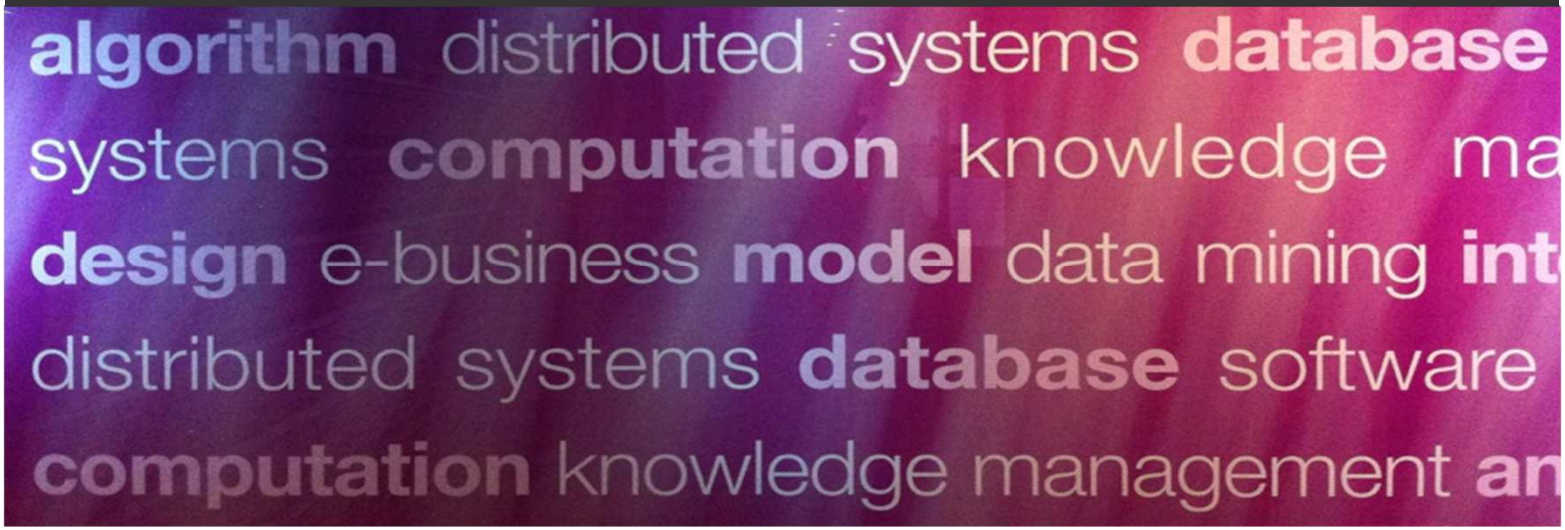
MONASH University

Information Technology

FIT3176 Advanced Database Design

Topic 4 –PL/SQL & Triggers

Dr. Minh Le
Minh.Le@monash.edu



*Adapted from slides developed by Lindsay Smith

Learning Objectives and References

Learning Objectives

By the end of this week you should be able to:

- Code Oracle PL/SQL blocks making use of
  - variables, constants
  - dbms_output
  - raise and raise_application_error, and
  - a range of control structures
- Code Oracle triggers to satisfy a requirements specification

Learning Objectives and References

References

1. Coronel & Morris, Database Systems: Design, Implementation & Management, 11th Edition 2015, Thomson Course Technology. Chapter 8 (8.7 & 8.7.1)
2. Oracle PL/SQL Manual

PL/SQL Basic Structure

- The basic unit of a PL/SQL source program is the ***block***, with the general structure
  - DECLARE (optional)
    - Declaration of variables, etc
  - BEGIN (required)
    - Procedural statements which are to be executed
    - EXCEPTION (optional)
      - Handlers for exceptions raised
  - END;
- An anonymous block, is a PL/SQL block which is not stored in the database – it is compiled each time it is loaded into memory
- Blocks can also be stored in the database within triggers, procedures and functions

PL/SQL Variables and Constants

- Declared within the declare section of the block
- Wide range of data types available, we will make use of
  - SQL Data types (CHAR, VARCHAR2, NUMBER and INTEGER)
  - BOOLEAN PL/SQL data type
  - CURSOR PL/SQL variables used to hold multiple rows from a select

    part_desc VARCHAR2(100);

    part_number NUMBER(6);

    in_stock BOOLEAN;
- Assignment via :=
  - initial values can be set in DECLARE

    max_credit_limit **CONSTANT** NUMBER(7,2) := 50000;

    in_stock BOOLEAN := FALSE;

PL/SQL %TYPE attributes

- Allows you to declare a data item of the same data type as a column or a previously declared variable
  - Referencing item inherits data type and size and constraints (if not column) but not initial value
  - If the referenced column data type is changed then the declared type changes accordingly
  - Particularly useful to hold database values
    - Syntax for declaration is:
      - *referencing_item referenced_item*%TYPE;

```
part_desc  product.prod_description%TYPE

name VARCHAR2(25) NOT NULL := 'Smith';
surname name%TYPE := 'Jones';
```

PL/SQL Output

- Output from PL/SQL can be obtained via a package
  - A PL/SQL package is a collection of related PL/SQL objects, you can write your own and/or make use of Oracle supplied packages such as DBMS_OUTPUT
  - DBMS_OUTPUT ***– main use debugging***
    - DBMS_OUTPUT.put
      - Outputs without line feed
    - DBMS_OUTPUT.put_line
      - Outputs with line feed
    - Data for output is concatenated via ||
      - dbms_output.put_line ('Product description is: ' || part_desc)
  - Must be turned on in SQL Developer
    - View – DBMS Output
      - Select +, select connection

PL/SQL Output

- Output can also be generated by raising an exception
  - RAISE
    - Handle raised exception with coded exception handler
      - eg. Subsitute default value
    - RAISE salary_too_high;
  - RAISE_APPLICATION_ERROR
    - Procedure defined within DBMS_STANDARD package
    - Returns error code and error message to the invoker
      - Error code is an integer in the range -20000 … -20999
      - RAISE_APPLICATION_ERROR(-20000, 'Error Msg');

PL/SQL Control Structures

- IF THEN statement

  IF *condition* THEN

  statements

  END IF;

- IF THEN ELSE statement

  IF *condition* THEN

  statements

  ELSE

  else_statements

  END IF;

- IF THEN ELSE statements can be nested

PL/SQL Control Structures

- IF THEN ELSE alternative structure using IF THEN ELSIF

```
IF condition_1 THEN
        statements_1
ELSIF condition_2 THEN
        statements_2
[ ELSIF condition_3 THEN
        statements_3
]...
[ ELSE
        else_statements
]
END IF;
```

- Easier to understand than equivalent nested IF THEN ELSE statements

PL/SQL Control Structures

```
DECLARE
        grade CHAR(1);
BEGIN
        grade := 'B';
        IF grade = 'A' THEN
            DBMS_OUTPUT.PUT_LINE('Excellent');
        ELSIF grade = 'B' THEN
           DBMS_OUTPUT.PUT_LINE('Very Good');
        ELSIF grade = 'C' THEN
           DBMS_OUTPUT.PUT_LINE('Good');
        ELSIF grade = 'D' THEN
           DBMS_OUTPUT. PUT_LINE('Fair');
        ELSIF grade = 'F' THEN
            DBMS_OUTPUT.PUT_LINE('Poor');
        ELSE
           DBMS_OUTPUT.PUT_LINE('No such grade');
        END IF;
END;
```

PL/SQL Control Structures

- CASE statement also could be used:

```
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');
          WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
          WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
          ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
        END CASE;
END;
```

PL/SQL SELECT … INTO structure

- Standard SQL select statement simply retrieves and displays value/s
- We need to select and store value in a variable so that subsequent PL/SQL code can reference the value
- General form:

  SELECT *select_item* [, *select_item*]...

  INTO *variable_name* [, *variable_name*]...

  FROM *table_name …*;
- For each *select_item* there must be a type compatible *variable_name*

```
declare
 emp_bonus number(8,2);

begin
  select emp_salary * 0.10 into emp_bonus
  from employee
  where emp_no = 1;

  dbms_output.put_line('Bonus for employee 1 is ' ||
                       ltrim(to_char(emp_bonus,'$999999.99')));
end;
/
```

Q1: An anonymous PL/SQL block is called anonymous because:

A. The user running the block is hidden

B. The block has no access to the current declared variables

C. Anonymous is the incorrect term, it should be called a synonymous block

D. The block is not stored in the database

Q2: Declaring a PL/SQL variable using a type attribute such as:

part_desc product.product_description%TYPE

has the advantage/s listed below

A. If the referenced column data type changes then the declared type changes accordingly
B. It allows the variable to take on any suitable type based on the type of data assigned to it
C. It is a quicker way of making nay type of variable declaration
D. It is independent of the database attribute and consequently more efficient
E. None of these
F. More than one of these

Q3: DBMS_OUTPUT is a package supplied by Oracle as part of a standard install. The included procedures *put* and *put_line* are used for

A. Controlling SQL Developer SVN access
B. Displaying data for debugging
C. Displaying data for users
D. To raise application errors via the standard error codes (-20000 … -20999)
E. More than one of the above

Oracle Triggers

- A trigger is PL/SQL code associated with a table, which performs an action when a row in a table is inserted, updated, or deleted.
- Triggers are used to implement some types of data integrity constraints that cannot be enforced at the DBMS design and implementation levels
- A trigger is a stored procedure/code block associated with a table
- Triggers specify a condition and an action to be taken whenever that condition occurs
- The DBMS automatically executes the trigger when the condition is met
- A Trigger can be ENABLE'd or DISABLE'd via the ALTER command
  - ALTER TRIGGER *trigger_name* ENABLE;

```
select cust_order_value / cust_num_orders as avg_order_value
  from custorders
  where cust_no = 2;

declare
  avg_order_value number(6,2);
  req_custno number(3) := 2;

begin

  select cust_order_value / cust_num_orders
  into avg_order_value
  from custorders
  where cust_no = req_custno;

  dbms_output.put_line('Customer Number ' || req_custno ||
       ' has an average order value of ' ||
       ltrim(to_char(avg_order_value,'$999999.99')));

  exception
     when zero_divide then
       dbms_output.put_line('Customer Number ' || req_custno ||
       ' has not placed any orders');
end;
/
```

Oracle Triggers

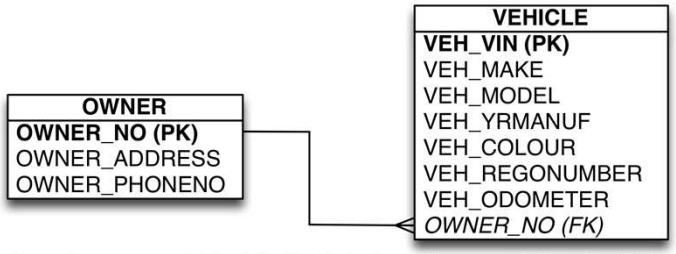
- Use triggers where:
  - a specific operation is performed, to ensure related actions are also performed
  - to enforce integrity where data has been denormalised
  - to maintain an audit trail
  - global operations should be performed, regardless of who performs the operation
  - they do NOT duplicate the functionality built into the DBMS
  - their size is reasonably small (< 50 - 60 lines of code)
- Do not create triggers where:
  - they are recursive
  - they modify or retrieve information from triggering tables

Oracle Triggers

```
CREATE OR REPLACE TRIGGER triggername
BEFORE|AFTER  INSERT|UPDATE [of colname]|DELETE  [OR ...]  ON
    Table
WHEN  condition
REFERENCING ....
FOR EACH ROW
DECLARE
    var_name          datatype [, ...]
BEGIN
    .....
END;
```

- Oracle triggers are fired on modification (insert or update or delete) of a single table – this is a ROW trigger vs STATEMENT trigger
- The triggers may be fired either before or after the table modification.

Common use of triggers



- In the model above OWNER is the PARENT (PK end) and VEHICLE is the CHILD (FK end)
- What should the database do to maintain integrity if the user:
  - attempts to UPDATE the owner_no of the parent
  - attempts to DELETE an owner who still has vehicles in the vehicle table
- Oracle, by default, takes the safe approach
  - UPDATE RESTRICT (no update of PK permitted if child records)
  - DELETE RESTRICT (no delete permitted if child records)
  - what if you as the developer want UPDATE CASCADE?

Oracle Triggers

```
CREATE OR REPLACE TRIGGER Dept_Upd_Cas
BEFORE UPDATE OF deptno ON department
FOR EACH ROW
BEGIN
    UPDATE employee
    SET        deptno = :new.deptno
    WHERE  deptno = :old.deptno;
    DBMS_OUTPUT.PUT_LINE ('Corresponding department number in the EMPLOYEE table
    has also been updated');
END;
/
```

Implement UPDATE CASCADE rule
DEPARTMENT 1 ---- has --- M EMPLOYEE
:new.deptno – value of deptno after update (new value)
:old.deptno – value of deptno before update (old value)

- SQL Window: To CREATE triggers, include the RUN command (/) after the last line of the file
  - Accessing errors and code at the command line (in an SQL Window) requires the use of a range of select statements.
- SQL Developer provides a GUI which is a more powerful environment to work in and debug trigger code.
  - Often use the SQL Window to enter initially and then swap to SQL Developer GUI

Triggering Statement

BEFORE|AFTER INSERT|UPDATE [of colname]|DELETE [OR ...]
ON Table

- The triggering statement specifies:
  - the type of SQL statement that fires the trigger body.
  - the possible options include DELETE, INSERT, and UPDATE. One, two, or all three of these options can be included in the triggering statement specification.
  - the table associated with the trigger.
- Column List for UPDATE
  - if a triggering statement specifies UPDATE, *an optional list of columns can be included in the triggering statement*.
  - if you include a column list, the trigger is fired on an UPDATE statement only when one of the specified columns is updated.
  - if you omit a column list, the trigger is fired when any column of the associated table is updated

Trigger Body

```
BEGIN
    .....
END;
```

- is a PL/SQL block that can include SQL and PL/SQL statements. These statements are executed if the triggering statement is issued and the trigger restriction (if included) evaluates to TRUE.
- Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement. Two correlation names exist for every column of the table being modified: one for the old column value and one for the new column value.

Correlation Names

- Oracle uses two correlation names in conjunction with every column value of the current row being affected by the triggering statement. These are denoted by:

  OLD.ColumnName & NEW.ColumnName

  - For DELETE, only OLD.ColumnName is meaningful
  - For INSERT, only NEW.ColumnName is meaningful
  - For UPDATE, both are meaningful
- A colon must precede the OLD and NEW qualifiers when they are used in a trigger's body, but a colon is not allowed when using the qualifiers in the WHEN clause or the REFERENCING option.
- Old and new values are available in both BEFORE and AFTER row triggers.

FOR EACH ROW Option

- The FOR EACH ROW option determines whether the trigger is a row trigger or a statement trigger. If you specify FOR EACH ROW, the trigger fires once for each row of the table that is affected by the triggering statement. The absence of the FOR EACH ROW option means that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement.

```
CREATE OR REPLACE TRIGGER display_salary_increase
AFTER UPDATE OF empmsal ON employee
FOR EACH ROW
WHEN (new.empmsal > 1000)
BEGIN
   DBMS_OUTPUT.PUT_LINE ( 'Employee:  '|| :new.empno ||' Old salary:
    '|| :old.empmsal ||  ' New salary:  '|| :new.empmsal);
END;
```

FOR EACH ROW Option

- The following trigger fires only once for each UPDATE of the EMP table:

```
CREATE OR REPLACE TRIGGER log_salary_increase
AFTER UPDATE OF empmsal ON employee
BEGIN
   DBMS_OUTPUT.PUT_LINE ( 'Employees salaries were
    updated on '|| SYSDATE);
END;
```

WHEN Clause

- a trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause
- if included, the expression in the WHEN clause is evaluated for each row that the trigger affects. If the expression evaluates to TRUE for a row, the trigger body is fired on behalf of that row. However, if the expression evaluates to FALSE or NOT TRUE (that is, unknown, as with nulls) for a row, the trigger body is not fired for that row.
- For example, in the display_salary_increase trigger, the trigger body would not be executed if the new value of msal is less than or equal to 1000
- The expression in a WHEN clause of a row trigger can include correlation names.

WHEN Clause

- Can be used to limit the scope of a trigger in FOR EACH ROW triggers
- new.attribute and old.attribute do not have a : in the when clause
- For example, when checking if the FK attribute in a child table has a matching PK attribute in the parent table, this only needs to be tested if the FK attribute is not null
- For example:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE INSERT OR UPDATE OF FK_attribute ON child_table
FOR EACH ROW
WHEN (NEW.FK_attribute IS NOT NULL)
DECLARE
   local_counter     NUMBER;
BEGIN
   SELECT COUNT(*) INTO local_counter
      FROM parent_table WHERE …
    …
END;
```

Conditional Predicates

- If more than one type of DML operation can fire a trigger (for example, "ON INSERT OR DELETE OR UPDATE OF employee"), the trigger body can use the conditional predicates INSERTING, DELETING, and UPDATING to execute specific blocks of code, depending on the type of statement that fires the trigger

  E.g. INSERT OR UPDATE ON employee

- Within the code of the trigger body, you can include the following conditions:

IF INSERTING THEN . . . END IF;

IF UPDATING THEN . . . END IF;

Conditional Predicates

- In an UPDATE trigger, a column name can be specified with an UPDATING conditional predicate to determine if the named column is being updated.

```
CREATE OR REPLACE TRIGGER log_salary_increase
AFTER UPDATE OF empsal, empcomm ON employee
FOR EACH ROW
BEGIN
IF UPDATING ( 'empsal' ) THEN
   dbms_output.put_line ( 'Employee:  '|| :new.empno ||' Old salary:  '|| :old.empsal ||
     'New salary:  '|| :new.empsal);
END IF;
IF UPDATING ( 'comm' ) THEN
   dbms_output.put_line ( 'Employee:  '|| :new.empno ||' Old comm:  '|| :old.empcomm ||
     'New comm:  '|| :new.empcomm);
END IF;
END;
```

Error Conditions and Exceptions

- If a predefined or user-defined error condition is raised during the execution of a trigger body, all effects of the trigger body, as well as the triggering statement, are rolled back (unless the error is trapped by an exception handler).
- A trigger body can thus be designed to prevent the execution of the triggering statement by raising an error.

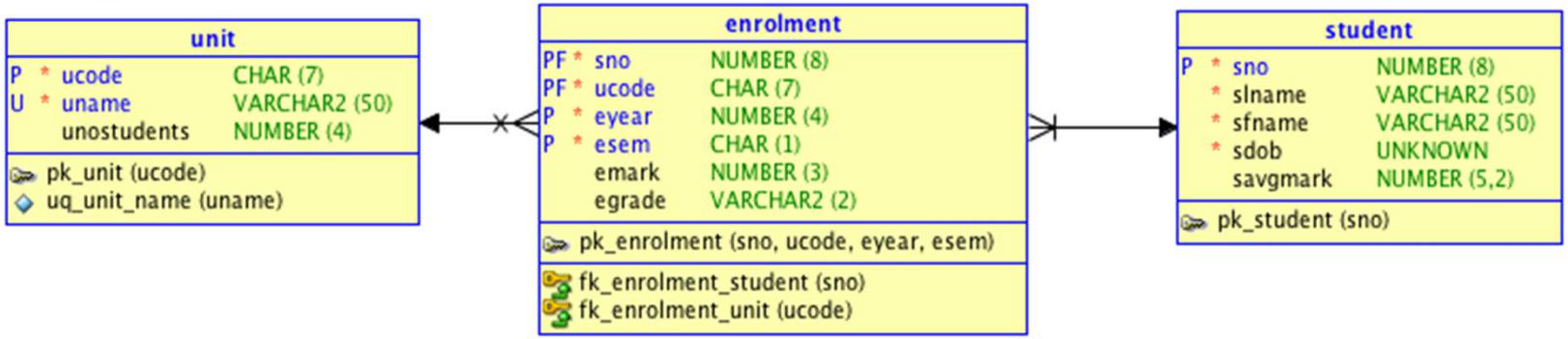
```
CREATE OR REPLACE TRIGGER dept_upd_restrict
BEFORE UPDATE OF deptno ON department
FOR EACH ROW
DECLARE
        emp_count        NUMBER;
BEGIN
        SELECT count(*) INTO emp_count
        FROM employee
        WHERE deptno = :old.deptno;

          IF emp_count > 0 THEN
                RAISE_APPLICATION_ERROR(-20001, 'Cannot update as employees present in'|| ' Department '
                        || TO_CHAR(:old.deptno));
          ELSE
                DBMS_OUTPUT.PUT_LINE ('No employees in department '||:old.deptno|| ' therefore it has
                   been updated');
          END IF;
END;
/
```

Error Conditions and Exceptions

```
CREATE OR REPLACE TRIGGER dept_del_restrict
BEFORE DELETE ON department
FOR EACH ROW
DECLARE
  employees_present EXCEPTION;
  employees_not_present EXCEPTION;
  emp_count NUMBER;
BEGIN
        SELECT count(*) INTO emp_count
        FROM employee WHERE deptno = :old.deptno;
          IF emp_count > 0 THEN
              RAISE employees_present;
            ELSE
              RAISE employees_not_present;
          END IF;
        EXCEPTION
          WHEN employees_present THEN
            RAISE_APPLICATION_ERROR(-20001, 'Cannot delete as employees present in' || ' Department ' ||
             TO_CHAR(:old.deptno));
        WHEN employees_not_present THEN
            DBMS_OUTPUT.PUT_LINE( 'The department has been deleted as no employees are present in'|| ' Department
             ' || TO_CHAR(:old.deptno));
END;
```

You can create your own named exceptions – which can be RAISEd. Control is then transferred to an exception handling routine, EXCEPTION at the end of the PL/SQL block

Case Study



- The student enrolment database contains two derived attributes unostudents (total number of students in a unit) and savgmark (a students average mark).
- The total number of students in a unit is updated when an enrolment is added or deleted.
- The average mark is updated when an update on attribute emark is performed.
- For audit purpose, any deletion of enrolment needs to be recorded. The recorded information includes the username who performed the deletion, the date and time of the deletion, the student no and unit code.

Q4. Based on the rule to maintain the integrity of the unostudents attribute in the UNIT table as well as keeping the audit record, a trigger needs to be created for _____ table. The trigger will update a value on _____ table and insert a row to _____ table.

A. UNIT, ENROLMENT, AUDIT
B. ENROLMENT, UNIT, AUDIT
C. STUDENT, ENROLMENT, AUDIT
D. AUDIT, UNIT, ENROLMENT

Q5. What would be an appropriate condition for the trigger described on the previous slide?

A. BEFORE INSERT OR DELETE ON enrolment.
B. AFTER INSERT OR DELETE ON enrolment.
C. BEFORE UPDATE OF mark ON enrolment.
D. AFTER UPDATE OF mark ON enrolment

```
CREATE OR REPLACE TRIGGER change_enrolment
AFTER INSERT OR DELETE ON ENROLMENT
FOR EACH ROW
DECLARE
      ??????
BEGIN
      ????????

END;
```

Q6. What would be the logic to update the unostudents attribute in the UNIT table when a new row is inserted to ENROLMENT?

A. UPDATE unit
SET unostudents = unostudents + 1
WHERE ucode = unit code of the
inserted row

B. UPDATE unit
SET unostudents = (SELECT count (sno)
FROM enrolment
WHERE ucode= unit code of the
inserted row)
WHERE unitcode = unit code of the
inserted row

C. UPDATE unit
SET unostudents = unostudents -1
WHERE unitcode = unit code of the
inserted row

D. UPDATE unit

```
CREATE OR REPLACE TRIGGER change_enrolment
AFTER INSERT OR DELETE ON ENROLMENT
FOR EACH ROW
DECLARE
      ??????
BEGIN
      IF INSERTING THEN
            UPDATE unit
            SET unostudents = unostudents + 1
            WHERE unitcode = :new.ucode
      ENDIF;
      ?????


END;
```

Q7. What would be the logic for the trigger to deal with a deletion of a row in enrolment? Assume that a table audit_trail contains audit_time, user, sno and unitcode attributes.

A. UPDATE unit
SET unostudents = unostudents -1
WHERE ucode = :old.ucode;

B. INSERT INTO audit_trail VALUES
(SYSDATE, USER, :old.sno, :old.ucode);

C. UPDATE unit
SET unostudents = unostudents – 1
WHERE unitcode = :new.ucode;

D. a and b.

E. b and c.

```
CREATE OR REPLACE TRIGGER change_enrolment
AFTER INSERT OR DELETE ON ENROLMENT
FOR EACH ROW
DECLARE
      ??????
BEGIN
      IF INSERTING THEN
            UPDATE unit
            SET unostudents = unostudents + 1
            WHERE ucode = :new.ucode;
      END IF;
      IF DELETING THEN
            UPDATE unit
            SET unostudents = unostudents -1
            WHERE ucode = :old.ucode;

            INSERT INTO audit_trail VALUES (SYSDATE,
                                                USER, :old.sno, :old.ucode);
      END IF;
END;
```

What is the difference here?

```
create or replace
TRIGGER UPDATE_STATEMENT
AFTER UPDATE ON ENROLMENT
BEGIN
INSERT INTO enrol_history VALUES (SYSDATE, USER, 'updating');
END;
```

```
create or replace
TRIGGER UPDATE_ENROLMENT
AFTER UPDATE ON ENROLMENT
FOR EACH ROW
BEGIN
   INSERT INTO audit_trail VALUES
            (SYSDATE, USER, :old.sno, :old.ucode);
END;
```

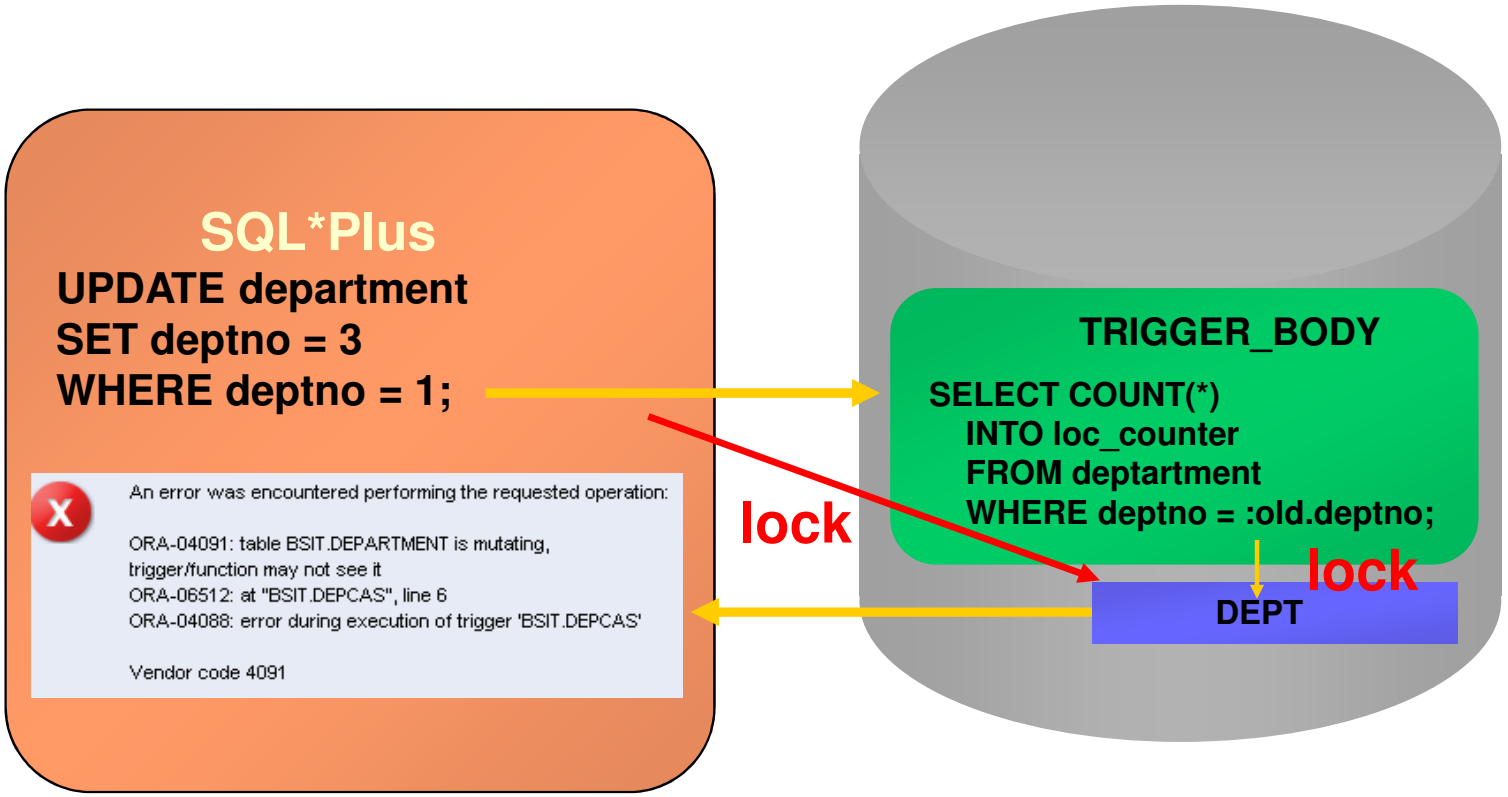
Mutating Table

- A table that is currently being modified through an INSERT, DELETE or UPDATE statement SHOULD NOT (***CANNOT***) be read from or written to by a row trigger because it is in a transition state between two stable states (before and after) where data integrity cannot be guaranteed.
  - Such a table is called a mutating table.
  - The ENROLMENT table is a mutating table in our example.
- How then to manage update of SAVGMARK on STUDENT?
  - A statement level trigger ?
    - AFTER UPDATE of ENROL_MARK on ENROLMENT
    - Highly inefficient
  - REALISE that triggers have their place but should be carefully evaluated to determine if they are the best approach in the given scenario

Mutating Table Example

```
CREATE OR REPLACE TRIGGER DEPCAS
BEFORE UPDATE OF DEPTNO ON DEPARTMENT
FOR EACH ROW
DECLARE
  Loc_Counter number;
BEGIN
 SELECT COUNT(*) INTO Loc_Counter
     FROM department
     WHERE deptno = :old.deptno;

  IF (Loc_Counter = 0) THEN
     DBMS_OUTPUT.PUT_LINE ('Not a valid department');
  ELSE
     UPDATE employee
        SET deptno = :new.deptno
        WHERE  deptno = :old.deptno;
     DBMS_OUTPUT.PUT_LINE ('Corresponding department records in the EMP table
   have also been updated');
  END IF;
END;
```

Mutating Tables



Trigger activity

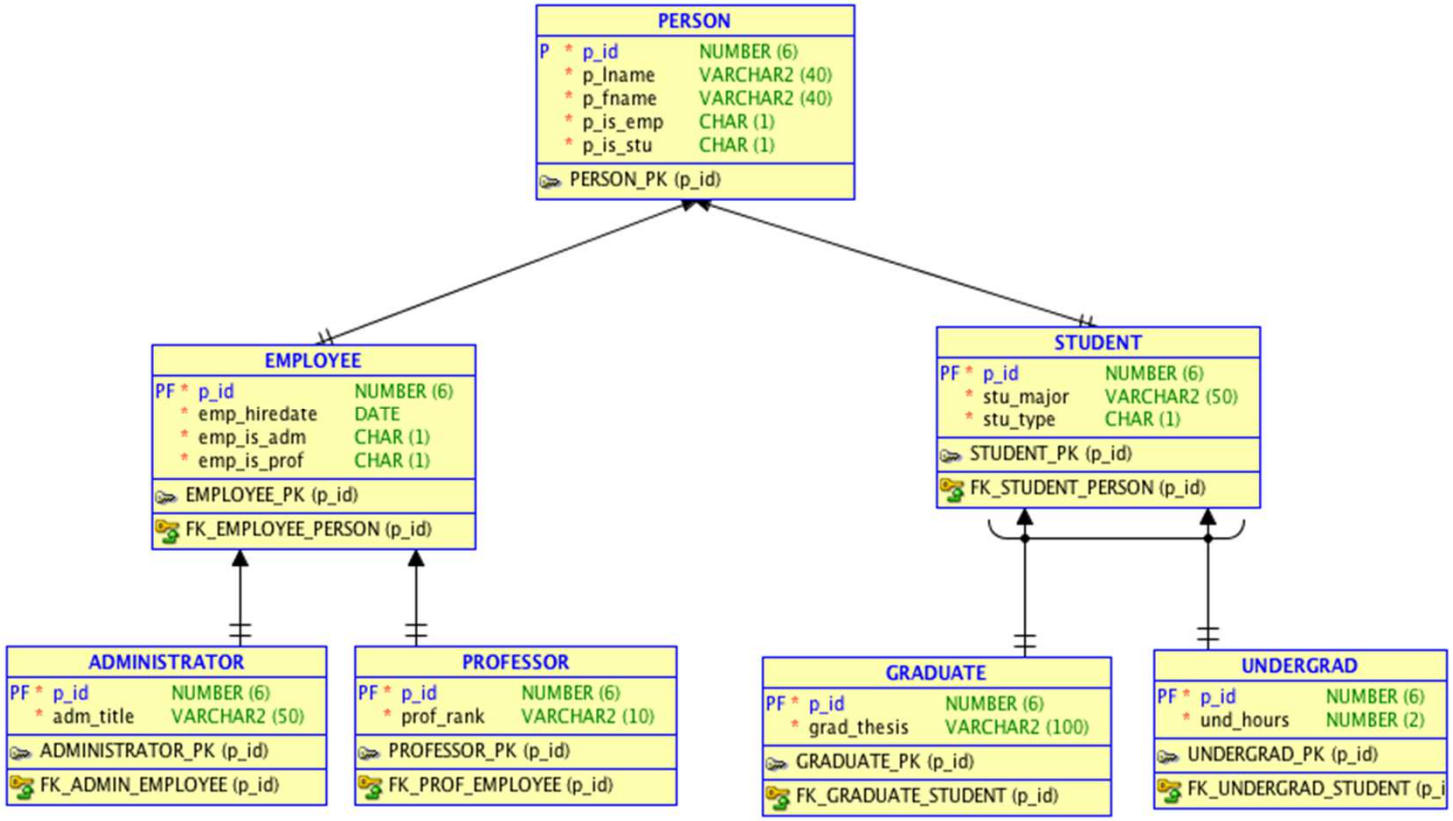
- Triggers are executed as part of the transaction which fires them
  - triggers do not contain *commit* – this is carried out by the managing transaction code
    - script contains

      insert into ENROLMENT values ('11111121','FIT2077', 2016,'1',null,null);

      » Trigger fires here

      commit;
- Trigger activity is managed by the SQL ALTER command
  - ALTER TRIGGER change_enrolment [ENABLE | DISABLE];
    - or
  - ALTER TABLE enrolment [ENABLE ALL| DISABLE ALL] TRIGGERS;

Trigger to validate *entry* of PROFESSOR?



```
CREATE OR REPLACE TRIGGER chk_professor before
    INSERT
     OR
    UPDATE
        OF p_id ON professor FOR EACH row DECLARE emp_is CHAR(1);

    BEGIN
        SELECT
            e.emp_is_prof
        INTO
            emp_is
        FROM
            employee e
        WHERE
            e.p_id = :new.p_id ;

        IF emp_is <> 'Y' THEN
            raise_application_error(-20001, 'EMPLOYEE is not a PROFESSOR' );
        END IF;

    END;
    /
```

Summary

- Discussed elements of Oracle PL/SQL language
  - basic structure, variables, constants
  - %TYPE attributes,
  - control structures (if … elsif … else … end if; if … then, etc.)
  - select … into structure
- Discussed PL/SQL Output using DBMS_OUTPUT package
- Discussed when to use and how to code Oracle triggers
- Discussed mutating table and its limitations
- Made use of raise and raise_application_error Oracle constructs for handling errors