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

CSE 4508

RELATIONAL DATABASE MANAGEMENT SYSTEM LAB SEPTEMBER 24, 2024

Contents

Winter 2024 Triggers, Cursors, Recursive Queries and Advanced Aggregation Features Lab 2

1 Triggers

A trigger is a special type of stored procedure that automatically executes in response to specific events occurring within a database (when an INSERT, UPDATE, or DELETE operation performed on a database table). It serves as a mechanism for responding to data modifications, thereby ensuring data integrity and enforcing business rules without the need for manual intervention.

Scope of a trigger can be:

- FOR EACH ROW: The trigger executes once for each row affected by the triggering event.
- FOR EACH STATEMENT: The trigger executes once per SQL statement, regardless of the num ber of rows affected.

Triggers are particularly useful for automating repetitive tasks. For instance, a trigger can be config ured to perform calculations following each specific database action. Additionally, triggers can be established to carry out data validation tasks on a table.

Real-World Examples:

1. Automatically updating last_modified timestamp column whenever a record is updated.

2. Validating data before insertion to ensure it adheres to specific business rules. 3. Cascading updates or deletions across related tables to maintain referential integrity. **1.1 Syntax of Triggers**

```
CREATE TRIGGER trigger_name
        { BEFORE | AFTER } { INSERT | UPDATE | DELETE } ON table_name [ FOR
        EACH ROW | FOR EACH STATEMENT ]
        [ WHEN ( condition ) ]
BEGIN
        -- Trigger actions
END ;

/
```

Note: Remember to SET SERVEROUTPUT ON to see the results of the blocks. **1.2 Trigger Examples**

Trigger Example 1:

In the takes table, if a grade is updated to a blank space (“ ”), automatically convert it to NULL to maintain data consistency.

```
CREATE TRIGGER setnull_trigger
        BEFORE UPDATE ON takes
        REFERENCING NEW ROW AS nrow
        FOR EACH ROW
        WHEN ( nrow . grade = ' ')
BEGIN ATOMIC
        SET nrow . grade = NULL ;
END ;
/
```

- Trigger Timing: BEFORE UPDATE ensures the trigger fires before the update operation on the takes table.

- Condition: WHEN (nrow.grade = ' ') checks if the new grade value is a blank space.
- Action: SET nrow.grade = NULL; sets the grade to NULL if the condition is met.

Trigger Example 2

When a student's grade changes from 'F' or NULL to a passing grade, automatically update their total credits earned (tot_cred) based on the course's credit value.

```
CREATE TRIGGER credits_earned
AFTER UPDATE OF grade ON takes
REFERENCING NEW ROW AS nrow
                OLD ROW AS orow
FOR EACH ROW
WHEN nrow . grade <> 'F' AND nrow . grade IS NOT NULL
      AND ( orow . grade = 'F' OR orow . grade IS NULL )
BEGIN ATOMIC
      UPDATE student
      SET tot_cred = tot_cred + (
            SELECT credits
            FROM course
            WHERE course . course_id = nrow . course_id
      )
      WHERE student . id = nrow . id ;
END ;
/
```

- Trigger Timing: AFTER UPDATE OF grade ensures the trigger fires after the grade column is updated in the takes table.
- Condition: nrow.grade <> 'F' AND nrow.grade IS NOT NULL: The new grade is passing. (orow.grade = 'F' OR orow.grade IS NULL): The previous grade was failing or undefined.
- Action: Updates the tot_cred in the student table by adding the course's credit value.Uses a subquery to fetch the credits from the course table based on the course_id.

Trigger Example 3

Implement a trigger that executes once per UPDATE statement on the employee table, regardless of how many rows are affected. This is useful for logging bulk operations or enforcing constraints that apply to the entire operation.

```
CREATE TRIGGER bulk_update_trigger
AFTER UPDATE ON employee
REFERENCING NEW TABLE AS new_table
                OLD TABLE AS old_table
FOR EACH STATEMENT
BEGIN ATOMIC
      -- Example Action : Log the number of rows updated
      INSERT INTO update_log ( table_name , operation , row_count , update_time )
      VALUES ('employee ', 'UPDATE ', ( SELECT COUNT (*) FROM new_table ) ,
            CURRENT_TIMESTAMP );
END ;
/
```

- Trigger Timing: AFTER UPDATE ensures the trigger fires after any UPDATE operation on the employee table.

- Scope: FOR EACH STATEMENT means the trigger executes once per SQL statement, not per affected row.

- Action: Inserts a log entry into the update_log table, recording the number of rows updated and the time of the operation.
- Uses new_table to reference the set of new rows affected by the UPDATE.

Note: “BEGIN ATOMIC" ensures that all actions in the block are treated as one task—either every thing succeeds, or nothing happens. It acts as a safety lock, preventing partial changes and keeping the database consistent. In triggers, it’s essential to ensure all small changes during an event occur together or not at all.

2 Recursive Queries

Recursive queries allow SQL databases to handle hierarchical or transitive relationships, such as finding all prerequisite courses, even when the prerequisites span multiple levels.

2.1 Recursive Query Example

To illustrate recursive queries, consider the problem of finding all the prerequisites (direct or indi rect) for a course. The following example query defines a recursive view that generates the transitive closure of a prereq relation:

```
WITH RECURSIVE rec_prereq ( course_id , prereq_id ) AS (
        -- Part 1: Base case
        SELECT course_id , prereq_id
        FROM prereq

        UNION

        -- Part 2: Recursive case
        SELECT rec_prereq . course_id , prereq . prereq_id
        FROM rec_prereq , prereq
        WHERE rec_prereq . prereq_id = prereq . course_id
)
SELECT *
FROM rec_prereq ;
/
```

1. Base Case (First SELECT)

The first SELECT query serves as the starting point of the recursion. It selects all direct prerequisite relationships from the prereq table. This defines the initial content of the rec_prereq view.

```
SELECT course_id , prereq_id
FROM prereq
```

This query fetches all course_id and prereq_id pairs that represent direct prerequisites. These rows form the **base case** for the recursive view.

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Table 1: Initial Course Prerequisites

2. Recursive Case (Second SELECT)

The second SELECT defines the recursive part of the query. It uses the results from the base case to find indirect prerequisites by joining rec_prereq with the prereq table.

```
SELECT rec_prereq . course_id , prereq . prereq_id
FROM rec_prereq , prereq
WHERE rec_prereq . prereq_id = prereq . course_id
```

- It takes the results from the rec_prereq view (which contains both direct prerequisites from the base case and any previously discovered indirect prerequisites).
- Then, it checks if any of these prerequisites (rec_prereq.prereq_id) are themselves courses with prerequisites in the prereq table.
- If a match is found, it adds this indirect prerequisite relationship to rec_prereq.

This process continues until no new prerequisites can be found, at which point the recursion stops. This is called reaching the **fixed point**. The final output will include rows such as:

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-315	CS-101
CS-319	CS-101
CS-319	CS-190
CS-319	CS-315
CS-347	CS-101
CS-347	CS-190
CS-347	CS-315
CS-347	CS-319

Table 2: Recursive Course Prerequisites

3 Cursor

In response to any DML statement, the database creates a memory area, known as the context area, for processing an SQL statement, which contains all information needed for processing the state ment, for example, the number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by an SQL statement. The set of rows the cursor holds is referred to as the active set.
You can name a cursor so that it can be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors.
- Explicit cursors.

3.1 Implicit Cursor

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE, and DELETE) is issued, an implicit cursor is as sociated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be

affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT.

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE state ment affected one or more rows or a SELECT INTO state ment returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Other wise, it returns FALSE.
%ISOPEN	Always returns FALSE for implicit cursors, because Ora cle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

```
DECLARE
        total_rows number (2) ;
BEGIN
        UPDATE emp
        SET salary = salary + 500;
        IF sql % notfound THEN
                dbms_output . put_line (’no customers selected ’) ;
        ELSIF sql % found THEN
                total_rows := sql % rowcount ;
```



```
                dbms_output . put_line ( total_rows || ’ customers selected ’) ; END IF ;
END ;
/
```

Code 1: Implicit Cursor

3.2 Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT statement which returns one or more rows.

CURSOR cursor_name IS select_statement;

3.2.1 Steps for Cursors

1. Declaring the cursor for initializing in the memory
2. Opening the cursor for allocating memory

3. Fetching the cursor for retrieving data

4. Closing the cursor to release allocated memory

```
DECLARE
        c_id customers . id % type ;
        c_name customers . name % type ;
        c_addr customers . address % type ;
        CURSOR c_customers is
        SELECT id , name , address FROM customers ;
BEGIN
        OPEN c_customers ;
        LOOP
                FETCH c_customers into c_id , c_name , c_addr ;
                EXIT WHEN c_customers % notfound ;
                dbms_output . put_line ( c_id || ' ' || c_name || ' ' || c_addr ) ; END LOOP ;
        CLOSE c_customers ;
END ;
/
```

Code 2: Explicit Cursor

3.3 Cursor FOR Loop

The cursor FOR loop is an elegant and natural extension of the numeric FOR loop in PL/SQL. With a numeric FOR loop, the body of the loop executes once for every integer value between the low and high values specified in the range. With a cursor FOR loop, the body of the loop is executed for each row returned by the query.

```
FOR record_index IN cursor_name
LOOP
        {... statements ...}
END LOOP ;
```

```
CREATE OR REPLACE Function TotalIncome
( name_in IN varchar2 )
RETURN varchar2
IS
        total_val number (6) ;
```



```
        cursor c1 is
        SELECT monthly_income
        FROM employees
        WHERE name = name_in ;
BEGIN
        total_val := 0;
        FOR employee_rec IN c1
        LOOP
                total_val := total_val + employee_rec . monthly_income ;
        END LOOP ;
        RETURN total_val ;
END ;
```

3.3.1 Variables in Explicit Cursor Queries

An explicit cursor query can reference any variable in its scope. When you open an explicit

cursor, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

```
DECLARE
        sal employees . salary % TYPE ;
        sal_multiple employees . salary % TYPE ;
        factor INTEGER := 2;
        CURSOR c1 is
        SELECT salary , salary * factor FROM employees
        WHERE job_id LIKE 'AD_ %';
BEGIN
        OPEN c1 ; -- PL/SQL evaluates factor
        LOOP
                FETCH c1 INTO sal , sal_multiple ;
                EXIT WHEN c1 % NOTFOUND ;
                DBMS_OUTPUT . PUT_LINE ('factor = ' || factor ) ;
                DBMS_OUTPUT . PUT_LINE ('sal = ' || sal ) ;
                DBMS_OUTPUT . PUT_LINE (' sal_multiple = ' || sal_multiple ) ; factor :=
                factor + 1; -- Does not affect sal_multiple
        END LOOP ;
        CLOSE c1 ;
END ;
/

factor = 2
sal = 4451
sal_multiple = 8902
factor = 3
sal = 26460
sal_multiple = 52920
factor = 4
sal = 18742.5
sal_multiple = 37485
factor = 5
sal = 18742.5
sal_multiple = 37485
```

3.3.2 Explicit Cursors that Accept Parameters

You can create an explicit cursor that has formal parameters, and then pass different actual parame ters to the cursor each time you open it. In the cursor query, you can use a formal cursor parameter anywhere that you can use a constant. Outside the cursor query, you cannot reference formal cursor parameters.

Example:

```
DECLARE
        CURSOR c ( job VARCHAR2 , max_sal NUMBER ) IS
        SELECT last_name , first_name , ( salary - max_sal ) overpayment FROM
        employees
        WHERE job_id = job
        AND salary > max_sal
        ORDER BY salary ;
        PROCEDURE print_overpaid IS
                last_name_ employees . last_name % TYPE ;
                first_name_ employees . first_name % TYPE ;
                overpayment_ employees . salary % TYPE ;
        BEGIN
```

```
            LOOP
                    FETCH c INTO last_name_ , first_name_ , overpayment_ ;
                EXIT WHEN c % NOTFOUND ;
                DBMS_OUTPUT . PUT_LINE ( last_name_ || ', ' || first_name_ || ' (by ' ||
                 overpayment_ || ')') ;
            END LOOP ;
        END print_overpaid ;
BEGIN
        DBMS_OUTPUT . PUT_LINE (' ----------------------') ;
        DBMS_OUTPUT . PUT_LINE ('Overpaid Stock Clerks :') ;
        DBMS_OUTPUT . PUT_LINE (' ----------------------') ;
        OPEN c ('ST_CLERK ', 2500) ;
        print_overpaid ;
        CLOSE c ;
        DBMS_OUTPUT . PUT_LINE (' ------------------------------') ; DBMS_OUTPUT .
        PUT_LINE ('Overpaid Sales Representatives :') ; DBMS_OUTPUT . PUT_LINE
        (' ------------------------------') ; OPEN c ('SA_REP ', 10000) ;
        print_overpaid ;
        CLOSE c ;
END ;
/
```


4 Advanced Ranking Features

4.1 Ranking in SQL

Ranking functions in SQL allow you to assign a unique rank to each row within a result set based on specified criteria. These functions are typically used in conjunction with the ORDER BY clause to determine the order of ranking.

```
SELECT ID , RANK () OVER ( ORDER BY GPA DESC ) AS s_rank
FROM student_grades ;
```

This query assigns ranks to students based on their GPA in descending order.

4.2 Ranking Functions: RANK(), DENSE_RANK(), ROW_NUMBER()

- RANK(): Assigns the same rank to rows with identical values, creating gaps in the ranking sequence.

```
SELECT ID , GPA , RANK () OVER ( ORDER BY GPA DESC ) AS rank
FROM student_grades ;
```

- DENSE_RANK(): Similar to RANK(), but without gaps in the ranking sequence when there are ties.

```
SELECT ID , GPA , DENSE_RANK () OVER ( ORDER BY GPA DESC ) AS dense_rank
FROM student_grades ;
```

- ROW_NUMBER(): Assigns a unique sequential number to each row, regardless of ties.

```
SELECT ID , GPA , ROW_NUMBER () OVER ( ORDER BY GPA DESC ) AS row_num
FROM student_grades ;
```

ROW_NUMBER	GPA	RANK	DENSE_RANK
1	4.0	1	1
2	4.0	1	1
3	3.8	3	2

4.3 Partitioned Ranking

Ranks students within each department based on their GPA using window functions, which is more efficient than subqueries.

```
SELECT ID , dept_name , RANK () OVER ( PARTITION BY dept_name ORDER BY GPA DESC )
    AS dept_rank
FROM dept_grades
ORDER BY dept_name , dept_rank ;
```

ID	dept_name	GPA
1	Math	3.8
2	Science	3.5
3	Math	3.8
4	Science	3.2

ID	dept_name	dept_rank
1	Math	1
3	Math	1
2	Science	1
4	Science	2

Table 3: Sample InputTable 4: Sample Output 11

5 Windowing Functions

Windowing functions perform calculations across a set of table rows that are related to the current row. They are useful for tasks like calculating moving averages, running totals, and more.

5.1 3-Day Moving Average

Calculates a 3-day moving average of sales by summing the current day's value with the values of the preceding and following days.

```
SELECT date , SUM ( value ) OVER ( ORDER BY date ROWS BETWEEN 1 PRECEDING AND 1
    FOLLOWING ) AS moving_avg
FROM sales ;
```

date	value
2024-09-17	100
2024-09-18	150
2024-09-19	200
2024-09-20	250
2024-09-21	300

Table 5: Sample Input

date	moving_avg
2024-09-17	250
2024-09-18	450
2024-09-19	600
2024-09-20	750
2024-09-21	550

Table 6: Sample Output

5.2 Running Total by Account

Calculates a running total of transaction values for each account.

```
SELECT account_number , date_time ,
        SUM ( value ) OVER (
            PARTITION BY account_number
            ORDER BY date_time
            ROWS UNBOUNDED PRECEDING
        ) AS running_total
FROM transaction
ORDER BY account_number , date_time ;
```

ac_no	date_time	total
A001	2024-09-17 10:00:00	100
A001	2024-09-18 12:00:00	150
A002	2024-09-17 09:30:00	200
A001	2024-09-19 14:00:00	250
A002	2024-09-18 11:00:00	300

acc_no	date_time	run_total
A001	2024-09-17 10:00:00	100
A001	2024-09-18 12:00:00	250
A001	2024-09-19 14:00:00	500
A002	2024-09-17 09:30:00	200
A002	2024-09-18 11:00:00	500

Table 7: Sample Input Table 8: Sample Output 12

6 Tasks

Task 1

You have an employee (ID, Name, Salary, and Designation) table where salary is an attribute. Try to increase it by 10% for employees having the designation “manager” with salary <30000 and decrease it by 10% for “assistant manager” with salary>20000 and show how many rows got affected using an **implicit cursor.**

```
DECLARE
    rows_affected NUMBER := 0;
BEGIN
    UPDATE employees
    SET salary = salary * 1.1
    WHERE designation = 'manager' AND salary < 30000;

    rows_affected := SQL%ROWCOUNT;

    UPDATE employees
    SET salary = salary * 0.9
    WHERE designation = 'assistant manager' AND salary > 20000;

       rows_affected := rows_affected + SQL%ROWCOUNT;

       DBMS_OUTPUT.PUT_LINE('Total rows affected: ' || rows_affected);
END;
/
```

Task 2

Create a table of transactions (User_ID, Amount, T_Date) which stores all bank transactions of all the users in our hypothetical bank. Fill up the table with a few transactions of your choice. Create another table loan_type (Scheme, Installment_Number, Charge, Min_Trans). Loan_type will have the loan schemes as shown below. For simplicity, you can store the Scheme as a number, such as 1, 2, or 3 instead of “S-A/S-B/S-C”. Insert only those 3 specific rows into the table. Now, create a function that takes as input a User_ID, calculates his/her total transactions, and checks against the loan_type table **(use a cursor here)** to determine the correct present loan scheme for this person. Return and display the loan_scheme number.

Scheme	Installments	Service Charge	Eligibility
S-A	30	5%	Transactions ≥ 2000000
S-B	20	10%	Transactions ≥ 1000000
S-C	15	15%	Transactions ≥ 500000

```
DROP TABLE transactions;
DROP TABLE loan_type;

CREATE TABLE transactions (
    User_ID INT,
    Amount NUMBER(10, 2),
    T_Date DATE
);


CREATE TABLE loan_type (
    Scheme INT,
    Installment_Number INT,
    Charge NUMBER(5, 2),
    Min_Trans NUMBER(15, 2)
);


INSERT INTO transactions (User_ID, Amount, T_Date)
VALUES (101, 500000, TO_DATE('2023-01-01', 'YYYY-MM-DD'));

INSERT INTO transactions (User_ID, Amount, T_Date)
VALUES (102, 1200000, TO_DATE('2023-02-15', 'YYYY-MM-DD'));

INSERT INTO transactions (User_ID, Amount, T_Date)
VALUES (101, 1800000, TO_DATE('2023-03-10', 'YYYY-MM-DD'));
```

```sql
INSERT INTO transactions (User_ID, Amount, T_Date)
VALUES (103, 700000, TO_DATE('2023-04-20', 'YYYY-MM-DD'));

INSERT INTO loan_type (Scheme, Installment_Number, Charge, Min_Trans)
VALUES (1, 30, 5, 2000000);

INSERT INTO loan_type (Scheme, Installment_Number, Charge, Min_Trans)
VALUES (2, 20, 10, 1000000);

INSERT INTO loan_type (Scheme, Installment_Number, Charge, Min_Trans)
VALUES (3, 15, 15, 500000);




CREATE OR REPLACE FUNCTION get_loan_scheme(p_user_id IN NUMBER) RETURN NUMBER
IS
    total_transactions NUMBER := 0;
    loan_scheme NUMBER := 0;

    CURSOR c_loan_types IS
        SELECT scheme, min_trans
        FROM loan_type
        ORDER BY min_trans DESC;
BEGIN
    SELECT SUM(amount)
    INTO total_transactions
    FROM transactions
    WHERE user_id = p_user_id;

    FOR loan IN c_loan_types LOOP
        IF total_transactions >= loan.min_trans THEN
            loan_scheme := loan.scheme;
            EXIT;
        END IF;
    END LOOP;

    RETURN loan_scheme;
END;
/
```

Task 3

Consider the following relational schema that manages the telephone bills of a mobile phone com pany.

- CUSTOMER (SSN, Name, Surname, PhoneNum, Plan)
- PRICINGPLAN (Code, ConnectionFee, PricePerSecond)
- PHONECALL (SSN, Date, Time, CalledNum, Seconds)
- BILL (SSN, Month, Year, amount)

Task 3.1

Write a **trigger** that automatically initializes the Bill for a particular customer to 0 whenever a new customer is added to the system.

```
CREATE OR REPLACE TRIGGER new_customer_bill_initialize

AFTER INSERT ON customer

FOR EACH ROW

BEGIN

    INSERT INTO bill (SSN, Month, Year, amount)

    VALUES (:NEW.SSN, TO_CHAR(SYSDATE, 'MM'), TO_CHAR(SYSDATE, 'YYYY'), 0);

END;

/
```

Task 3.2

Write a **trigger** that updates the customers' bill based on their Pricing plan after each phone Call.

```
CREATE OR REPLACE TRIGGER update_bill

AFTER INSERT ON phonecall

FOR EACH ROW

DECLARE

    pps NUMBER;

    conn_fee NUMBER;

    call_cost NUMBER;
```

```
BEGIN

    SELECT p.PricePerSecond, p.ConnectionFee

    INTO pps, conn_fee

    FROM pricingplan p, customer c

    WHERE c.SSN = :NEW.SSN

      AND c.Plan = p.Code;



    call_cost := conn_fee + (pps * :NEW.Seconds);



    UPDATE bill

    SET amount = amount + call_cost

    WHERE SSN = :NEW.SSN

    AND Month = TO_NUMBER(TO_CHAR(SYSDATE, 'MM'))

    AND Year = TO_NUMBER(TO_CHAR(SYSDATE, 'YYYY'));

END;

/
```

Task 4

Task 4.1

IUT wants to create a database for all students. The student table consists of four rows: ID (var char2), Date of Admission (date), Department (char), Program (char), Section (char). Department,

Program and Section are 1 character strings with values ranging from ‘1’ to ‘6’.

Now create a function Gen_ID which generates an ID (a string) automatically in the following for mat:

YY00DPSXX

YY are the last two digits of the year, extracted from the Date of Admission. D is Dept number, P is Program number and S is Section Number each of 1 character. XX is an automatically generated number (you can use a sequence, like 01 for first entry, then 02 for next and so on)

that increments as you add new members.

Once you have made Gen_ID, create a **trigger** such that, every time you insert into the table Student with only the Date of Admission, Department, Program, and Section, the ID is not given as input and rather automatically generated by calling Gen_ID inside the trigger and then inserted automatically along with the other components.

```
-- Create the STUDENT table

CREATE TABLE STUDENT (

    ID VARCHAR2(10) PRIMARY KEY,

    DATE_OF_ADMISSION DATE,

    DEPARTMENT CHAR(1) CHECK (DEPARTMENT BETWEEN '1' AND '6'),

    PROGRAM CHAR(1) CHECK (PROGRAM BETWEEN '1' AND '6'),

    SECTION CHAR(1) CHECK (SECTION BETWEEN '1' AND '6')

);

-- Create sequence for ID generation

CREATE SEQUENCE STUDENT_SEQ

    START WITH 1

    INCREMENT BY 1

    NOCACHE

    NOCYCLE;

CREATE OR REPLACE FUNCTION Gen_ID(

    p_date_of_admission DATE,

    p_dept CHAR,

    p_prog CHAR,

    p_section CHAR

) RETURN VARCHAR2

IS
```

```
    v_year VARCHAR2(2);
    v_seq_num VARCHAR2(2);
    v_id VARCHAR2(10);
BEGIN
    -- Extract last two digits of the year from date
    v_year := TO_CHAR(p_date_of_admission, 'YY');

    -- Get next sequence number and format as 2 digits
    v_seq_num := LPAD(TO_CHAR(STUDENT_SEQ.NEXTVAL), 2, '0');

    -- Construct the ID in format YY00DPSXX
    v_id := v_year || '00' || p_dept || p_prog || p_section || v_seq_num;

    RETURN v_id;
END;
/
```

Task 4.2

Accounts (ID, Name, AccCode, Balance, LastDateofInterest) AccountProperties (ID, Name, intere stRate, GP)

- GP = 1 means daily
- GP = 2 means monthly
- GP = 3 means yearly

Using an explicit cursor, write a procedure that updates all account current balances adding interest if it satisfies the condition.

```
CREATE OR REPLACE PROCEDURE UpdateAccountBalances AS
```

```
    -- Cursor declaration
    CURSOR account_cursor IS
        SELECT a.ID, a.Balance, a.LastDateofInterest, p.interestRate, p.GP
        FROM Accounts a
        JOIN AccountProperties p ON a.AccCode = p.ID;

    -- Variables
    v_id Accounts.ID%TYPE;
    v_balance Accounts.Balance%TYPE;
    v_last_date Accounts.LastDateofInterest%TYPE;
    v_interest_rate AccountProperties.interestRate%TYPE;
    v_grace_period AccountProperties.GP%TYPE;
    v_days_passed NUMBER;
    v_interest NUMBER := 0;
    v_apply_interest BOOLEAN := FALSE;

BEGIN
    -- Open the cursor
    OPEN account_cursor;

    -- Loop through each account
    LOOP
        FETCH account_cursor INTO v_id, v_balance, v_last_date,
v_interest_rate, v_grace_period;
        EXIT WHEN account_cursor%NOTFOUND;

        -- Calculate days passed since last interest application
        v_days_passed := TRUNC(SYSDATE) - TRUNC(v_last_date);
```

```
    v_apply_interest := FALSE;

    -- Check if interest should be applied based on grace period
    CASE v_grace_period
        WHEN 1 THEN -- Daily
            IF v_days_passed >= 1 THEN
                v_apply_interest := TRUE;
            END IF;
        WHEN 2 THEN -- Monthly
            IF MONTHS_BETWEEN(SYSDATE, v_last_date) >= 1 THEN
                v_apply_interest := TRUE;
            END IF;
        WHEN 3 THEN -- Yearly
            IF MONTHS_BETWEEN(SYSDATE, v_last_date) >= 12 THEN
                v_apply_interest := TRUE;
            END IF;
    END CASE;

    -- Apply interest if conditions are met
    IF v_apply_interest THEN
        -- Calculate interest based on balance and rate
        v_interest := v_balance * (v_interest_rate / 100);

        -- Update account with new balance and reset last interest date
        UPDATE Accounts
        SET Balance = Balance + v_interest,
            LastDateofInterest = SYSDATE
```

```
            WHERE ID = v_id;


            DBMS_OUTPUT.PUT_LINE('Updated Account ID: ' || v_id ||
                                 ' - Added interest: ' || v_interest ||
                                 ' - New balance: ' || (v_balance +
v_interest));
        END IF;
    END LOOP;


    -- Close cursor
    CLOSE account_cursor;


    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Account balance update completed successfully.');
EXCEPTION
    WHEN OTHERS THEN
        -- Error handling
        IF account_cursor%ISOPEN THEN
            CLOSE account_cursor;
        END IF;
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error updating account balances: ' || SQLERRM);
END UpdateAccountBalances;
/
```

Submission Guidelines

You are required to submit a report that includes your code, a detailed explanation, and the corre sponding output. Rename your report as <StudentID_Lab_1.pdf>.

• Your lab report must be generated in LaTeX. Recommended tool: Overleaf. You can use some of the available templates in Overleaf.

• Include all code/pseudo code relevant to the lab tasks in the lab report.

• Please provide citations for any claims that are not self-explanatory or commonly

understood. • It is recommended to use vector graphics (e.g., .pdf, .svg) for all visualizations.

• In cases of high similarities between two reports, both reports will be discarded.