**(6428317 Mukesh P)**

**WEEK 2 : PL\_SQL PRACTICE**

**MANDATORY 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.

**PROGRAM:**

DECLARE

    CURSOR cur\_customers IS

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

        TRUNC(MONTHS\_BETWEEN(SYSDATE, c.DOB) / 12) AS age

        FROM Customers c

        JOIN Loans l ON c.CustomerID = l.CustomerID;

BEGIN

    FOR cust IN cur\_customers LOOP

        IF cust.age > 60 THEN

            UPDATE Loans

            SET InterestRate = InterestRate - 1

            WHERE LoanID = cust.LoanID;

            DBMS\_OUTPUT.PUT\_LINE('Discount applied to Customer ID: ' || cust.CustomerID ||

                                 ' | New Interest Rate: ' || (cust.InterestRate - 1));

        END IF;

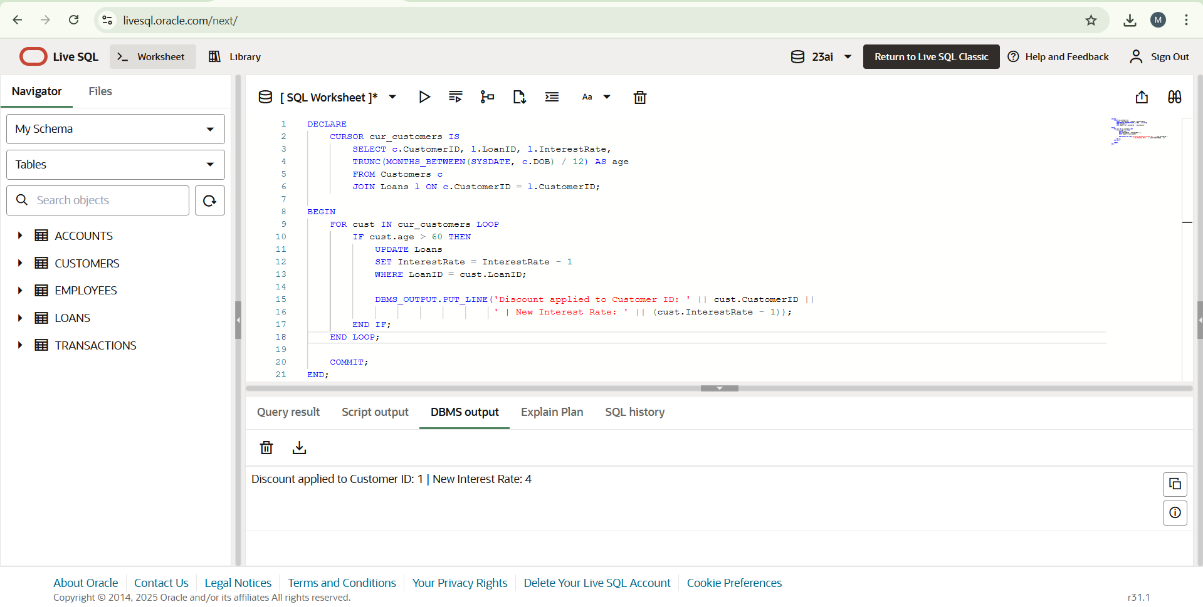
    END LOOP;

    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.

**PROGRAM:**

BEGIN

    FOR cust IN (SELECT CustomerID, Balance FROM Customers) LOOP

        IF cust.Balance > 10000 THEN

            UPDATE Customers

            SET IsVIP = 'Y'

            WHERE CustomerID = cust.CustomerID;

            DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || cust.CustomerID || ' promoted to VIP.');

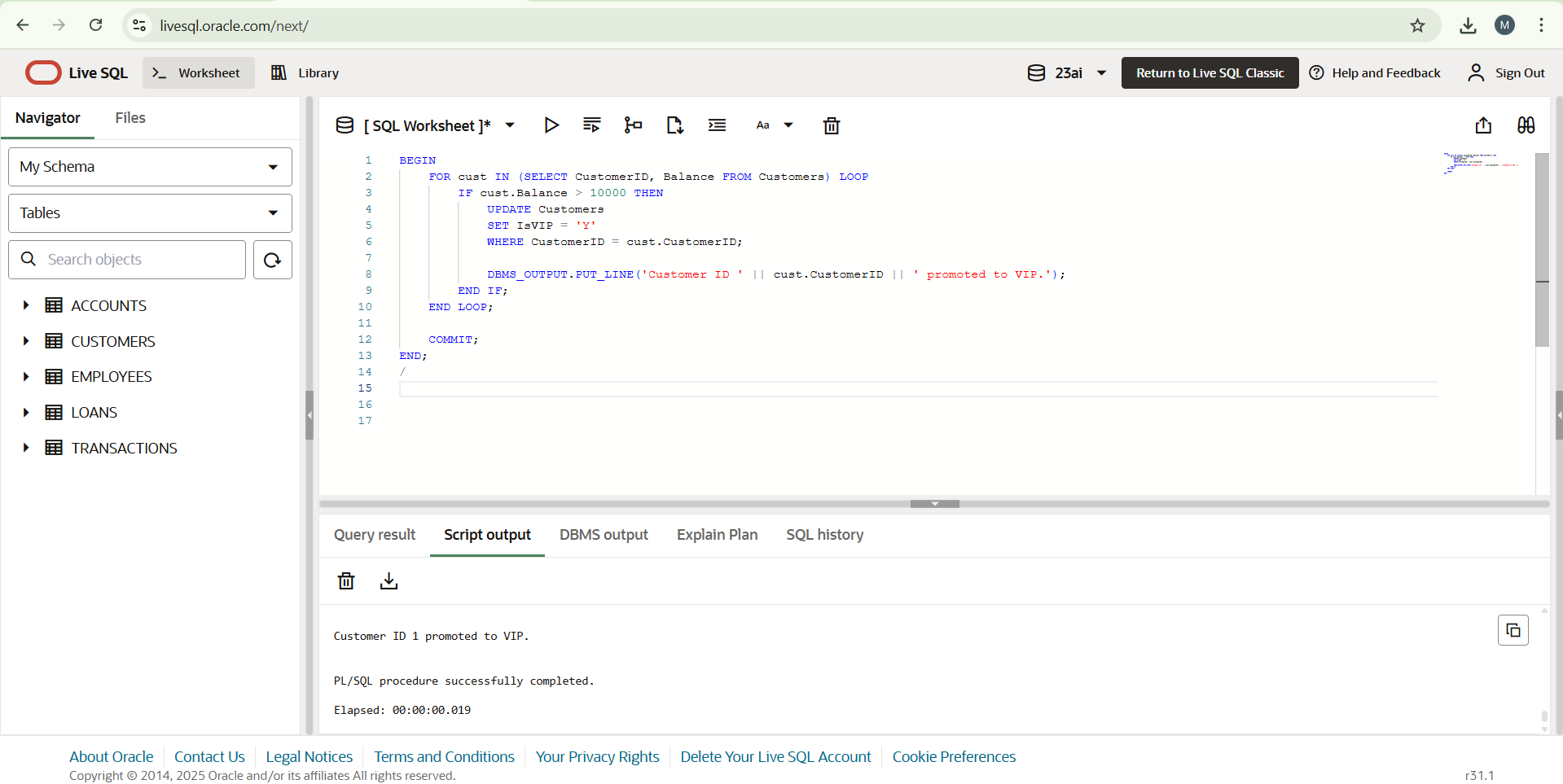
        END IF;

    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.

**PROGRAM:**

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

VALUES (3, 1, 3000, 5, SYSDATE, SYSDATE + 10);

COMMIT;

BEGIN

    FOR rec IN (

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

        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: Loan ID ' || rec.LoanID ||

            ' for Customer ' || rec.Name ||

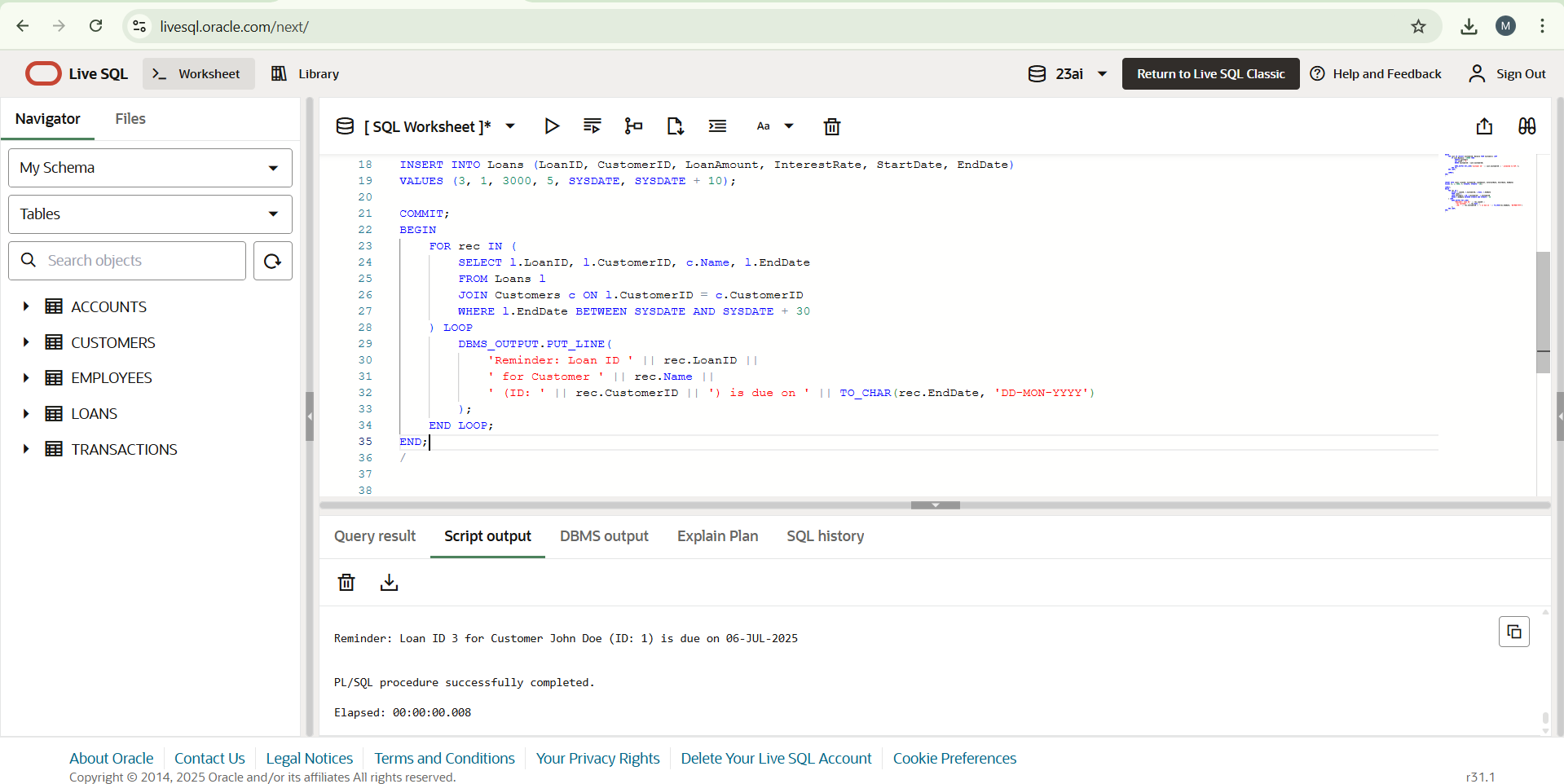
            ' (ID: ' || rec.CustomerID || ') is due on ' || TO\_CHAR(rec.EndDate, 'DD-MON-YYYY')

        );

    END LOOP;

END;

**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.

**PROGRAM:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

    UPDATE ACCOUNTS

    SET BALANCE=BALANCE+(BALANCE\*0.01),

        LASTMODIFIED=SYSDATE

    WHERE ACCOUNTTYPE='Savings';

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('interest applied to all accounts');

EXCEPTION

    WHEN OTHERS THEN

    ROLLBACK;

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

END;

/

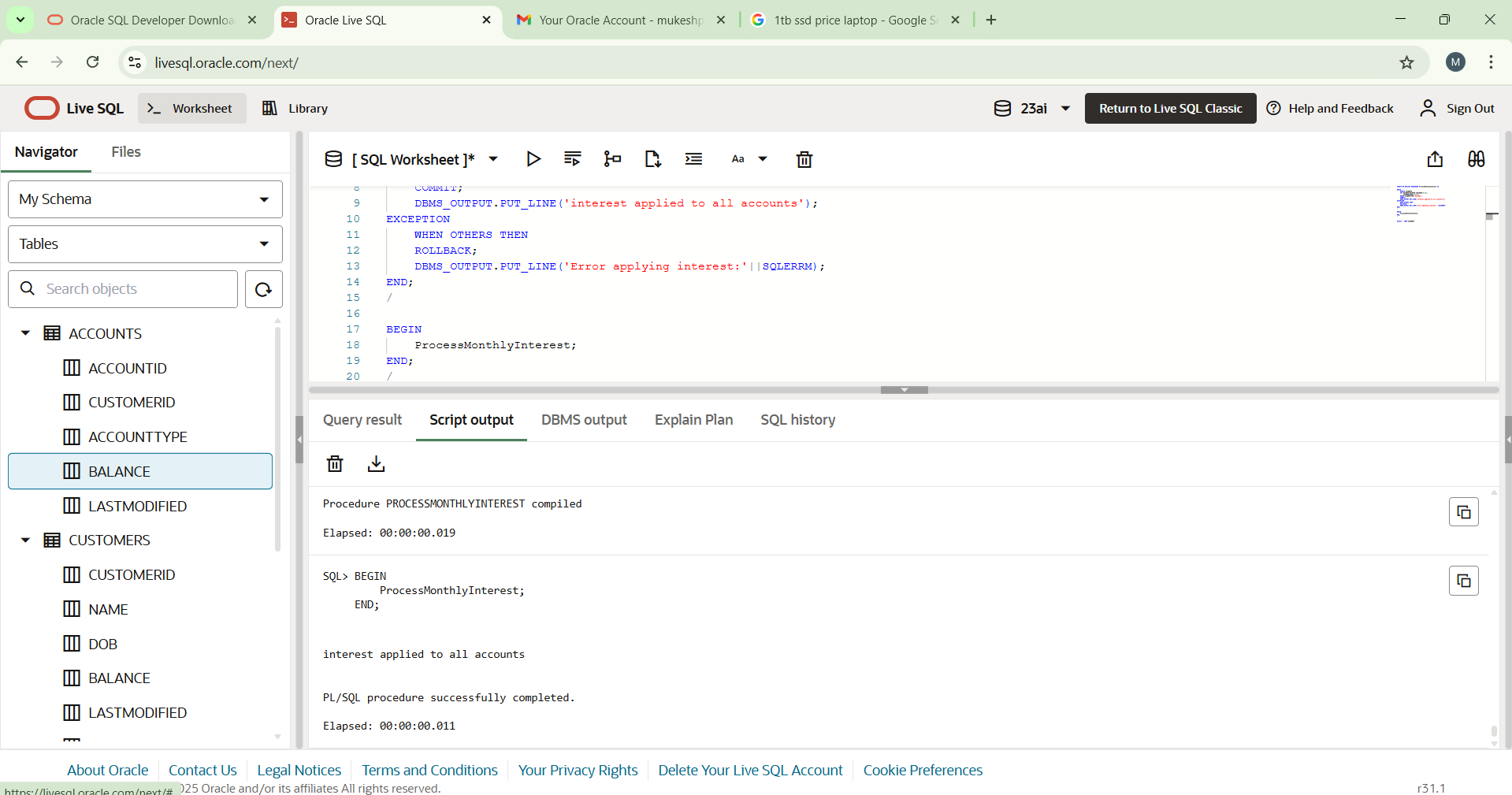
BEGIN

    ProcessMonthlyInterest;

END;

/

SELECT \* FROM ACCOUNTS

**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.

**PROGRAM:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

    p\_department IN VARCHAR2,

    p\_bonus IN NUMBER

)

IS

BEGIN

    UPDATE EMPLOYEES

    SET SALARY=SALARY+(SALARY\*p\_bonus/100)

    WHERE DEPARTMENT=p\_department;

IF SQL%ROWCOUNT=0 THEN

    DBMS\_OUTPUT.PUT\_LINE('No employee found');

ELSE

    DBMS\_OUTPUT.PUT\_LINE(SQL%ROWCOUNT || ' EMPLOYEES IN DEPARTMENT '||p\_department||' received bonus');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

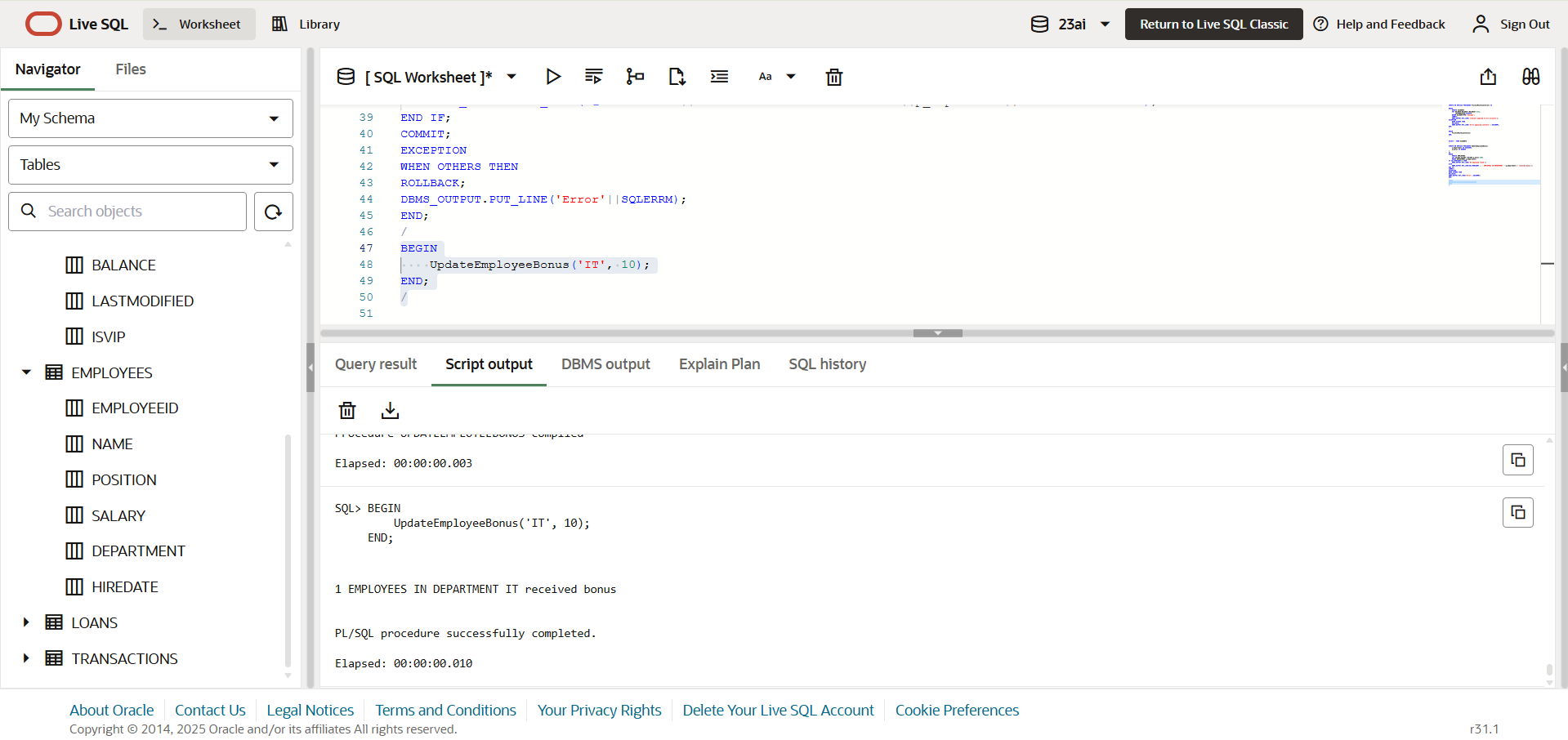
BEGIN

    UpdateEmployeeBonus('IT', 10);

END;

/

**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.

**PROGRAM :**

CREATE OR REPLACE PROCEDURE TransferFunds(

    p\_from IN NUMBER,

    p\_to IN NUMBER,

    p\_amount IN NUMBER

)

IS

    v\_balance NUMBER;

BEGIN

SELECT BALANCE INTO v\_balance

FROM ACCOUNTS

WHERE ACCOUNTID=p\_from

FOR UPDATE;

IF v\_balance<p\_amount THEN

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

END IF;

UPDATE ACCOUNTS

SET BALANCE=BALANCE+p\_amount,

    LASTMODIFIED=SYSDATE

WHERE ACCOUNTID=p\_to;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transferred');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

    TransferFunds(1, 2, 100);

END;

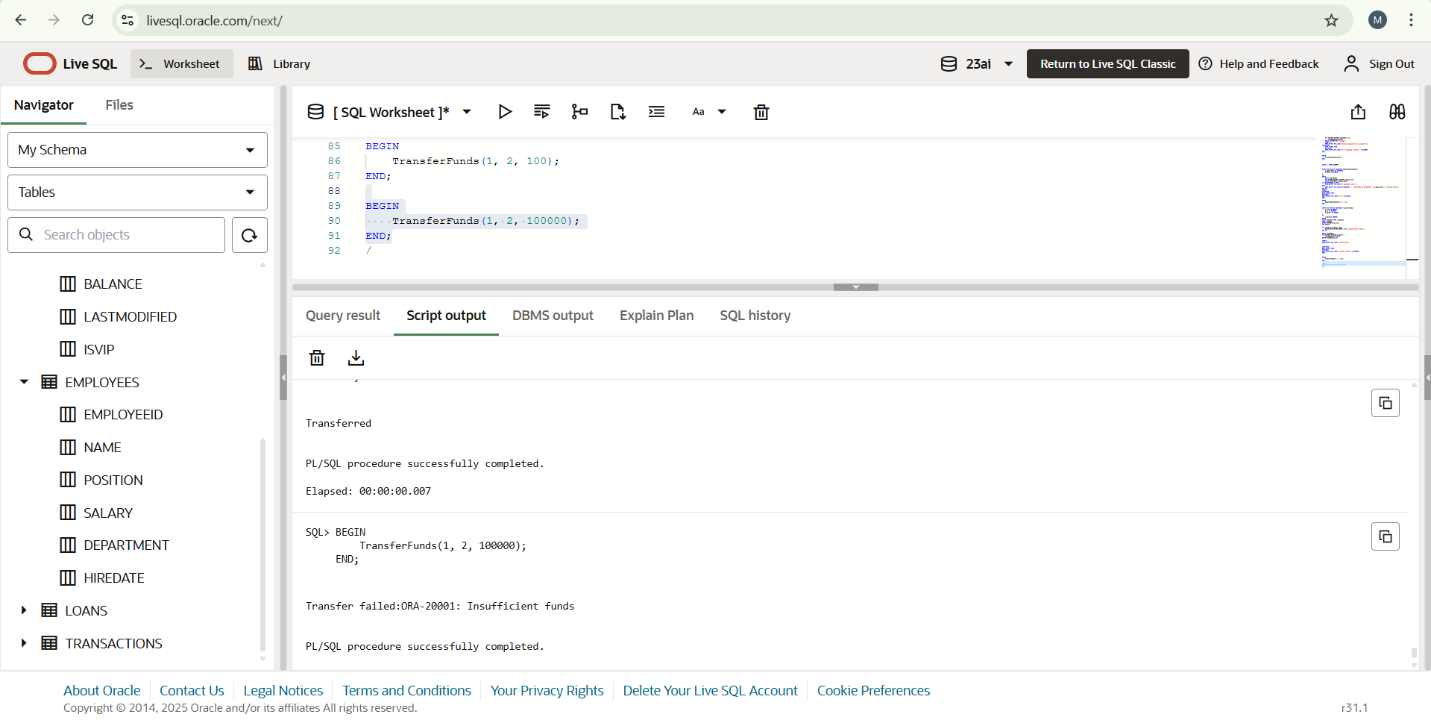
BEGIN

    TransferFunds(1, 2, 100000);

END;

/

**OUTPUT:**



**-------------------------------------------------------------------------------------------------------------**

**ADDITIONAL HANDS\_ON :**

**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.**PROGRAM :**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

    p\_from\_account\_id IN NUMBER,

    p\_to\_account\_id IN NUMBER,

    p\_amount IN NUMBER

) IS

    v\_balance NUMBER;

BEGIN

    SELECT Balance INTO v\_balance

    FROM Accounts

    WHERE AccountID = p\_from\_account\_id

    FOR UPDATE;

    IF v\_balance < p\_amount THEN

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

    END IF;

    UPDATE Accounts

    SET Balance = Balance - p\_amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_from\_account\_id;

    UPDATE Accounts

    SET Balance = Balance + p\_amount,

        LastModified = SYSDATE

    WHERE AccountID = p\_to\_account\_id;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully.');

EXCEPTION

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

BEGIN

    SafeTransferFunds(1, 2, 100);

END;

/

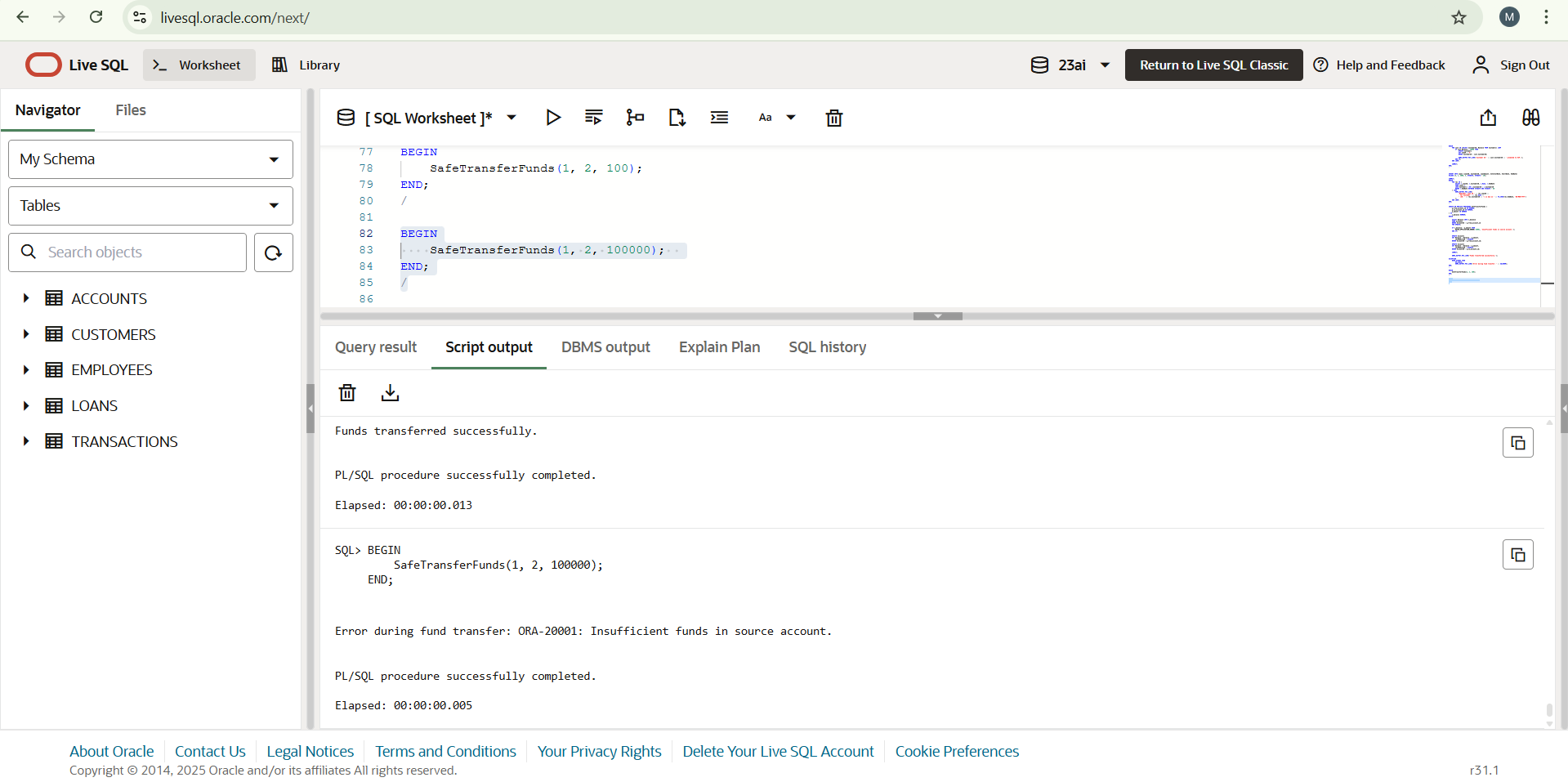
BEGIN

    SafeTransferFunds(1, 2, 100000);

END;

/

**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.

**PROGRAM:**

CREATE OR REPLACE PROCEDURE UpdateSalary(

    p\_employee\_id in NUMBER,

    p\_percentage IN NUMBER

)

IS

BEGIN

    UPDATE EMPLOYEES

    SET SALARY=SALARY+(SALARY\*p\_percentage/100)

    WHERE EMPLOYEEID=p\_employee\_id;

    IF SQL%ROWCOUNT=0 THEN

        RAISE\_APPLICATION\_ERROR(-20002, 'Employee not found.');

    END IF;

    COMMIT;

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

EXCEPTION

    WHEN OTHERS THEN

    ROLLBACK;

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

END;

BEGIN

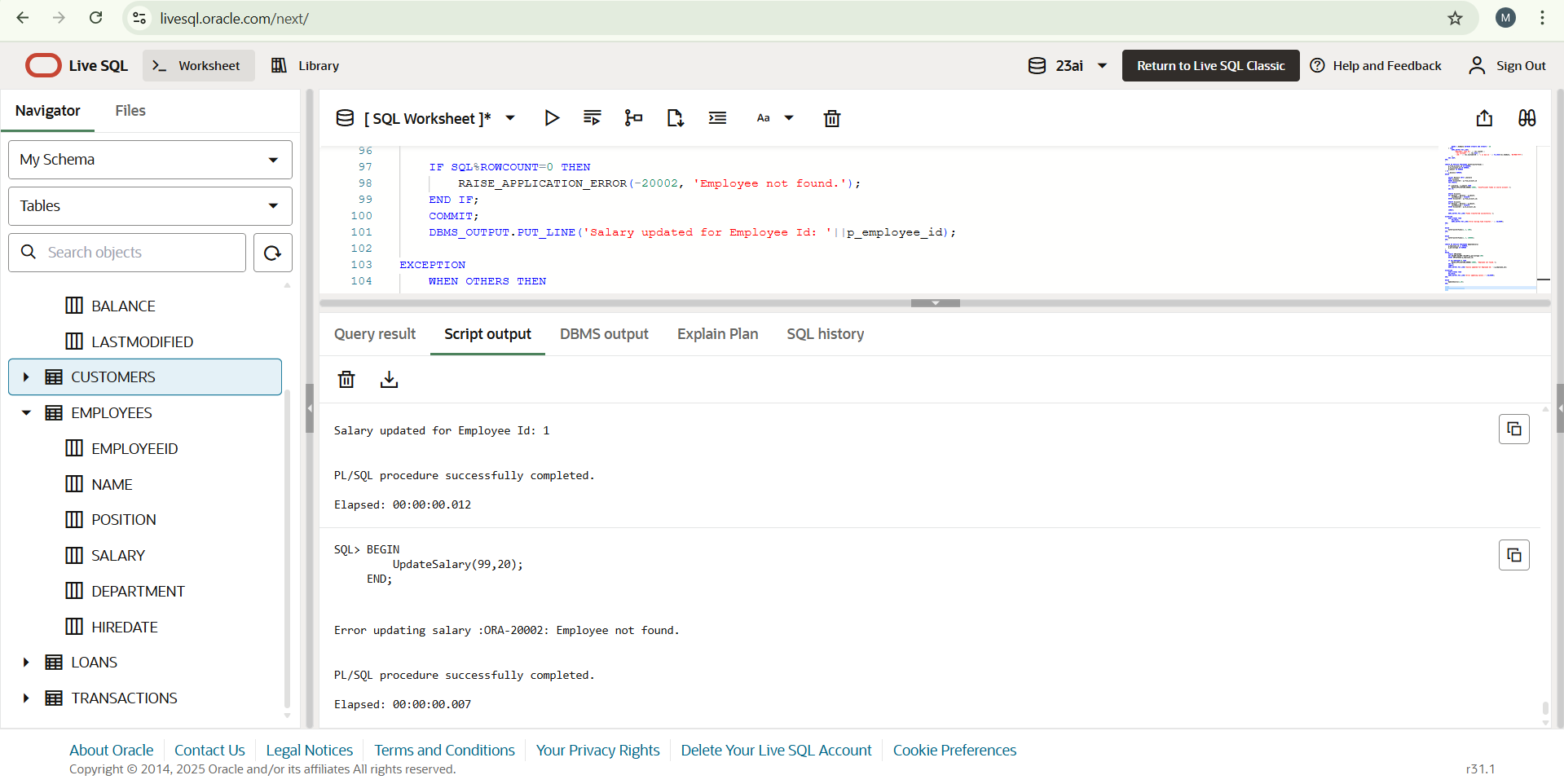
    UpdateSalary(1,10);

END;

BEGIN

    UpdateSalary(99,20);

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.

**PROGRAM:**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

    p\_customerid 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\_customerid,p\_name,p\_dob,p\_balance,SYSDATE);

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Customer added succesfully');

EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

    ROLLBACK;

    DBMS\_OUTPUT.PUT\_LINE('Error: Customer Id '||p\_customerid||'ALREADY EXISTS.');

    WHEN OTHERS THEN

    ROLLBACK;

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

    END;

BEGIN

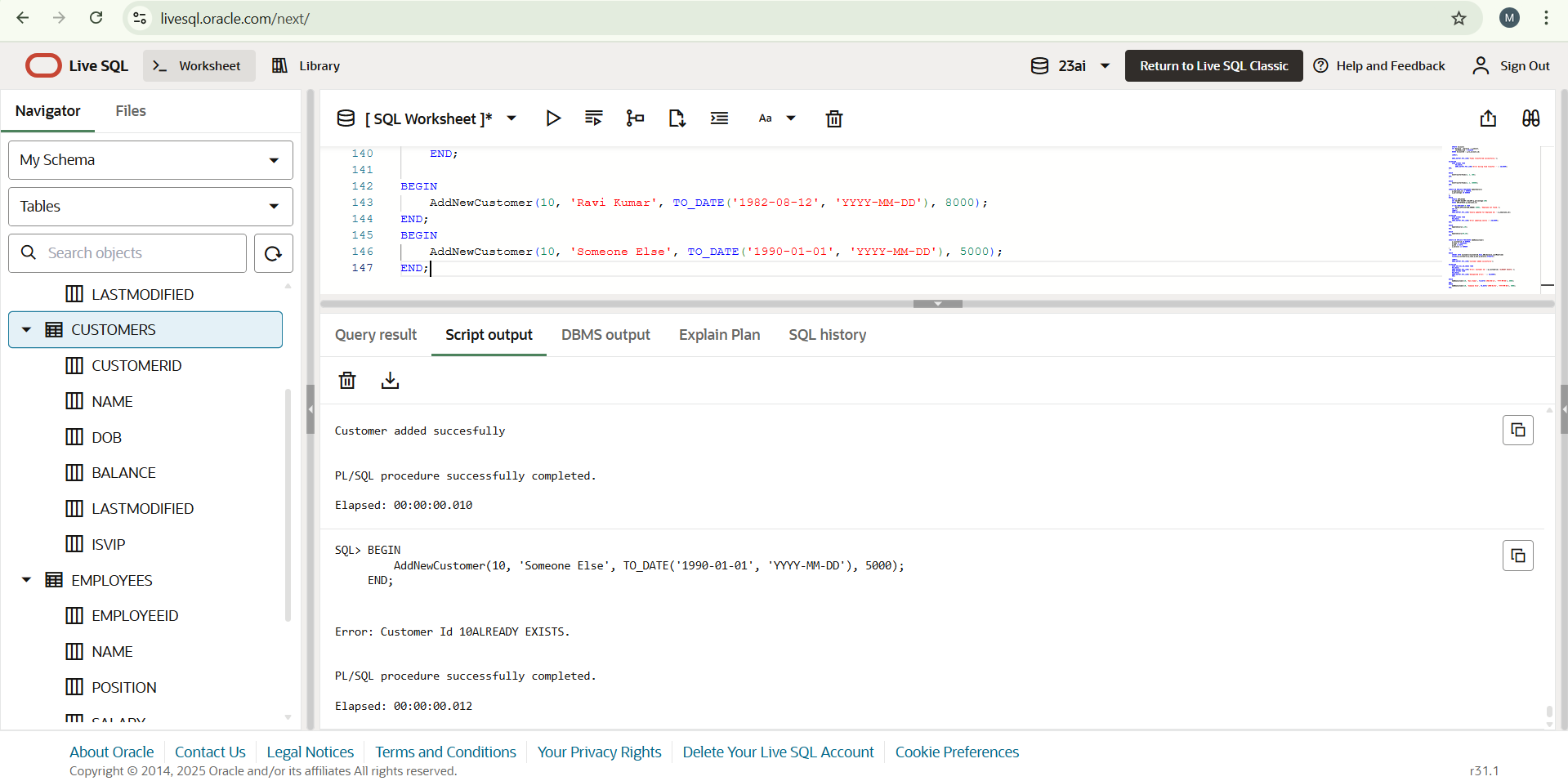
    AddNewCustomer(10, 'Ravi Kumar', TO\_DATE('1982-08-12', 'YYYY-MM-DD'), 8000);

END;

BEGIN

    AddNewCustomer(10, 'Someone Else', TO\_DATE('1990-01-01', '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.

**PROGRAM:**

CREATE OR REPLACE FUNCTION CalculateAge(

    p\_dob IN DATE

)RETURN NUMBER IS

    v\_age NUMBER;

BEGIN

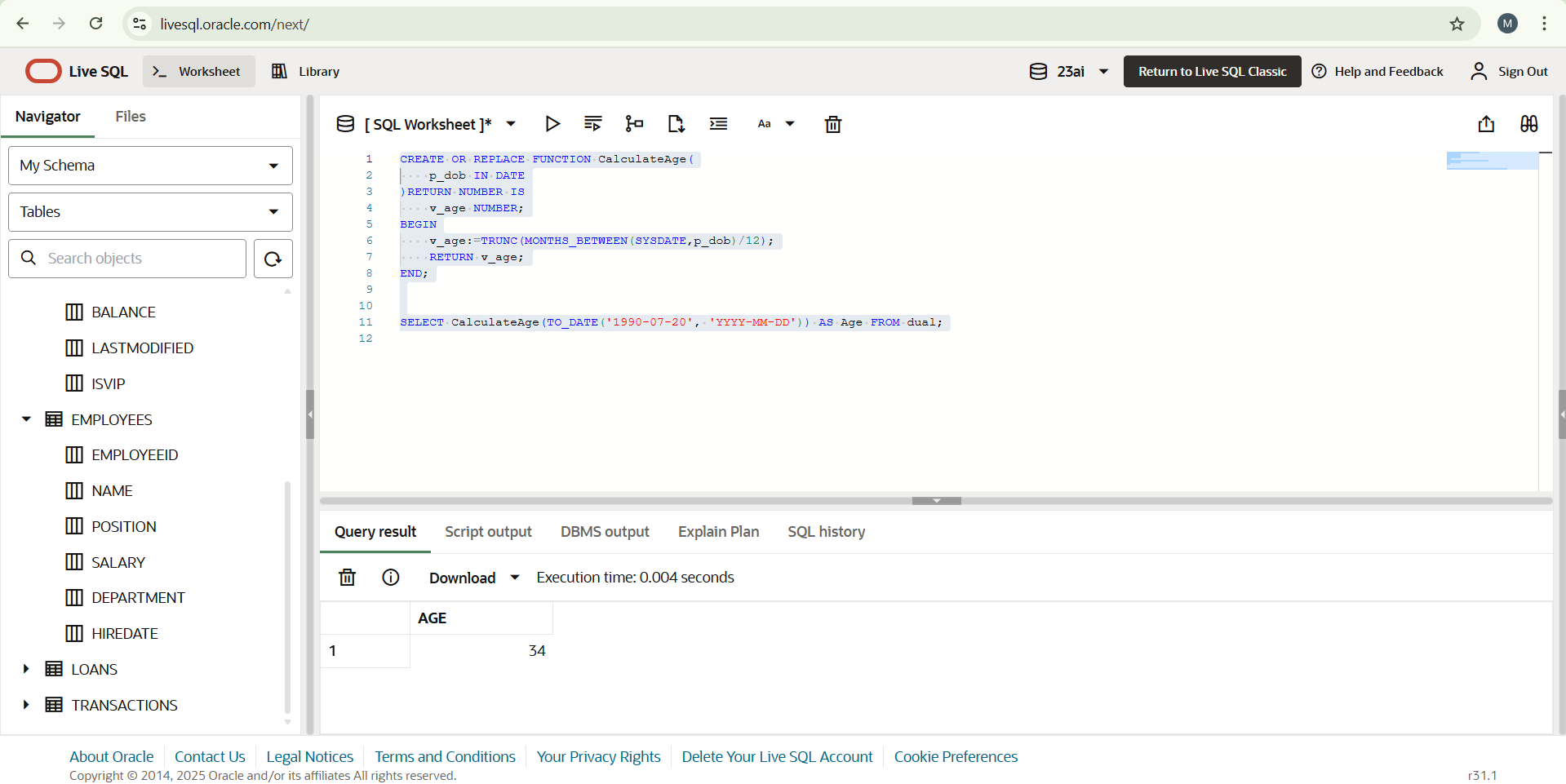
    v\_age:=TRUNC(MONTHS\_BETWEEN(SYSDATE,p\_dob)/12);

    RETURN v\_age;

END;

SELECT CalculateAge(TO\_DATE('1990-07-20', 'YYYY-MM-DD')) AS Age FROM dual;

**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.

**PROGRAM:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

    p\_loan\_amount IN NUMBER,

    p\_annual\_interest\_rate IN NUMBER,

    p\_duration\_years IN NUMBER

)

RETURN NUMBER IS

    v\_monthly\_rate NUMBER;

    v\_months NUMBER;

    v\_emi NUMBER;

BEGIN

    v\_monthly\_rate := p\_annual\_interest\_rate / 12 / 100;

    v\_months := p\_duration\_years \* 12;

    IF v\_monthly\_rate = 0 THEN

        v\_emi := p\_loan\_amount / v\_months;

    ELSE

        v\_emi := p\_loan\_amount \* v\_monthly\_rate \*

                 POWER(1 + v\_monthly\_rate, v\_months) /

                 (POWER(1 + v\_monthly\_rate, v\_months) - 1);

    END IF;

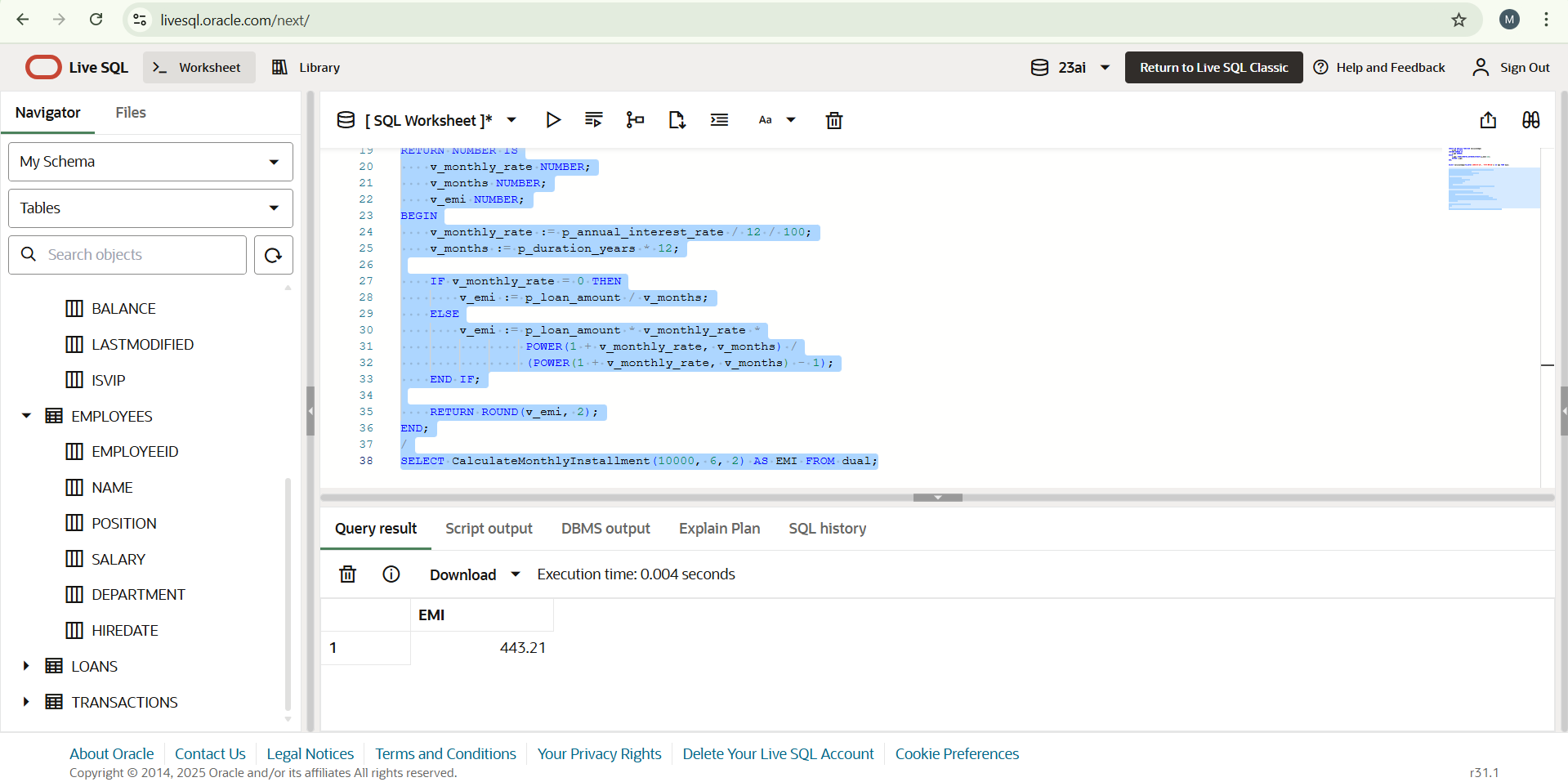
    RETURN ROUND(v\_emi, 2);

END;

/

SELECT CalculateMonthlyInstallment(10000, 6, 2) AS EMI FROM dual;

**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.

**PROGRAM:**

CREATE OR REPLACE FUNCTION HasSufficientBalance (

    p\_account\_id IN NUMBER,

    p\_required\_amount IN NUMBER

) RETURN BOOLEAN IS

    v\_balance NUMBER;

BEGIN

    SELECT Balance INTO v\_balance

    FROM Accounts

    WHERE AccountID = p\_account\_id;

    RETURN v\_balance >= p\_required\_amount;

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        RETURN FALSE;

    WHEN OTHERS THEN

        RETURN FALSE;

END;

/

DECLARE

    v\_result BOOLEAN;

BEGIN

    v\_result := HasSufficientBalance(1, 1000);

    IF v\_result THEN

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

    ELSE

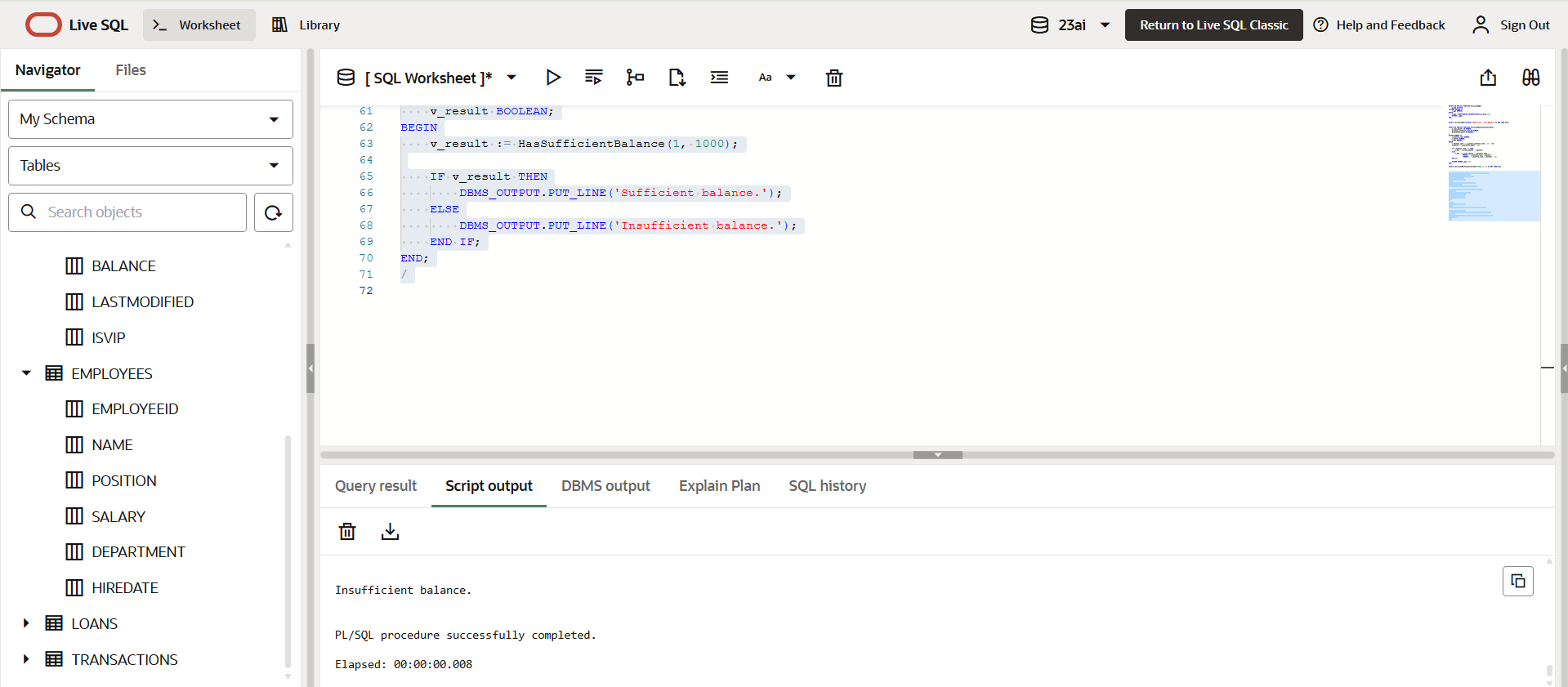
        DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

    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.

**PROGRAM:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

    :NEW.LastModified := SYSDATE;

END;

/

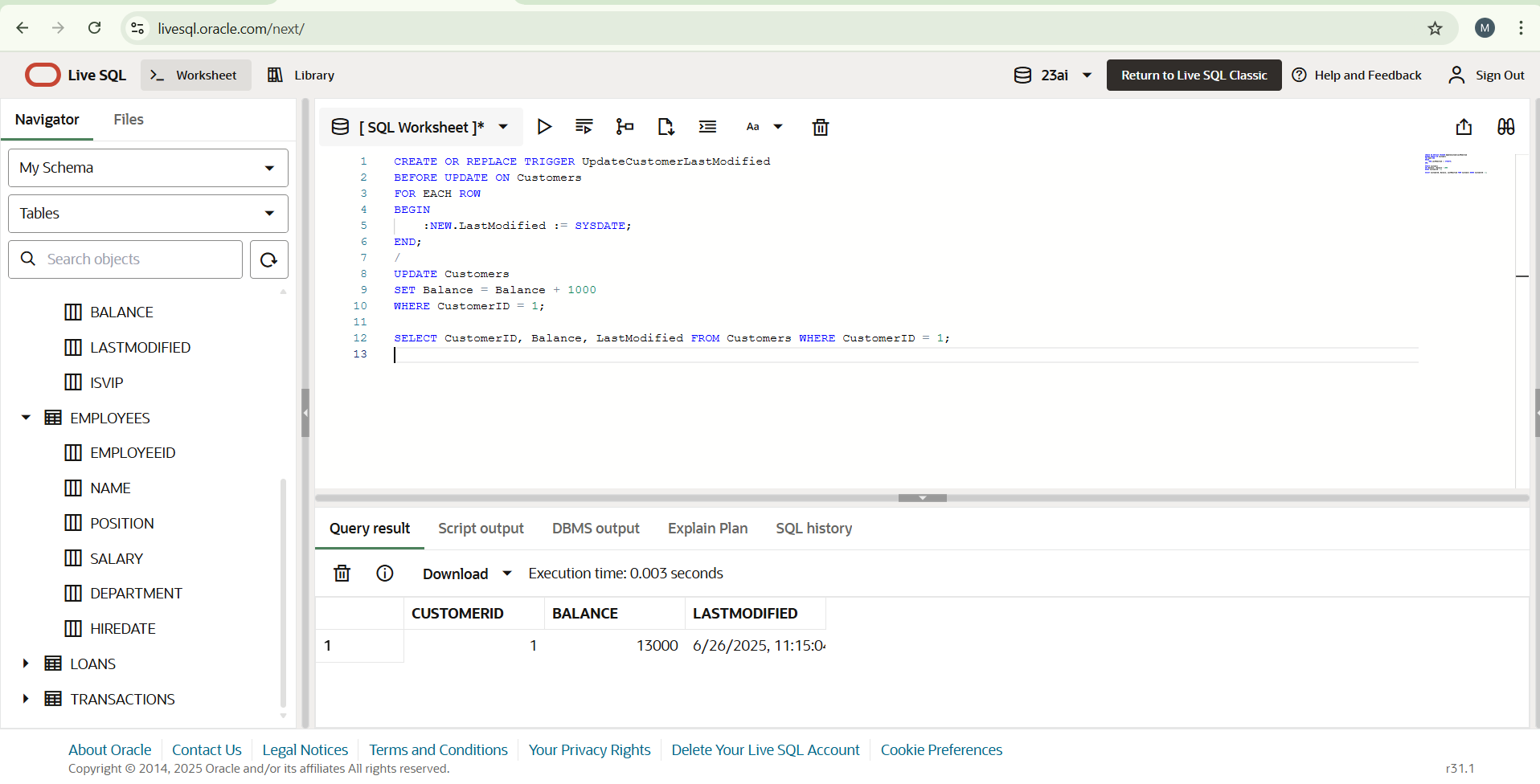
UPDATE Customers

SET Balance = Balance + 1000

WHERE CustomerID = 1;

SELECT CustomerID, Balance, LastModified FROM Customers WHERE CustomerID = 1;

**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.

**PROGRAM:**

CREATE TABLE AuditLog (

    LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

    TransactionID NUMBER,

    ActionDate DATE,

    ActionType VARCHAR2(20),

    Amount NUMBER

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

    INSERT INTO AuditLog (TransactionID, ActionDate, ActionType, Amount)

    VALUES (:NEW.TransactionID, SYSDATE, :NEW.TransactionType, :NEW.Amount);

END;

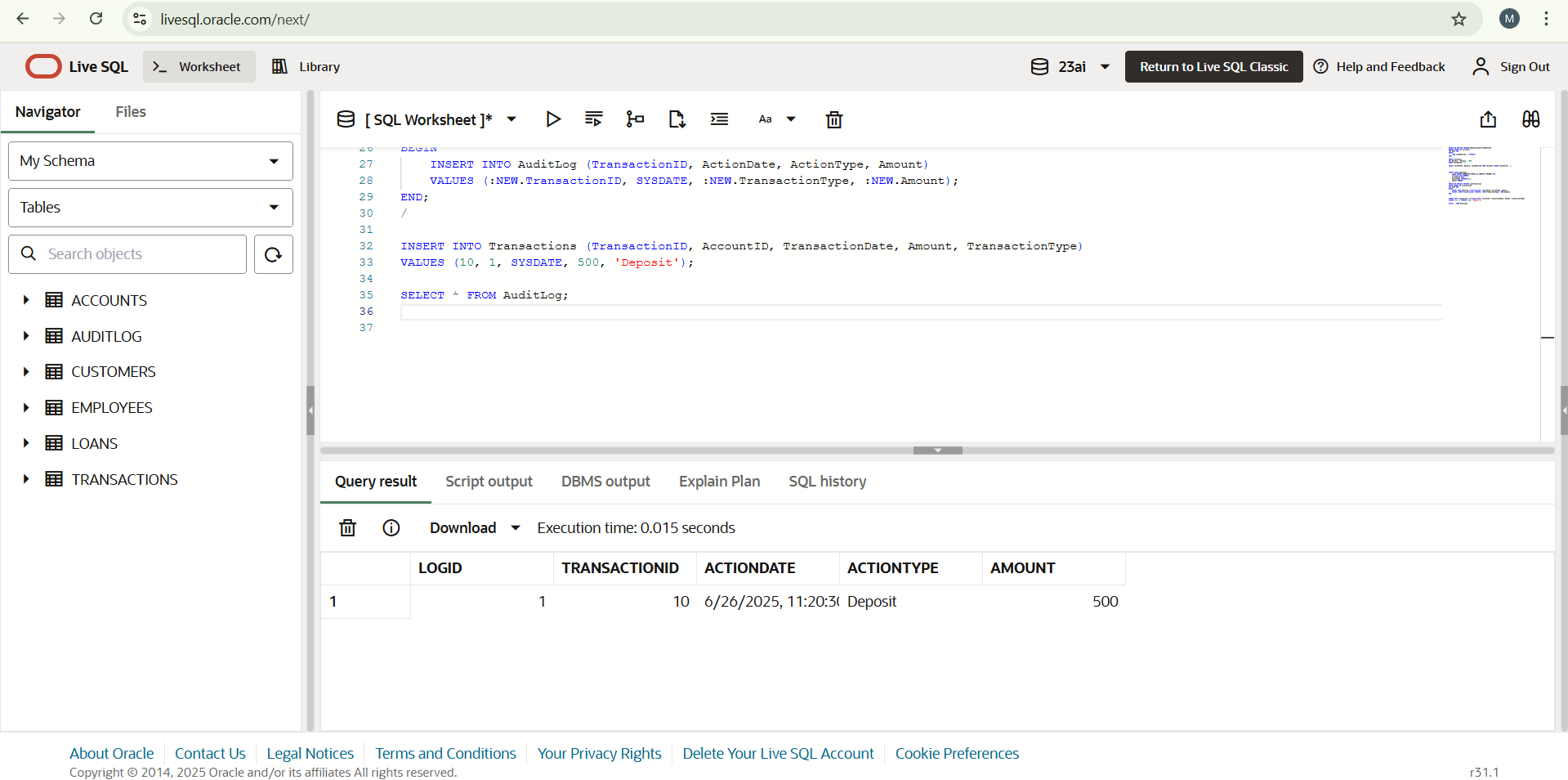
/

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

VALUES (10, 1, SYSDATE, 500, 'Deposit');

SELECT \* FROM AuditLog;

**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.

**PROGRAM:**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

    v\_balance NUMBER;

BEGIN

    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 amount exceeds account balance.');

    END IF;

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

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

    END IF;

END;

/

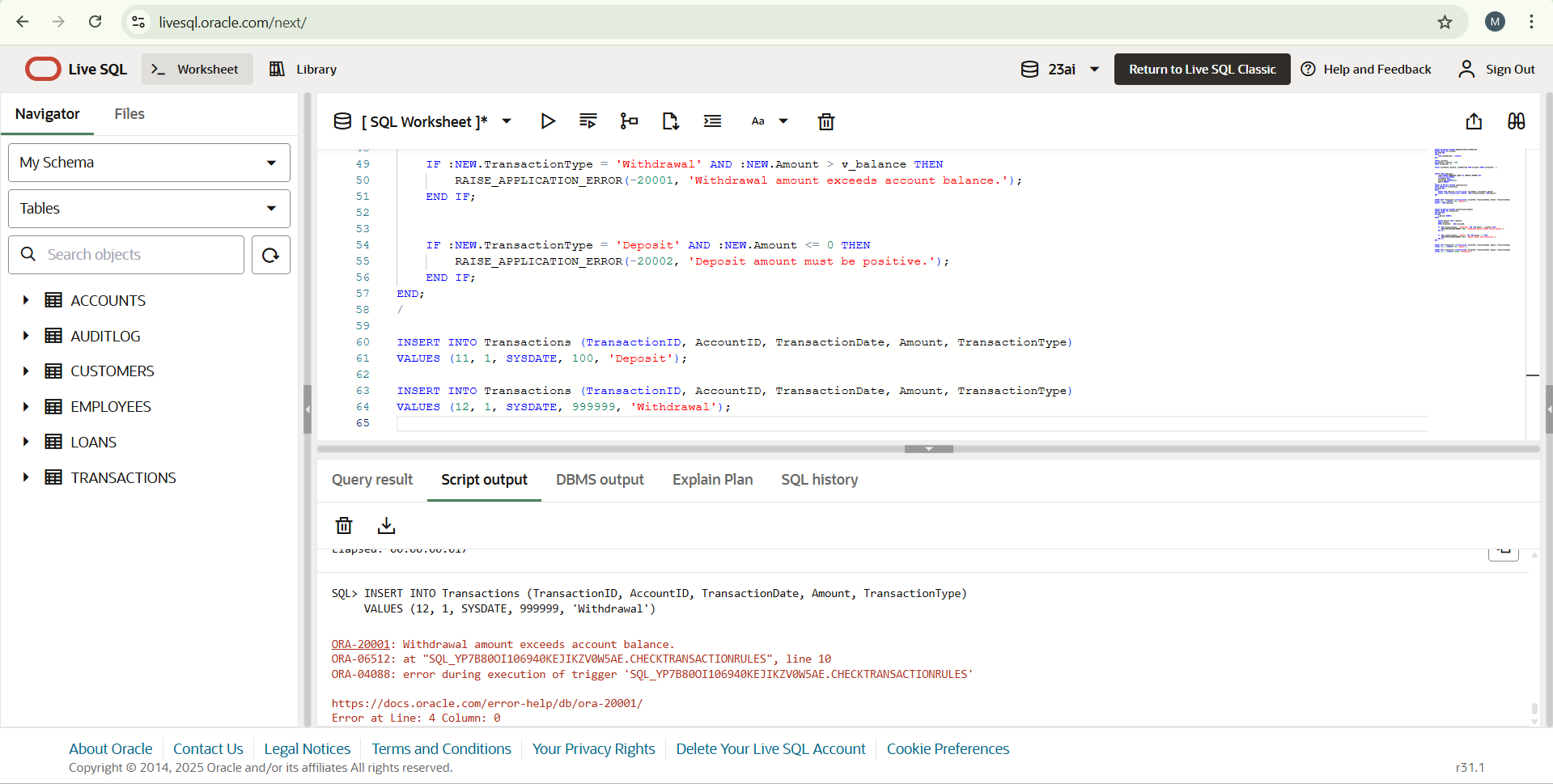
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (11, 1, SYSDATE, 100, 'Deposit');

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

VALUES (12, 1, SYSDATE, 999999, 'Withdrawal');

**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.

**PROGRAM:**

DECLARE

    CURSOR cur\_transactions IS

        SELECT c.CustomerID, c.Name, t.TransactionType, t.Amount, 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, 'MMYYYY') = TO\_CHAR(SYSDATE, 'MMYYYY');

    v\_row cur\_transactions%ROWTYPE;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('Monthly Statement for ' || TO\_CHAR(SYSDATE, 'Month YYYY'));

    OPEN cur\_transactions;

    LOOP

        FETCH cur\_transactions INTO v\_row;

        EXIT WHEN cur\_transactions%NOTFOUND;

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

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

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

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

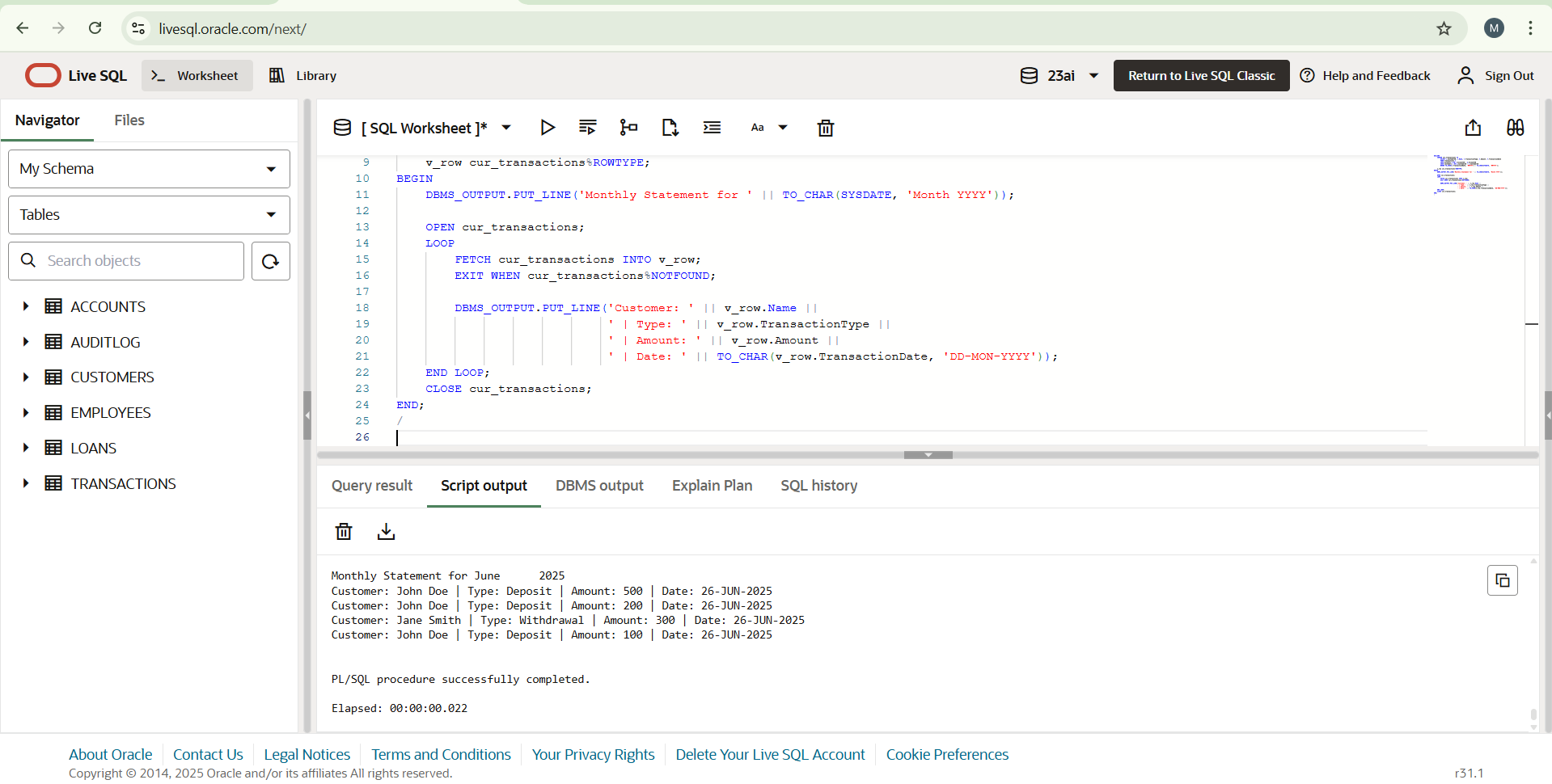
    END LOOP;

    CLOSE cur\_transactions;

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.

**PROGRAM:**

DECLARE

    CURSOR cur\_accounts IS

        SELECT AccountID, Balance FROM Accounts;

    v\_acct cur\_accounts%ROWTYPE;

    v\_fee NUMBER := 200;

BEGIN

    OPEN cur\_accounts;

    LOOP

        FETCH cur\_accounts INTO v\_acct;

        EXIT WHEN cur\_accounts%NOTFOUND;

        UPDATE Accounts

        SET Balance = Balance - v\_fee,

            LastModified = SYSDATE

        WHERE AccountID = v\_acct.AccountID;

        DBMS\_OUTPUT.PUT\_LINE('Annual fee of ' || v\_fee ||

                             ' applied to Account ID: ' || v\_acct.AccountID);

    END LOOP;

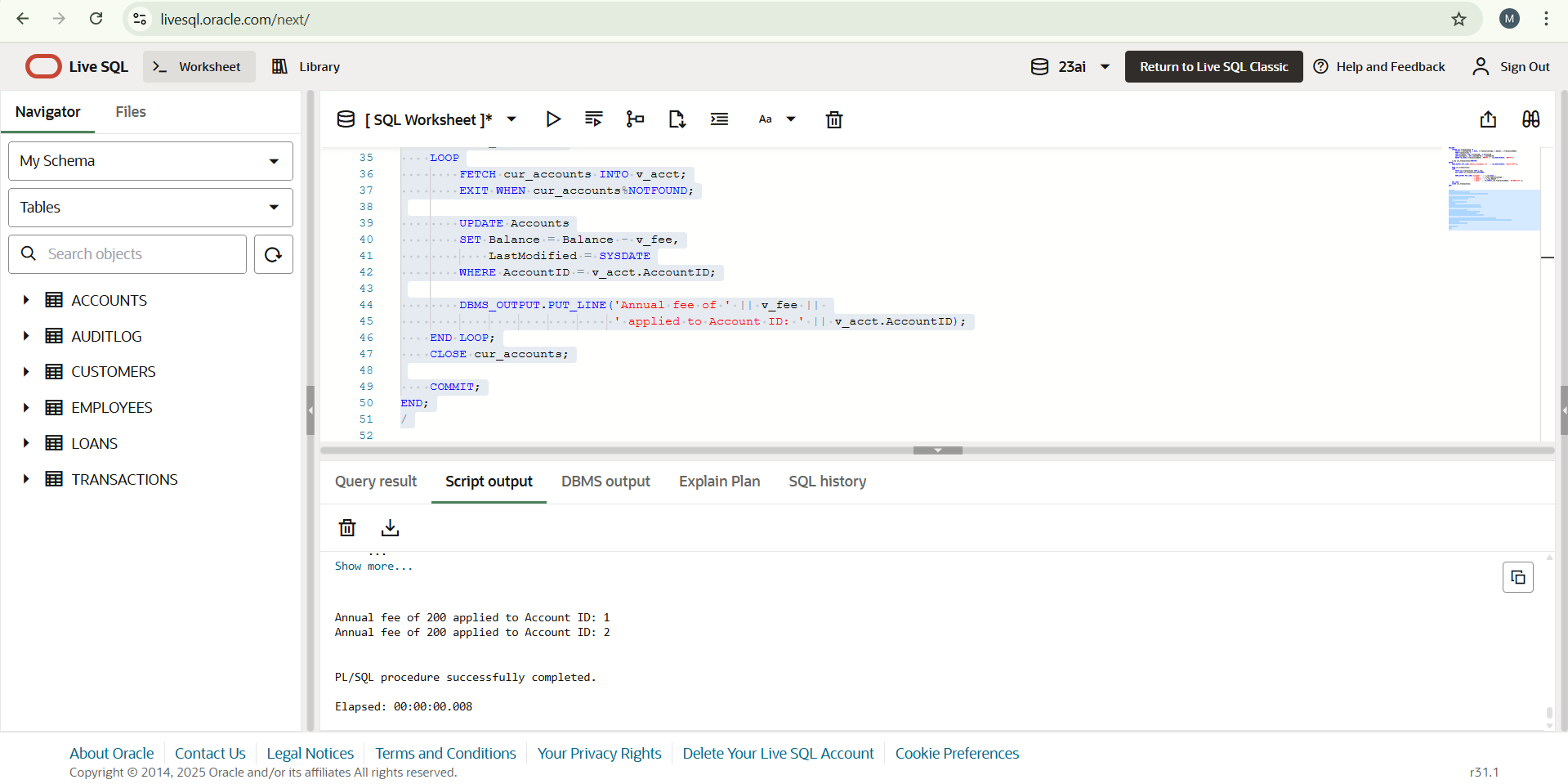
    CLOSE cur\_accounts;

    COMMIT;

END;

/

**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.

**PROGRAM:**

DECLARE

    CURSOR cur\_loans IS

        SELECT LoanID, InterestRate FROM Loans;

    v\_loan cur\_loans%ROWTYPE;

    v\_increment NUMBER := 0.5;

BEGIN

    OPEN cur\_loans;

    LOOP

        FETCH cur\_loans INTO v\_loan;

        EXIT WHEN cur\_loans%NOTFOUND;

        UPDATE Loans

        SET InterestRate = v\_loan.InterestRate + v\_increment

        WHERE LoanID = v\_loan.LoanID;

        DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || v\_loan.LoanID ||

                             ' | New Interest Rate: ' || (v\_loan.InterestRate + v\_increment));

    END LOOP;

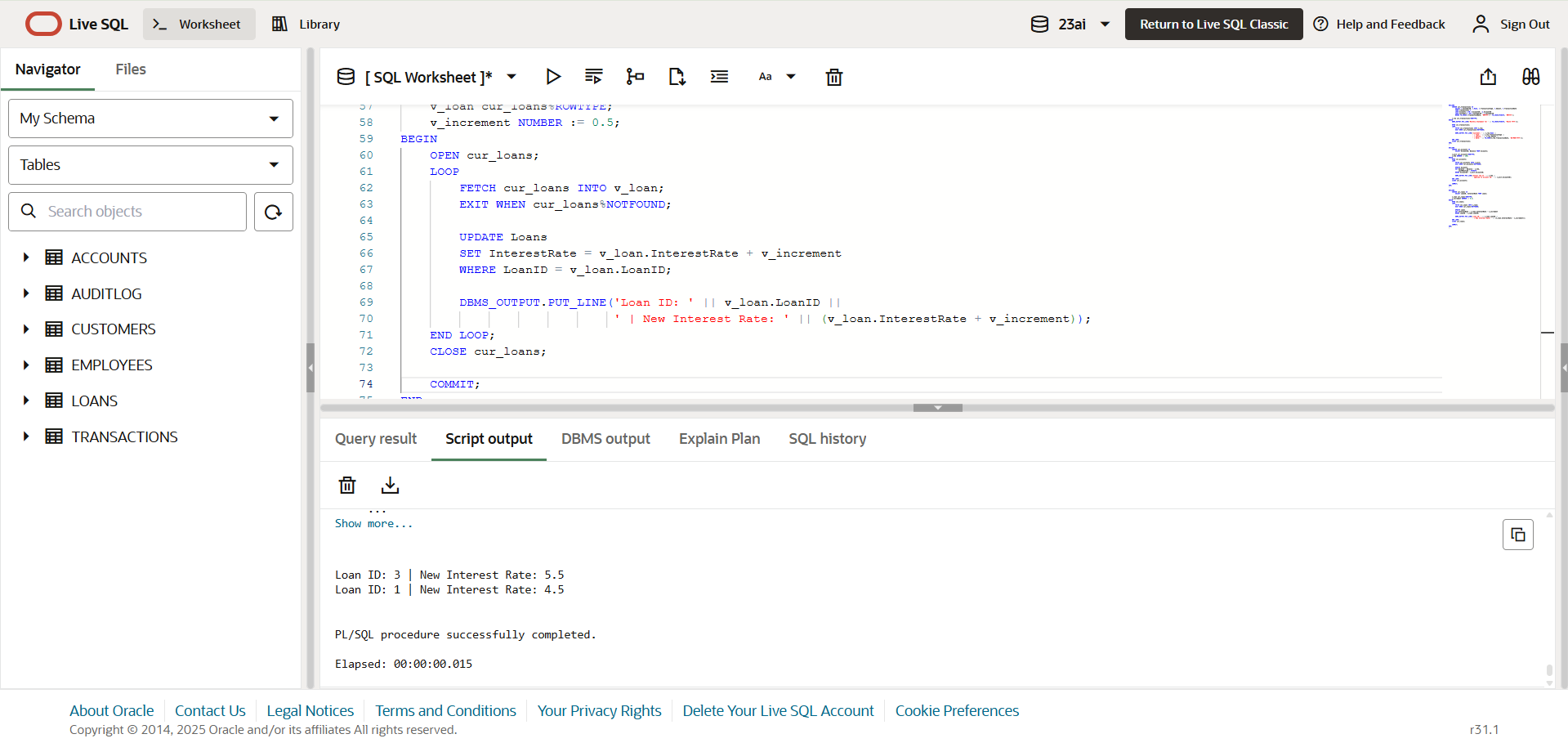
    CLOSE cur\_loans;

    COMMIT;

END;

/

**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.

**PROGRAM:**

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, p\_balance NUMBER);

    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 successfully');

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

    END;

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

    BEGIN

        UPDATE Customers

        SET Name = p\_name,

            Balance = p\_balance,

            LastModified = SYSDATE

        WHERE CustomerID = p\_id;

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Customer updated');

        END IF;

    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;

/

BEGIN

    CustomerManagement.AddCustomer(5, 'Suresh', TO\_DATE('1992-03-25', 'YYYY-MM-DD'), 4000);

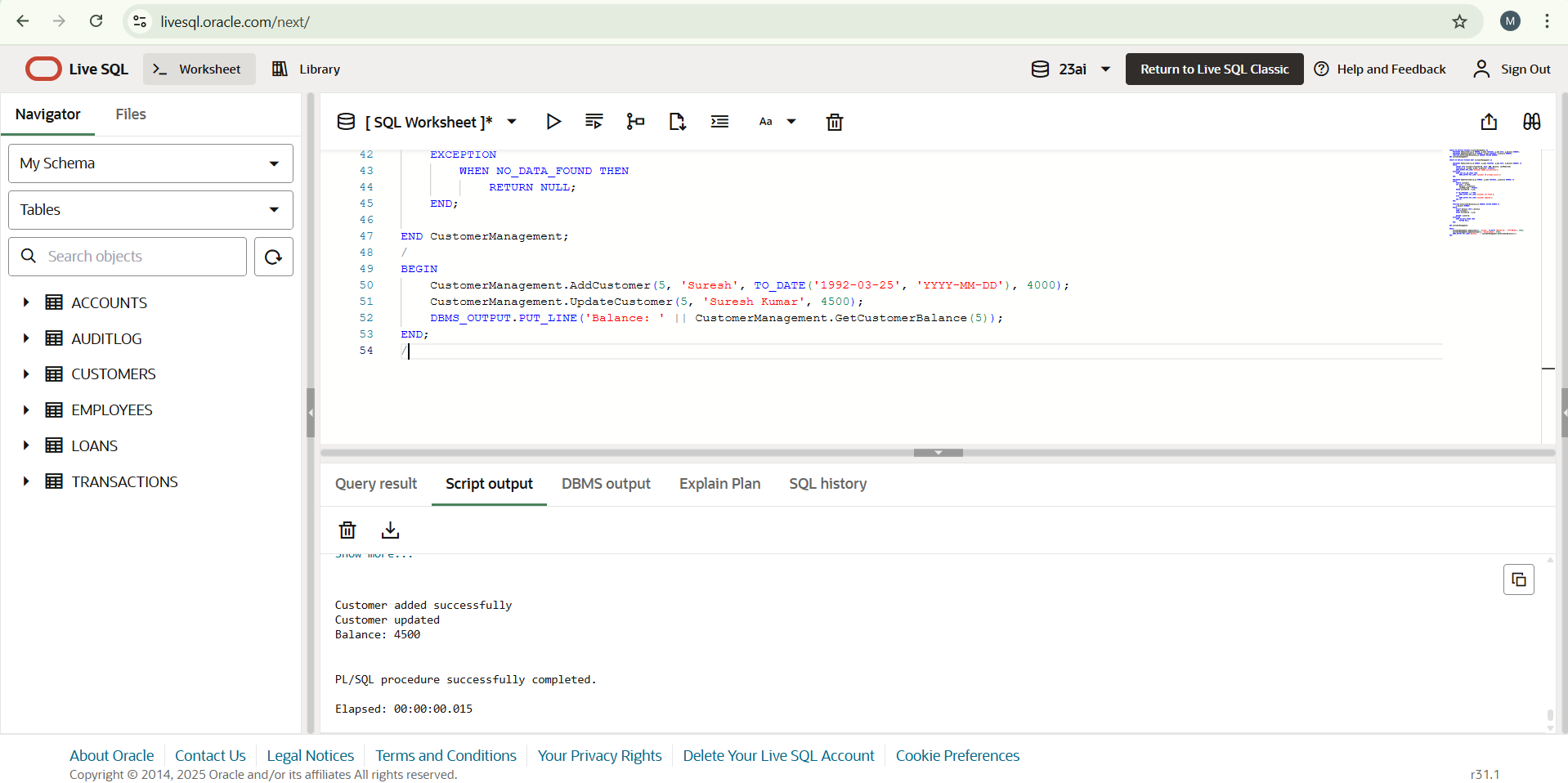
    CustomerManagement.UpdateCustomer(5, 'Suresh Kumar', 4500);

    DBMS\_OUTPUT.PUT\_LINE('Balance: ' || CustomerManagement.GetCustomerBalance(5));

END;

/

**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.

**PROGRAM:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

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

    BEGIN

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

        VALUES(p\_id, p\_name, p\_position, p\_salary, p\_dept, p\_date);

        DBMS\_OUTPUT.PUT\_LINE('Employee hired successfully');

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

    END;

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

    BEGIN

        UPDATE Employees

        SET Salary = p\_salary

        WHERE EmployeeID = p\_id;

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Employee salary updated');

        END IF;

    END;

    FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

        v\_salary NUMBER;

    BEGIN

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

        RETURN v\_salary \* 12;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

            RETURN NULL;

    END;

END EmployeeManagement;

/

BEGIN

    EmployeeManagement.HireEmployee(3, 'Mukesh P', 'Tester', 50000, 'QA', SYSDATE);

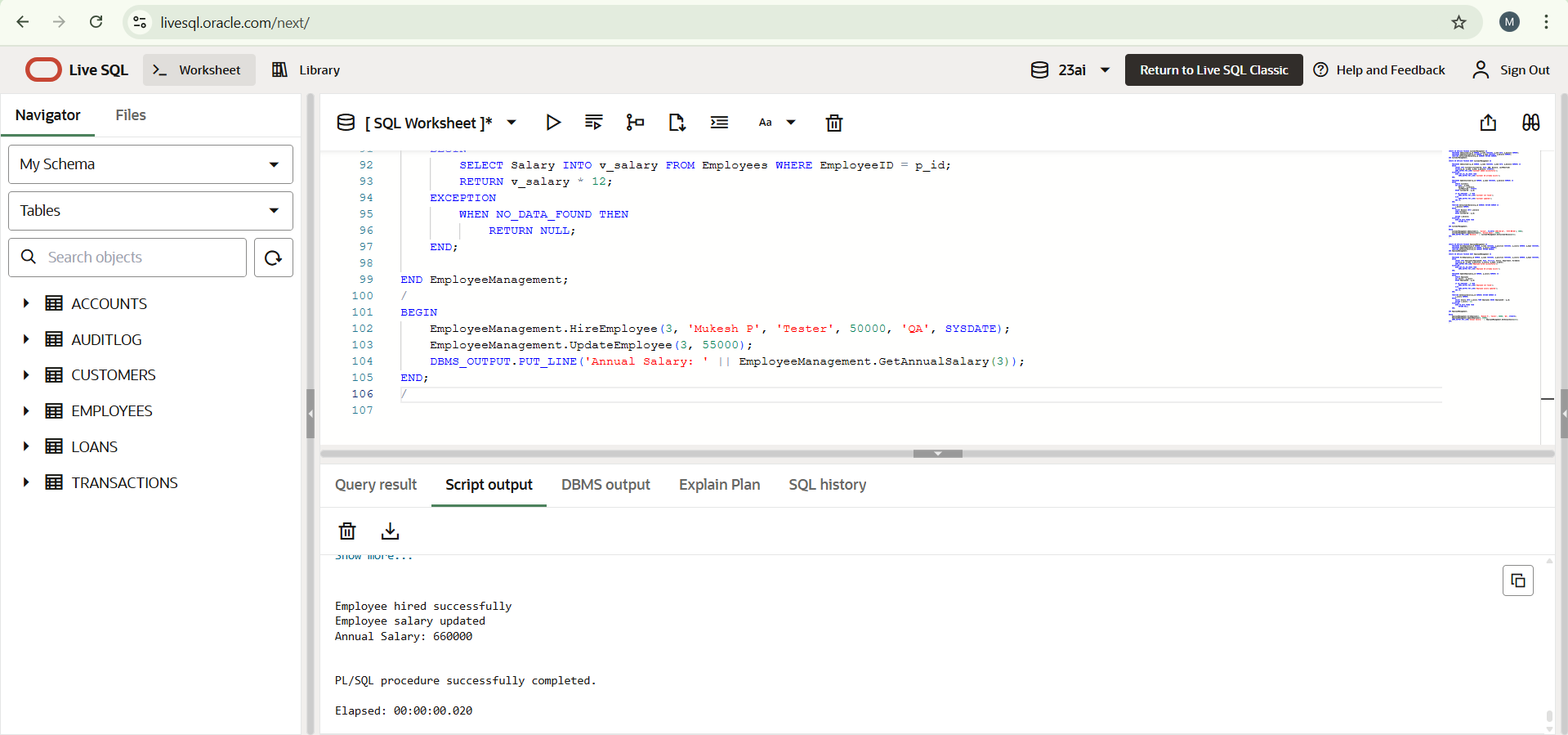
    EmployeeManagement.UpdateEmployee(3, 55000);

    DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || EmployeeManagement.GetAnnualSalary(3));

END;

/

**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.

**PROGRAM:**

CREATE OR REPLACE PACKAGE AccountOperations AS

    PROCEDURE OpenAccount(p\_accountid NUMBER, p\_customerid NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

    PROCEDURE CloseAccount(p\_accountid NUMBER);

    FUNCTION GetTotalBalance(p\_customerid NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

    PROCEDURE OpenAccount(p\_accountid NUMBER, p\_customerid NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

    BEGIN

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

        VALUES(p\_accountid, p\_customerid, p\_type, p\_balance, SYSDATE);

        DBMS\_OUTPUT.PUT\_LINE('Account opened');

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

    END;

    PROCEDURE CloseAccount(p\_accountid NUMBER) IS

    BEGIN

        DELETE FROM Accounts WHERE AccountID = p\_accountid;

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Account closed');

        END IF;

    END;

    FUNCTION GetTotalBalance(p\_customerid NUMBER) RETURN NUMBER IS

        v\_total NUMBER;

    BEGIN

        SELECT SUM(Balance) INTO v\_total

        FROM Accounts

        WHERE CustomerID = p\_customerid;

        RETURN NVL(v\_total, 0);

    END;

END AccountOperations;

/

BEGIN

    AccountOperations.OpenAccount(10, 1, 'Savings', 3000);

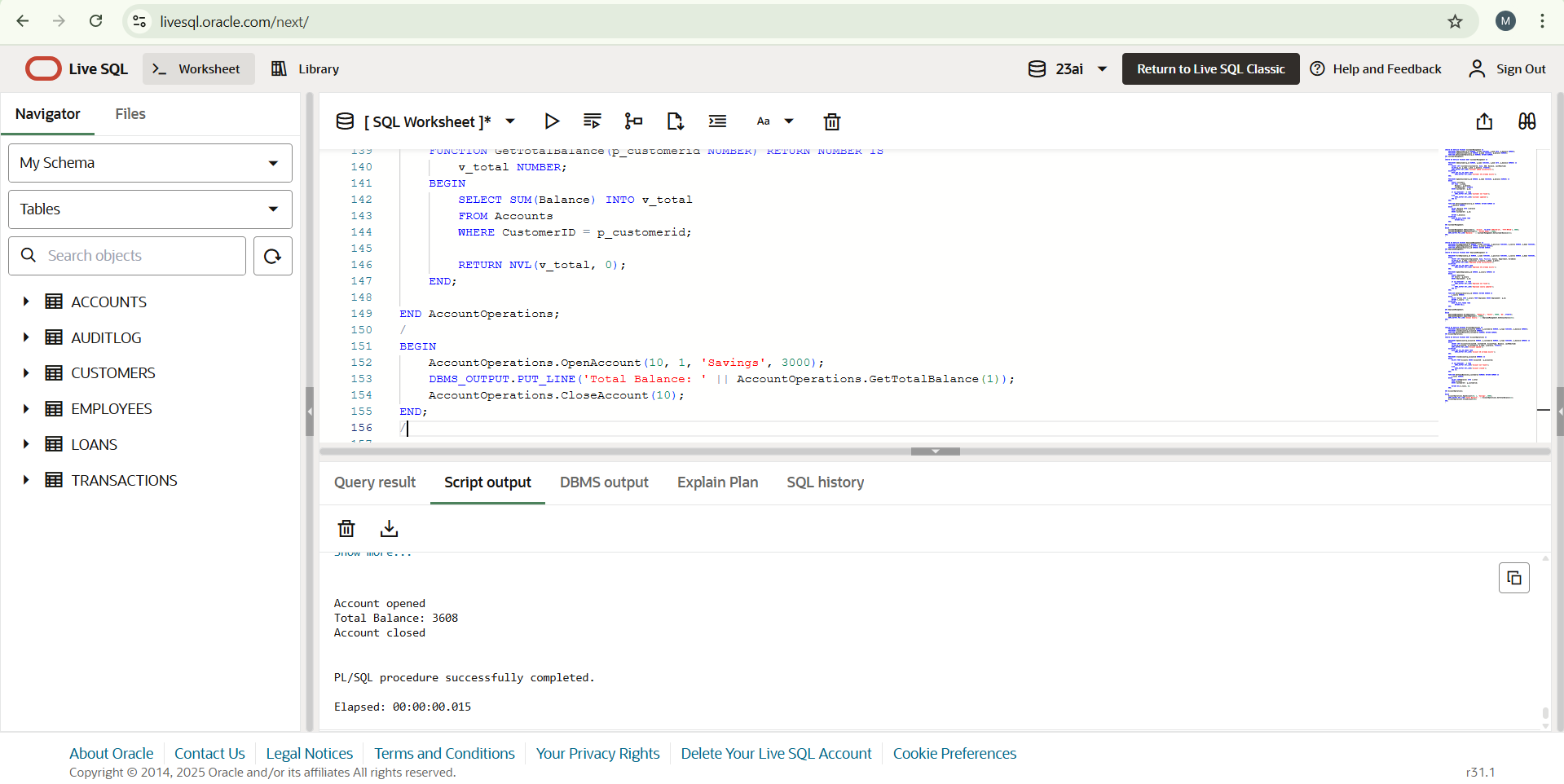
    DBMS\_OUTPUT.PUT\_LINE('Total Balance: ' || AccountOperations.GetTotalBalance(1));

    AccountOperations.CloseAccount(10);

END;

/

**OUTPUT:**



**Schema to be Created**

*CREATE TABLE Customers (*

*CustomerID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*DOB DATE,*

*Balance NUMBER,*

*LastModified DATE*

*);*

*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*

*);*

**Example Scripts for 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 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'));*