Chapter 6 (cont.):
Trigger, Store Procedure, Function & Cursor in Oracle

Contents

Contents

Trigger Overview

- A trigger is a procedure which is executed implicitly whenever the triggering event happens.
- Executing a trigger is to “fire” the trigger.
- Triggering Events are:
  - DML Commands: INSERT, UPDATE, DELETE
  - DDL Commands : CREATE, ALTER, DROP
  - Database Events: SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN

Trigger Overview

- Uses for triggers:
  - Automatically generate derived column values.
  - Maintain complex integrity constraints.
  - Enforce complex business rules.
  - Record auditing information about database changes.
  - Invoke a program when database changes.

Simple DML Trigger Syntax

CREATE [OR REPLACE] TRIGGER *schema.trigger_name*
BEFORE | AFTER | INSTEAD OF
DELETE | INSERT | UPDATE [*OF columns list*] [OR …]
ON *schema.table_name*
[*REFERENCING OLD [AS] <old_name> | NEW [AS] <new_name>*]
[**FOR EACH ROW**]
[**WHEN** *(condition)*]
BEGIN
    PL/SQL_block | call_procedure_statement;
END *trigger_name*;

Types of Triggers

Category	Values	Comments
DML	`Insert`	Type of DML which makes the trigger fire.
	`Update`	
	`Delete`	
Timing	`Before`	When the trigger fires.
	`After`	
	`Instead of`	
Level	Row	Row level triggers fire for each affected row. Identified by keywords `FOR EACH ROW`
	Statement	Statement level triggers fire once per DML Statement

Trigger Firing Order

1. **Before statement** triggers fire.
2. For Each Row:
   A) **Before row** triggers fire.
   B) Execute the Insert/Update/Delete.
   C) **After row** triggers fire.
3. **After statement** triggers fire.

REFERCING Clause: Old and New Data

- When **row-triggers** fire, there are 2 pseudo-records created called new and old.

```
new table_name%ROWTYPE;
old table_name%ROWTYPE;
```

- old and new are of datatype ROWTYPE from the affected table. Use dot notation to reference columns from old and new.
- old is undefined for insert statements.
- new is undefined for delete statements.

REFERCING Clause: Old and New Data

- Instead of a REFERENCING clause, Oracle assumes that new tuples are referred to as "new" and old tuples by "old."
- Also, for statement-level triggers: "newtable" and "oldtable".
- In actions, *but not in conditions*, you must prefix "new," etc., by a colon
  - :new
  - :old

Example: Row Level Trigger

```
CREATE TRIGGER    NoLowerPrices
AFTER UPDATE OF  price  ON Product
FOR EACH ROW
WHEN (old.price > new.price)
BEGIN
   UPDATE  Product
   SET  price = :old.price
   WHERE  p_name = :new.p_name;
END;
```

Bad Things Can Happen

```
CREATE TRIGGER  Bad_trigger
AFTER UPDATE OF price ON Product
FOR EACH ROW
WHEN   (new.price > 50)
BEGIN
   UPDATE  Product
   SET  price = :new.price * 2
   WHERE  p_name = :new.p_name;
END;
```

Contents

Database Stored Procedures

- **Stored procedures**
  - Program modules stored by the DBMS at the database server
  - Can be functions or procedures
- **Persistent stored modules**
  - Stored persistently by the DBMS

Stored Procedures & Functions

- Useful:
  - When database program is needed by several applications
  - To reduce data transfer and communication cost between client and server in certain situations
  - To enhance modeling power provided by views

Stored Procedures & Functions

- Declaring stored procedures:

```
CREATE [OR REPLACE] PROCEDURE
  procedure_name
[(parameter_name [IN | OUT | IN OUT]
  datatype )]
{IS | AS}
BEGIN
  procedure_body
END procedure_name;
```

Stored Procedures & Functions

- Parameter:
  - **Data type**: one of the SQL data types.
  - **Parameter mode**: `IN`, `OUT`, or `IN OUT`
    - **IN:** you must supply a value for the parameter when calling the procedure.
    - **OUT:** procedure passes a value for this parameter back to its calling environment after execution.
    - **IN OUT:** you must supply a value for the parameter when calling the procedure and that the procedure passes a value back to its calling environment after execution.
    - **Defaults: IN**.

Stored Procedures & Functions

- Example of store procedure:

```
CREATE OR REPLACE PROCEDURE update_salary
   (p_emp_id IN EMPLOYEE.SSN%type,
    p_factor IN NUMBER)
AS
  v_emp_count INTEGER;
BEGIN
   SELECT COUNT(*) INTO v_emp_count
    FROM employee
   WHERE SSN = p_emp_id;
   IF v_emp_count = 1 THEN
         UPDATE employee
         SET salary = salary * p_factor
         WHERE SSN = p_emp_id;
         COMMIT;
   END IF;
END update_salary;
```

Stored Procedures & Functions

- Calling a store procedure:
  - EXECUTE **update_salary (‘123456789’, 1.5);**
  - BEGIN
    update_salary (‘123456789’, 1.5);
    END;

Stored Procedures & Functions

- Declaring function:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT]
  datatype )]
RETURN datatype
{IS | AS}
BEGIN
  function_body
END function_name;
```

Stored Procedures & Functions

- Example of Function:

```
CREATE OR REPLACE FUNCTION get_salary
  (p_emp_id IN EMPLOYEE.SSN%TYPE)
RETURN NUMBER
AS
  v_sal NUMBER;
BEGIN
  SELECT salary into v_sal
  FROM EMPLOYEE
  WHERE SSN = p_emp_id;
  RETURN v_sal;
END get_salary;
```

Stored Procedures & Functions

- Calling a function:
  - SELECT * FROM EMPLOYEE
    WHERE salary = **get_salary (‘123456789’);**
  - SELECT **get_salary (‘123456789’)** FROM dual;

Contents

Database Access Using Cursors

- When the result of an SQL query (select statement) consists of more than one row, the simple select into statement can not be used.
- A PL/SQL **cursor** allows the program to fetch and process information from the database into the PL/SQL program, **one row at a time**.

Explicit Cursor

- Explicit cursor: used for processing a query resulting in more than one row.
- Implicit cursor: is automatically defined by PL/SQL for the select into statements, which result in one or fewer rows.
- Syntax of explicit cursor:

```
cursor <cname>  [return-spec]
is <select-statement>;
```

Cursor Example

```
cursor c1 return customers%rowtype is
select * from customers;
```

has return clause

```
cursor c2 is
select pno, pname, price*markdown sale_price
from parts;
```

Use PL/SQL variable markdown

Process cursor

- One a cursor has been declared, it can be processed using the **open**, **fetch**, and **close** statements.

open <cname>;

fetch <cname> into <Record-or-VariableList>;

close <cname>;

Explicit Cursor Attributes

- Obtain status information about a cursor.

%FOUND	Returns TRUE if the last fetch returned a row, or FALSE if the last fetch failed to return a row.
%NOTFOUND	The logical opposite of %FOUND.
%ROWCOUNT	Before the first fetch, returns 0. When a cursor is opened, %ROWCOUNT is zeroed. Thereafter, returns **the number of rows fetched** so far. The number is incremented if the latest fetch returned a row.
%ISOPEN	If a cursor is open, returns TRUE; otherwise, it returns FALSE.

Explicit Cursor Attributes example

```
IF c1%ISOPEN THEN
   FETCH c1 INTO v_ename, v_sal,
   v_hiredate;
ELSE
      OPEN c1;
END IF;
```

```
LOOP
   FETCH c1 INTO v_ename, v_sal,
   v_hiredate;
   EXIT WHEN c1%ROWCOUNT > 10;
END LOOP;
```

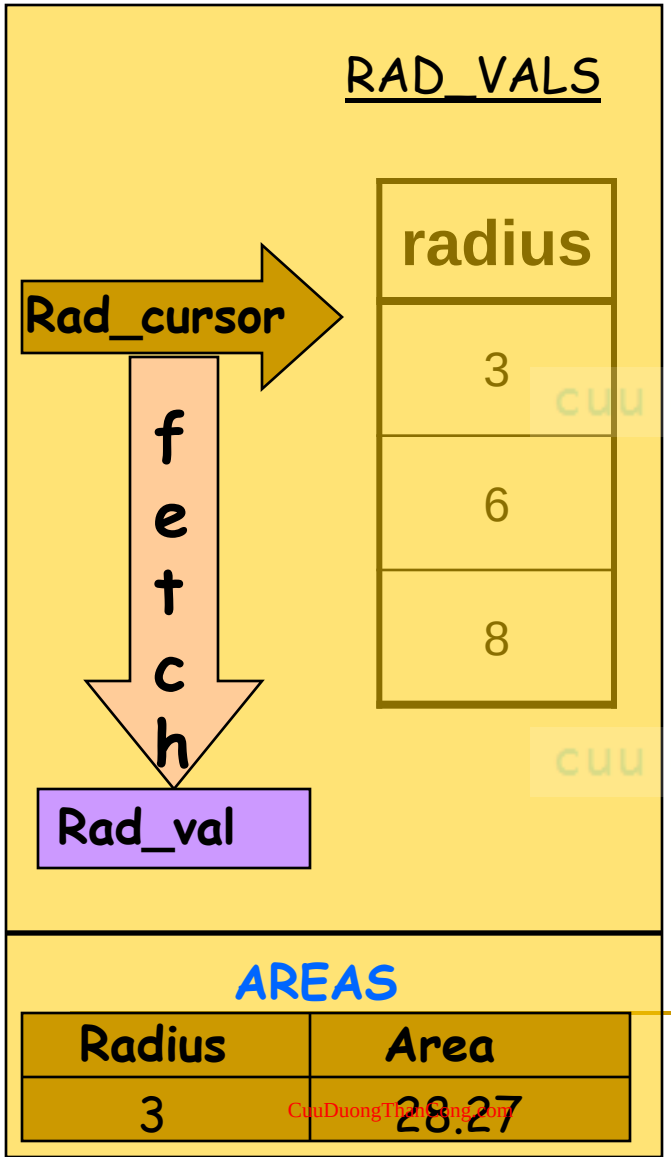
Cursor Example

```
DECLARE
    cursor c is select * from sailors;
     sailorData sailors%ROWTYPE;

BEGIN
    open c;
    fetch c into sailorData;
```


sailorData is a variable that can hold a ROW from the sailors table

Here the first row of sailors is inserted into sailorData

Cursor Example




```
DECLARE
  Pi constant NUMBER(8,7) := 3.1415926;
  area NUMBER(14,2);
  cursor rad_cursor is select * from RAD_VALS;
  rad_val rad_cursor%ROWTYPE;

BEGIN
  open rad_cursor;
  fetch rad_cursor into rad_val;
  area:=pi*power(rad_val.radius,2);
  insert into AREAS values (rad_val.radius,
     area);
  close rad_cursor;
END;
/
```

Cursor FOR LOOP statement

- This loop is very useful when all rows of the cursors are to be processed.

```
for <record_index> in <cursor name>
  loop
  <loop-body>;
end loop;
```

- `<record_index>` is a record variable that is implicitly declared by PL/SQL. Its scope is the for loop, and it can not be accessed outside the for loop.

Cursor FOR LOOP statement

- The loop terminates automatically when all rows of the cursor have been fetched.
- There is no need to open, fetch, or close the cursor, and there is no need to declare the record into which the cursor rows are to be fetched.

Cursor FOR LOOP example

```
declare
  cursor c1 is
    select cno, cname, city
    from customers, zipcodes
    where customers.zip = zipcodes.zip;
begin
  for c1_rec in c1 loop
     dbms_output.put_line('Row number ' ||
  c1%rowcount ||     '> ' || c1_rec.cno || '
  ' || c1_rec.cname || ' ' || c1_rec.city);
  end loop
end;
```


c1_rec

No declare for the record into which the cursor rows are to be fetched

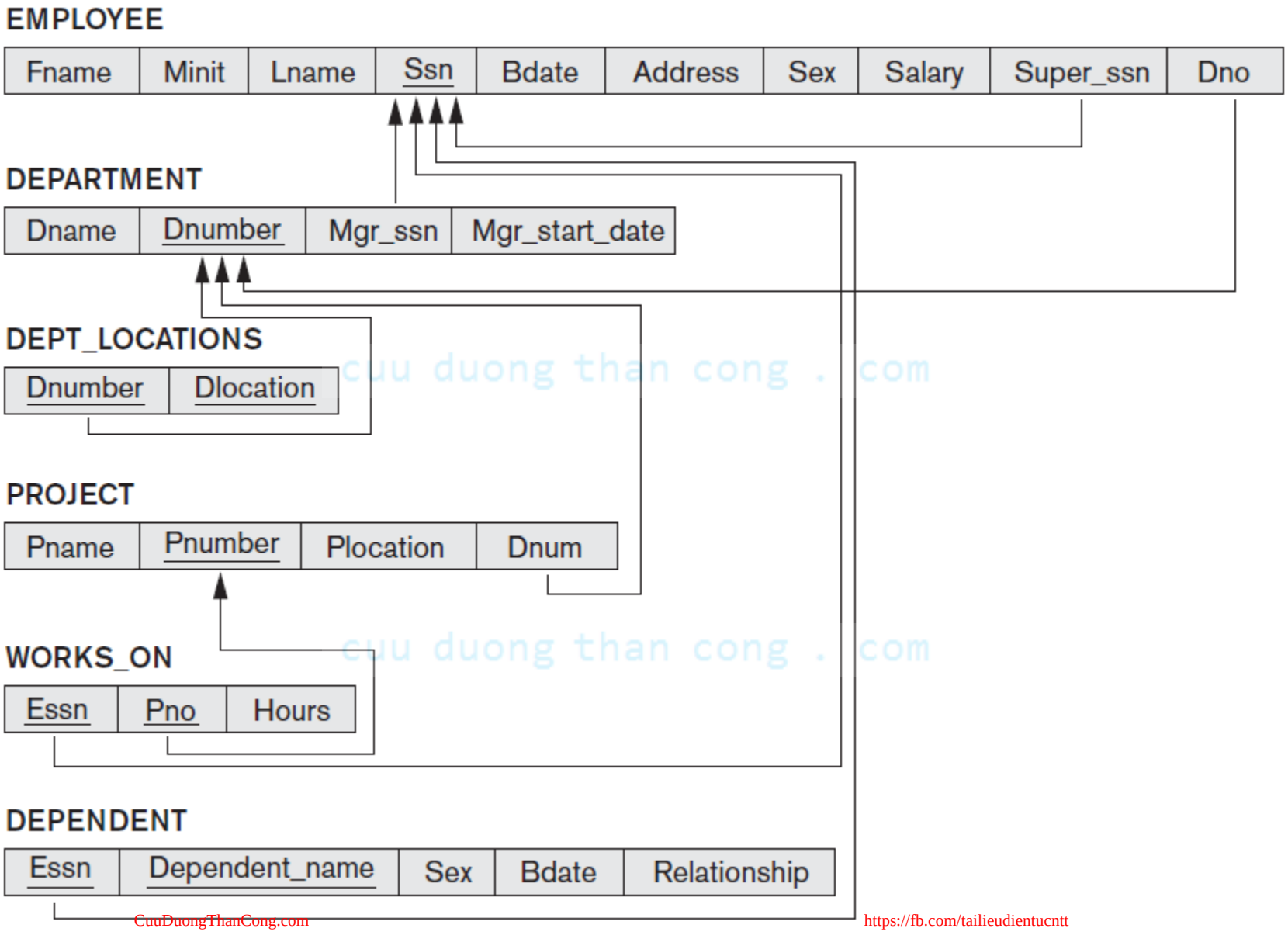
Another controlling Cursor Example

```
OPEN c_1;
LOOP
   -- fetch from cursor variable
   FETCH c_1 INTO a, b, c;
   -- exit when last row is fetched
   EXIT WHEN c_1%NOTFOUND;

   -- process data record

END LOOP;
```

Contents

Q & A

Exercise



1. Write a trigger for ensuring that the employee's ages must be between18 and 60.
2. Write a trigger to enforce that when an employee has a new project, his or her salary will be increased by 10% * number of hours per week working on that project.
3. Write a store procedure to read an employee's id and print the names of his/her dependents.
4. Write a function to read a project's id and return the total number of employees who work for that project.