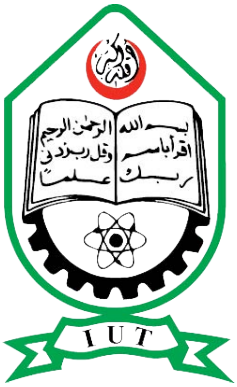
Islamic University of Technology



RELATIONAL DATABASE MANAGEMENT SYSTEM LAB

CSE 4508

Lab 2: Triggers, Cursors, Recursive Queries and Advanced Aggregation Features

Author:
Ahmed M. S. Albreem (210041258)

October 8, 2024

Contents

1 Introduction

This report outlines solutions to various SQL tasks involving employee salary updates, bank transaction processing, mobile phone billing, and student database management. The tasks utilize SQL features like stored procedures, triggers, and functions to efficiently handle database operations.

2 Task 1: Updating Employee Salaries

The first task is focused on updating employee salaries based on specific criteria. Managers with a salary below $30,000 will receive a 10% salary increase, while assistant managers with a salary above $20,000 will receive a 10% decrease.

```sql
UPDATE employees
SET salary = CASE
                WHEN job_title = 'manager' AND salary < 30000
    THEN salary * 1.10  -- 10% increase for managers
                WHEN job_title = 'assistant manager' AND
   salary > 20000 THEN salary * 0.90  -- 10% decrease for
   assistant managers
                ELSE salary  -- No change for other cases
            END
WHERE job_title IN ('manager', 'assistant manager');

SELECT COUNT(*) AS affected_rows
FROM employees
WHERE job_title IN ('manager', 'assistant manager');
```

Listing 1: Task 1: Updating Employee Salaries

In this solution:

- **Salary Update**: The `CASE` statement handles conditional salary adjustments based on designation and salary range.
- **Count Affected Rows**: The final query verifies how many rows were impacted by the update.

2.1 Error Handling

To avoid potential errors during the update, it is recommended to encapsulate the update in a transaction, allowing for rollback in case of failure.

2.2 Variable Names and Values

- `salary`: Represents the employee's current salary.
- `designation`: Indicates the employee's job title.
- The `CASE` statement checks conditions to determine the new salary.

3 Task 2: Bank Transactions and Loan Scheme

This task involves setting up bank transaction tables and defining a function to assign loan schemes based on transaction history. The 'loan$_type$'$table specifies minimum transact$

3.1 Table Creation

The first step is to create tables to store transaction data and loan types.

```
CREATE TABLE bank_transactions (
  customer_id INT,
  transaction_amount DECIMAL(10,2),
  transaction_date DATE
);

CREATE TABLE loan_schemes (
  scheme_id INT,
  installment_count INT,
  interest_rate DECIMAL(4,2),
  minimum_transactions DECIMAL(10,2)
);
```

3.2 Function to Determine Loan Scheme

Next, the function calculates the loan scheme based on the user's total transaction amount.

```
CREATE OR REPLACE FUNCTION get_loan_scheme(user_id IN INT)
RETURN INT
IS
  total_transactions DECIMAL(10,2);
  loan_scheme INT := NULL;
BEGIN
  SELECT SUM(Amount) INTO total_transactions
  FROM transactions
```

```
  WHERE User_ID = user_id;

  SELECT Scheme INTO loan_scheme
  FROM loan_type
  WHERE total_transactions >= Min_Trans
  ORDER BY Min_Trans DESC
  FETCH FIRST 1 ROWS ONLY;

  RETURN loan_scheme;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN NULL;
  WHEN OTHERS THEN
    RETURN NULL;
END;
/
```

3.3 Variable Names and Values

- `user_id_in`: Input parameter representing the user ID.
- `total_trans`: Variable holding the sum of transactions for the given user.

3.4 Purpose of Function

The function `get_loan_scheme` determines the loan scheme based on the user's transaction history.

3.5 Error Handling

Ensure that the `user_id_in` exists in the transactions table. Consider raising an exception if no transactions are found.

4 Task 3: Mobile Phone Billing System

This task simulates a mobile billing system, where triggers handle billing updates based on customer usage. Customers have plans with specific billing rates.

4.1 Table Creation and Trigger Setup

Tables are created for customers and bills, and triggers automate billing updates when new calls are made.

```
-- Create Customer Table
CREATE TABLE CUSTOMER (
  SSN VARCHAR(10) PRIMARY KEY,
  CustomerName VARCHAR(50),
  CustomerSurname VARCHAR(50),
  PhoneNumber VARCHAR(15),
  Plan VARCHAR(10)
);

-- Create Pricing Plan Table
CREATE TABLE PRICINGPLAN (
  PlanCode VARCHAR(10) PRIMARY KEY,
  ConnectionFee DECIMAL(10,2),
  RatePerSec DECIMAL(4,2)
);

-- Create Phone Call Table
CREATE TABLE PHONECALL (
  CustomerSSN VARCHAR(10),
  CallDate DATE,
  CallTime VARCHAR(8),
  CalledNumber VARCHAR(15),
  CallDuration INT,
  FOREIGN KEY (CustomerSSN) REFERENCES CUSTOMER(SSN)
);

-- Create Bill Table
CREATE TABLE BILL (
  CustomerSSN VARCHAR(10),
  BillMonth INT,
  BillYear INT,
  BillAmount DECIMAL(10,2),
  PRIMARY KEY (CustomerSSN, BillMonth, BillYear),
  FOREIGN KEY (CustomerSSN) REFERENCES CUSTOMER(SSN)
);

-- Create a Trigger to Update Bill Amount After a New Phone
   Call
CREATE OR REPLACE TRIGGER update_bill
AFTER INSERT ON PHONECALL
FOR EACH ROW
```

```
BEGIN
  UPDATE BILL
  SET BillAmount = BillAmount +
      (SELECT RatePerSec
       FROM PRICINGPLAN
       WHERE PlanCode = (SELECT Plan
                         FROM CUSTOMER
                         WHERE SSN = :NEW.CustomerSSN)) * :NEW.
    CallDuration
  WHERE CustomerSSN = :NEW.CustomerSSN
    AND BillMonth = EXTRACT(MONTH FROM :NEW.CallDate)
    AND BillYear = EXTRACT(YEAR FROM :NEW.CallDate);
END;
/
```

4.2 Variable Names and Values

- `SSN`: The unique identifier for customers.
- `Plan`: Indicates the customer's billing plan.

4.3 Purpose of Trigger

The `update_bill` trigger updates the customer's bill automatically when a new phone call record is inserted.

4.4 Error Handling

Check for potential errors when accessing data from `CUSTOMER` and `PRICINGPLAN` tables. Use exception handling to manage cases where no matching records are found.

5 Task 4: Student Database Management

In this task, we create a database for student records with automated ID generation. IDs incorporate admission year, department, and program information.

5.1 Function and Trigger for Auto-generating ID

The function and trigger automate ID creation when a new student is added.

```
CREATE OR REPLACE FUNCTION GenerateStudentID(
  AdmissionDate IN DATE,
  DepartmentCode IN CHAR,
  ProgramCode IN CHAR,
  SectionCode IN CHAR
)
RETURN VARCHAR2
IS
  YearPart VARCHAR2(2);
  StudentID VARCHAR2(8);
  SeqNum NUMBER;
BEGIN
  YearPart := SUBSTR(TO_CHAR(AdmissionDate, 'YYYY'), 3, 2);

  SeqNum := SEQ_ID.NEXTVAL;

  StudentID := YearPart || DepartmentCode || ProgramCode ||
   SectionCode || LPAD(SeqNum, 2, '0');


  RETURN StudentID;
END;
/
```

5.2 Variable Names and Values

- `Date_of_Admission_in`: Input parameter for the admission date.
- `Department_in`, `Program_in`, `Section_in`: Input parameters for department, program, and section.
- `YY`: Year extracted from the admission date.
- `SEQ_ID`: A sequence number used for ID generation.

5.3 Purpose of Function

The function `Gen_ID` generates a unique student ID by combining various attributes related to the student.

5.4 Error Handling

Validate the input parameters to ensure they conform to expected formats and values before proceeding with ID generation.

6 Task 5: Calculating Interest for Bank Accounts

The task is to create a procedure that calculates interest for each account and updates the balance accordingly.

```
CREATE OR REPLACE PROCEDURE CalculateAccountInterest
IS
  CURSOR accounts_cursor IS
    SELECT * FROM account;
  interest_amount NUMBER;
  new_balance NUMBER;
BEGIN
  FOR account_record IN accounts_cursor LOOP
    interest_amount := account_record.balance *
   account_record.interest_rate;

    new_balance := account_record.balance + interest_amount;

    UPDATE account
    SET balance = new_balance
    WHERE acc_num = account_record.acc_num;
  END LOOP;
END;
/
```

6.1 Variable Names and Values

- `rec`: A record variable that represents each row in the cursor.
- `interest`: The calculated interest for the account.
- `new_balance`: The updated balance after adding the interest.

6.2 Purpose of Procedure

The procedure `Interest_Calculation` automates the calculation of interest for all accounts, ensuring balances are updated accordingly.

6.3 Error Handling

Implement checks to ensure that the balance and interest rate are not null and that accounts exist before performing updates.

7 Conclusion

In summary, these SQL tasks demonstrate essential database operations for salary management, loan scheme assignments, billing systems, student ID management, and interest calculation. They highlight the utility of SQL procedures, triggers, and functions in automating complex database transactions.