ADVANCED SQL

- PL/SQL
- PROCEDURES
- FUNCTIONS
- TRIGGERS
- cursors

PL/SQL

PL/SQL is Oracle's *procedural* language extension to SQL, the non-procedural relational database language.

With PL/SQL, you can use SQL statements to **manipulate ORACLE data and the *flow* of control statements to process the data**. Moreover, you can declare constants and variables, define subprograms (procedures and functions), and trap runtime errors.

Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

DIFFERENCE BETWEEN PL/SQL AND SQL

When a SQL statement is issued on the client computer, the request is made to the database on the server, and the result set is sent back to the client.

As a result, a single SQL statement causes two trips on the network. If multiple SELECT statements are issued, the network traffic increase significantly very fast. For example, four SELECT statements cause eight network trips.

If these statements are part of the PL/SQL block, they are sent to the server as a single unit. The SQL statements in this PL/SQL program are executed at the server and the result set is sent back as a single unit. There is still only one network trip made as is in case of a single SELECT statement.

PL/SQL BLOCKS

PL/SQL blocks can be divided into two groups:

1. Named
2. Anonymous.

Named blocks are used when creating subroutines. These subroutines are procedures, functions, and packages.

The subroutines can be stored in the database and referenced by their names later on.

In addition, subroutines can be defined within the anonymous PL/SQL block.

Anonymous PL/SQL blocks do not have names. As a result,they cannot be stored in the database and referenced later.

PL/SQL BLOCK STRUCTURE

PL/SQL blocks contain three sections

1. Declare section
2. Executable section and
3. Exception-handling section.

The executable section is the only mandatory section of the block.

Both the declaration and exception-handling sections are optional.

PL/SQL BLOCK STRUCTURE

PL/SQL block has the following structure:

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

DECLARATION SECTION

The *declaration section* is the first section of the PL/SQL block.

It contains definitions of PL/SQL identifiers such as variables, constants, cursors and so on.

Example

```
DECLARE
   v_first_name VARCHAR2(35) ;
   v_last_name VARCHAR2(35) ;
   v_counter NUMBER := 0 ;
```

EXECUTABLE SECTION

The executable section is the next section of the PL/SQL block.

This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section.

```
BEGIN
  SELECT first_name, last_name
     FROM student
     WHERE student_id = 123 ;
  DBMS_OUTPUT.PUT_LINE
  ('Student name :' || first_name ||'  '|| last_name);
END;
```

EXCEPTION-HANDLING SECTION

The *exception-handling section* is the last section of the PL/SQL block.

This section contains statements that are executed when a runtime error occurs within a block.

Runtime errors occur while the program is running and cannot be detected by the PL/SQL compiler.

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
  DBMS_OUTPUT.PUT_LINE
     (‘ There is no student with student id 123 ’);
END;
```

```
DECLARE
-- variable declaration
message varchar2(20):= 'Hello, World!';
BEGIN
/* * PL/SQL executable statement(s) */
dbms_output.put_line(message);

END; /
```

```
Hello World
PL/SQL procedure successfully completed.
```

PL/SQL EXAMPLE

```
DECLARE
  v_first_name VARCHAR2(35);
  v_last_name VARCHAR2(35);
BEGIN
  SELECT first_name, last_name
  INTO v_first_name, v_last_name
  FROM student
  WHERE student_id = 123;
  DBMS_OUTPUT.PUT_LINE
  ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE
           ('There is no student with student id 123');
END;
```

PL/SQL Program Units

- A PL/SQL unit is any one of the following −
- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

Programmatic Control Constructs

- **If-Then-ElsIf-then-Else-End If :**

```
IF < condition> THEN
        < action >

ELSIF <condition> THEN
        < action >

ELSE
        < action >
END IF;
```

```
1  DECLARE
2    n_sales NUMBER := 300000;
3    n_commission NUMBER( 10, 2 ) := 0;
4  BEGIN
5    IF n_sales > 200000 THEN
6      n_commission := n_sales * 0.1;
7    ELSE
8      n_commission := n_sales * 0.05;
9    END IF;
10 END;
```

```
1  DECLARE
2    n_sales NUMBER := 300000;
3    n_commission NUMBER( 10, 2 ) := 0;
4  BEGIN
5    IF n_sales > 200000 THEN
6      n_commission := n_sales * 0.1;
7    ELSIF n_sales <= 200000 AND n_sales > 100000 THEN
8      n_commission := n_sales * 0.05;
9    ELSIF n_sales <= 100000 AND n_sales > 50000 THEN
10     n_commission := n_sales * 0.03;
11   ELSE
12     n_commission := n_sales * 0.02;
13   END IF;
14 END;
```

S.No	Loop Type & Description
1	PL/SQL Basic LOOPIn this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
2	PL/SQL WHILE LOOPRepeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
3	PL/SQL FOR LOOPExecute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	Nested loops in PL/SQLYou can use one or more loop inside any another basic loop, while, or for loop.

BASIC LOOPS IN PL/SQL

- sequence of statements is enclosed between the LOOP and the END LOOP statements.
- At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

```
Declare
Begin
Loop
   Statements
End loop
End;/
```

- https://www.youtube.com/watch?v=AFx6QYcY1CU

- Controlling the loops
- EXIT
- EXIT WHEN

While and For

While Loop:

```
WHILE <condition>
LOOP
        <Action>
END LOOP;
```

Syntax

```
FOR counter IN initial_value .. final_value LOOP
   sequence_of_statements;
END LOOP;
```

```
DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produce

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

```
DECLARE
   a number(2);
BEGIN
   FOR a in 10 .. 20 LOOP
      dbms_output.put_line('value of a: ' || a);
  END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

PL/SQL procedure successfully completed.
```

FOR LOOP Statement

```
DECLARE
   a number(2) ;
BEGIN
   FOR a IN REVERSE 10 .. 20 LOOP
      dbms_output.put_line('value of a: ' || a);
   END LOOP;
END;
/
```

When the above code is executed at the SQL prompt,

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10
```

Case Statement

```
CASE [TRUE | selector]
   WHEN expression1 THEN
sequence_of_statements1;
   WHEN expression2 THEN
sequence_of_statements2;
   ...
   WHEN expressionN THEN
sequence_of_statementsN;
  [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

// **Case statement**

```
DECLARE
      grade CHAR(1);
BEGIN
   grade := 'B';
CASE grade
   WHEN 'A' THEN
DBMS_OUTPUT.PUT_LINE('Excellent');
   WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very
Good');
   WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
END CASE;
```

```
DECLARE
  n_pct    employees.commission_pct%TYPE;
  v_eval   varchar2(10);
  n_emp_id employees.employee_id%TYPE := 145;
BEGIN
  -- get commission percentage
  SELECT commission_pct
  INTO n_pct
  FROM employees
  WHERE employee_id = n_emp_id;

  -- evalutate commission percentage
  CASE n_pct
    WHEN 0 THEN
      v_eval := 'N/A';
    WHEN 0.1 THEN
      v_eval := 'Low';
    WHEN 0.4 THEN
      v_eval := 'High';
    ELSE
      v_eval := 'Fair';
  END CASE;
  -- print commission evaluation
  DBMS_OUTPUT.PUT_LINE('Employee ' || n_emp_id ||
                       ' commission ' || TO_CHAR(n_pct) ||
                       ' which is '   || v_eval);
END;
```

Procedures and Functions in PL/SQL

- PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −
- **Functions** − These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

Procedure

- A Procedure is a subprogram unit that consists of a group of PL/SQL statements. Each procedure in Oracle has its own unique name by which it can be referred. This subprogram unit is stored as a database object.
- A stored procedure or in simple term, a proc is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

PROCEDURES

- The syntax for creating a procedure is as follows:

```
CREATE OR REPLACE PROCEDURE name
```

(<parameterl IN/OUT <datatype>)

```
[AS | IS]
   [local declarations]
BEGIN
   executable statements
[EXCEPTION
   exception handlers]
END [name];
```

- CREATE PROCEDURE instructs the compiler to create new procedure. Keyword 'OR REPLACE' instructs the compile to replace the existing procedure (if any) with the current one.
- Procedure name should be unique.
- Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning

- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
dbms_output.line("Hello world");
END;
```

When the above code is executed using the SQL prompt, it will produce the following result − PROCEDURE CREATED

```
EXECUTE greetings;
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
greetings;
END; /
```

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement.

Syntax for deleting a procedure is

```
DROP PROCEDURE procedure-name;
```

Parameter

- The **parameter is variable or placeholder** of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.
- These parameters should be **defined** along with the subprograms at the **time of creation**.
- These parameters are **included in the calling statement** of these subprograms to interact the values with the subprograms.
- The datatype of the parameter in the subprogram and the calling statement **should be same**.
- The **size of the datatype should not mention at the time of parameter declaration,** as the size is dynamic for this type.

PARAMETERS

Parameters are the means to pass values to and from the calling environment to the server.

These are the values that will be processed or returned via the execution of the procedure.

- There are three types of parameters:

  IN, OUT, and IN OUT.

IN passes value into the procedure, OUT passes back from the procedure and INOUT does both.

Types of Parameters

Mode	Description	Usage
IN	Passes a value into the program	Read only value Constants, literals, expressions Cannot be changed within program Default mode
OUT	Passes a value back from the program	Write only value Cannot assign default values Has to be a variable Value assigned only if the program is successful
IN OUT	Passes values in and also send values back	Has to be a variable Value will be read and then written

S.No	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

create table named emp have two column id and salary with number datatype.

```
CREATE OR REPLACE  PROCEDURE emp_insert(id IN NUMBER, sal IN NUMBER) AS
BEGIN
 INSERT INTO emp VALUES(id, sal);
 DBMS_OUTPUT.PUT_LINE('VALUE INSERTED.');
END;
/
```

Procedure created.

```
SET SERVEROUTPUT ON;
EXECUTE  emp_insert(101,2000);
```

procedure successfully completed.

// **Insert a person information into a PERSON table provided his information does not exist already.**

```
CREATE OR REPLACE PROCEDURE insertPerson ( id IN VARCHAR,
 DOB IN DATE, fname IN VARCHAR,  lname IN VARCHAR)
IS
   counter      INTEGER;     --declaration part
   BEGIN
   SELECT COUNT(*) INTO counter FROM person p WHERE      p.pid = id;
   IF (counter > 0) THEN
   -- person with the given pid already exists
   DBMS_OUTPUT.PUT_LINE('WARNING Inserting    person: person with
 pid ' || id || ' already exists!');
   ELSE
   INSERT INTO person VALUES (id, DOB, fname, lname);
   DBMS_OUTPUT.PUT_LINE('Person with pid ' || id || ' is inserted.');
    END IF;
END;
/
```

// **Procedure calls another procedure**

```
CREATE OR REPLACE PROCEDURE insertFaculty (pid IN VARCHAR,
 DOB IN DATE, fname IN VARCHAR, lname IN VARCHAR, rank IN
 VARCHAR, dept IN VARCHAR) IS
BEGIN
    insertPerson (pid, DOB, fname, lname);     // User defined Procedure
    insert into facultyEDB values(pid, rank, dept);
    DBMS_OUTPUT.PUT_LINE('Faculty with pid ' || pid || ' is inserted.');
END insertFaculty;
```

EXECUTE insertFaculty('121-11-1111', '21-OCT-1961', 'Susan', 'Urban', 'Emeritus', 'CSE');

-- from sql prompt

Example

- In order to execute a procedure use the following syntax:

EXECUTE Procedure_name;

Or EXEC Procedure_name;

```
SQL> EXECUTE insertPerson ('p1', '10-10-2000', 'John', 'Smith') ;
```

- To see the o/p on the screen use following command

SET SERVEROUTPUT ON

FUNCTIONS

Functions are a type of stored code and are very similar to procedures.

The significant difference is that a function is a PL/SQL **block that *returns* a single value.**

Functions can accept one, many, or no parameters, but a function **must have a return clause** in the executable section of the function.

The datatype of the return value must be declared in the header of the function.

A function is not a stand-alone executable in the way that a procedure is: It must be used in some context.

A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.

FUNCTIONS

The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.

The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception).

FUNCTIONS

- The syntax for creating a function is as follows:

```
CREATE [OR REPLACE] FUNCTION
    function_name
  (parameter_list)
RETURN datatype
IS
BEGIN
  <body>
  RETURN (return_value);
END;
```

```
Select * from customers;
```

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

Function to fetch no of customers.

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;

   RETURN total;
END;
/
```

```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

call the function

When the above code is executed at the SQL prompt, it produces the following result −

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

```
1.  CREATE OR REPLACE FUNCTION welcome_msg_func ( p_name IN VARCHAR2)
2.  RETURN VARCHAR2
3.  IS
4.  BEGIN
5.  RETURN ('Welcome '|| p_name);
6.  END;
7.  /
```

Output:

Function created

Function Created

```
8.  DECLARE
9.  lv_msg VARCHAR2(250);
10. BEGIN
11. lv_msg := welcome_msg_func ('Guru99');
12. dbms_output.put_line(lv_msg);
13. END;
```

Calling function with 'Guru99' as parameter

Output:

Welcome Guru99

Example

```
CREATE OR REPLACE FUNCTION show_description (i_course_no IN
        number)
RETURN varchar2
IS
  v_description varchar2(50);
BEGIN
  SELECT description INTO v_description
  FROM course WHERE course_no = i_course_no;
  RETURN v_description;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
     RETURN('The Course is not in the database');
  WHEN OTHERS    THEN
     RETURN('Error in running show_description');
END;
```

Making Use Of Functions

In a anonymous block

```
DECLARE
  v_description VARCHAR2(50);
BEGIN
  v_description := show_description(&sv_cnumber);
  DBMS_OUTPUT.PUT_LINE(v_description);
END;
```

In a SQL statement

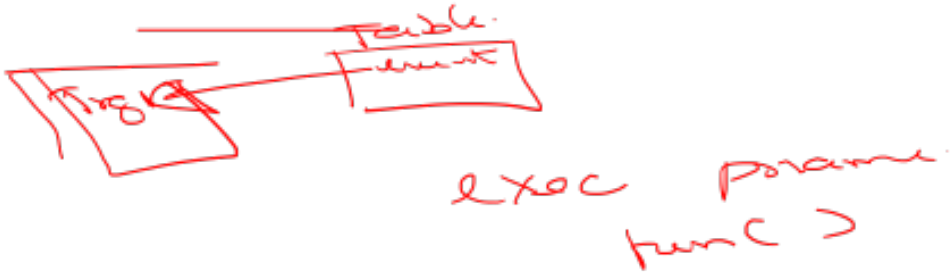
```
SELECT course_no, show_description(course_no)
  FROM course;
```

SET SERVEROUTPUT ON

TRIGGERS & CURSORS

included for exams

Triggers



- An SQL trigger is a mechanism that automatically executes a specified PL/SQL block when a triggering event occurs on a table.
- The triggering event may be one of insert, delete, or update.
- The trigger is associated with a database table and is fired when the triggering event takes place on the table.

TRIGGERS

You can associate up to 12 database triggers with a given table.

A database trigger has three parts:

a **triggering event**, an **optional trigger constraint**, and a **trigger action**.

When an event occurs, a database trigger is fired, and an predefined PL/SQL block will perform the necessary action.

Triggers

create [or replace] trigger trigger-name
{before | after}
{delete | insert | update [of column [, column] …]}
ON table-name
[[**referencing** {old [as] <old> [new [as] <new>]
| new [as] <new> [old [as] <old> }]
for each row
[**when** (condition)]]
pl/sql_block

TRIGGERS

The trigger_name references the name of the trigger.

BEFORE or AFTER specify when the trigger is fired (before or after the triggering event).

The triggering_event references a DML statement issued against the table (e.g., INSERT, DELETE, UPDATE).

The table_name is the name of the table associated with the trigger.

The clause, FOR EACH ROW, specifies a trigger is a row trigger and fires once ~~for each modified~~ row.

Bear in mind that if you drop a table, all the associated triggers for the table are dropped as well.

Triggers

- referencing specifies correlation names that can be used to refer to the old and new values of the row components that are being affected by the trigger
- for each row designates the trigger to be a row trigger, i.e., the trigger is fired once for each row that is affected by the triggering event and meets the optional trigger constraint defined in the when clause.
- when specifies the trigger restriction.

example

```
mysql> desc Student;
+-------+-------------+------+-----+---------+
| Field | Type        | Null | Key | Default |
+-------+-------------+------+-----+---------+
| tid   | int(4)      | NO   | PRI | NULL    |
| name  | varchar(30) | YES  |     | NULL    |
| subj1 | int(2)      | YES  |     | NULL    |
| subj2 | int(2)      | YES  |     | NULL    |
| subj3 | int(2)      | YES  |     | NULL    |
| total | int(3)      | YES  |     | NULL    |
| per   | int(3)      | YES  |     | NULL    |
+-------+-------------+------+-----+---------+
```

```
create trigger stud_marks
before INSERT
on
Student
for each row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.p
```

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)

mysql> select * from Student;
+-----+-------+-------+-------+-------+-------+------+
| tid | name  | subj1 | subj2 | subj3 | total | per  |
+-----+-------+-------+-------+-------+-------+------+
| 100 | ABCDE |    20 |    20 |    20 |    60 |   36 |
+-----+-------+-------+-------+-------+-------+------+
1 row in set (0.00 sec)
```

TYPES OF TRIGGERS

A trigger may be a ROW or STATEMENT type.

If the statement FOR EACH ROW is present in the CREATE TRIGGER clause of a trigger, the trigger is a row trigger. A row trigger is fired for each row affected by an triggering statement.

A statement trigger, however, is fired only once for the triggering statement, regardless of the number of rows affected by the triggering statement

- Suppose, you want to restrict users to update credit of customers from 28th to 31st of every month so that you can close the financial month.

To enforce this rule, you can use this statement-level trigger:

```
1  CREATE OR REPLACE TRIGGER customers_credit_trg
2      BEFORE UPDATE OF credit_limit
3      ON customers
4  DECLARE
5      l_day_of_month NUMBER;
6  BEGIN
7      -- determine the transaction type
8      l_day_of_month := EXTRACT(DAY FROM sysdate);
9
10     IF l_day_of_month BETWEEN 28 AND 31 THEN
11         raise_application_error(-20100,'Cannot update customer credit from 28th to 31st');
12     END IF;
13 END;
```

TYPES OF TRIGGERS

Example 2: statement trigger

```
CREATE OR REPLACE TRIGGER mytrig1
BEFORE DELETE OR INSERT OR UPDATE ON employee
BEGIN
IF     (TO_CHAR(SYSDATE, 'day') IN ('sat', 'sun'))      OR
       (TO_CHAR(SYSDATE,'hh:mi') NOT BETWEEN '08:30' AND '18:30')

THEN     RAISE_APPLICATION_ERROR(-20500, 'table is secured');
END IF;
END;
/
```

Raise_application_error

- Used inside trigger
- Purpose:
  - output an error message and
  - immediately **stop** the event that fired the trigger
  - For example, data insertion
- Can include variables/trigger values, see previous slide
- E.g.

```
RAISE_APPLICATION_ERROR(-20000,'trigger violated')
```

label

Message text

Referencing the values inside trigger

- ROW LEVEL TRIGGER allows to track old and new values
- When a DML statement changes a column, the old and new values are visible to the executing code
- This is done by prefixing the table column with :old or :new

:new is useful for INSERT and UPDATE

:old is useful for DELETE and UPDATE

Eg :old.emp_salary :new.emp_salary

- triggers may fire other triggers in which case they are CASCADING. Try not to create too many interdependencies with triggers!

:OLD.column_name
:NEW.column_name

IF :NEW.credit_limit > :OLD.credit_limit THEN
--few statements
ENDIF

Example: ROW Trigger

```
CREATE OR REPLACE TRIGGER mytrig2
AFTER DELETE OR INSERT OR UPDATE ON employee
FOR EACH ROW
BEGIN
IF DELETING THEN
   INSERT INTO xemployee (emp_ssn,  emp_last_name,  emp_first_name,  del_date)
   VALUES (:old.emp_ssn,  :old.emp_last_name,  :old.emp_first_name,  sysdate);

ELSIF INSERTING THEN
   INSERT INTO nemployee (emp_ssn,  emp_last_name,  emp_first_name,  add_date)
   VALUES (:new.emp_ssn,  :new.emp_last_name ,  :new.emp_first_name,  sysdate);

ELSIF UPDATING('emp_salary') THEN
      INSERT INTO cemployee (emp_ssn,  oldsalary,  newsalary,  up_date)
      VALUES (:old.emp_ssn,  :old.emp_salary,  :new.emp_salary,  sysdate);

END IF;
END;
```

// **If person's department changes and earlier he is the chairperson for that department, then for that department chair is set to NULL in department table.**

```
CREATE OR REPLACE TRIGGER faculty_after_update_row
AFTER UPDATE ON facultyEDB        // Here facultyEDB has dept and pid attributes.
  FOR EACH ROW
  BEGIN
     IF UPDATING ('dept') AND :old.dept <> :new.dept
     THEN UPDATE department SET chair = NULL WHERE chair = :old.pid;
 END IF;
END; /
```

Ex. 4

// **When faculty is deleted from a faculty table , delete his information from Person table**

```
CREATE OR REPLACE TRIGGER faculty_after_delete_row
AFTER DELETE ON facultyEDB
FOR EACH ROW BEGIN
  DELETE FROM person WHERE pid = :old.pid;
END; /
```

ENABLING, DISABLING, DROPPING TRIGGERS

SQL>ALTER TRIGGER trigger_name DISABLE;
SQL>ALTER TABLE table_name DISABLE ALL TRIGGERS;

To enable a trigger, which is disabled, we can use the following syntax:

SQL>ALTER TABLE table_name ENABLE trigger_name;
All triggers can be enabled for a specific table by using the following command

SQL> ALTER TABLE table_name ENABLE ALL TRIGGERS;
SQL> DROP TRIGGER trigger_name

Referential integrity trigger example

```
UPDATE author SET aID='9' WHERE aID='5';

ERROR at line 1:
ORA-02292: integrity constraint
   (MCTPL.BOOK_FK) violated - child record found
```

Without trigger - referential integrity prevents changes to aID

```
CREATE OR REPLACE TRIGGER author_trg
AFTER UPDATE OF aID ON author
FOR EACH ROW
BEGIN
UPDATE Book SET authID = :new.aID WHERE authID= :old.aID;
END;
```

Trigger automatically applies corresponding changes to aID in child table

```
UPDATE author SET aID='9' WHERE aID='5';
1 row updated.
```

With trigger - changes to aID are now allowed!

Example trigger: Incorporating business rules

To raise a message if more than 1000 books from the same publisher has been added to the "Book" table.

```
CREATE OR REPLACE TRIGGER publish_trg
BEFORE INSERT OR UPDATE ON book
FOR EACH ROW

DECLARE
   how_many NUMBER;
BEGIN
   SELECT COUNT(*) INTO how_many FROM book
          WHERE publisher = :new.publisher;
   IF how_many >= 1000 then
     Raise_application_error(-20000,'Publisher' ||
       :new.publisher || 'already has 1000 books');
   END IF;
END;
```

Compilation errors

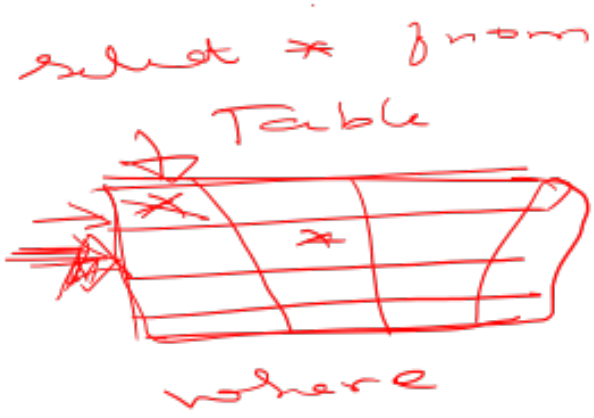
- When you create a trigger, Oracle responds
  - "Trigger created" or
  - "Warning: Trigger created with compilation errors."
- Type in the command

  SHOW ERRORS

  on its own to make the error messages visible

Database Access Using Cursors

Cursors



- A pointer to the context area
- Cursor Types:
- Explicit: user-defined
- Implicit: system-defined

Types of Cursor

- **Implicit Cursors:**
  Implicit Cursors are also known as Default Cursors of SQL SERVER. These Cursors are allocated by SQL SERVER when the user performs DML operations.
- **Explicit Cursors :**
  Explicit Cursors are Created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.

Explicit Cursors

- To use explicit cursors…
- Declare the cursor
- Open the cursor
- Fetch the results into PL/SQL variables
- Close the cursor

- https://www.youtube.com/watch?v=_snAMqCBitg

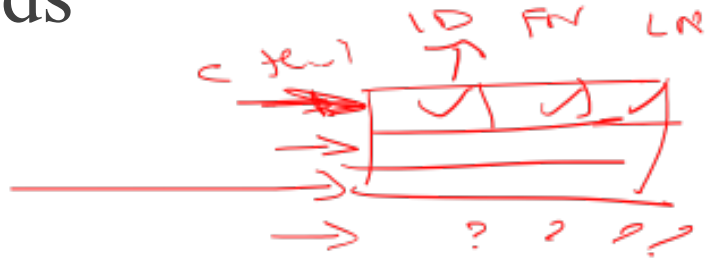
Declaring Cursors

```
DECLARE
  v_StudentID    students.id%TYPE;
  v_FirstName    students.first_name%TYPE;
  v_LastName     students.last_name%TYPE;

  CURSOR c_HistoryStudents IS
    SELECT id, first_name, last_name
    FROM students
    WHERE major = 'History';
BEGIN
  -- open cursor, fetch records & then close cursor here
END;
/
```

OPEN Cursor

```
DECLARE
  v_StudentID    students.id%TYPE;
  v_FirstName    students.first_name%TYPE;
  v_LastName     students.last_name%TYPE;

  CURSOR c_HistoryStudents IS
    SELECT id, first_name, last_name
    FROM students
    WHERE major = 'History';
BEGIN
  OPEN c_HistoryStudents;
  -- fetch records & then close cursor here
END;
/
```

FETCH Records



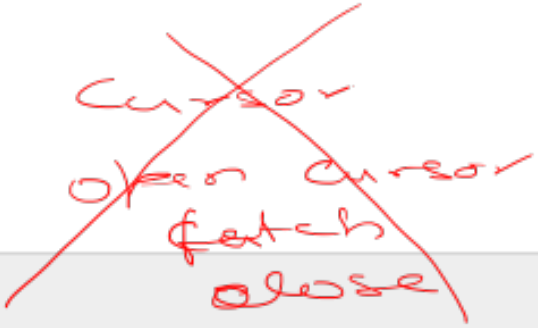

```
DECLARE
  v_StudentID    students.id%TYPE;
  v_FirstName    students.first_name%TYPE;
  v_LastName     students.last_name%TYPE;
  CURSOR c_HistoryStudents IS
    SELECT id, first_name, last_name
    FROM students
    WHERE major = 'History';
BEGIN
  OPEN c_HistoryStudents;
  LOOP
    FETCH c_HistoryStudents INTO v_StudentID, v_FirstName, v_LastName;
    EXIT WHEN c_HistoryStudents%NOTFOUND;
    -- do something with the values that are now in the variables
  END LOOP
  CLOSE c_HistoryStudents;
END;/
```

Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open.
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row.
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far.

Implicit cursor




```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   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;
/
```

default cursor

data types

%type and % rowtype

- The %TYPE attribute, used in PL/SQL variable and parameter declarations, is supported by the data server. Use of this attribute ensures that type compatibility between table columns and PL/SQL variables is maintained.

%TYPE

Table: Agents (aid, aname, city, percent)

```
DECLARE
    percent_val     agents.percent%TYPE;

BEGIN

  SELECT percent INTO percent_val FROM agents WHERE aid = 'a02';

IF percent_val > 50 THEN
            INSERT INTO agents (aid, aname, city) VALUES (
                                                    'a07', 'John', 'Corpus');
END IF;

END; /
```

%ROWTYPE

```
DECLARE

   dept_rec    departments%ROWTYPE;

BEGIN

    SELECT * INTO dept_rec       -- Can access only one tuple
      FROM departments
       WHERE department_id = 30 ;
  DBMS_OUTPUT.PUT_LINE
      (‘Dept infor:  ' || dept_rec .Name ||’  ‘|| dept_rec.Head_Name) ;

END;
 /
```

- Write a program in PL/SQL to display a cursor based detail information of employees from employees table.

```
DECLARE
CURSOR z_emp_info IS
SELECT employee_id, first_name, last_name, salary FROM
employees;
r_emp_info z_emp_info%ROWTYPE;
BEGIN
OPEN z_emp_info;
LOOP FETCH z_emp_info
INTO r_emp_info;
EXIT WHEN z_emp_info%NOTFOUND;
dbms_output.Put_line('Employees Information:: ' ||' ID: '
||r_emp_info.employee_id ||' Name: ' ||r_emp_info.first_name ||' '
||r_emp_info.last_name);
END LOOP;
dbms_output.Put_line('Total number of rows : '
||z_emp_info%rowcount); CLOSE z_emp_info;
END; /
```

- Write a program in PL/SQL to create an implicit cursor with for loop.

```
BEGIN
FOR emprec IN
(SELECT department_name, d.department_id, first_name, last_name,
job_id, salary FROM departments d join employees e ON
e.department_id = d.department_id WHERE job_id = 'ST_CLERK'
AND salary > 3000) LOOP
dbms_output.Put_line('Name: ' ||emprec.first_name ||' '
||emprec.last_name||chr(9) ||' Department: '
||emprec.department_name||chr(9) ||' Department ID: '
||emprec.department_id||chr(9) ||' Job ID: ' ||emprec.job_id||chr(9) ||'
Salary: ' ||emprec.salary);
Exit when sql%notfound
END LOOP;
END; /
```

Workout

Using Functions (Not cursors)

Consider an employee database with two relations

employee (*employee name*, *street*, *city*)

works (*employee name*, *company name*, *salary*)

where the primary keys are underlined. Write a query to find companies whose employees earn a higher salary, on average, than the average salary at “First Bank Corporation”.