Unit 5
Procedures, Functions,Packages and Triggers

Procedures and Functions

- Procedures and functions:
  - Normally stored in the database within package specifications – a package is a sort of wrapper for a group of named blocks.
  - Can be stored as individual database objects.
  - Are parsed and compiled at the time they are stored.
  - Compiled objects execute faster than nonprocedural SQL scripts because nonprocedural scripts require extra time for compilation.

Procedures

- Procedures are named PL/SQL blocks.
- Created/owned by a particular schema
- Privilege to execute a specific procedure can be granted to or revoked from application users in order to control data access.
- Requires CREATE PROCEDURE (to create in your schema) or CREATE ANY PROCEDURE privilege (to create in other schemas).

CREATE PROCEDURE Syntax

```
CREATE [OR REPLACE] PROCEDURE <procedure_name> (<parameter1_name> <mode>
 <data type>, <parameter2_name> <mode> <data type>, ...) {AS|IS}
    <Variable declarations>
BEGIN
    Executable statements
[EXCEPTION
    Exception handlers]
END <optional procedure name>;
```

- Unique procedure name is required.
- OR REPLACE clause facilitates testing.
- Parameters are optional – enclosed in parentheses when used.
- AS or IS keyword is used – both work identically.
- Procedure variables are declared prior to the BEGIN keyword.
- DECLARE keyword is NOT used in named procedure.

Parameters

- Both procedures and functions can take parameters.
- Values passed as parameters to a procedure as arguments in a calling statement are termed *actual parameters*.
- The parameters in a procedure declaration are called *formal parameters*.
- The values stored in actual parameters are values passed to the formal parameters – the formal parameters are like placeholders to store the incoming values.
- When a procedure completes, the actual parameters are assigned the values of the formal parameters.
- A formal parameter can have one of three possible modes: (1) IN, (2), OUT, or (3) IN OUT.

Defining the IN, OUT, and IN OUT Parameter Modes

- **IN** – this parameter type is passed to a procedure as a read-only value that cannot be changed within the procedure – this is the default mode.
- **OUT** – this parameter type is write-only, and can only appear on the left side of an assignment statement in the procedure – it is assigned an initial value of NULL.
- **IN OUT** – this parameter type combines both IN and OUT; a parameter of this mode is passed to a procedure, and its value can be changed within the procedure.
- If a procedure raises an exception, the formal parameter values are not copied back to their corresponding actual parameters.

Procedure to find square of a given number

```
CREATE OR REPLACE PROCEDURE squareNum(x IN number ,square out number) IS
BEGIN
  square := x * x;
END;
/
```

To compile the procedure:

```
SQL> start F:\advdbms\2020\lab\program\procedure\proc_sqr.sql;
                              OR
SQL> @ F:\advdbms\2020\lab\program\procedure\proc_sqr.sql;

Procedure created.
```

Showing errors

```
 Warning: Procedure created with compilation errors.
```

SQL> select * from user_errors;

```
NAME          TYPE    SEQUENCE      LINE   POSITION TEXT ATTRIBUTE MESSAGE_NUMBER
--------- --------------
SQUARENUM  PROCEDURE       1          7          1
PLS-00103: Encountered the symbol ";"
ERROR                 103
```

SQL> show errors;

```
Errors for PROCEDURE SQUARENUM:

LINE/COL ERROR
-------- -----------------------------------------------------------
7/1      PLS-00103: Encountered the symbol ";"
```

To execute Procedure

- Call procedure in a PL/SQL Program

```
DECLARE
sq number:=0;
x number:=&x;
BEGIN
  squareNum(x,sq);
  dbms_output.put_line(' Square is:'||sq);
END;
/
```

```
SQL> start F:\advdbms\2020\lab\program\procedure\call_sqr.sql;
Enter value for x: 4
old   3: x number:=&x;
new   3: x number:=4;
Square is:16

PL/SQL procedure successfully completed.
```

To execute Procedure

- **Use Exec/Execute**

```
SQL> var sqr number;
SQL> exec squarenum(8,:sqr);
PL/SQL procedure successfully completed.

SQL> print sqr;

       SQR
----------
        64
```

Procedure with No Parameters

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE DisplaySalary IS
 temp_Salary NUMBER(10,2);
 temp_name emp.ename%type;
BEGIN
     SELECT Sal,ename INTO temp_Salary,temp_name FROM emp WHERE
  empno=102;
     DBMS_OUTPUT.PUT_LINE ('Salary of '||temp_name||' is
  '||temp_salary);
END;
/
```

Executing *DisplaySalary* Procedure

```
SQL> start F:\advdbms\2020\lab\program\procedure\proc_dissal.sql;

Procedure created.

SQL> execute displaysalary;
Salary of Ramesh is 35000

PL/SQL procedure successfully completed.
```

Passing IN and OUT Parameters

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE displaysalary1(p_eno in number,p_sal
 out number) IS
BEGIN
     SELECT Sal INTO p_sal FROM emp WHERE empno=p_eno;
EXCEPTION
     WHEN NO_DATA_FOUND THEN
          DBMS_OUTPUT.PUT_LINE ('Employee not found.');
END displaysalary1;
/
```

```
SQL> start F:\advdbms\2020\lab\program\procedure\proc_dissal1.sql;

Procedure created
```

Calling *DisplaySalary1*

```
DECLARE
v_sal emp.sal%type;
v_eno emp.empno%type:=&v_eno;
BEGIN
 displaysalary1(v_eno,v_sal);
 dbms_output.put_line('Actual salary of employee '||v_eno||' is
 '||v_sal);
END;
/
```

Executing

```
SQL> start F:\advdbms\2020\lab\program\procedure\call_dissal.sql;
Enter value for v_eno: 101
old   3: v_eno emp.empno%type:=&v_eno;
new   3: v_eno emp.empno%type:=101;
Actual salary of employee 101 is 30000

PL/SQL procedure successfully completed.

SQL> start F:\advdbms\2020\lab\program\procedure\call_dissal.sql;
Enter value for v_eno: 102
old   3: v_eno emp.empno%type:=&v_eno;
new   3: v_eno emp.empno%type:=102;
Actual salary of employee 102 is 35000
```

Executing using bind variables

```
SQL> var v_sal number;
SQL> execute displaysalary1(103,:v_sal);
PL/SQL procedure successfully completed.
SQL> print v_sal;

     V_SAL
----------
     55000
```

```
SQL> var v_sal number;
SQL> execute displaysalary1(100,:v_sal);
Employee not found.

PL/SQL procedure successfully completed.
```

Cursor in Procedure

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE displaysalary2(p_dno in varchar2) IS
cursor cur_dis is select ename,sal from emp where deptno=p_dno;
BEGIN
for i in cur_dis
loop
dbms_output.put_line(i.ename||' earns '||i.sal);
end loop;
END displaysalary2;
/
```

Executing

```
SQL> start F:\advdbms\2020\lab\program\procedure\proc_dissal2.sql;

Procedure created.

SQL> execute displaysalary2('D1');
Sona earns 55000
Tina earns 25000

PL/SQL procedure successfully completed.
```

Dropping a Procedure

- The SQL statement to drop a procedure is the straight-forward DROP PROCEDURE <procedureName> command.
- This is a data definition language (DDL) command, and so an implicit commit executes prior to and immediately after the command.

```
SQL> DROP PROCEDURE DisplaySalary2;
Procedure dropped.
```

- Write a procedure to calculate simple interest, taking principle, rate and year as inputs.
- Write a procedure to take department name as input to display project being handled by the department and name of the employees working under those projects belonging to the department.

Create Function Syntax

- Like a procedure, a function can accept multiple parameters, and the data type of the return value must be declared in the header of the function.

```
CREATE [OR REPLACE] FUNCTION <function_name>
  (<parameter1_name> <mode> <data type>,
    <parameter2_name> <mode> <data type>, ...)
RETURN <function return value data type> {AS|IS}
    <Variable declarations>
BEGIN
    Executable Commands
    RETURN (return_value);
    . . .
[EXCEPTION
    Exception handlers]
END;
```

- The general syntax of the RETURN statement is:

```
RETURN <expression>;
```

Example1-To retrieve salary

```
SET SERVEROUTPUT ON
CREATE OR REPLACE function Display_Salary(v_eno in  number)
Return number IS
 temp_Salary NUMBER(10,2);
BEGIN
     SELECT Sal INTO temp_Salary FROM emp WHERE empno=v_eno;
     return temp_salary;
END;
/
```

Executing Functions

PL/SQL Block

```
DECLARE
v_sal emp.sal%type;
v_eno emp.empno%type:=&v_eno;
BEGIN
 v_sal:=display_salary(v_eno);
 dbms_output.put_line('Salary of '||v_eno||' is '||v_sal);
 END;
/
```

Executing Functions

PL/SQL Block

```
DECLARE
v_sal emp.sal%type;
v_eno emp.empno%type:=&v_eno;
BEGIN
 v_sal:=display_salary(v_eno);
 dbms_output.put_line('Salary of '||v_eno||' is '||v_sal);
 END;
/
```

```
SQL> start F:\advdbms\2020\lab\program\procedure\call_func.sql;
Enter value for v_eno: 103
old   3: v_eno emp.empno%type:=&v_eno;
new   3: v_eno emp.empno%type:=103;
Salary of 103 is 55000
```

Executing Functions

SELECT

```
SQL> select display_salary(103) from dual;

DISPLAY_SALARY(103)
-------------------
              55000
```

Executing Functions

EXECUTE/EXEC

```
SQL> var v_sal number;
SQL> exec :v_sal:=display_salary(103);

PL/SQL procedure successfully completed.

SQL> print v_sal;
     V_SAL
----------
     55000
```

Dropping a Function

- As with the DROP PROCEDURE statement, the DROP FUNCTION <functionName> is also straight-forward.
- As with DROP PROCEDURE, the DROP FUNCTION statement is a DDL command that causes execution of an implicit commit prior to and immediately after the command.

```
SQL> DROP FUNCTION FullName;
Function dropped.
```

Write a function to display the name of the employee drawing less salary than the average salary of any given department

```
--function to display the name of employee earning less salary
SET SERVEROUTPUT ON
CREATE OR REPLACE function Display_ename(d_no in  varchar2)
Return varchar2 IS
 v_ename emp.ename%type;
BEGIN
    SELECT ename INTO v_ename FROM emp WHERE sal<(select avg(sal) from emp
where deptno=d_no);
    return v_ename;
END;
/
```

Note:You can write this program using cursor

```
SQL> start F:\advdbms\2020\lab\program\procedure\func_dispename.sql;

Function created.

SQL> select * from emp;

     EMPNO ENAME                  SAL DEPT NO
---------- --------------- ---------- --
       101 Ravi                 30000  D2
       102 Ramesh               35000  D2
       103 Sona                 55000  D1
       104 Tina                 25000  D1
       105 Bindu                35000  D3
       106 Bahabur              35000  D4

6 rows selected.
```

```
SQL> select display_ename('D2') from dual;

DISPLAY_ENAME('D2')
--------------------------------------------------------------------------------

Ravi

SQL> select display_ename('D1') from dual;

DISPLAY_ENAME('D1')
--------------------------------------------------------------------------------

Tina
```

Function Vs Procedure

Stored Procedure	**Function**
May or may not returns a value to the calling part of program.	Returns a value to the calling part of the program.
Uses IN, OUT, IN OUT parameter.	Uses only IN parameter.
Returns a value using “ OUT” parameter.	Returns a value using “RETURN”.
Does not specify the datatype of the value if it is going to return after a calling made to it.	Necessarily specifies the datatype of the value which it is going to return after a calling made to it.
Cannot be called from the function block of code.	Can be called from the procedure block of code.

PACKAGES

- A *package* is a collection of PL/SQL objects grouped together under one package name.
- Packages provide a means to collect related procedures, functions, cursors, declarations, types, and variables into a single, named database object that is more flexible than the related database objects are by themselves.
- Package variables – can be referenced in any procedure, function, (other object) defined within a package.

Package Specification and Scope

- A package consists of a package specification and a package body.
  - The *package specification*, also called the package header.
  - Declares global variables, cursors, exceptions, procedures, and functions that can be called or accessed by other program units.
  - A package specification must be a uniquely named database object.
  - Elements of a package can declared in any order. If element "A" is referenced by another element, then element "A" must be declared before it is referenced by another element. For example, a variable referenced by a cursor must be declared before it is used by the cursor.
- Declarations of subprograms must be forward declarations.
  - This means the declaration only includes the subprogram name and arguments, but does not include the actual program code.

Create Package Syntax

- Basically, a package is a named declaration section.
  - Any object that can be declared in a PL/SQL block can be declared in a package.
  - Use the CREATE OR REPLACE PACKAGE clause.
  - Include the specification of each named PL/SQL block header that will be public within the package.
  - Procedures, functions, cursors, and variables that are declared in the package specification are *global*.
- The basic syntax for a package is:

```
CREATE [OR REPLACE PACKAGE[ <package name> {AS|IS}
    <variable declarations>;
    <cursor declarations>;
    <procedure and function declarations>;
END <package name>;
```

Declaring Procedures and Functions within a Package

- To declare a procedure in a package – specify the procedure name, followed by the parameters and variable types:

```
PROCEDURE <procedure_name> (param1 param1datatype,
    param2 param2datatype, ...);
```

- To declare a function in a package, you must specify the function name, parameters and return variable type:

```
FUNCTION <function_name> (param1 param1datatype,
    param2 param2datatype, ...)
RETURN <return data type>;
```

Package Body

- Contains the code for the subprograms and other constructs, such as exceptions, declared in the package specification.
- Is optional – a package that contains only variable declarations, cursors, and the like, but no procedure or function declarations does not require a package body.
- Any subprograms declared in a package must be coded completely in the package body. The procedure and function specifications of the package body must match the package declarations including subprogram names, parameter names, and parameter modes.

Create Package Body Syntax

- Use the CREATE OR REPLACE PACKAGE BODY clause to create a package body. The basic syntax is:

```
CREATE [OR REPLACE] PACKAGE BODY <package
 name> AS
    <cursor specifications>
    <subprogram specifications and code>
END <package name>;
```

```
CREATE OR REPLACE PACKAGE Calculate1 IS



FUNCTION squrs1(Num1 IN NUMBER) return Number;
PROCEDURE Cubes1(Num1 IN NUMBER,Num2 OUT NUMBER);


END Calculate1;
/
```

```
CREATE OR REPLACE PACKAGE BODY Calculate1 IS
FUNCTION squrs1(Num1 IN NUMBER) return Number IS
RESULT NUMBER(3);


BEGIN
RESULT:= NUM1*NUM1;
RETURN( RESULT);
END squrs1;

PROCEDURE Cubes1(Num1 IN NUMBER,Num2 OUT NUMBER) IS

BEGIN
NUM2:= NUM1*NUM1*NUM1;
END Cubes1;
END Calculate1;
/
```

```
Set Serveroutput on;
DECLARE

M1 NUMBER(3);

RESULT NUMBER(4,1);

BEGIN
M1:=& M1;

Result:=Calculate1.squrs1 (M1); -- Function CALL
DBMS_OUTPUT.PUT_LINE ('SQuare IS '|| Result);
Calculate1.Cubes1(M1,Result);
DBMS_OUTPUT.PUT_LINE ('Cube IS '|| Result);
END;
/
```

Example Package

```
CREATE OR REPLACE PACKAGE ManageEmployee AS
    PROCEDURE FindEmployee(
      emp_ID        IN employee.EmployeeID%TYPE,
      emp_FirstName OUT employee.FirstName%TYPE,

      emp_LastName  OUT employee.LastName%TYPE);
       e_EmployeeIDNotFound EXCEPTION;

FUNCTION GoodIdentifier(
     emp_ID        IN employee.EmployeeID%TYPE)
        RETURN BOOLEAN;
END ManageEmployee;
/
```

Package Body

```
CREATE OR REPLACE PACKAGE BODY ManageEmployee AS
    -- Procedure to find employees
    PROCEDURE FindEmployee(
        emp_ID        IN employee.EmployeeID%TYPE,
        emp_FirstName OUT employee.FirstName%TYPE,
        emp_LastName  OUT employee.LastName%TYPE ) AS
    BEGIN
        SELECT FirstName, LastName
        INTO emp_FirstName, emp_LastName
        FROM Employee
        WHERE EmployeeID = emp_ID;

        -- Check for existence of employee
        IF SQL%ROWCOUNT = 0 THEN
            RAISE e_EmployeeIDNotFound;
        END IF;
    END FindEmployee;
```

Example 13.13 (2 of 2) – Package Body

```
FUNCTION GoodIdentifier(
        emp_ID         IN employee.EmployeeID%TYPE)
        RETURN BOOLEAN
    IS
        v_ID_Count NUMBER;
    BEGIN
        SELECT COUNT(*) INTO v_ID_Count
        FROM Employee
        WHERE EmployeeID = emp_ID;

        -- return TRUE if v_ID_COUNT is 1
        RETURN (1 = v_ID_Count);
    EXCEPTION
        WHEN OTHERS THEN
            RETURN FALSE;
    END GoodIdentifier;
END ManageEmployee;
```

Calling Package Procedure

```
/* PL SQL */
DECLARE
    v_FirstName   employee.FirstName%TYPE;
    v_LastName    employee.LastName%TYPE;
    search_ID     employee.EmployeeID%TYPE;
BEGIN
    ManageEmployee.FindEmployee (&search_ID, v_FirstName,
        v_LastName);
    DBMS_OUTPUT.PUT_LINE ('The employee name is: ' ||
        v_LastName || ', ' || v_FirstName);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('Cannot find an employee
  with that ID.');
END;
/
```

Results of Calling Package Procedure

- When the employee identifier is valid, the code displays the employee name as shown here.

```
Enter value for search_id: '01885'
The employee name is: Bock, Douglas
PL/SQL procedure successfully completed.
```

- When the identifier is not valid, the exception raised within the called procedure is propagated back to the calling procedure and is trapped by the EXCEPTION section’s WHEN OTHERS clause and an appropriate message is displayed as shown here.

```
Enter value for search_id: '99999'
Cannot find an employee with that ID.
PL/SQL procedure successfully completed.
```

DATABASE TRIGGERS

- Database trigger – a stored PL/SQL program unit that is associated with a specific database table, or with certain view types – can also be associated with a system event such as database startup.
- Two sections:
  - A named database event
  - A PL/SQL block that will execute when the event occurs
- Triggers execute (fire) automatically for specified SQL DML operations – INSERT, UPDATE, or DELETE affecting one or more rows of a table.

DATABASE TRIGGERS -tasks

- Database triggers can be used to perform any of the following tasks:
  - Audit data modification.
  - Log events transparently.
  - Enforce complex business rules.
  - Derive column values automatically.
  - Implement complex security authorizations.
  - Maintain replicate tables.
  - Publish information about events for a publish-subscribe environment such as that associated with web programming.

- Triggers:
  - are named PL/SQL blocks with declarative, executable, and exception handling sections.
  - are stand-alone database objects – they are not stored as part of a package and cannot be local to a block.
  - do not accept arguments.

- To create/test a trigger, you (not the system user of the trigger) must have appropriate access to all objects referenced by a trigger action.

- Example: To create a BEFORE INSERT trigger for the *employee* table requires you to have INSERT ROW privileges for the table.

Create Trigger Syntax

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE|AFTER|INSTEAD OF} triggering_event
  [referencing_clause] ON {table_name | view_name}
[WHEN condition] [FOR EACH ROW]
[DECLARE
     Declaration statements]
BEGIN
     Executable statements
[EXCEPTION
     Exception-handling statements]
END;
```

The trigger body must have at least the executable section.

- The declarative and exception handling sections are optional.
- When there is a declarative section, the trigger body must start with the DECLARE keyword.
- The WHEN clause specifies the condition under which a trigger should fire.

Trigger Types

- BEFORE and AFTER Triggers – trigger fires before or after the triggering event. Applies only to tables.
- INSTEAD OF Trigger – trigger fires instead of the triggering event. Applies only to views.
- Triggering_event – a DML statement issued against the table or view named in the ON clause – example: INSERT, UPDATE, or DELETE.
- DML triggers are fired by DML statements and are referred to sometimes as row triggers.
- FOR EACH ROW clause – a ROW trigger that fires once for each modified row.
- STATEMENT trigger – fires once for the DML statement.
- Referencing_clause – enables writing code to refer to the data in the row currently being modified by a different name.

Conditional Predicates for Detecting Triggering DML Statement

Conditional Predicate	TRUE if and only if:
INSERTING	An INSERT statement fired the trigger.
UPDATING	An UPDATE statement fired the trigger.
UPDATING ('*column*')	An UPDATE statement that affected the specified column fired the trigger.
DELETING	A DELETE statement fired the trigger.

```
SET SERVEROUTPUT On
CREATE OR REPLACE TRIGGER t
 BEFORE
   INSERT OR
   UPDATE OF salary, deptno OR
   DELETE   ON emp
BEGIN
 CASE
   WHEN INSERTING THEN
     DBMS_OUTPUT.PUT_LINE('Inserting');
   WHEN UPDATING('salary') THEN
     DBMS_OUTPUT.PUT_LINE('Updating salary');
   WHEN UPDATING(‘deptno') THEN
     DBMS_OUTPUT.PUT_LINE('Updating department ID');
   WHEN DELETING THEN
     DBMS_OUTPUT.PUT_LINE('Deleting');
 END CASE;
END;
/
```

ROW Trigger – Accessing Rows

- Access data on the row currently being processed by using two correlation identifiers named **:old** and **:new**. These are special Oracle bind variables.
- The PL/SQL compiler treats the **:old** and **:new** records as records of type trigger_Table_Name%ROWTYPE.
- To reference a column in the triggering table, use the notation shown here where the *ColumnName* value is a valid column in the triggering table.

```
:new.ColumnName
:old.ColumnName
```

Empno	Ename	Sal
100	Raj	129399

```
:old.sal is 123999
```

Empno	Ename	Sal
100	Raj	116000

```
:new.sal is 111600
```

```
SET SERVEROUTPUT On
CREATE OR REPLACE TRIGGER t1
  BEFORE INSERT OR UPDATE OF salary, deptno OR
    DELETE ON emp FOR EACH ROW
BEGIN
  CASE
    WHEN INSERTING THEN
      DBMS_OUTPUT.PUT_LINE('Inserting '|| :NEW.EMPNO||:NEW.SAL);
    WHEN UPDATING('sal') THEN
      DBMS_OUTPUT.PUT_LINE('Updating salary'||:OLD.SAL||'---'||:NEW.SAL);
    WHEN UPDATING('DEPTNO') THEN
      DBMS_OUTPUT.PUT_LINE('Updating department ID'||:OLD.DEPTNO||:NEW.DEPTNO);
    WHEN DELETING THEN
      DBMS_OUTPUT.PUT_LINE('Deleting');
      DBMS_OUTPUT.PUT_LINE(:OLD.EMPNO||'--'||:OLD.ENAME||'--'||:OLD.SAL||'--
'||:OLD.DEPTNO);
  END CASE;
END;
/
```

Bind Variables :old and :new Defined

DML Statement	**:old**	**:new**
INSERT	Undefined – all column values are NULL as there is no “old” version of the data row being inserted.	Stores the values that will be inserted into the new row for the table.
UPDATE	Stores the original values for the row being updated before the update takes place.	Stores the new values for the row – values the row will contain after the update takes place.
DELETE	Stores the original values for the row being deleted before the deletion takes place.	Undefined – all column values are NULL as there will not be a “new” version of the row being deleted.

Example

```
CREATE TABLE Emp_log (
  Emp_id     NUMBER(4),
  Log_date   DATE,
  New_salary NUMBER(7,2),
  Action     VARCHAR2(20));
```

Example-

```
CREATE OR REPLACE TRIGGER log_salary_increase
  AFTER UPDATE OF salary ON employee
  FOR EACH ROW
BEGIN
  INSERT INTO Emp_log (Emp_id, Log_date, New_salary,
Action)
  VALUES (:NEW.empno, SYSDATE, :NEW.salary, ‘Insert
New Salary');
END;
/
```

Example-Statement Trigger

Create the following table and do not insert records.

```
CREATE TABLE users_log ( User_name varchar2(10),Operation varchar2(10), Login_Date Date ) ;
```

```
CREATE OR REPLACE TRIGGER note_hr_logoff_trigger
  BEFORE LOGOFF
  ON m2021.SCHEMA
BEGIN
  INSERT INTO users_log VALUES (USER, 'LOGOFF', SYSDATE);
END;
/
```

Example-Statement Trigger

Create the following table and do not insert records.

```
CREATE TABLE users_log ( User_name varchar2(10),Operation varchar2(10), Login_Date Date ) ;


create or replace trigger emp_sal_update before update of sal on emp
begin
if to_char(sysdate,'DY') = 'SUN' then
raise_application_error(-20111,'No changes can be made on sunday.');
else
dbms_output.put_line(' Today is not SUNDAY, Let us WOrk');
end if;
end;
/
```

t.sql

Dropping a Trigger

- The DROP TRIGGER statement drops a trigger from the database.
- If you drop a table, all associated table triggers are also dropped.
- The syntax is:

```
DROP TRIGGER trigger_name;
```