Database Management System Lab(CS 29006)

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

School of Computer Engineering

1 Credit

Course Committee

Lab Contents

Sr #	Major and Detailed Coverage Area	Lab#
1	PL/SQL Programming Language ❑ Exception ❑ Cursor ❑ Trigger	10

Exception

An error condition during a program execution is called an **exception** in PL/SQL. PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
     exception1-handling-statements
  /*continuation of program */
  WHEN exception2  THEN
     exception2-handling-statements
  ........
  WHEN others THEN
     exception-handling-statements
END;
```

Exception Example


```
DECLARE
  c_id customer.id%type := 8;
  c_name  customer.name%type;
  c_addr customer.address%type;
BEGIN
  SELECT  name, address INTO  c_name, c_addr FROM customer WHERE id = c_id;
  DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

Since there is no customer with ID value 8 in the table, the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

Customer

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

System-defined Exception

These exception are pre-defined and are automatically raised by Oracle whenever an exception is encountered. Each exception is assigned a unique number and a name.

Error Name	Error No	Description
ACCESS_INTO_NULL	ORA-06530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	ORA-06592	It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
ZERO_DIVIDE	ORA-01476	It is raised when an attempt is made to divide a number by zero.
TOO_MANY_ROWS	ORA-01422	It is raised when s SELECT INTO statement returns more than one row.
INVALID_NUMBER	ORA-01722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
NO_DATA_FOUND	ORA-01403	It is raised when a SELECT INTO statement returns no rows.
PROGRAM_ERROR	ORA-06504	It is raised when PL/SQL has an internal problem.

System-defined Exception Cont...

Error Name	Error No	Description
CURSOR_ALREADY_OPEN	ORA-06511	A program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers, so your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	ORA-00001	A program attempts to store duplicate values in a column that is constrained by a unique index.
VALUE_ERROR	ORA-06502	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)

System-defined Exception Example


```
DECLARE
  stock_price NUMBER := 9.73;
  net_earnings NUMBER := 0;
  pe_ratio NUMBER;
BEGIN
  pe_ratio := stock_price / net_earnings; -- Calculation might cause division-by-zero error.
  DBMS_OUTPUT.PUT_LINE('Price/earnings ratio = ' || pe_ratio);
EXCEPTION  -- exception handlers begin
-- Only one of the WHEN blocks is executed.
  WHEN ZERO_DIVIDE THEN  -- handles 'division by zero' error
    DBMS_OUTPUT.PUT_LINE('Company must have had zero earnings.');
    pe_ratio := NULL;
  WHEN OTHERS THEN  -- handles all other errors
    DBMS_OUTPUT.PUT_LINE('Some other kind of error occurred.');
    pe_ratio := NULL;
END;  -- exception handlers and block end here
/
```

Raising Exceptions

Exceptions are raised by the database automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE.** Following is the simple syntax of raising an exception:

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

User-defined Exception

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using RAISE statement.

Example

```
DECLARE
  c_id customer.id%type := &cc_id;
  c_name  customer.name%type;
  c_addr customer.address%type;
  -- user defined exception
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
     SELECT   name, address INTO   c_name, c_addr FROM customer WHERE id =
c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;
/*Continuation of program */
EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

Assigning name and error number to user-defined exception

A user-defined exception can be assigned a name and an error number by using PRAGMA pre-compiler directive. This directive binds the specified error number to a user-defined exception name. You can use more than one PRAGMA EXCEPTION_INIT directives. The syntax is:

exceptionname EXCEPTION;

PRAGMA EXCEPTION_INIT(exceptionname, errorcode);

Example

```
DECLARE
  vcomm Employee.comm%TYPE; veno Employee.empno%TYPE;
  Invalid_comm EXCEPTION;
  PRAGMA EXCEPTION_INIT(Invalid_comm, -20000);
BEGIN
   veno: =&veno;
   SELECT comm INTO vcomm FROM Employee WHERE empno=veno;
   IF vcomm<0 THEN
      RAISE Invalid_comm;
   ELSE
       DBMS_OUTPUT.PUT_LINE(vcomm);
END IF;
/*Continuation of program */
EXCEPTION
   WHEN Invalid_comm THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM||' '||'Negative commission);
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('No such id');
END;
/
```

Guidelines for Avoiding and Handling PL/SQL Errors and Exceptions

Because reliability is crucial for database programs, use both error checking and exception handling to ensure your program can handle all possibilities:

- Add exception handlers whenever there is any possibility of an error occurring. Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors could also occur at other times, for example if a hardware failure with disk storage or memory causes a problem that has nothing to do with your code; but your code still needs to take corrective action.
- Add error-checking code whenever you can predict that an error might occur if your code gets bad input data. Expect that at some time, your code will be passed incorrect or null parameters, that your queries will return no rows or more rows than you expect.
- Carefully consider whether each exception handler should commit the transaction, roll it back, or let it continue. Remember, no matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data.
- Test your code with different combinations of bad data to see what potential errors arise.

Cursor

A cursor is a temporary work area created in the system memory when a SQL statement is executed. This temporary work area known as the **context area** and is used to store the data retrieved from the database and manipulate this data. **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 a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

There are two types of cursors in PL/SQL:

- **Implicit cursors**: These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.
- **Explicit cursors:** They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as **current row**. When you fetch a row the current row position moves to next row.

Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

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 associated 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 attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement

Implicit Cursor Cont...

The following table provides the description of the most used attributes –

Sr #	Attribute and Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%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. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Implicit Cursor Example


```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE Customer
  SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('No Customer Updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' Customer updated');
  END IF;
END;
/
```

Customer

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

Explicit Cursor

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 more than one row.

Working with an explicit cursor includes the following steps –

1. Declaring the cursor for initializing the memory
2. Opening the cursor for allocating the memory
3. Fetching the cursor for retrieving the data
4. Closing the cursor to release the allocated memory

1. Declaring the cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
CURSOR Customer_C IS
  SELECT id, name, address FROM Customer;
```

Explicit Cursor

2. Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows –

OPEN Customer_C;

3. Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows –

FETCH Customer_C INTO c_id, c_name, c_addr;

4. Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows –

CLOSE Customer_C;

Explicit Cursor Example


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

Declaring the

Opening the cursor

Fetching the cursor

Closing the cursor

Explicit Cursor For Loop

In cursor FOR loop, you do not have to explicitly open and close the cursor. It is automatically done by FOR loop. **Syntax**:

FOR variable IN cursorname

LOOP

Statements;

END LOOP

Example:

```
DECLARE
   CURSOR EMP_C IS SELECT ID, SALARY FROM EMP WHERE DEPT=30;
BEGIN
  FOR i IN EMP_C
    LOOP
      UPDATE EMP SET salary=i.salary*1.05 WHERE ID=i.ID;
      INSERT INTO Emp_raise VALUES(i.ID, SYSDATE, i.SALARY *0.05);
   END LOOP;
   COMMIT;
END;
/
```

Parameterized Cursors

Parameters in cursors are useful when a cursor is required to be opened based on different set of parameter values. The parameter makes the cursor more reusable. A cursor with parameter can be opened and closed several times. Each time a new active set is loaded in the memory and the pointer is placed at first.

Syntax: CURSOR cursorname (parametername type, parametername type) IS SELECT statement;
Example –

```
DECLARE
   vname EMPLOYEE.name%TYPE;
   vdesg EMPLOYEE.designation%TYPE;
   did NUMBER(2);
   CURSOR EMPLOYEE_C(deptno EMPLOYEE.dno%TYPE) IS SELECT name, designation FROM
   EMPLOYEE   WHERE dno=deptno;
BEGIN
    did: =&did;
   OPEN EMPLOYEE_C (did);
   /* Some Operation */
   CLOSE EMPLOYEE_C;
END;
/
```

Cursor Disadvantages

1. Uses more resources because each time you fetch a row from the cursor, it results in a network roundtrip
2. Because of the round trips, performance and speed is slow.
3. If the cursor is not properly closed, the resources will not be freed until the SQL session itself is closed.
4. It is returned only one row at a time.

How cursors can be avoided?

1. **Using the SQL while loop:** Using a while loop we can insert the result set into the temporary table.
2. **User defined functions:** Cursors are sometimes used to perform some calculation on the resultant row set. This cam also be achieved by creating a user defined function to suit the needs.

Trigger

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)
- A database definition (DDL) statement (CREATE, ALTER, or DROP)
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP or SHUTDOWN)

Triggers can be defined on the table, view, schema, or database with which the event is associated. It is written for the following purposes:

- Generating some derived column values automatically.
- Enforcing referential integrity.
- Event logging and storing information on table access.
- Auditing.
- Synchronous replication of tables.
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is

CREATE [OR REPLACE] TRIGGER trigger_name // Creates or replaces an existing trigger with the trigger_name.

{BEFORE | AFTER | INSTEAD OF } // specifies when the trigger will be executed. INSTEAD OF is used for a view.

{INSERT [OR] | UPDATE [OR] | DELETE} // specifies the DML operation.

[OF col_name] // specifies the column name that will be updated.

ON table_name // specifies the name of the table associated with the trigger.

[REFERENCING OLD AS o NEW AS n] // refer new and old values for various DML statements.

[FOR EACH ROW] // specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

WHEN (condition) // provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

```
DECLARE
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

Creating Triggers cont...

SELECT * FROM CUSTOMER;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

Now the task is to create a row-level trigger for the CUSTOMER table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMER table. The trigger to display the salary difference between the old values and new values.

Creating Triggers cont...


```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON CUSTOMER
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

Triggering a Trigger

Now some DML operations on the CUSTOMER table is performed.

Here is one **INSERT** statement, which will create a new record in the table:

```
INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY) VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMER table, **display_salary_changes** will be fired and it will display the following result:

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null.

Triggering a Trigger cont...

Let's perform one more DML operation on the CUSTOMER table.

The UPDATE statement will update an existing record in the table i.e.,

```
UPDATE CUSTOMER
SET salary = salary + 500
WHERE ID = 2;
```

When a record is updated in the CUSTOMER table, **display_salary_changes** will be fired and it will display the following result:

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

More operations on Trigger

DROP:

Once a trigger is created, one might find that it needs to be removed. So, **DROP TRIGGER** statement is used to drop the trigger.

Syntax: DROP TRIGGER trigger_name; where trigger_name is the name of the trigger that needs to be dropped.

Example: DROP TRIGGER display_salary_changes;

DISABLE:

Once a trigger is created, one might find to disable it. So, it can be done with the **ALTER TRIGGER** statement.

Syntax: ALTER TRIGGER trigger_name **DISABLE**; where trigger_name is the name of the trigger that needs to be disabled.

Example: **ALTER TRIGGER** display_salary_changes **DISABLE;**

To disable all triggers, the syntax is **ALTER TABLE** table_name **DISABLE ALL TRIGGERS**; where table_name is the name of the table that all triggers should be disabled.

More operations on Trigger

ENABLE:

Once a trigger is disabled, one might find to enable it. So, it can be done with the **ALTER TRIGGER** statement.

Syntax: ALTER TRIGGER trigger_name **ENABLE**; where trigger_name is the name of the trigger that needs to be disabled.

Example: **ALTER TRIGGER** display_salary_changes **ENABLE;**

To enable all triggers, the syntax is **ALTER TABLE** table_name **ENABLE ALL TRIGGERS**; where table_name is the name of the table that all triggers should be enabled.

Thank You
End of Lab 10

Experiments

Reference Tables

- **EMPLOYEE** table with the attributes:

  ID, LAST_NAME, FIRST_NAME, MIDDLE_NAME, FATHER_NAME, MOTHER_NAME, SEX, HIRE_DATE, ADDRESS, CITY, STATE, ZIP, PHONE, PAGER, SUPERVISOR_ID, DOB, INJECTED_DATE

- **SCHOOL** table with the attributes:

  ID, NAME, INJECTED_DATE

- **EMPLOYEE _ALIGNMENT** table with the attributes:

  EMPLOYEE_ID, SCHOOL_ID, INJECTED_DATE

- **JOB** table with the attributes:

  ID, NAME, TITLE, SALARY, BONUS , INJECTED_DATE

- **EMPLOYEE _PAY** table with the attributes:

  EMPLOYEE_ID, JOB_ID, INJECTED_DATE

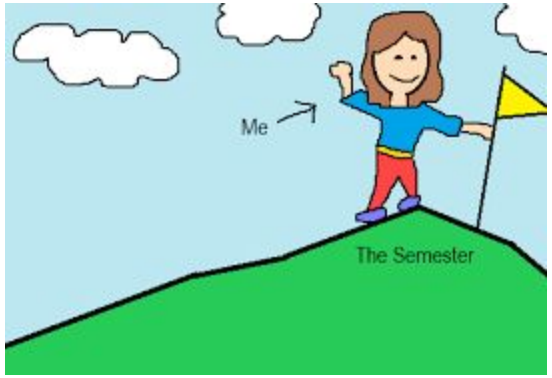
Experiments

1. Write PL/SQL block that asks user to input first number, second number and an arithmetic operator (+, -, *, or /). If operator is invalid, throw and handle a user defined exception. If second number is 0 and the operator is /, handle ZERO_DIVIDE predefined server exception.
2. Write a PL/SQL block that retrieves entire EMPLOYEE table into a cursor. Then, ask user to input a Employee Id to search. If employee exists then print its information, but if employee does not exist throw a user-defined exception and handle it with a message 'Employee does not exist'.
3. Create a PL/SQL block to increase salary of employees in school of Computer Engineering. The salary increase is 15% for the employees making less than 100K and 10% for the employees making 100K or more.
4. Create a PL/SQL block to declare a cursor to select last name, first name, salary and hire date of the employee. Retrieve each row from the cursor and print the employee's information if the employee's salary is greater than 50000 and the hire date is before 31-APR-2021.
5. Create a PL/SQL block to declare a cursor and retrieve each row from the cursor to display employee information, drawing more than the average salary in "Professor" rank.

Experiments

6. Develop BEFORE INSERT trigger on EMPLOYEE.
7. Develop AFTER INSERT trigger on SCHOOL.
8. Develop BEFORE UPDATE trigger on EMPLOYEE.
9. Develop AFTER UPDATE trigger on SCHOOL.
10. Develop BEFORE DELETE and AFTER DELETE trigger on EMPLOYEE and SCHOOL.

Thank You
End of the Lab




HELLO.
EXAMS