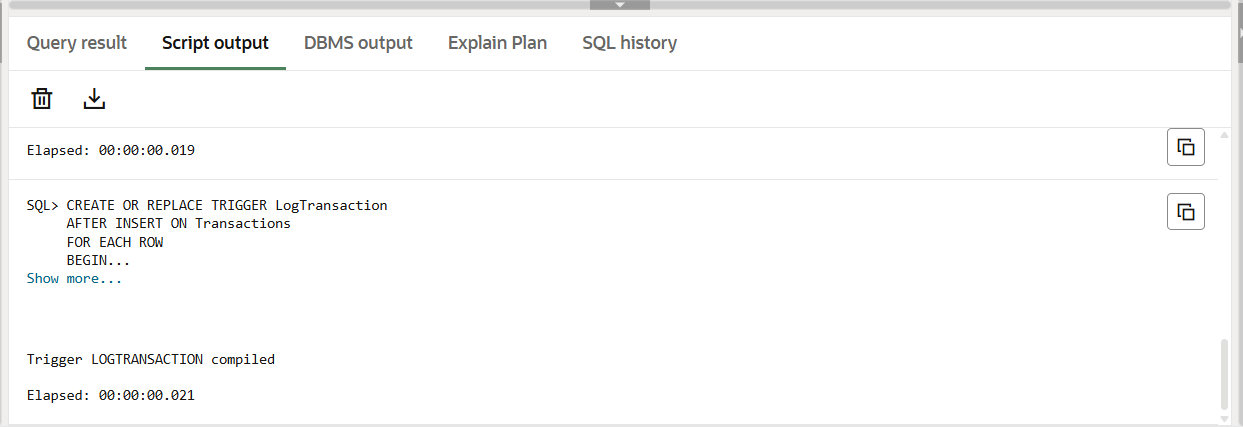
INSERT INTO AuditLog (TransactionID, Action, LogDate)

VALUES (:NEW.TransactionID, 'INSERT', SYSDATE);

END;

/

**OUTPUT:**

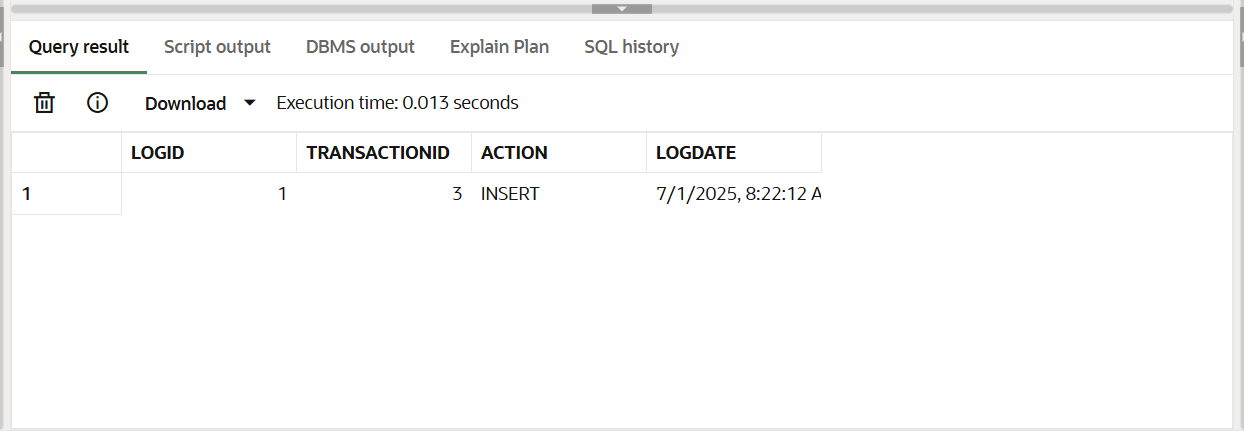


INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount,TransactionType)

VALUES (3, 6, SYSDATE, 1500, 'Deposit');

COMMIT;

SELECT \* FROM AuditLog;



**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.

**PL SQL CODE:**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

acc\_balance NUMBER;

BEGIN

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

RAISE\_APPLICATION\_ERROR(-20001, 'Deposit amount must be greater than 0');

END IF;

IF :NEW.TransactionType = 'Withdrawal' THEN

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

IF :NEW.Amount > acc\_balance THEN

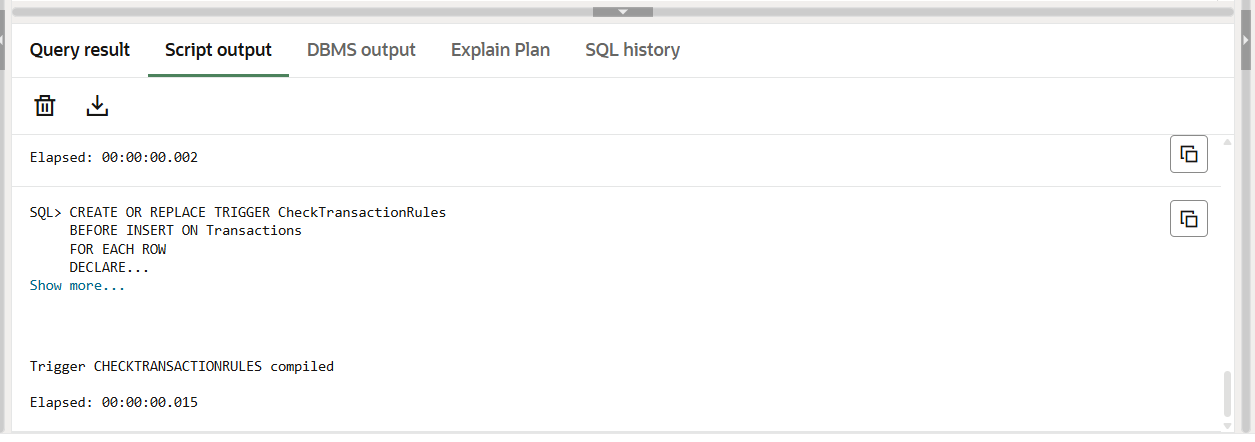
RAISE\_APPLICATION\_ERROR(-20002, 'Withdrawal exceeds account balance');

END IF;

END IF;

END;

/



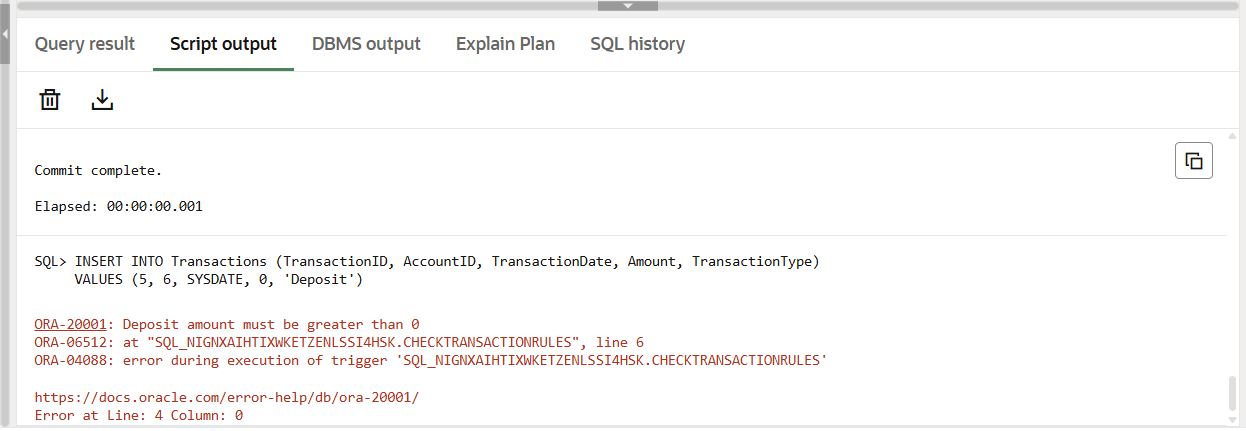
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (4, 6, SYSDATE, 500, 'Deposit');

COMMIT;

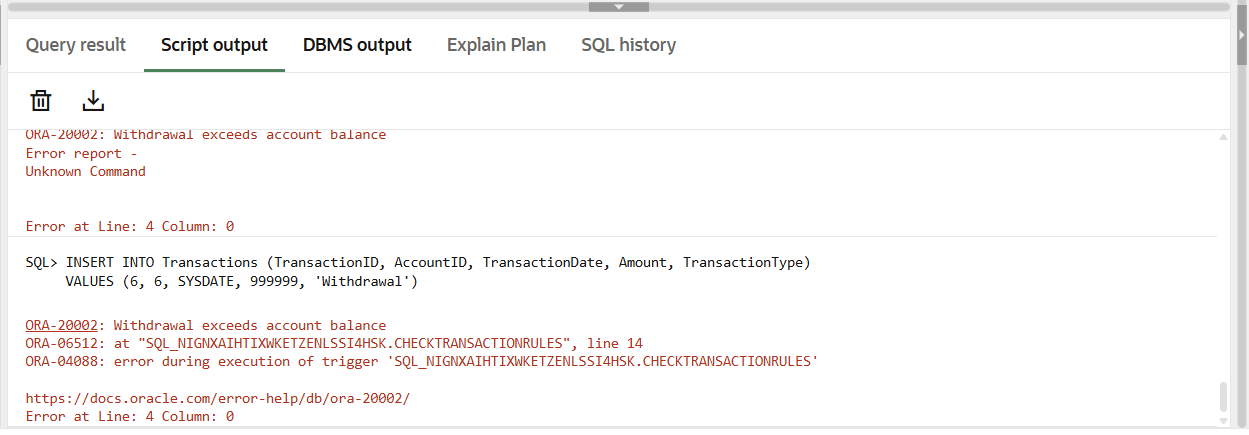
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (5, 6, SYSDATE, 0, 'Deposit');

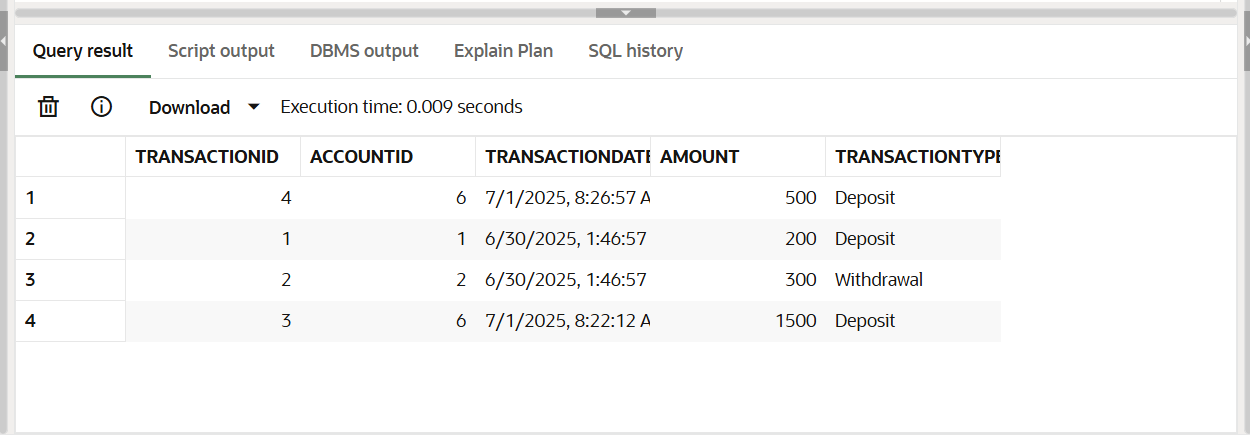


INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (6, 6, SYSDATE, 999999, 'Withdrawal');



SELECT \* FROM Transactions;



**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.

**CODE:**

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

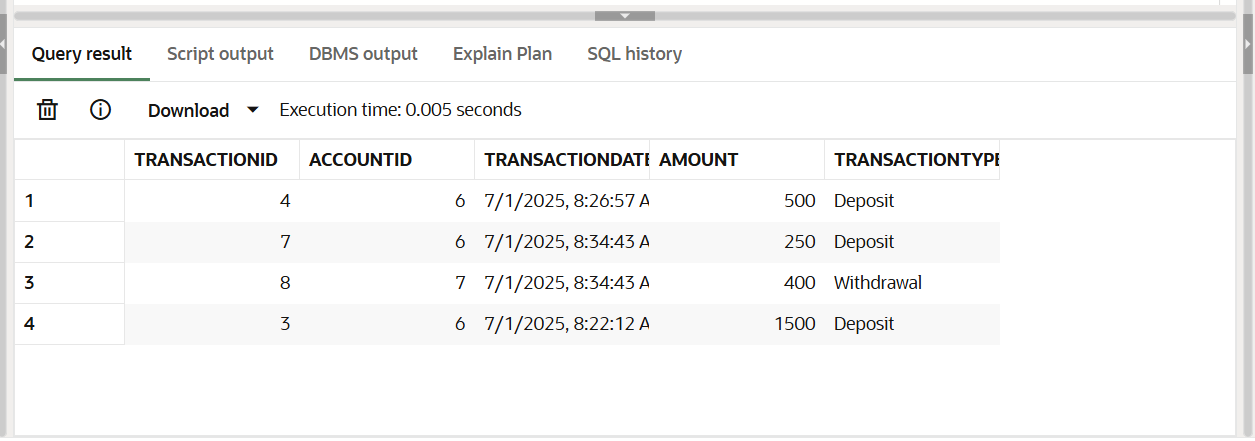
VALUES (7, 6, SYSDATE, 250, 'Deposit');

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

VALUES (8, 7, SYSDATE, 400, 'Withdrawal');

COMMIT;

SELECT \* FROM Transactions WHERE TO\_CHAR(TransactionDate, 'MM-YYYY') = TO\_CHAR(SYSDATE, 'MM-YYYY');



**PL SQL CODE:**

DECLARE

CURSOR txn\_cursor IS

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

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

JOIN Customers c ON a.CustomerID = c.CustomerID

WHERE TO\_CHAR(t.TransactionDate, 'MM-YYYY') = TO\_CHAR(SYSDATE, 'MM-YYYY');

v\_row txn\_cursor%ROWTYPE;

BEGIN

OPEN txn\_cursor;

LOOP

FETCH txn\_cursor INTO v\_row;

EXIT WHEN txn\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || v\_row.Name ||

', Txn ID: ' || v\_row.TransactionID ||

', Type: ' || v\_row.TransactionType ||

', Amount: ' || v\_row.Amount ||

', Date: ' || TO\_CHAR(v\_row.TransactionDate, 'DD-MON-YYYY'));

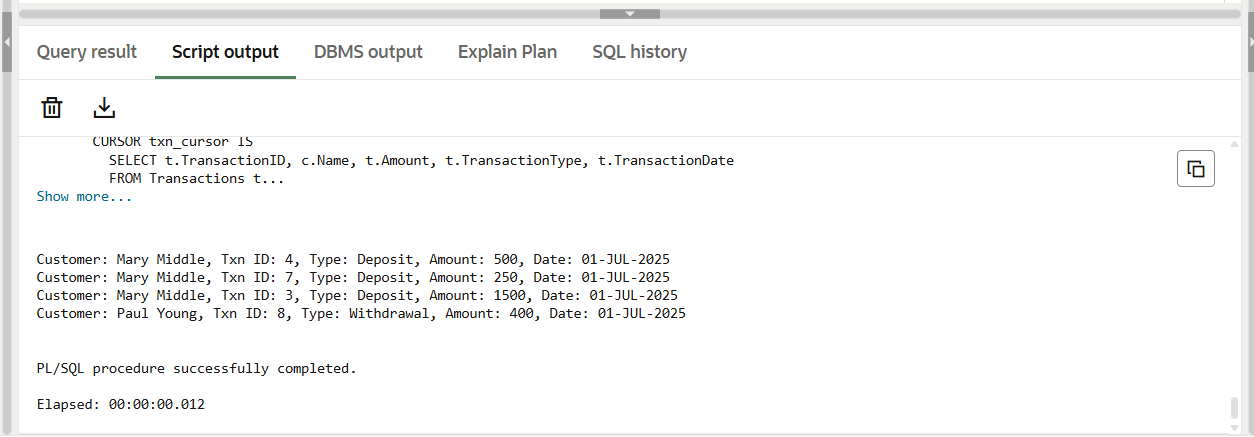
END LOOP;

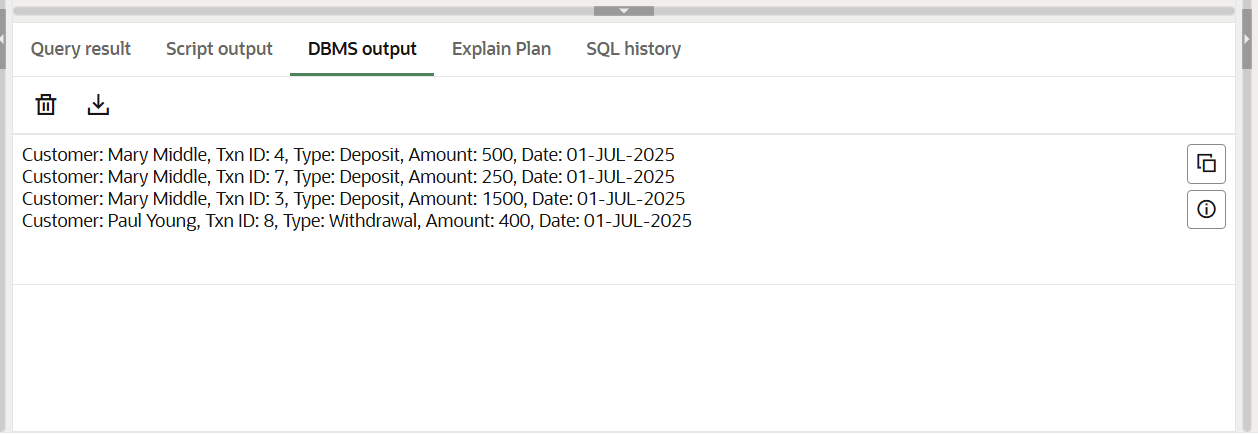
CLOSE txn\_cursor;

END;

/

**OUTPUT:**

****

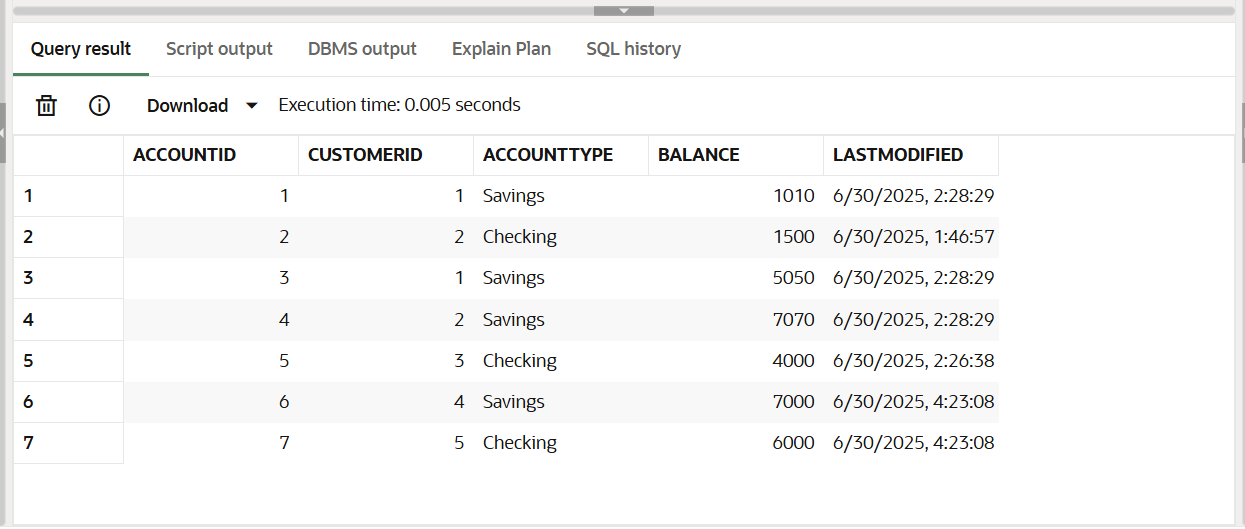


**Scenario 2:** Apply annual fee to all accounts.

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

**CODE:**

SELECT \* FROM Accounts;



**PL SQL CODE:**

DECLARE

CURSOR acc\_cursor IS

SELECT AccountID, Balance FROM Accounts;

v\_acc acc\_cursor%ROWTYPE;

BEGIN

OPEN acc\_cursor;

LOOP

FETCH acc\_cursor INTO v\_acc;

EXIT WHEN acc\_cursor%NOTFOUND;

UPDATE Accounts

SET Balance = Balance - 100,

LastModified = SYSDATE

WHERE AccountID = v\_acc.AccountID;

END LOOP;

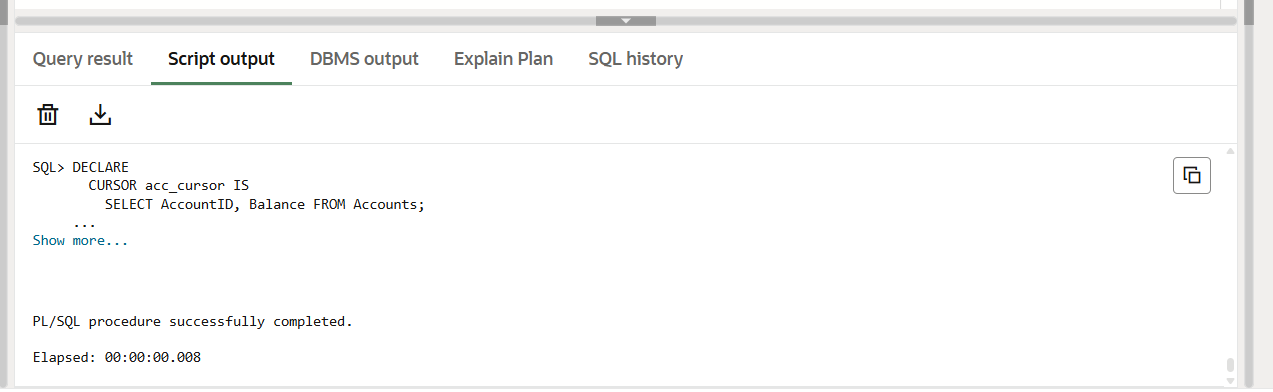
CLOSE acc\_cursor;

COMMIT;

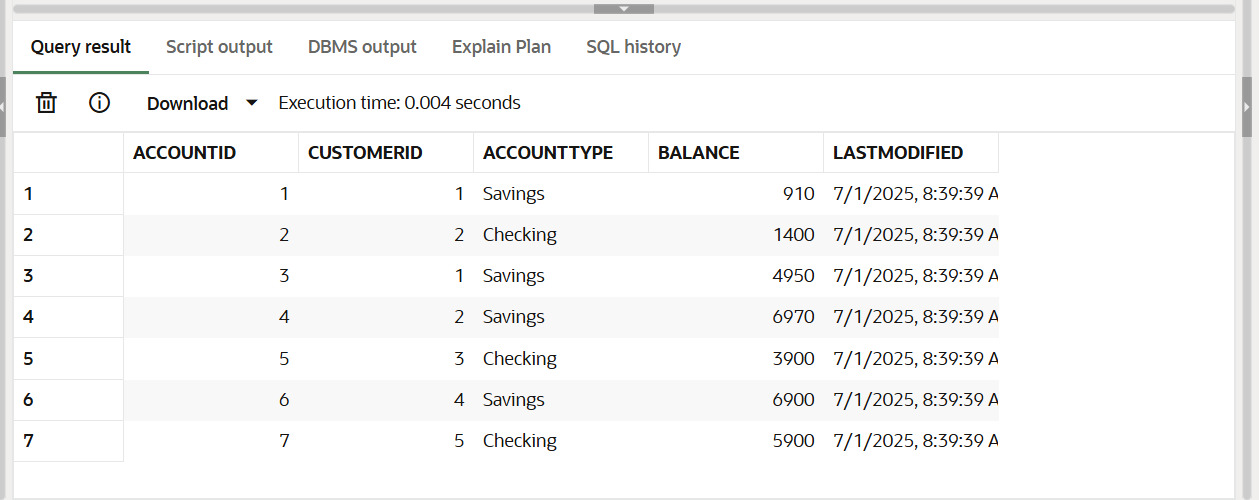
END;

/

**OUTPUT:**

****

SELECT \* FROM Accounts;

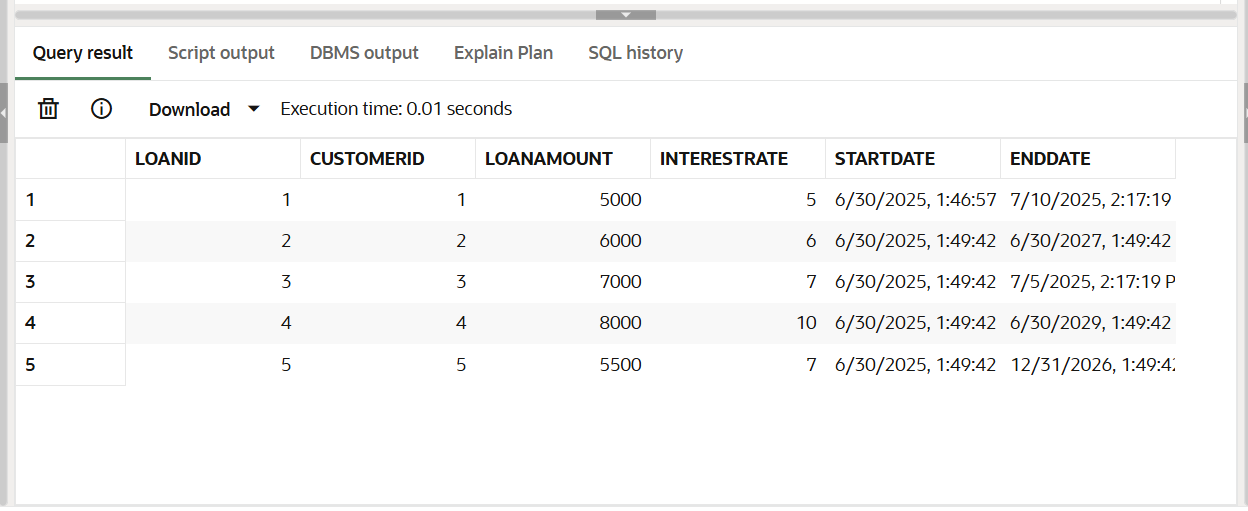


**Scenario 3:** Update the interest rate for all loans based on a new policy.

**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.

**CODE:**

SELECT \* FROM Loans;



**PL SQL CODE:**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate FROM Loans;

v\_loan loan\_cursor%ROWTYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loan;

EXIT WHEN loan\_cursor%NOTFOUND;

IF v\_loan.InterestRate < 10 THEN

UPDATE Loans

SET InterestRate = InterestRate + 1

WHERE LoanID = v\_loan.LoanID;

ELSE

UPDATE Loans

SET InterestRate = InterestRate - 0.5

WHERE LoanID = v\_loan.LoanID;

END IF;

END LOOP;

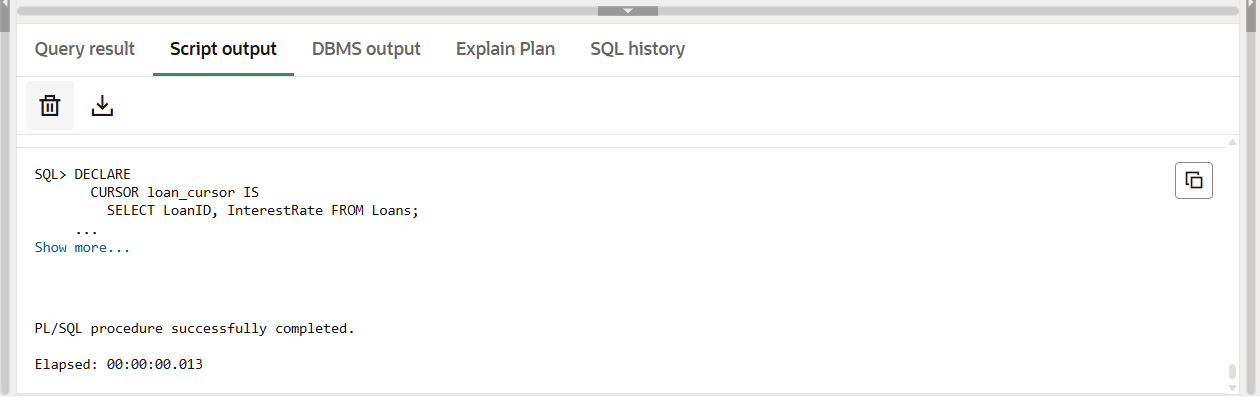
CLOSE loan\_cursor;

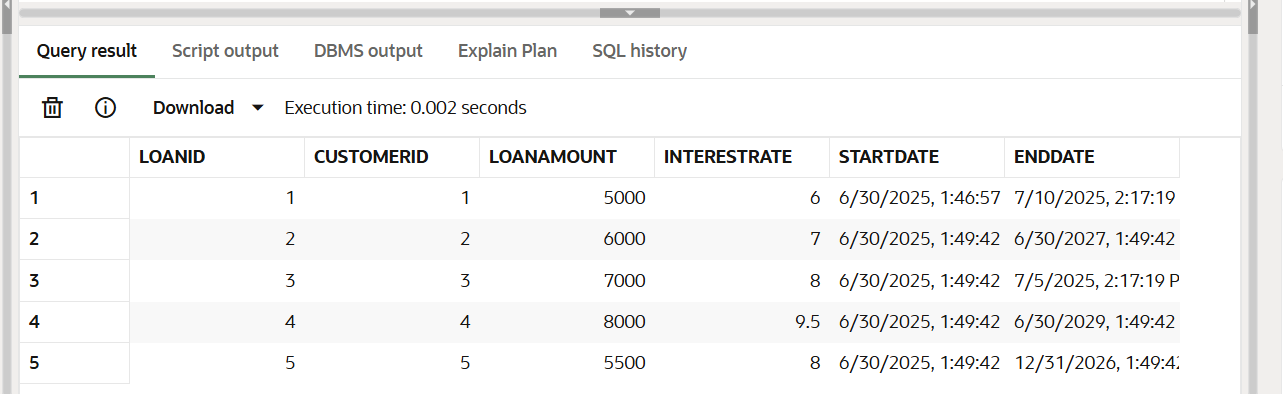
COMMIT;

END;

/

**OUTPUT:**



SELECT \* FROM Loans;

**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.

**PL SQL CODE:**

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(

p\_CustomerID NUMBER,

p\_Name VARCHAR2,

p\_DOB DATE,

p\_Balance NUMBER

);

PROCEDURE UpdateCustomerDetails(

p\_CustomerID NUMBER,

p\_Name VARCHAR2,

p\_Balance NUMBER

);

FUNCTION GetCustomerBalance(

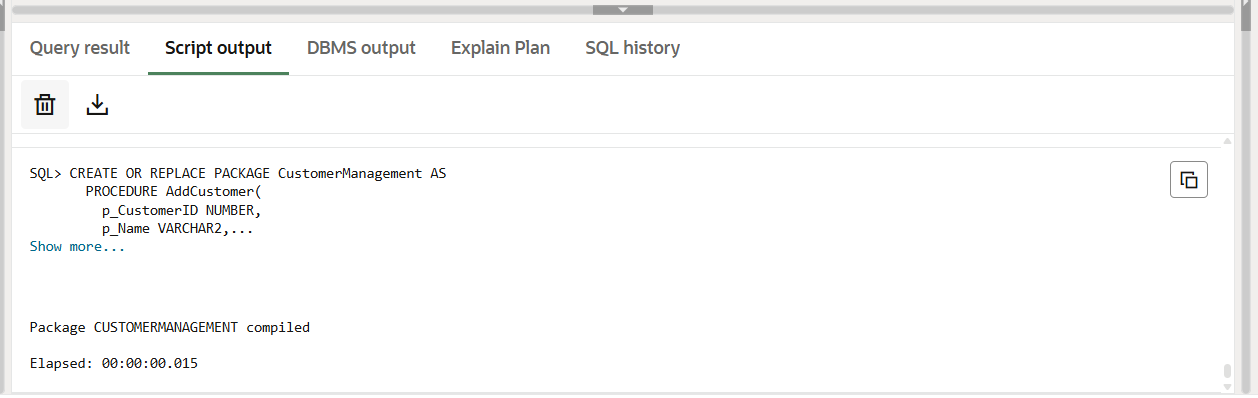
p\_CustomerID NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

**OUTPUT:**



CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER) IS

BEGIN

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

VALUES (p\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END;

PROCEDURE UpdateCustomerDetails(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_Balance NUMBER) IS

BEGIN

UPDATE Customers

SET Name = p\_Name,

Balance = p\_Balance,

LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

END;

FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

bal NUMBER;

BEGIN

SELECT Balance INTO bal FROM Customers WHERE CustomerID = p\_CustomerID;

RETURN bal;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

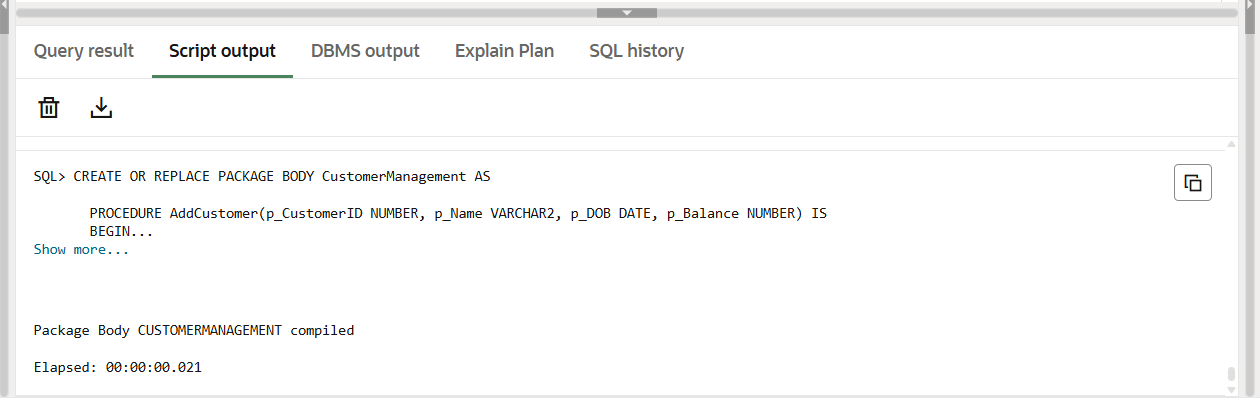
RETURN NULL;

END;

END CustomerManagement;

/

**OUTPUT:**



BEGIN

CustomerManagement.AddCustomer(20, 'Package Test', TO\_DATE('1990-01-01','YYYY-MM-DD'), 5000);

END;

/

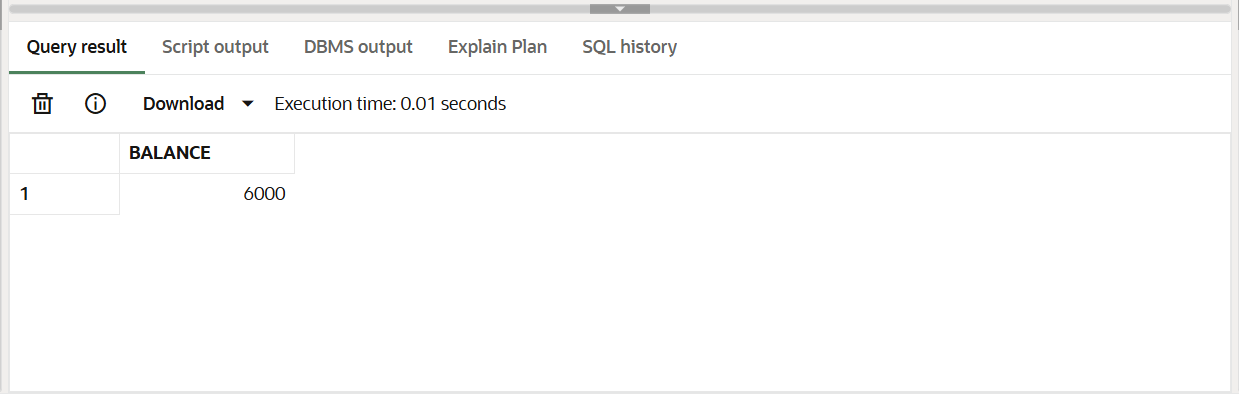
BEGIN

CustomerManagement.UpdateCustomerDetails(20, 'Updated Test', 6000);

END;

/

SELECT CustomerManagement.GetCustomerBalance(20) AS Balance FROM dual;



**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.

**PL SQL CODE:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(

p\_EmployeeID NUMBER,

p\_Name VARCHAR2,

p\_Position VARCHAR2,

p\_Salary NUMBER,

p\_Department VARCHAR2,

p\_HireDate DATE

);

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID NUMBER,

p\_Name VARCHAR2,

p\_Salary NUMBER

);

FUNCTION CalculateAnnualSalary(

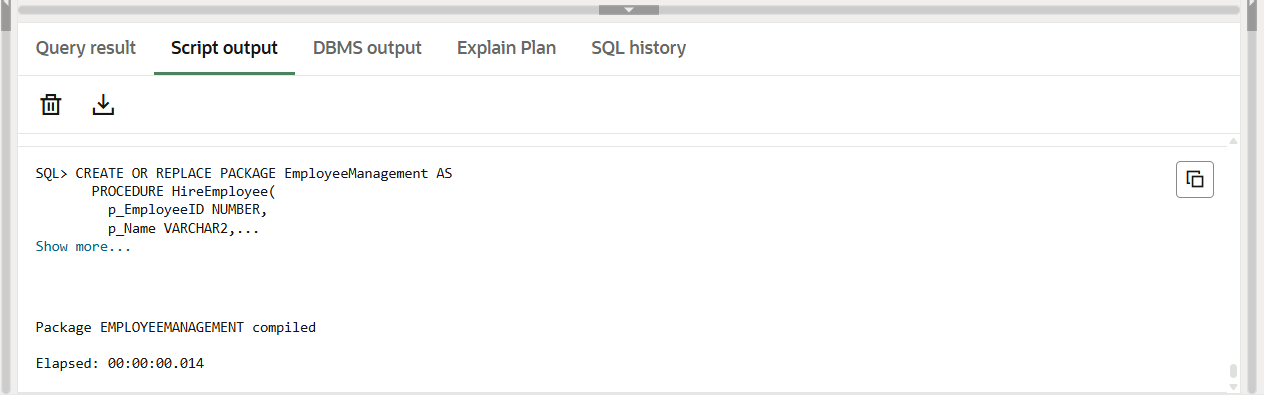
p\_EmployeeID NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

**OUTPUT:**



CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee (

p\_EmployeeID NUMBER,

p\_Name VARCHAR2,

p\_Position VARCHAR2,

p\_Salary NUMBER,

p\_Department VARCHAR2,

p\_HireDate DATE

) IS

BEGIN

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

VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Salary, p\_Department, p\_HireDate);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END;

PROCEDURE UpdateEmployeeDetails (

p\_EmployeeID NUMBER,

p\_Name VARCHAR2,

p\_Salary NUMBER

) IS

BEGIN

UPDATE Employees

SET Name = p\_Name,

Salary = p\_Salary

WHERE EmployeeID = p\_EmployeeID;

END;

FUNCTION CalculateAnnualSalary (p\_EmployeeID NUMBER) RETURN NUMBER IS

monthly\_salary NUMBER;

BEGIN

SELECT Salary INTO monthly\_salary FROM Employees WHERE EmployeeID = p\_EmployeeID;

RETURN monthly\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

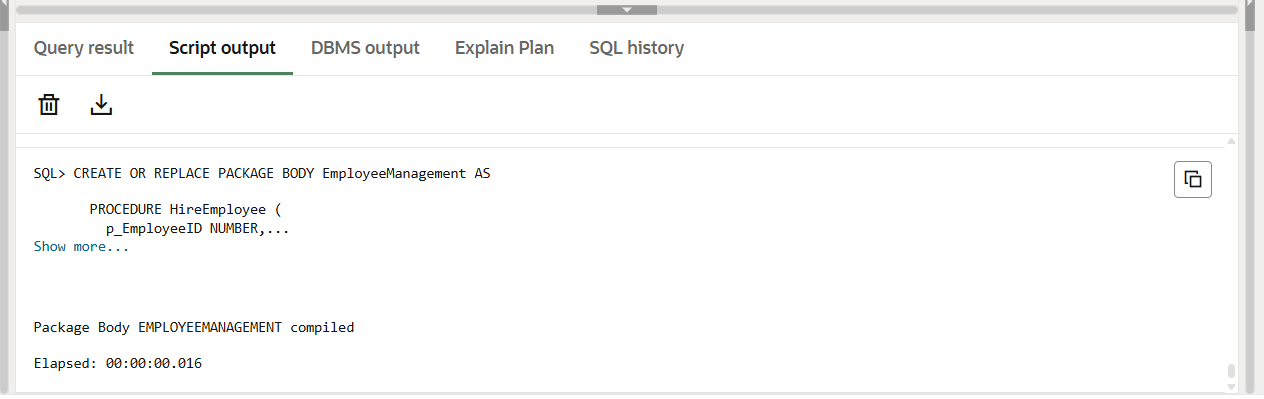
RETURN NULL;

END;

END EmployeeManagement;

/

**OUTPUT:**



BEGIN

EmployeeManagement.HireEmployee(10, 'Test Emp', 'Analyst', 5000, 'QA', SYSDATE);

END;

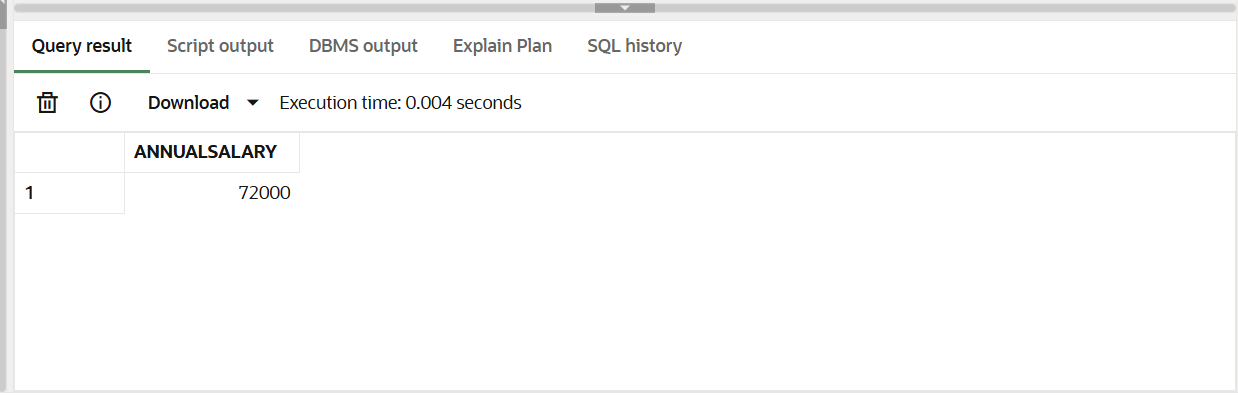
/

BEGIN

EmployeeManagement.UpdateEmployeeDetails(10, 'Updated Emp', 6000);

END;

/

SELECT EmployeeManagement.CalculateAnnualSalary(10) AS AnnualSalary FROM dual;

**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.

**PL SQL CODE:**

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(

p\_AccountID NUMBER,

p\_CustomerID NUMBER,

p\_AccountType VARCHAR2,

p\_Balance NUMBER

);

PROCEDURE CloseAccount(

p\_AccountID NUMBER

);

FUNCTION GetTotalBalance(

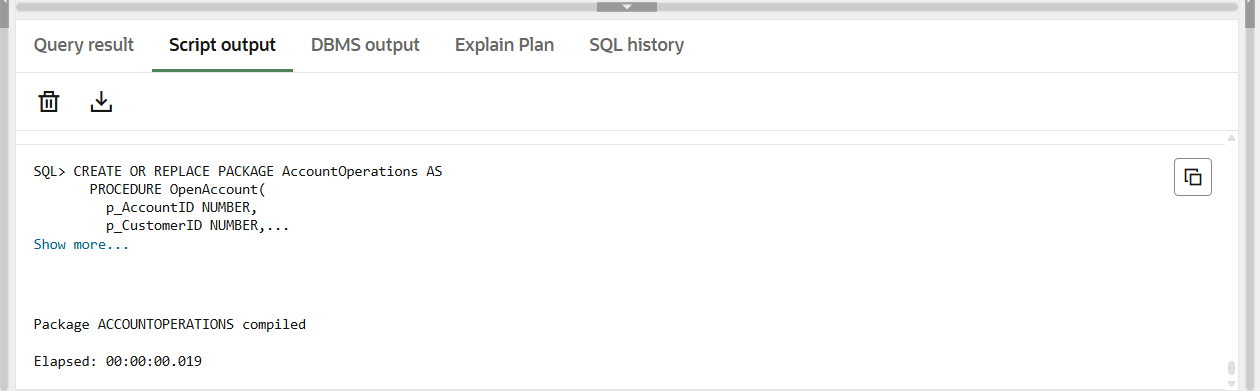
p\_CustomerID NUMBER

) RETURN NUMBER;

END AccountOperations;

/

**OUTPUT:**

****

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount (

p\_AccountID NUMBER,

p\_CustomerID NUMBER,

p\_AccountType VARCHAR2,

p\_Balance NUMBER

) IS

BEGIN

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

VALUES (p\_AccountID, p\_CustomerID, p\_AccountType, p\_Balance, SYSDATE);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END;

PROCEDURE CloseAccount (

p\_AccountID NUMBER

) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_AccountID;

END;

FUNCTION GetTotalBalance (

p\_CustomerID NUMBER

) RETURN NUMBER IS

total NUMBER;

BEGIN

SELECT SUM(Balance) INTO total

FROM Accounts

WHERE CustomerID = p\_CustomerID;

RETURN NVL(total, 0);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

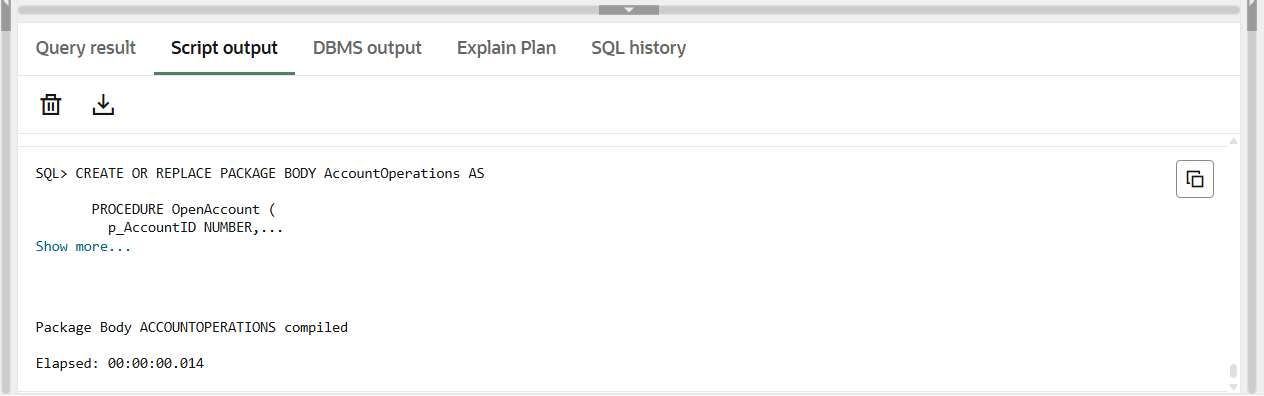
RETURN 0;

END;

END AccountOperations;

/

**OUTPUT:**



BEGIN

AccountOperations.OpenAccount(30, 1, 'Savings', 5000);

END;

/

SELECT AccountOperations.GetTotalBalance(1) AS TotalBalance FROM dual;

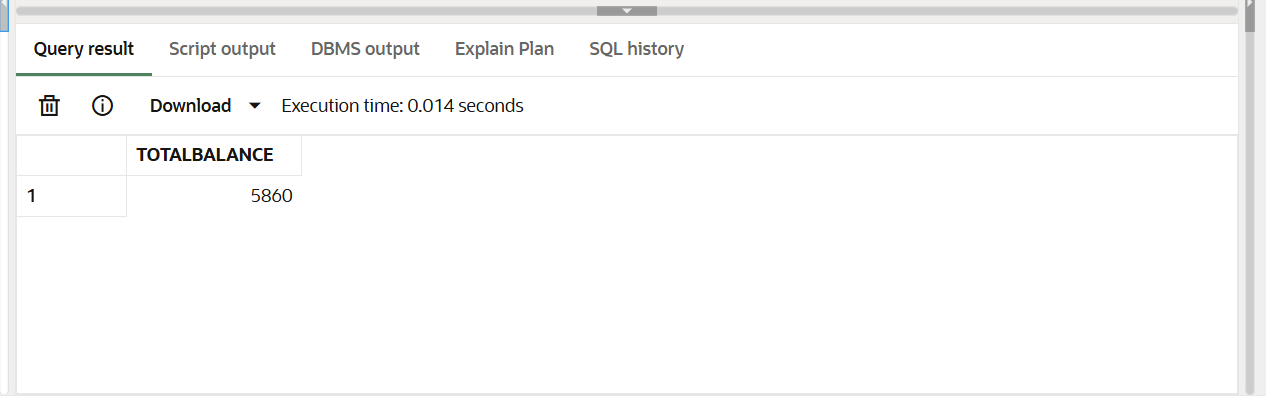
BEGIN

AccountOperations.CloseAccount(30);

END;

/

**OUTPUT:**

****