PL/SQL : INTRODUCTION

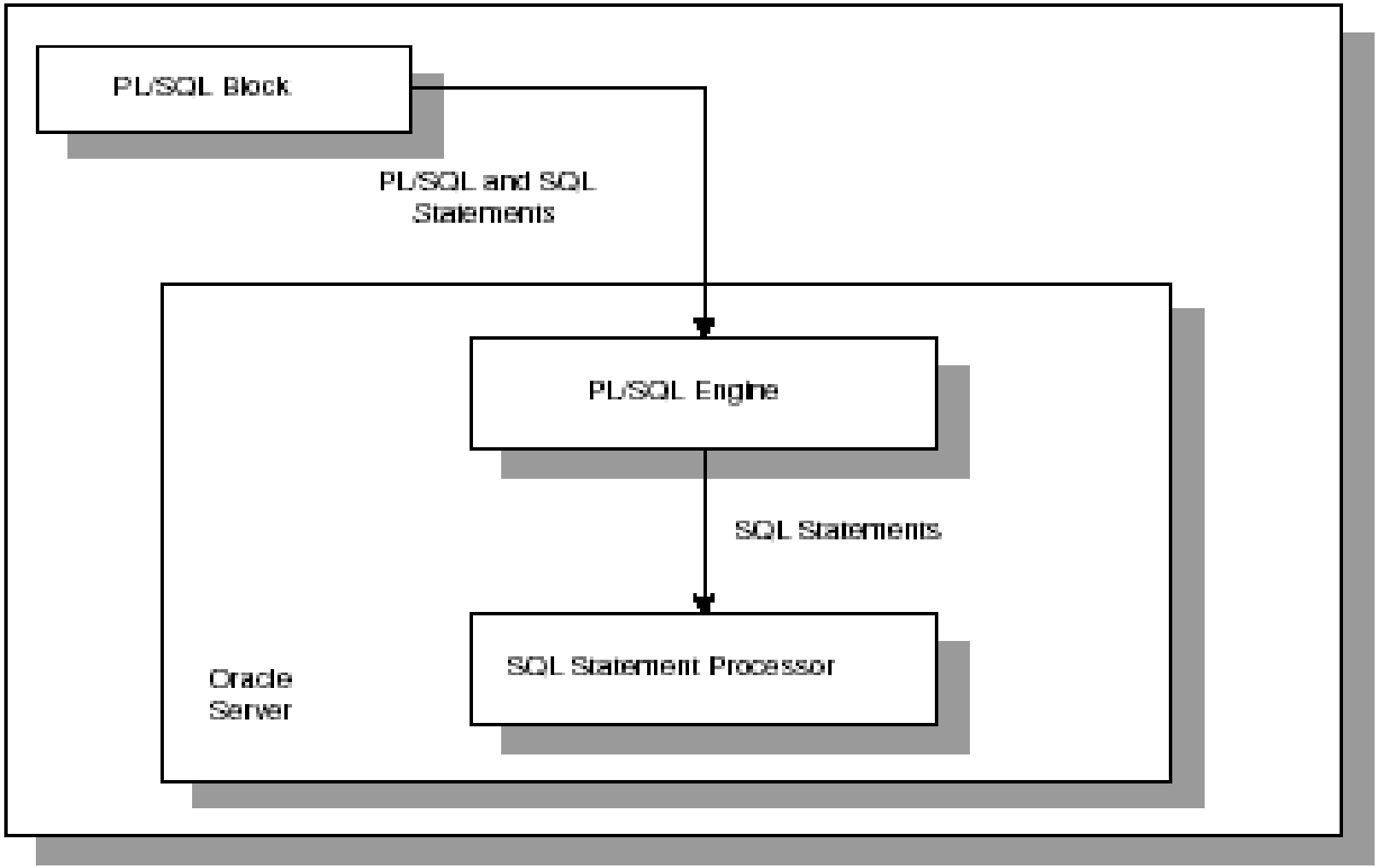

Figure 2.1 ■ The PL/SQL engine and Oracle server.

Comparison of SQL*PLUS and PL/SQL

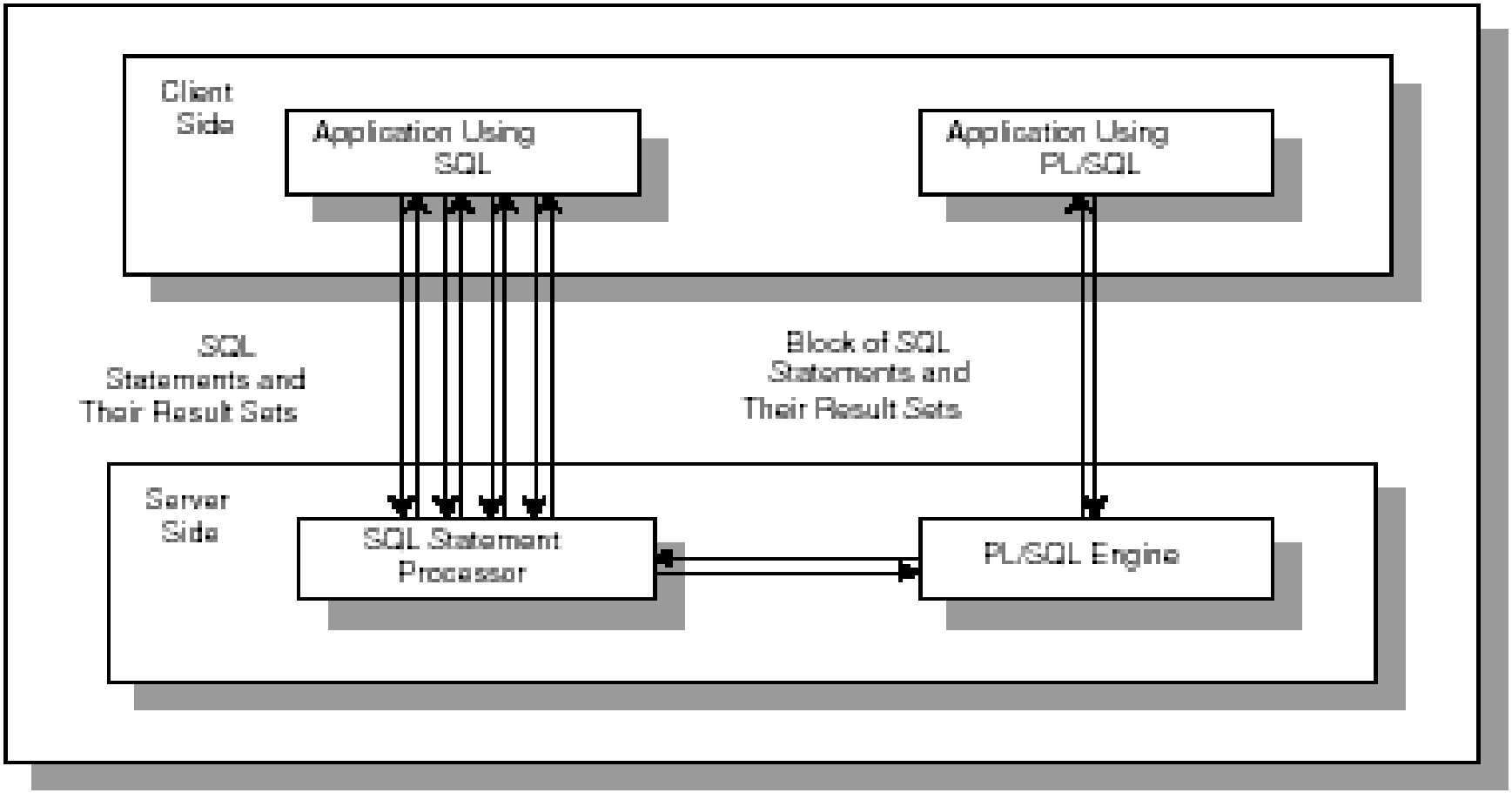

Figure 2.2 ■ PL/SQL in client-server architecture.

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
EXCETION
        Exception-handling statements
END ;
```

Comments:

Not executed by interpreter
Enclosed between /* and */
On one line beginning with --

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
        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);
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;
```

HOW PL/SQL GETS EXECUTED

- Every time an anonymous block is executed, the code is sent to the PL/SQL engine on the server where it is compiled.
- The named PL/SQL block is compiled only at the time of its creation, or if it has been changed.
- The compilation process includes syntax checking, binding and p-code generation.
- Syntax checking involves checking PL/SQL code for syntax or compilation errors.
- Once the programmer corrects syntax errors, the compiler can assign a storage address to program variables that are used to hold data for Oracle. This process is called ***Binding.***

HOW PL/SQL GETS EXECUTED

- After binding, p-code is generated for the PL/SQL block.
- P-code is a list of instructions to the PL/SQL engine.
- For named blocks, p-code is stored in the database, and it is used the next time the program is executed.
- Once the process of compilation has completed successfully, the status for a named PL/SQL block is set to VALID, and also stored in the database.
- If the compilation process was not successful, the status for a named PL/SQL block is set to INVALID.

SQL EXAMPLE

```
SELECT first_name, last_name
FROM student;
```

- The semicolon terminates this SELECT statement. Therefore, as soon as you type semicolon and hit the ENTER key, the result set is displayed to you.

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;
```

- Bordoloi and Bock

/

PL/SQL EXAMPLE

- There are two additional lines at the end of the block containing “.” and “/”. The “.” marks the end of the PL/SQL block and is optional.
- The “/” executes the PL/SQL block and is required.
- When SQL*Plus reads SQL statement, it knows that the semicolon marks the end of the statement. Therefore, the statement is complete and can be sent to the database.
- When SQL*Plus reads a PL/SQL block, a semicolon
- marks the end of the individual statement within the block. In other words, it is not a block terminator.

PL/SQL EXAMPLE

- Therefore, SQL*Plus needs to know when the block has ended. As you have seen in the example, it can be done with period and forward slash.

EXECUTING PL/SQL

PL/SQL can be executed directly in SQL*Plus. A PL/SQL program is normally saved with an .sql extension. To execute an anonymous PL/SQL program, simply type the following command at the SQL prompt:

SQL> @DisplayAge

GENERATING OUTPUT

Like other programming languages, PL/SQL provides a procedure (i.e. PUT_LINE) to allow the user to display the output on the screen. For a user to able to view a result on the screen, two steps are required.

First, before executing any PL/SQL program, type the following command at the SQL prompt (Note: you need to type in this command only once for every SQL*PLUS session):

```
SQL>    SET SERVEROUTPUT ON;
```

or put the command at the beginning of the program, right before the declaration section.

GENERATING OUTPUT

Second, use **DBMS_OUTPUT.PUT_LINE** in your executable section to display any message you want to the screen.

Syntax for displaying a message:

DBMS_OUTPUT.PUT_LINE(<string>);

in which PUT_LINE is the procedure to generate the output on the screen, and DBMS_OUTPUT is the package to which the PUT_LINE belongs.

DBMS_OUTPUT_PUT_LINE('My age is ' || num_age);

SUBSTITUTIONVARIABLES

- SQL*Plus allows a PL/SQL block to receive input information with the help of substitution variables.
- Substitution variables cannot be used to output the values because no memory is allocated for them.
- SQL*Plus will substitute a variable before the PL/SQL block is sent to the database.
- Substitution variables are usually prefixed by the ampersand(&) character or double ampersand (&&) character.

EXAMPLE

```
DECLARE
        v_student_id NUMBER := &sv_student_id;
        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 = v_student_id;
        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 such student');
END;
```

EXAMPLE

- When this example is executed, the user is asked to provide a value for the student ID.
- The example shown above uses a single ampersand for the substitution variable.
- When a single ampersand is used throughout the PL/SQL block, the user is asked to provide a value for each occurrence of the substitution variable.

EXAMPLE

```
BEGIN
      DBMS_OUTPUT.PUT_LINE('Today is '||'&sv_day');
      DBMS_OUTPUT.PUT_LINE('Tomorrow will be '||' &sv_day');
END;
```

This example produces the following output:

```
Enter value for sv_day: Monday
old 2: DBMS_OUTPUT.PUT_LINE('Today is '||' &sv_day');
new 2: DBMS_OUTPUT.PUT_LINE('Today is '||' Monday');
Enter value for sv_day: Tuesday
old 3: DBMS_OUTPUT.PUT_LINE('Tomorrow will be '||' &sv_day');
new 3: DBMS_OUTPUT.PUT_LINE('Tomorrow will be '||' Tuesday');
Today is Monday
Tomorrow will be Tuesday
PL/SQL procedure successfully completed.
```

EXAMPLE

- When a substitution variable is used in the script, the output produced by the program contains the statements that show how the substitution was done.
- If you do not want to see these lines displayed in the output produced by the script, use the SET command option before you run the script as shown below:

SET VERIFY OFF;

EXAMPLE

- Then, the output changes as shown below:

Enter value for sv_day: Monday
Enter value for sv_day: Tuesday
Today is Monday
Tomorrow will be Tuesday
PL/SQL procedure successfully completed.

- The substitution variable sv_day appears twice in this PL/SQL block. As a result, when this example is run, the user is asked twice to provide the value for the same variable.

Variables and Data Types

- Variables
  - Used to store numbers, character strings, dates, and other data values
  - Avoid using keywords, table names and column names as variable names
  - Must be declared with data type before use: *variable_name data_type_declaration;*

Scalar Data Types

- Represent a single value

Data Type	Description	Sample Declaration
VARCHAR2	Variable-length character string	`current_s_last VARCHAR2(30);`
CHAR	Fixed-length character string	`student_gender CHAR(1);`
DATE	Date and time	`todays_date DATE;`
INTERVAL	Time interval	`curr_time_enrolled INTERVAL YEAR TO MONTH;` `curr_elapsed_time INTERVAL DAY TO SECOND;`
NUMBER	Floating-point, fixed-point, or integer number	`current_price NUMBER(5,2);`

Table 4-2 Scalar database data types

Scalar Data Types

Data Type	Description	Sample Declaration
Integer number subtypes (BINARY_INTEGER, INTEGER, INT, SMALLINT)	Integer	`counter BINARY_INTEGER;`
Decimal number subtypes (DEC, DECIMAL, DOUBLE PRECISION, NUMERIC, REAL)	Numeric value with varying precision and scale	`student_gpa REAL;`
BOOLEAN	True/False value	`order_flag BOOLEAN;`

Table 4-3 General scalar data types

Composite Variables

- Composite variables
  - RECORD: contains multiple scalar values, similar to a table record
  - TABLE: tabular structure with multiple columns and rows
  - VARRAY: variable-sized array

Reference Variables

- Reference variables
  - Directly reference a specific database field or record and assume the data type of the associated field or record
  - %TYPE: same data type as a database field
  - %ROWTYPE: same data type as a database record

Arithmetic Operators

Operator	Description	Example	Result
**	Exponentiation	2 ** 3	8
* /	Multiplication Division	2 * 3 9/2	6 4.5
+ -	Addition Subtraction	3 + 2 3 - 2	5 1
-	Negation	-5	-5

Table 4-5 PL/SQL arithmetic operators in describing order of precedence

Assignment Statements

- Assigns a value to a variable
- *variable_name* := *value*;
- Value can be a literal:
  - current_s_first_name := 'John';
- Value can be another variable:
  - current_s_first_name := s_first_name;

```
Set serveroutput on
DECLARE

  v_inv_value  number(10,2);
   v_price          number(8,2)  :=  10.25;
  v_quantitynumber(8,0)  :=  400;

BEGIN
  v_inv_value := v_price * v_quantity;
  dbms_output.put('The value is: ');
  dbms_output.put_line(v_inv_value);
END;
/
```

```
Set serveroutput on

Accept p_price Prompt 'Enter the Price: ‘

DECLARE
  v_inv_value  number(8,2);
  v_price                number(8,2);
  v_quantity   number(8,0)  :=  400;

BEGIN
   v_price := &p_price;
   v_inv_value := v_price * v_quantity;
   dbms_output.put_line('******');
   dbms_output.put_line('price * quantity=');
   dbms_output.put_line(v_inv_value);
END;
/
```

Conditional Statements

- IF … Then … ELSE

```
If <condition1>  Then
        <Code>
ELSIF <Condition2> Then
        <Code>
ELSE
        <Code>
END IF;
```

- Note here that for one IF we only need one END IF;
- No END IF is required for ELSIF i.e for one set of IF condition only one END IF; is required

Conditional Statements

- IF … Then … ELSE

```
If  v_deptno = 10  Then
        DBMS_OUTPUT.PUT_LINE ('Accounting');
ELSIF v_deptno = 20 Then
        DBMS_OUTPUT.PUT_LINE (‘ESG');
ELSE
        DBMS_OUTPUT.PUT_LINE (‘Invalid');
END IF;
```

Conditional Statements

- CASE : This is available from ORACLE 8i onwards only , not in ORACLE 8 and version prior to that.

```
CASE
 WHEN <Variable> = <Value1> Then
        <Code>
WHEN <Variable> =  <Value2> Then
        <Code>
ELSE
        <Code>

END CASE;
```

Conditional Statements

- CASE :

```
CASE
      When v_deptno  =10  Then
                DBMS_OUTPUT.PUT_LINE ('Accounting');
      When v_deptno  =20  Then
                DBMS_OUTPUT.PUT_LINE (‘ESG');
      ELSE
                 DBMS_OUTPUT.PUT_LINE (‘Invalid');
END CASE;
```

TYPES OF LOOPS

- Simple Loop

  Loop

  Exit When <Condition>

  <Code>

  End Loop;

- Exit when is required to give the condition to end the loop

- It is pre tested as condition is checked first and then code is executed

TYPES OF LOOPS

- Simple Loop

```
Loop
        Exit When i = 10
        dbms_output.put_line (i);
End Loop;
```

--Pre Tested

TYPES OF LOOPS

- Simple Loop

```
Loop
        <Code>
        Exit When <Condition>
End Loop;
```

- Exit when is required to give the condition to end the loop
- It is post tested as condition is checked after the code is executed

TYPES OF LOOPS

- Simple Loop

```
Loop
        dbms_output.put_line (i);
        Exit When i = 10
End Loop;
```

--Post Tested

TYPES OF LOOPS

- While Loop

  While <Condition>
  Loop
  <Code>
  End Loop;

- While is required for condition to end the Loop
- This is also pre tested.

TYPES OF LOOPS

- While Loop

```
While i < 10
Loop
      dbms_output.put_line (i);
End Loop;
```

TYPES OF LOOPS

- FOR Loop

  FOR <Variable> IN <Min> .. <Max>
  Loop
  <Code>
  End Loop;

- This Loop is used when we know the number of time the loop is to be executed.
- This is also pre tested.

TYPES OF LOOPS

- FOR Loop

```
FOR  i IN 1 .. 100
Loop
        <Code>
End Loop;
```

- This Loop will execute the given code 100 times for i = 1 to 100

TYPES OF LOOPS

- FOR Loop Reverse

```
FOR  i IN Reverse 1 .. 100
Loop
        <Code>
End Loop;
```

- This Loop will execute the given code 100 times for i = 100 to 1
- This is reverse i.e from last value to first value

Cursors

- Pointer to a memory location that the DBMS uses to process a SQL query
- Use to retrieve and manipulate database data

Implicit Cursor

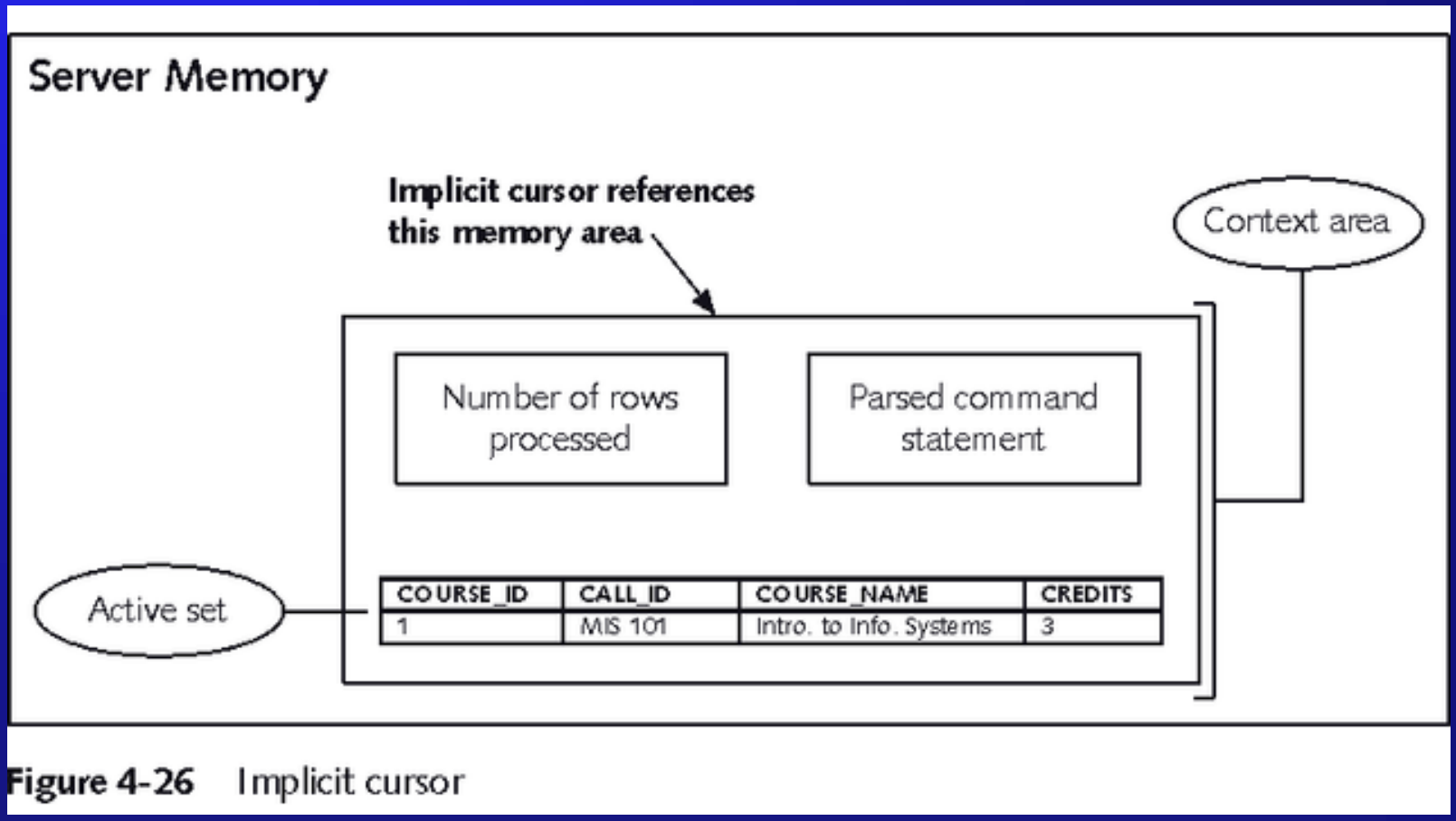

Figure 4-26 Implicit cursor

Using an Implicit Cursor

- Executing a SELECT query creates an implicit cursor
- To retrieve it into a variable use INTO:
  - SELECT *field1, field2, ...*
    INTO *variable1, variable2, ...*
    FROM *table1, table2, ...*
    WHERE *join_ conditions*
    AND *search_condition_to_retrieve_1_record;*
- Can only be used with queries that return exactly one record

Explicit Cursor

- Use for queries that return multiple records or no records
- Must be explicitly declared and used
- A cursor is a private set of records
- An Oracle Cursor = VB recordset = JDBC ResultSet
- Implicit cursors are created for every query made in Oracle
- Explicit cursors can be declared by a programmer within PL/SQL.

Cursor Attributes

• cursorname%ROWCOUNT	Rows returned so far
• cursorname%FOUND	One or more rows retrieved
• cursorname%NOTFOUND	No rows found
• Cursorname%ISOPEN	Is the cursor open

Explicit Cursor Control

- Declare the cursor
- Open the cursor
- Fetch a row
- Test for end of cursor
- Close the cursor

Note: there is a FOR LOOP available with an implicit fetch

Using an Explicit Cursor

- Declare the cursor
  - *CURSOR cursor_name IS select_query;*
- Open the cursor
  - *OPEN cursor_name;*
- Fetch the data rows
  - *LOOP*

    FETCH cursor_name INTO variable_name(s);

    EXIT WHEN cursor_name%NOTFOUND;
- Close the cursor
  - *CLOSE cursor_name;*

Sample Cursor Program

```
DECLARE
  CURSOR students_cursor IS
    SELECT * from students;
  v_student students_cursor%rowtype;
  /* instead we could do v_student students%rowtype */
BEGIN
  DBMS_OUTPUT.PUT_LINE ('********************');
  OPEN students_cursor;
  FETCH students_cursor into v_student;
  WHILE students_cursor%found LOOP
     DBMS_OUTPUT.PUT_LINE (v_student.last);
     DBMS_OUTPUT.PUT_LINE (v_student.major);
     DBMS_OUTPUT.PUT_LINE ('********************');
     FETCH students_cursor into v_student;
  END LOOP;
  CLOSE students_cursor;
END;
/
```

Cursor FOR Loop

- Automatically opens the cursor, fetches the records, then closes the cursor
- *FOR variable_name(s) IN cursor_name LOOP*

  processing commands

  END LOOP;
- Cursor variables cannot be used outside loop

Handling Runtime Errors in PL/SQL Programs

- Runtime errors cause exceptions
- Exception handlers exist to deal with different error situations
- Exceptions cause program control to fall to exception section where exception is handled

```
EXCEPTION
  WHEN exception1_name THEN
     exception1 handler commands;
  WHEN exception2_name THEN
     exception2 handler commands;
  …
  WHEN OTHERS THEN
     other handler commands;
END;
```

Figure 4-34 Exception handler syntax

Predefined Exceptions

Oracle Error Code	Exception Name	Description
ORA-00001	DUP_VAL_ON_INDEX	Command violates primary key unique constraint
ORA-01403	NO_DATA_FOUND	Query retrieves no records
ORA-01422	TOO_MANY_ROWS	Query returns more rows than anticipated
ORA-01476	ZERO_DIVIDE	Division by zero
ORA-01722	INVALID_NUMBER	Invalid number conversion (such as trying to convert "2B" to a number)
ORA-06502	VALUE_ERROR	Error in truncation, arithmetic, or data conversion operation

Table 4-10 Common PL/SQL predefined exceptions

Undefined Exceptions

- Less common errors
- Do not have predefined names
- Must declare your own name for the exception code in the declaration section
  - *DECLARE*

    e_exception_name EXCEPTION;

    PRAGMA EXCEPTION_INIT(e_exception_name, -Oracle_error_code);

Stored Procedures

- PL/SQL code stored in the database and executed when called by the user.
- Called by procedure name from another PL/SQL block or using EXECUTE from SQL+. For example **EXEC SQR(50)**
- Example:

```
Create procedure SQR (v_num_to_square IN number)
AS
   v_answer number(10);
BEGIN
   v_answer := v_num_to_square * v_num_to_square;
   dbms_output.put_line(v_answer);
END;
/
```

Function

- PL/SQL user defined function stored in the database and executed when a function call is made in code: example x := SQUARED(50)
- Example:

```
Create or Replace Function SQUARED
  (p_number_to_square IN number)
   RETURN number
   IS
      v_answer number(10);
   BEGIN
      v_answer := p_number_to_square * p_number_to_square;
      RETURN(v_answer);
   END;
/
```

Another Stored Procedure Example

```
Create or replace procedure mytabs
AS
   CURSOR table_cursor IS
        Select table_name from user_tables;
   v_tablename varchar2(30);
BEGIN
   open table_cursor;
   fetch table_cursor into v_tablename;
   while table_cursor%found loop
        dbms_output.put_line(v_tablename);
        fetch table_cursor into v_tablename;
   end loop;
   close table_cursor;
END;
```

Triggers

- PL/SQL code executed automatically in response to a database event, typically DML.
- Like other stored procedures, triggers are stored in the database.
- Often used to:
  - enforce complex constraints, especially multi-table constraints. Financial posting is an example of this.
  - Trigger related actions
  - implement auditing “logs”
  - pop a sequence when creating token keys
- Triggers do not issue transaction control statements (such as commit). Triggers are part of the SQL transaction that invoked them.
- USER_TRIGGERS provides a data dictionary view of triggers.

Triggers

```
CREATE OR REPLACE TRIGGER <trigger_name>
   [BEFORE/AFTER][DELETE/INSERT/UPDATE of <column_name |, column_name… |>
   ON <table_name>
   |FOR EACH ROW|
   |WHEN <triggering condition>|
   |DECLARE|
   BEGIN
      trigger statements
      …….
   END;

To delete a trigger use:

DROP TRIGGER <trigger_name>;
```

Log Trigger Example

```
CREATE OR REPLACE TRIGGER LOGSTUDENTCHANGES
  BEFORE INSERT OR DELETE OR UPDATE of Major ON STUDENTS
  FOR EACH ROW
DECLARE
  v_ChangeType CHAR(1);
  v_sid varchar2(10);
BEGIN
  IF INSERTING THEN
     V_ChangeType := 'I';
     v_sid := :new.sid;
  ELSIF UPDATING THEN
     V_ChangeType := 'U';
     v_sid := :new.sid;
  ELSE
     V_ChangeType := 'D';
     v_sid := :old.sid;
END IF;
INSERT INTO MAJ_AUDIT (change_type, changed_by, timestamp,
   SID, old_major, new_major)
   VALUES (v_ChangeType, USER, SYSDATE, v_sid, :old.major, :new.major);
END LOGSTUDENTCHANGES;
```

UpperCase Trigger Example

```
CREATE OR REPLACE TRIGGER UPPERCASE
   BEFORE INSERT OR UPDATE ON STUDENTS
   FOR EACH ROW
DECLARE
BEGIN
   :new.lastname:=UPPER(:new.lastname);
   :new.firstname:=UPPER(:new.firstname);
END UPPERCASE;
/
```

SEQUENCE

```
CREATE SEQUENCE <sequence_name>
|INCREMENT BY <number>|
|START WITH <start_value>|
|MAXVALUE <maximum_value>|NOMAXVALUE|
|MINVALUE <minimum_value>|
|CYCLE|NOCYLE|
|CACHE <number of values>|NOCACHE|
|ORDER|NOORDER|
```

To pop the next sequence use:

SEQUENCENAME.NEXTVAL (CURRVAL shows last pop)