GALGOTIAS UNIVERSITY

School of Computing Science and Engineering

Program: BCA - IOP

Course Code: BCAS3031

Course Name: PL/SQL & Cursors and Triggers

Dr. T. Poongodi

Associate Professor

Architecture of PL/SQL

The PL/SQL architecture mainly consists of following three components:

- PL/SQL block
- PL/SQL Engine
- Database Server

PL/SQL block:

- Actual PL/SQL code
- This consists of different sections to divide the code logically
  - declarative section for declaring purpose
  - execution section for processing statements
  - exception handling section for handling errors
- It also contains the SQL instruction that used to interact with the database server.
- All the PL/SQL units are treated as PL/SQL blocks

The different type of PL/SQL units.

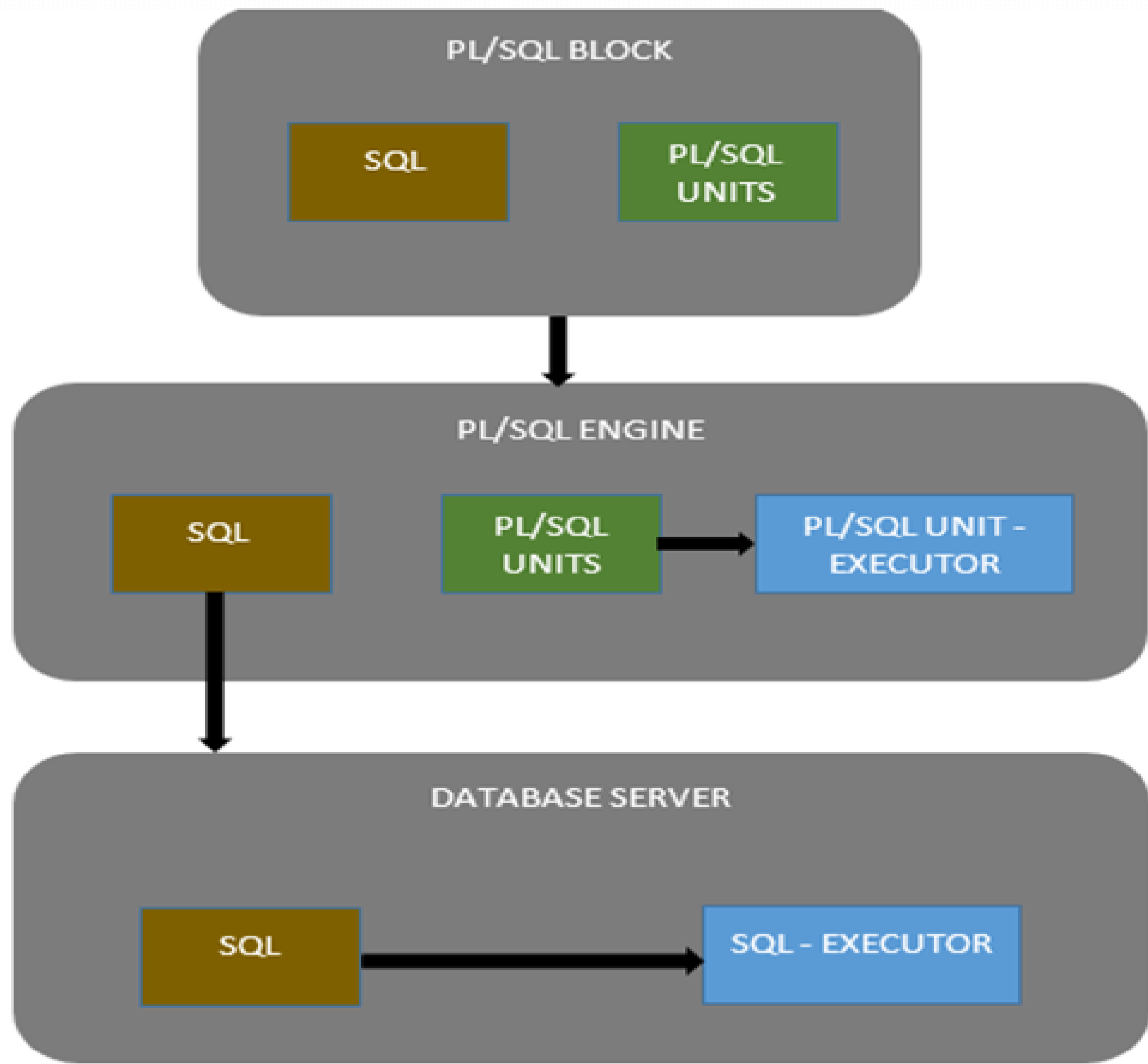
- Anonymous Block
- Function
- Library
- Procedure
- Package Body
- Package Specification
- Trigger
- Type
- Type Body

PL/SQL Engine

- PL/SQL engine is the component where the actual processing of the codes takes place.
- PL/SQL engine separates PL/SQL units and SQL part in the input.
- The separated PL/SQL units will be handled by the PL/SQL engine itself.
- The SQL part will be sent to database server where the actual interaction with database takes place.
- It can be installed in both database server and in the application server.

Database Server:

- This is the most important component of PL/SQL unit which stores the data.
- The PL/SQL engine uses the SQL from PL/SQL units to interact with the database server.
- It consists of SQL executor which parses the input SQL statements and execute the same.


PL/SQL BLOCK
SQL
PL/SQL UNITS
PL/SQL ENGINE
SQL
PL/SQL UNITS
PL/SQL UNIT - EXECUTOR
DATABASE SERVER
SQL
SQL - EXECUTOR

Advantages of using PL/SQL

- Better performance, as SQL is executed in bulk rather than a single statement
- High Productivity
- Tight integration with SQL
- Full Portability
- More Secured
- Support Object Oriented Programming concepts.

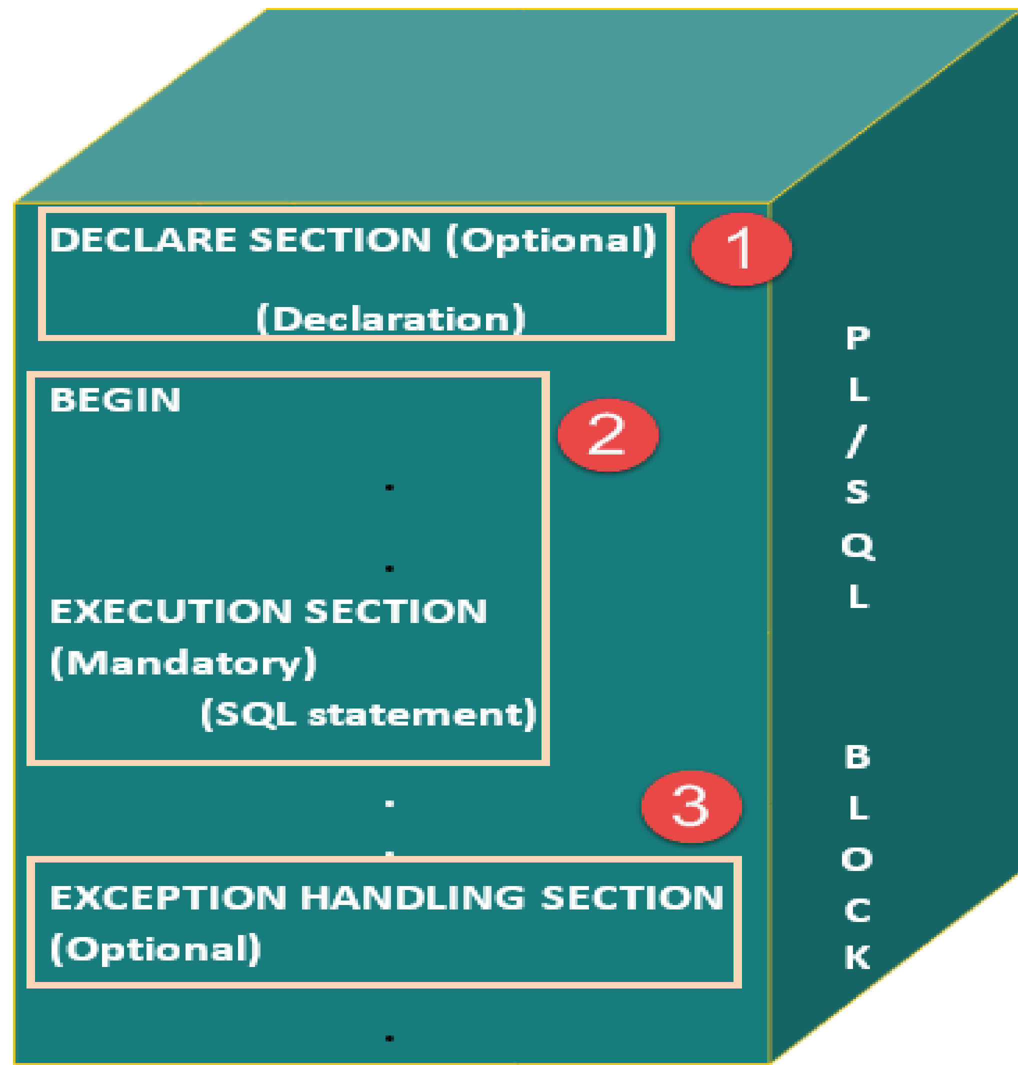
PL/SQL block

- In PL/SQL, the code is not executed in single line format, but it is always executed by grouping the code into a single element called Blocks.
- Blocks contain both PL/SQL as well as SQL instruction.
- All these instruction will be executed as a whole rather than executing a single instruction at a time.

Block Structure

PL/SQL blocks have a pre-defined structure in which the code is to be grouped. Below are different sections of PL/SQL blocks.

- Declaration section
- Execution section
- Exception-Handling section


DECLARE SECTION (Optional)
(Declaration)
1
BEGIN
2
EXECUTION SECTION
(Mandatory)
(SQL statement)
3
EXCEPTION HANDLING SECTION
(Optional)
PL/SQL BLOCK

Declaration Section

Declaration of variables, cursors, exceptions, subprograms, instructions and collections that are needed in the block will be declared. Some characteristics,

- This particular section is optional and can be skipped if no declarations are needed.
- This should be the first section in a PL/SQL block, if present.
- This section starts with the keyword 'DECLARE' for triggers and anonymous block. For other subprograms, this keyword will not be present. Instead, the part after the subprogram name definition marks the declaration section.
- This section should always be followed by execution section.

Execution Section

- Execution part is the main and mandatory part which actually executes the code that is written inside it. Since the PL/SQL expects the executable statements from this block this cannot be an empty block, i.e., it should have at least one valid executable code line in it. Some characteristics,
- This can contain both PL/SQL code and SQL code.
- This can contain one or many blocks inside it as a nested block.
- This section starts with the keyword 'BEGIN'.
- This section should be followed either by 'END' or Exception-Handling section (if present)

Exception-Handling Section:

- The exception is unavoidable in the program which occurs at run-time
- This is the section where the exception raised in the execution block is handled.
- This section is the last part of the PL/SQL block.
- Control from this section can never return to the execution block.
- This section starts with the keyword 'EXCEPTION'.
- This section should always be followed by the keyword 'END'.
- The Keyword 'END' marks the end of PL/SQL block.

Syntax of PL/SQL Block Structure:

```
DECLARE         --optional
        <declarations>
        .
        .
        .
BEGIN           --mandatory
        <executable statements. At least one executable statement is mandatory>
        .
        .
        .
EXCEPTION       --optional
        <exception handler>
        .
        .
        .
END;            --mandatory
/
```

```
DECLARE --optional
    <declarations>

BEGIN   --mandatory
    <executable statements. At least one executable statement
is mandatory>

EXCEPTION --optional
    <exception handles>

END;   --mandatory
/
```

A block should always be followed by '/' which sends the information to the compiler about the end of the block.

Types of PL/SQL block

- Anonymous blocks
- Named Blocks

Anonymous blocks:

- Anonymous blocks are PL/SQL blocks which do not have any names assigned to them.
- They need to be created and used in the same session because they will not be stored in the server as database objects.
- Since they need not store in the database, they need no compilation steps.
- They are written and executed directly, and compilation and execution happen in a single process.

Characteristics of Anonymous blocks,

- Don't have any reference name specified for them.
- These blocks start with the keyword 'DECLARE' or 'BEGIN'.
- Since these blocks do not have any reference name, these cannot be stored for later purpose. They shall be created and executed in the same session.
- They can call the other named blocks, but call to anonymous block is not possible as it is not having any reference.
- It can have nested block in it which can be named or anonymous. It can also be nested in any blocks.
- These blocks can have all three sections of the block, in which execution section is mandatory, the other two sections are optional.

Named blocks:

- Named blocks have a specific and unique name for them.
- They are stored as the database objects in the server. Since they are available as database objects, they can be referred to or used as long as it is present on the server.
- The compilation process for named blocks happens separately while creating them as a database objects.

Some characteristics of Named blocks are,

- These blocks can be called from other blocks.
- The block structure is same as an anonymous block, except it will never start with the keyword 'DECLARE'. Instead, it will start with the keyword **'CREATE'** which instruct the compiler to create it as a database object.
- These blocks can be nested within other blocks. It can also contain nested blocks.

Named blocks are basically of two types:

- Procedure
- Function

PL/SQL %TYPE Attribute

- The %TYPE attribute allow you to declare a constant, variable, or parameter to be of the same data type as previously declared variable, record, nested table, or database column.

Syntax:

```
identifier Table.column_name%TYPE;
```

```
declare
v_name employee.lastname%TYPE;
v_dep  number;
v_min_dep v_dep%TYPE:=31;
begin
select  lastname into  v_name from  EMPLOYEE
where  DEPARTMENTID=v_min_dep;
DBMS_OUTPUT.PUT_LINE('v_name: '||v_name);
end;
```

- 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.
- A qualified column name in dot notation or the name of a previously declared variable must be specified as a prefix to the %TYPE attribute.
- The data type of this column or variable is assigned to the variable being declared.
- The %TYPE attribute can also be used with formal parameter declarations.

Example:

A procedure that queries the EMP table using an employee number, displays the employee's data, finds the average salary of all employees in the department to which the employee belongs, and then compares the chosen employee's salary with the department average.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno          IN CHAR(6)
)
IS
    v_lastname       VARCHAR2(15);
    v_job            VARCHAR2(8);
    v_hiredate       DATE;
    v_salary         NUMBER(9,2);
    v_workdept       CHAR(3);
    v_avgsal         NUMBER(9,2);

BEGIN
    SELECT lastname, job, hiredate, salary, workdept
        INTO v_lastname, v_job, v_hiredate, v_salary,
v_workdept
        FROM emp WHERE empno = p_empno;
```

```
DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
DBMS_OUTPUT.PUT_LINE('Name       : ' || v_lastname);
DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_salary);
DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_workdept);
```

```
SELECT AVG(salary) INTO v_avgsal
   FROM emp WHERE workdept = v_workdept;

IF v_salary > v_avgsal THEN
   DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than
the department '|| 'average of ' || v_avgsal);

ELSE

   DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed
the department '|| 'average of ' || v_avgsal);

   END IF;
END
```

Example:

A procedure could be rewritten without explicitly coding the EMP table data types in the declaration section

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno             IN emp.empno%TYPE
)
IS
    v_lastname          emp.lastname%TYPE;
    v_job               emp.job%TYPE;
    v_hiredate          emp.hiredate%TYPE;
    v_salary            emp.salary%TYPE;
    v_workdept          emp.workdept%TYPE;
    v_avgsal            v_salary%TYPE;
BEGIN
    SELECT lastname, job, hiredate, salary, workdept
        INTO v_lastname, v_job, v_hiredate, v_salary,
v_workdept
        FROM emp WHERE empno = p_empno;
```

```
DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
DBMS_OUTPUT.PUT_LINE('Name       : ' || v_lastname);
DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_salary);
DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_workdept);
```

```
 SELECT AVG(salary) INTO v_avgsal
        FROM emp WHERE workdept = v_workdept;
 IF v_salary > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more
than the department '|| 'average of ' || v_avgsal);
 ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not
exceed the department '|| 'average of ' || v_avgsal);
    END IF;
END
```

The p_empno parameter is an example of a formal parameter that is defined using the %TYPE attribute. The v_avgsal variable is an example of the %TYPE attribute referring to another variable instead of a table column.

The following sample output is generated by a call to the EMP_SAL_QUERY procedure:

```
SET SERVEROUTPUT ON@
CALL emp_sal_query('200340')@

Employee # : 200340
Name       : ALONZO
Job        : FIELDREP
Hire Date  : 1997-07-05-00.00.00
Salary     : 31840
Dept #     : E21
Employee's salary does not exceed the department average
of 47086.67
```

SQL SEQUENCES

- Sequence is a set of integers 1, 2, 3, … that are generated and supported by some database systems to produce unique values on demand.
- A sequence is a user defined schema bound object that generates a **sequence of numeric values**.
- Sequences are frequently used in many databases because many applications require **each row in a table to contain a unique value** and sequences provides an easy way to generate them.
- The sequence of numeric values is generated in an a**scending or descending order** at defined intervals and can be configured to restart when exceeds max_value.

```
CREATE SEQUENCE sequence_name
START WITH initial_value
INCREMENT BY increment_value
MINVALUE minimum value
MAXVALUE maximum value
CYCLE|NOCYCLE ;
```

sequence_name: Name of the sequence.

initial_value: starting value from where the sequence starts.
Initial_value should be greater than or equal to minimum value and less than equal to maximum value.

increment_value: Value by which sequence will increment itself.
Increment_value can be positive or negative.

minimum_value: Minimum value of the sequence.
maximum_value: Maximum value of the sequence.

cycle: When sequence reaches its set_limit
It starts from beginning.

nocycle: An exception will be thrown
if sequence exceeds its max_value.

Sequence query creating sequence in ascending order.

```
CREATE SEQUENCE sequence_1
start with 1
increment by 1
minvalue 0
maxvalue 100
cycle;
```

- Query will create a sequence named *sequence_1*.
- Sequence will start from 1 and will be incremented by 1 having maximum value 100.
- Sequence will repeat itself from start value after exceeding 100.

Sequence query creating sequence in descending order.

```
CREATE SEQUENCE sequence_2
start with 100
increment by -1
minvalue 1
maxvalue 100
cycle;
```

- Query will create a sequence named *sequence_2*.
- Sequence will start from 100 and should be less than or equal to maximum value and will be incremented by -1 having minimum value 1.

Example to use sequence : create a table named students with columns as id and name

```
CREATE TABLE students
(
ID number(10),
NAME char(20)
);
```

Insert values into table:

```
INSERT into students
VALUES(sequence_1.nextval,'Ramesh’);

INSERT into students
VALUES(sequence_1.nextval,'Suresh');
```

where *sequence_1.nextval* will insert id's in id column in a sequence as defined in sequence_1.

Output:

```
________________________
| ID | NAME |
-------------------------
| 1 | Ramesh |
| 2 | Suresh |
-----------------------
```

Thank You