**PLSQL**

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL. PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

Advantages of PL/SQL

PL/SQL has the following advantages −

* SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
* PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
* PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
* PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
* Applications written in PL/SQL are fully portable.
* PL/SQL provides high security level.
* PL/SQL provides access to predefined SQL packages.
* PL/SQL provides support for Object-Oriented Programming.
* PL/SQL provides support for developing Web Applications and Server Pages.
* Following is the basic structure of a PL/SQL block −

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling>

END;

The 'Hello World' Example

DECLARE

message varchar2(20):='Hello, World!';

BEGIN

dbms\_output.put\_line(message);

END;

/

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result −

Hello World

PL/SQL procedure successfully completed.

**PL/SQL - Operators**

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators −

* Arithmetic operators
* Relational operators
* Comparison operators
* Logical operators
* String operators

Here, we will understand the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed in a later chapter − **PL/SQL - Strings**.

Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 5, then −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_arithmetic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 15 |
| - | Subtracts second operand from the first | A - B will give 5 |
| \* | Multiplies both operands | A \* B will give 50 |
| / | Divides numerator by de-numerator | A / B will give 2 |
| \*\* | Exponentiation operator, raises one operand to the power of other | A \*\* B will give 100000 |

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 20, then −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A = B) is not true. |
| !=  <>  ~= | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true |

Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either **TRUE, FALSE** or **NULL**.

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_comparison_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| LIKE | The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not. | If 'Zara Ali' like 'Z% A\_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A\_i' returns a Boolean false. |
| BETWEEN | The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b. | If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false. |
| IN | The IN operator tests set membership. x IN (set) means that x is equal to any member of set. | If x = 'm' then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true. |
| IS NULL | The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL. | If x = 'm', then 'x is null' returns Boolean false. |

Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us assume **variable A** holds true and**variable B** holds false, then −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Examples** |
| and | Called the logical AND operator. If both the operands are true then condition becomes true. | (A and B) is false. |
| or | Called the logical OR Operator. If any of the two operands is true then condition becomes true. | (A or B) is true. |
| not | Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false. | not (A and B) is true. |

**PL/SQL Operator Precedence**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, **x = 7 + 3 \* 2**; here, **x** is assigned **13**, not 20 because operator \* has higher precedence than +, so it first gets multiplied with **3\*2** and then adds into **7**.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_operators_precedence.htm)

|  |  |
| --- | --- |
| **Operator** | **Operation** |
| \*\* | exponentiation |
| +, - | identity, negation |
| \*, / | multiplication, division |
| +, -, || | addition, subtraction, concatenation |
| comparison |  |
| NOT | logical negation |
| AND | conjunction |
| OR | Inclusion |

**DECISION CONTROL STRUCTURE**

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.

When using **IF-THEN-ELSIF** statements there are a few points to keep in mind.

* It's ELSIF, not ELSEIF.
* An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
* An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
* Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax

The syntax of an **IF-THEN-ELSIF** Statement in PL/SQL programming language is −

IF(boolean\_expression 1)THEN

S1; -- Executes when the boolean expression 1 is true

ELSIF( boolean\_expression 2) THEN

S2; -- Executes when the boolean expression 2 is true

ELSIF( boolean\_expression 3) THEN

S3; -- Executes when the boolean expression 3 is true

ELSE

S4; -- executes when the none of the above condition is true

END IF;

Example

DECLARE

a number(3):=100;

BEGIN

IF ( a =10) THEN

dbms\_output.put\_line('Value of a is 10');

ELSIF ( a =20) THEN

dbms\_output.put\_line('Value of a is 20');

ELSIF ( a =30) THEN

dbms\_output.put\_line('Value of a is 30');

ELSE

dbms\_output.put\_line('None of the values is matching');

END IF;

dbms\_output.put\_line('Exact value of a is: '|| a );

END;

/

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

None of the values is matching

Exact value of a is: 100

PL/SQL procedure successfully completed.

**SWITCH-CASE LIKE STRUCTURE**

The searched **CASE** statement has no selector and the **WHEN**clauses of the statement contain search conditions that give Boolean values.

Syntax

The syntax for the searched case statement in PL/SQL is −

CASE

WHEN selector = 'value1' THEN S1;

WHEN selector = 'value2' THEN S2;

WHEN selector = 'value3' THEN S3;

...

ELSE Sn; -- default case

END CASE;

Flow Diagram

Example

DECLARE

grade char(1):='B';

BEGIN

case

when grade ='A'then dbms\_output.put\_line('Excellent');

when grade ='B'then dbms\_output.put\_line('Very good');

when grade ='C'then dbms\_output.put\_line('Well done');

when grade ='D'then dbms\_output.put\_line('You passed');

when grade ='F'then dbms\_output.put\_line('Better try again');

else dbms\_output.put\_line('No such grade');

endcase;

END;

/

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

Very good

PL/SQL procedure successfully completed.

**Loop Control**

PL/SQL - Basic Loop Statement

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax

The syntax of a basic loop in PL/SQL programming language is −

LOOP

Sequence of statements;

END LOOP;

Here, the sequence of statement(s) may be a single statement or a block of statements. An **EXIT statement** or an **EXIT WHEN statement** is required to break the loop.

Example

DECLARE

x number :=10;

BEGIN

LOOP

dbms\_output.put\_line(x);

x := x +10;

IF x >50 THEN

exit;

END IF;

END LOOP;

-- after exit, control resumes here

dbms\_output.put\_line('After Exit x is: '|| x);

END;

/

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

10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.

SQL> DECLARE

x number := 10;

BEGIN

LOOP

dbms\_output.put(x); --does not change line

x := x + 10;

IF x > 50 THEN

exit;

END IF;

END LOOP;

-- after exit, control resumes here

dbms\_output.put\_line('After Exit x is: ' || x);

END;

/

1020304050After Exit x is: 60

PL/SQL procedure successfully completed.

SQL> DECLARE

x number := 10;

BEGIN

LOOP

dbms\_output.put(x);

dbms\_output.new\_line(); --to send cursor to new line

x := x + 10;

IF x > 50 THEN

exit;

END IF;

END LOOP;

-- after exit, control resumes here

dbms\_output.put\_line('After Exit x is: ' || x);

END;

/

10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.

You can use the **EXIT WHEN** statement instead of the **EXIT**statement −

DECLARE

x number :=10;

BEGIN

LOOP

dbms\_output.put\_line(x);

x := x +10;

exit WHEN x >50;

END LOOP;

-- after exit, control resumes here

dbms\_output.put\_line('After Exit x is: '|| x);

END;

/

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

10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.

PL/SQL - WHILE LOOP Statement

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

WHILE condition LOOP

sequence\_of\_statements

END LOOP;

Example

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 produces the following result −

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.

PL/SQL - FOR LOOP Statement

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

FOR counter IN initial\_value .. final\_value LOOP

sequence\_of\_statements;

END LOOP;

Following is the flow of control in a **For Loop** −

* The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
* Next, the condition, i.e., *initial\_value .. final\_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.
* After the body of the for loop executes, the value of the counter variable is increased or decreased.
* The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.

Following are some special characteristics of PL/SQL for loop −

* The *initial\_value* and *final\_value* of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE\_ERROR.
* The *initial\_value* need not be 1; however, the **loop counter increment (or decrement) must be 1**.
* PL/SQL allows the determination of the loop range dynamically at run time.

Example

DECLARE

a number(2);

BEGIN

FOR a in10..20 LOOP

dbms\_output.put\_line('value of a: '|| a);

END LOOP;

END;

/

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

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.

Reverse FOR LOOP Statement

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE**keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds in ascending (not descending) order. The following program illustrates this −

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, it produces the following result −

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

PL/SQL procedure successfully completed.

Labeling a PL/SQL Loop

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

The following program illustrates the concept −

DECLARE

i number(1);

j number(1);

BEGIN

<< outer\_loop >>

FOR i IN 1..3 LOOP

<< inner\_loop >>

FOR j IN 1..3 LOOP

dbms\_output.put\_line('i is: '|| i ||' and j is: '|| j);

END loop inner\_loop;

END loop outer\_loop;

END;

/

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

i is: 1 and j is: 1

i is: 1 and j is: 2

i is: 1 and j is: 3

i is: 2 and j is: 1

i is: 2 and j is: 2

i is: 2 and j is: 3

i is: 3 and j is: 1

i is: 3 and j is: 2

i is: 3 and j is: 3

PL/SQL procedure successfully completed.

The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

PL/SQL supports the following control statements. Labeling loops also help in taking the control outside a loop. Click the following links to check their details.

|  |  |
| --- | --- |
| **S.No** | **Control Statement & Description** |
| 1 | [**EXIT statement**](https://www.tutorialspoint.com/plsql/plsql_exit_statement.htm)  The Exit statement completes the loop and control passes to the statement immediately after the END LOOP. |
| 2 | [**CONTINUE statement**](https://www.tutorialspoint.com/plsql/plsql_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | [**GOTO statement**](https://www.tutorialspoint.com/plsql/plsql_goto_statement.htm)  Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program. |

The **EXIT** statement in PL/SQL programming language has the following two usages −

* When the EXIT statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
* If you are using nested loops (i.e., one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for an EXIT statement in PL/SQL is as follows −

EXIT;

Flow Diagram

Example

DECLARE

a number(2):=10;

BEGIN

--while loop execution

WHILE a <20 LOOP

dbms\_output.put\_line ('value of a: '|| a);

a := a +1;

IF a >15 THEN

-- terminate the loop using the exit statement

EXIT;

END IF;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

PL/SQL procedure successfully completed.

The EXIT WHEN Statement

The **EXIT-WHEN** statement allows the condition in the WHEN clause to be evaluated. If the condition is true, the loop completes and control passes to the statement immediately after the END LOOP.

Following are the two important aspects for the EXIT WHEN statement −

* Until the condition is true, the EXIT-WHEN statement acts like a NULL statement, except for evaluating the condition, and does not terminate the loop.
* A statement inside the loop must change the value of the condition.

Syntax

The syntax for an EXIT WHEN statement in PL/SQL is as follows −

EXIT WHEN condition;

The EXIT WHEN statement **replaces a conditional statement like if-then** used with the EXIT statement.

Example

DECLARE

a number(2):=10;

BEGIN

--while loop execution

WHILE a <20 LOOP

dbms\_output.put\_line ('value of a: '|| a);

a := a +1;

-- terminate the loop using the exitwhen statement

EXIT WHEN a >15;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

PL/SQL procedure successfully completed.

The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

Syntax

The syntax for a CONTINUE statement is as follows −

CONTINUE;

Flow Diagram

Example

DECLARE

a number(2):=10;

BEGIN

--while loop execution

WHILE a <20 LOOP

dbms\_output.put\_line ('value of a: '|| a);

a := a +1;

IF a =15 THEN

-- skip the loop using the CONTINUE statement

a := a +1;

CONTINUE;

END IF;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

PL/SQL procedure successfully completed.

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

**NOTE** − The use of GOTO statement is not recommended in any programming language because it makes it difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax

The syntax for a GOTO statement in PL/SQL is as follows −

GOTO label;

..

..

<< label >>

statement;

Flow Diagram

Example

DECLARE

a number(2):=10;

BEGIN

<<loopstart>>

--while loop execution

WHILE a <20 LOOP

dbms\_output.put\_line ('value of a: '|| a);

a := a +1;

IF a =15 THEN

a := a +1;

GOTO loopstart;

END IF;

END LOOP;

END;

/

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

PL/SQL procedure successfully completed.

Restrictions with GOTO Statement

GOTO Statement in PL/SQL imposes the following restrictions −

* A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
* A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
* A GOTO statement cannot branch from an outer block into a sub-block (i.e., an inner BEGIN-END block).
* A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.
* A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

**Some Example code**

P1: PL/SQL block to display total of 3 numbers with 3rd number entered by the user

Sol:

declare

a number(2):=10;

b number(2):=5;

d number(2);

str1 varchar2(20):=' The sum is';

sum1 number(2):=0;

begin

d:=&d;

sum1:=a+b+d;

dbms\_output.put\_line(str1 || sum1);

end;

/

P2: PL/SQL block to display even odd status of a number entered by the user

Sol:declare

a number(2):=10;

begin

a:=&a;

if a mod 2=0 then

dbms\_output.put\_line('The number is even');

else

dbms\_output.put\_line('The number is odd');

endif;

end;

/

P3: PL/SQL block to display numbers 1 to 10 via for loop

Sol:

declare

begin

for i in 1..10 loop

dbms\_output.put\_line('i='||i);

endloop;

end;

/

P4: PL/SQL block to display 1,3,5,7…10 times

Sol:

declare

i2 number(2):=1;

begin

for i in 1..10 loop

dbms\_output.put\_line('i='||i2);

i2:=i2+2;

endloop;

end;

/

P5: PL/SQL block to display 10,9,8,7,…1

Sol:

begin

for i inreverse 1..10 loop

dbms\_output.put\_line('i='||i);

endloop;

end;

/

P6: PL/SQL block to display 1 to 10 via while loop

Sol:

declare

i number(2):=1;

begin

while i<=10 loop

dbms\_output.put\_line('i='||i);

i:=i+1;

endloop;

end;

/

P7: PL/SQL block to display 1 to 10 via simple loop

Sol:declare

i number(2):=1;

begin

loop

dbms\_output.put\_line('i='||i);

i:=i+1;

exitwhen i>10;

endloop;

end;

/

P8: PL/SQL block to display whether a number entered by the user is Armstrong or not?

Sol:

declare

n number:=0;

s number:=0;

d number;

len number;

temp number;

begin

n:=&n;

temp:=n;

while n>0 loop

d:=mod(n,10);

s:=s+power(d,3);

n:=trunc(n/10);

endloop;

if temp=s then

dbms\_output.put\_line(' It is an armstrong number');

else

dbms\_output.put\_line('It is not an armstrong number');

endif;

end;

/

**Ques:With reference to oracle, define cursor and its uses. [GGSIPU 2011]**

**Ques:What are explicit cursors? [GGSIPU2012]**

**Ques:Differentiate between implicit and explicit cursors. [GGSIPU2013]**

**Ques:What do you mean by cursors and triggers. [GGSIPU2014]**

**CURSOR**

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement.

OR

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it.

This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is referred to as the **active set**.

There are two types of cursors:

* Implicit cursors
* Explicit cursors

**Explicit Cursors:**

Explicit cursors are programmer defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is :

CURSOR cursor\_name IS select\_statement;

Working with an explicit cursor involves four steps:

* Declaring the cursor for initializing in the memory
* Opening the cursor for allocating memory
* Fetching the cursor for retrieving data
* Closing the cursor to release allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

CURSOR c\_customers IS

SELECT id, name, address FROM customers;

Opening the Cursor

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above-defined cursor as follows:

OPEN c\_customers;

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above-opened cursor as follows:

FETCH c\_customers INTO c\_id, c\_name, c\_addr;

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close above-opened cursor as follows:

CLOSE c\_customers;

**Following Programs show the use of Explicit Cursors**

Ques: WAP to display all odd rows from emp table.

Sol:

**declare**

**cursor c1 is select \* from emp;**

**e1 c1%rowtype;--e1 is a variable of type c1**

**begin**

**open c1;**

**loop**

**fetch c1 into e1;**

**exit when c1%notfound;**

**if mod(c1%rowcount,2)!=0 then**

**dbms\_output.put\_line(e1.empno||' '||e1.ename);**

**end if;**

**end loop;**

**close c1;**

**end;**

**/**

Ques: WAP to increment sal in emp2 table as

if e1.sal<=1000 then incr:=1000;

else if e1.sal>=1001 and e1.sal<=2000 then incr:=500;

otherwise incr:=100;.

Sol:

**declare**

**cursor c1 is select \* from emp2;**

**e1 c1%rowtype;**

**incr number(7,2);**

**begin**

**open c1;**

**loop**

**fetch c1 into e1;**

**exit when c1%notfound;**

**if e1.sal<=1000 then**

**incr:=1000;**

**elsif e1.sal>=1001 and e1.sal<=2000 then**

**incr:=500;**

**else**

**incr:=100;**

**end if;**

**update emp2 set sal=e1.sal+incr where empno=e1.empno;**

**end loop;**

**commit;**

**close c1;**

**end;**

**/**

Ques: WAP to increment sal in emp2 table as

**Increment sal of Ist employee by 100 second by 200,and so on..**

Sol

**declare**

**cursor c1 is select \* from emp2;**

**e1 c1%rowtype;**

**incr number(7,2):=100;**

**begin**

**open c1;**

**loop**

**fetch c1 into e1;**

**exit when c1%notfound;**

**e1.sal:=e1.sal+incr;**

**incr:=incr+100;**

**update emp2 set sal=e1.sal where empno=e1.empno;**

**end loop;**

**close c1;**

**end;**

**/**

Ques : WAP to display names of top3 earners from emp;

Sol:

**declare**

**cursor c1 is select \* from emp order by sal desc;**

**e1 c1%rowtype;**

**begin**

**open c1;**

**loop**

**fetch c1 into e1;**

**exit when c1%rowcount>3;**

**dbms\_output.put\_line(e1.empno||' '||e1.sal);**

**end loop;**

**close c1;**

**end;**

**/**

**Implicit Cursors**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected. They are also created when a SELECT statement that returns just one row is executed.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK\_ROWCOUNT and %BULK\_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| %FOUND | Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| %NOTFOUND | The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. |
| %ISOPEN | Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| %ROWCOUNT | Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

**CURSOR FOR LOOP**

**CURSOR FOR LOOP** in Oracle with syntax and examples.

**DESCRIPTION**

You would use a CURSOR FOR LOOP when you want to fetch and process every record in a cursor. The CURSOR FOR LOOP will terminate when all of the records in the cursor have been fetched.

**SYNTAX**

The syntax for the CURSOR FOR LOOP in Oracle/PLSQL is:

FOR *record\_index* in *cursor\_name*

LOOP

{...statements...}

END LOOP;

Parameters or Arguments

**record\_index**

The index of the record.

**cursor\_name**

The name of the cursor that you wish to fetch records from.

**statements**

The statements of code to execute each pass through the CURSOR FOR LOOP.

**EXAMPLE**

**declare**

**cursor c1 is select \* from emp;**

**begin**

**for e1 in c1 loop**

**dbms\_output.put\_line(e1.ename);**

**end loop;**

**end;**

**/**

**Cursors with Parameters**

We can pass parameters into a cursor and use them in the query.*A parameter makes the cursor more reusable.* We can only pass values to the cursor; and cannot pass values out of the cursor through parameters. Only the datatype of the parameter is defined, not its length. Optionally, we can also give a default value for the parameter, which will take effect if no value is passed to the cursor.

**Display names of employees department wise.( Employees of deptno=10 then employees of deptno=20..so on)**

**declare**

**cursor c1(dno dept.deptno%type) is select \* from emp where deptno=dno;**

**cursor c2 is select \* from dept;**

**begin**

**for e2 in c2 loop**

**for e1 in c1(e2.deptno)loop**

**dbms\_output.put\_line(e1.ename);**

**end loop;**

**end loop;**

**end;**

**/**

**Ques:What are stored procedures in Oracle? Briefly describe.[GGSIPU 2011]**

**Ques: What are stored procedures in Oracle?[GGSIPU2012]**

**Ques: How is stored procedure created? Explain with example the three ways of passing Parameters with the stored procedure.[GGSIPU2013]**

**PL/SQL - Procedures**

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

**CREATE [OR REPLACE] PROCEDURE procedure\_name**

**[(parameter\_name [IN | OUT | IN OUT] type [,...])]**

**{IS | AS}**

**BEGIN**

**< procedure\_body >**

**END procedure\_name;**

Where,

* *procedure-name* specifies the name of the procedure.
* [OR REPLACE] option allows modifying an existing procedure.
* The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
* *procedure-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example:

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

When above code is executed using SQL prompt, it will produce the following result:

Procedure created.

Executing a Standalone Procedure

A standalone procedure can be called in two ways:

* Using the EXECUTE keyword
* Calling the name of the procedure from a PL/SQL block

The above procedure named 'greetings' can be called with the EXECUTE keyword as:

EXECUTE greetings;

The above call would display:

HelloWorld

PL/SQL procedure successfully completed.

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

BEGIN

greetings;

END;

/

The above call would display:

HelloWorld

PL/SQL procedure successfully completed.

Deleting a Standalone Procedure

A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is:

DROP PROCEDURE procedure-name;

So you can drop *greetings* procedure by using the following statement:

DROP PROCEDURE greetings;

Parameter Modes in PL/SQL Subprograms

|  |  |
| --- | --- |
| **S.N.** | **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**. |
| 2 | **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 its 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.** |

**IN & OUT Mode Example 1**

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

DECLARE

a number;

b number;

c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

IF x < y THEN

z:= x;

ELSE

z:= y;

END IF;

END;

BEGIN

a:=23;

b:=45;

findMin(a, b, c);

dbms\_output.put\_line(' Minimum of (23, 45) : '|| c);

END;

/

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

Minimum of (23,45):23

PL/SQL procedure successfully completed.

**IN & OUT Mode Example 2**

This procedure computes the square of value of a passed value. This example shows how we can use same parameter to accept a value and then return another result.

DECLARE

a number;

PROCEDURE squareNum(x IN OUT number) IS

BEGIN

x := x \* x;

END;

BEGIN

a:=23;

squareNum(a);

dbms\_output.put\_line(' Square of (23): '|| a);

END;

/

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

Square of (23):529

PL/SQL procedure successfully completed.

PL/SQL - Functions

A PL/SQL function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

CREATE [OR REPLACE] FUNCTION function\_name

[(parameter\_name [IN | OUT | IN OUT] type [,...])]

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END[function\_name];

Where,

* *function-name* specifies the name of the function.
* [OR REPLACE] option allows modifying an existing function.
* The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
* The function must contain a **return** statement.
* *RETURN* clause specifies that data type you are going to return from the function.
* *function-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone function.

**STAND ALONE FUNCTION**

create or replace function findMax23(x IN number, y IN number) return number

as

z number;

begin

if x > y then

z:= x;

else

Z:= y;

end if;

return z;

end;

/

Above function exists as a separate object in database and can be used in a sql query as shown below:

SQL> select findmax23(45,44) from dual;

FINDMAX23(45,44)

----------------

45

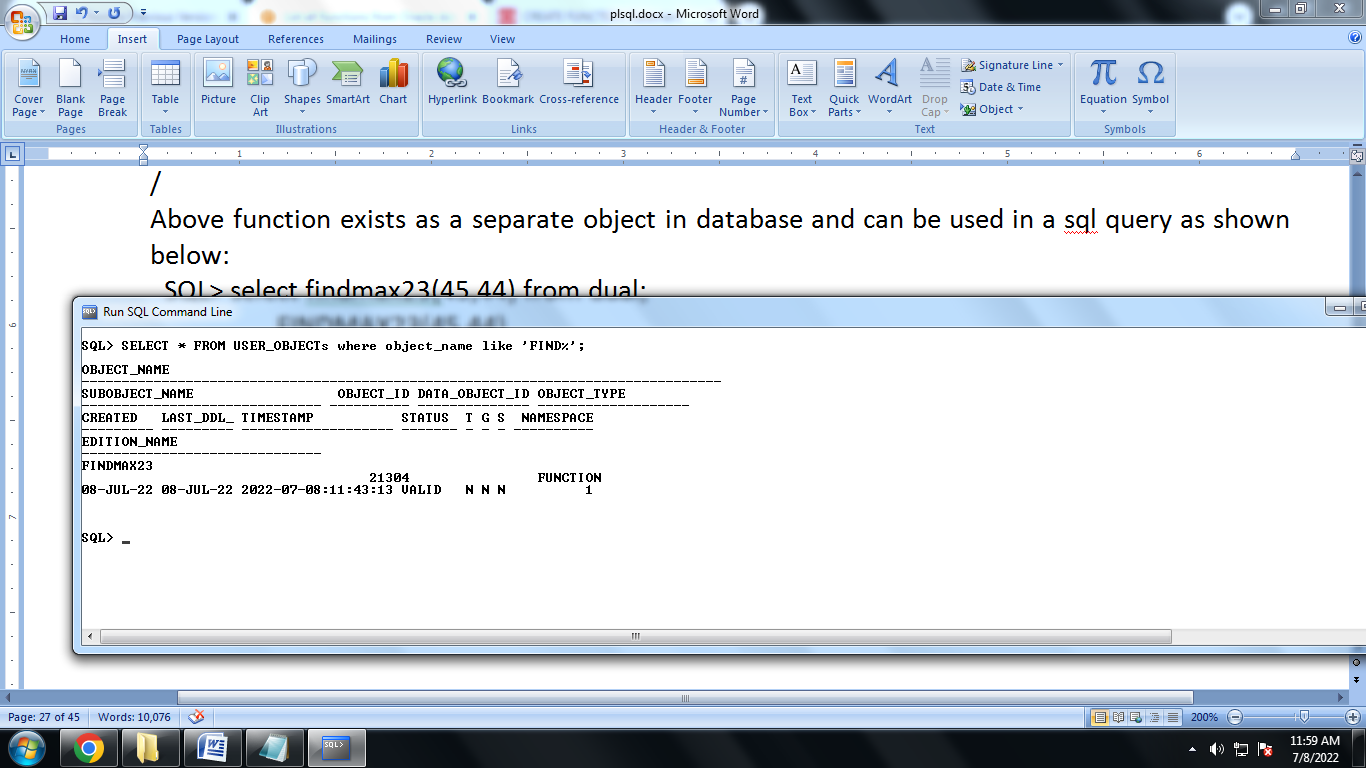
SQL> select findmax23(45,94) from dual;

FINDMAX23(45,94)

----------------

94

**Further if you wish to see whether it exists as separate object or not then use following sql command.**

****

**SQL> select \* from USER\_OBJECTS where OBJECT\_NAME like 'FIND%';**

**CREATE A FUNCTION WITHIN A BLOCK (NOT as STAND ALONE FUNCTION)**

**declare**

**a number;**

**b number;**

**c number;**

**function findMax(x IN number, y IN number) return number**

**is**

**z number;**

**begin**

**if x > y then**

**z:= x;**

**else**

**Z:= y;**

**end if;**

**return z;**

**end;**

**begin**

**a:= 23;**

**b:= 45;**

**c := findmax(a, b);**

**dbms\_output.put\_line(' maximum of (23,45): ' || c);**

**end;**

**/**

**Example2:Function to calculate Factorial of a number**

**declare**

**n number;**

**res number;**

**function fact(x IN number) return number**

**is**

**f1 number:=1;**

**begin**

**for i in 1..x loop**

**f1:=f1\*i;**

**end loop;**

**return f1;**

**end;**

**BEGIN**

**n:=&num;**

**res := fact(n);**

**dbms\_output.put\_line(' Factorial= ' || res);**

**END;**

**/**

Ques:Discuss Exception handling in PL/SQL.[GGSIPU2012]

Ques: Explain error handling in PL/SQL.[GGSIPU2013]

**Exception Handling**

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

* System-defined exceptions
* User-defined exceptions

**Syntax for Exception Handling**

The General Syntax for exception handling is as follows. Here you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

........

WHEN others THEN

exception3-handling-statements

END;

**Example**

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

DECLARE

c\_id customers.id%type :=8;

c\_name customers.name%type;

c\_addr customers.address%type;

BEGIN

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: '|| c\_addr);

EXCEPTION

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

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

No such customer!

PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception**NO\_DATA\_FOUND**, which is captured in **EXCEPTION** block.

**Raising Exceptions**

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax of raising an exception:

DECLARE

exception\_name EXCEPTION;

BEGIN

IF condition THEN

RAISE exception\_name;

END IF;

EXCEPTION

WHEN exception\_name THEN

statement;

END;

You can use above syntax in raising Oracle standard exception or any user-defined exception. Next section will give you an example on raising user-defined exception, similar way you can raise Oracle standard exceptions as well.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR.

The syntax for declaring an exception is:

DECLARE

my-exception EXCEPTION;

Example:

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid\_id is raised.

DECLARE

c\_id customers.id%type :=&cc\_id;

c\_name customers.name%type;

c\_addr customers.address%type;

-- user defined exception

ex\_invalid\_id EXCEPTION;

BEGIN

IF c\_id <=0 THEN

RAISE ex\_invalid\_id;

ELSE

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: '|| c\_addr);

END IF;

EXCEPTION

WHEN ex\_invalid\_id THEN

dbms\_output.put\_line('ID must be greater than zero!');

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

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

Enter value for cc\_id:-6(let's enter a value -6)

old 2: c\_id customers.id%type := &cc\_id;

new 2: c\_id customers.id%type := -6;

ID must be greater than zero!

PL/SQL procedure successfully completed.

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO\_DATA\_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Exception** | **Oracle Error** | **SQLCODE** | **Description** |
| ACCESS\_INTO\_NULL | 06530 | -6530 | It is raised when a null object is automatically assigned a value. |
| CASE\_NOT\_FOUND | 06592 | -6592 | It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION\_IS\_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP\_VAL\_ON\_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |
| INVALID\_CURSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID\_NUMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN\_DENIED | 01017 | -1017 | It is raised when s program attempts to log on to the database with an invalid username or password. |
| NO\_DATA\_FOUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT\_LOGGED\_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM\_ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE\_MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |
| SELF\_IS\_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE\_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO\_MANY\_ROWS | 01422 | -1422 | It is raised when s SELECT INTO statement returns more than one row. |
| VALUE\_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

**Ques: In oracle, we should use triggers and when should we not? Explain through examples. [GGSIPU2011]**

**Ques: When and how is the trigger created? [GGSIPU2013]**

**Ques:What do you mean by cursors and triggers. [GGSIPU2014]**

Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

* A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
* A database definition (DDL) statement (CREATE, ALTER, or DROP).
* A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes:

* Generating some derived column values automatically
* Enforcing referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is:

CREATE [OR REPLACE ] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR]| UPDATE [OR]| DELETE}

[OF col\_name]

ON table\_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

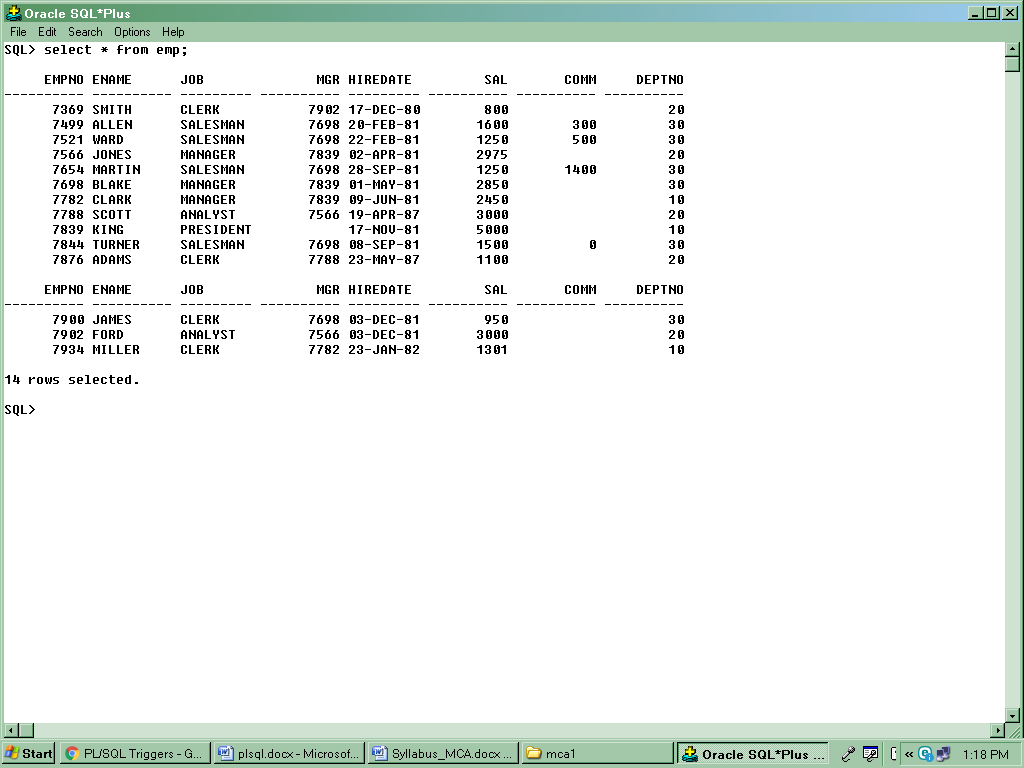
END;

Where,

* CREATE [OR REPLACE] TRIGGER trigger\_name: Creates or replaces an existing trigger with the *trigger\_name*.
* {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
* {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
* [OF col\_name]: This specifies the column name that would be updated.
* [ON table\_name]: This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
* [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
* WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

Example:

To start with, we will be using the emp table that comes with oracle.



The following program creates a **row level** trigger for the emp table that would fire for INSERT or UPDATE or DELETE operations performed on the emp table. This trigger will display the salary difference between the old values and new values:

CREATE OR REPLACE TRIGGER display\_salary\_changes

BEFORE DELETE or INSERT or UPDATe ON emp

FOR EACH ROW

DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.sal - :OLD.sal;

dbms\_output.put\_line('Old salary: ' || :OLD.sal);

dbms\_output.put\_line('New salary: ' || :NEW.sal);

dbms\_output.put\_line('Salary difference: ' || sal\_diff);

END;

/

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

Trigger created.

Here following two points are important and should be noted carefully:

* OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
* Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table

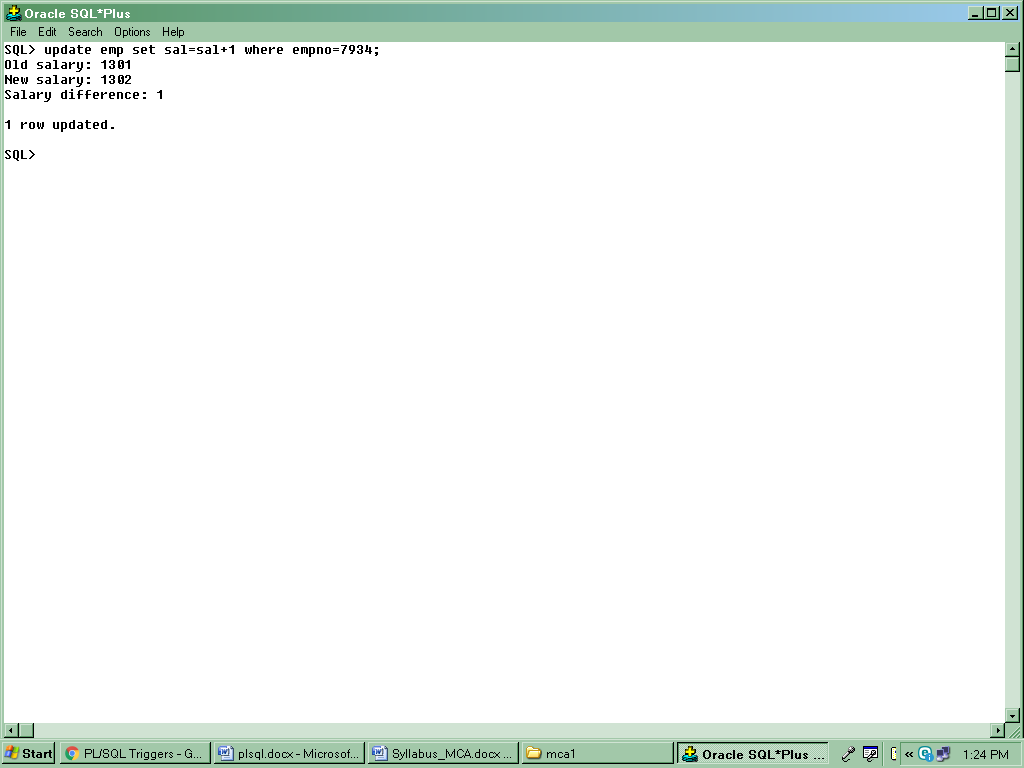
Here is one UPDATE statement, which will update an existing record in the table:

UPDATE emp

SET sal = sal + 1

WHERE empno =7934;

When a record is updated in emp table, above create trigger **display\_salary\_changes** will be fired and it will display the following result:



**LOGON TRIGGER**

SQL>

SQL> CREATE TABLE session\_logon\_statistics

2 (user\_logged VARCHAR2(30),

3 start\_time DATE,

4 end\_time DATE);

Table created.

SQL>

SQL>

SQL>

SQL> CREATE OR REPLACE TRIGGER logon\_log\_trigger

2 AFTER LOGON

3 ON DATABASE

4 BEGIN

5 INSERT INTO session\_logon\_statistics

6 (user\_logged, start\_time)

7 VALUES

8 (USER, SYSDATE);

9 END;

10 /

Trigger created.

SQL>

SQL>

SQL>

SQL> SELECT user\_logged,

2 TO\_CHAR(start\_time, 'MM/DD/YYYY HH24:MI:SS') "START TIME",

3 TO\_CHAR(end\_time, 'MM/DD/YYYY HH24:MI:SS') "END TIME"

4 FROM session\_logon\_statistics

5 where rownum < 50

6 order by user\_logged, start\_time;

no rows selected

SQL>

SQL>

SQL> drop trigger logon\_log\_trigger;

Trigger dropped.

**LOGOFF TRIGGER**

SQL>

SQL> CREATE TABLE session\_logon\_statistics

2 (user\_logged VARCHAR2(30),

3 start\_time DATE,

4 end\_time DATE);

Table created.

SQL>

SQL> CREATE OR REPLACE TRIGGER logoff\_log\_trigger

2 BEFORE LOGOFF

3 ON DATABASE

4 BEGIN

5 UPDATE session\_logon\_statistics

6 SET end\_time = SYSDATE

7 WHERE user\_logged = USER

8 AND end\_time IS NULL;

9 END;

10 /

Trigger created.

SQL>

SQL> SELECT user\_logged,

2 TO\_CHAR(start\_time, 'MM/DD/YYYY HH24:MI:SS') "START TIME",

3 TO\_CHAR(end\_time, 'MM/DD/YYYY HH24:MI:SS') "END TIME"

4 FROM session\_logon\_statistics

5 where rownum < 50

6 order by user\_logged, start\_time;

no rows selected

SQL>

SQL> drop trigger logoff\_log\_trigger;

Trigger dropped.

**BEFORE SHUTDOWN**

CREATE TABLE uptime\_log (

2 database\_name VARCHAR2(30),

3 event\_name VARCHAR2(20),

4 event\_time DATE,

5 triggered\_by\_user VARCHAR2(30)

6 );

Table created.

SQL>

SQL>

SQL> CREATE OR REPLACE TRIGGER log\_shutdown

2 BEFORE SHUTDOWN ON DATABASE

3 BEGIN

4 INSERT INTO uptime\_log

5 (database\_name,

6 event\_name,

7 event\_time,

8 triggered\_by\_user)

9 VALUES (sys.database\_name,

10 sys.sysevent,

11 sysdate,

12 sys.login\_user);

13 COMMIT;

14 END;

15 /

**Ques: Discuss the concurrency control techniques in details.[GGSIPU2012]**

**Ques: Explain the two phase locking protocol. What are its advantages and disadvantages. Define a transaction that satisfies 2PL. [GGSIPU2013][GGSIPU2014**

Answer: Purpose of Concurrency Control

* + To enforce Isolation (through mutual exclusion) among conflicting transactions.
  + To preserve database consistency through consistency preserving execution of transactions.
  + To resolve read-write and write-write conflicts.

A transaction is said to follow the two-phase locking protocol if all locking operations (read\_lock, write\_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phases:

Phase 1: Growing Phase

i)  transaction may obtain locks

ii)  transaction may not release locks

Phase 2: Shrinking Phase

i)  transaction may release locks

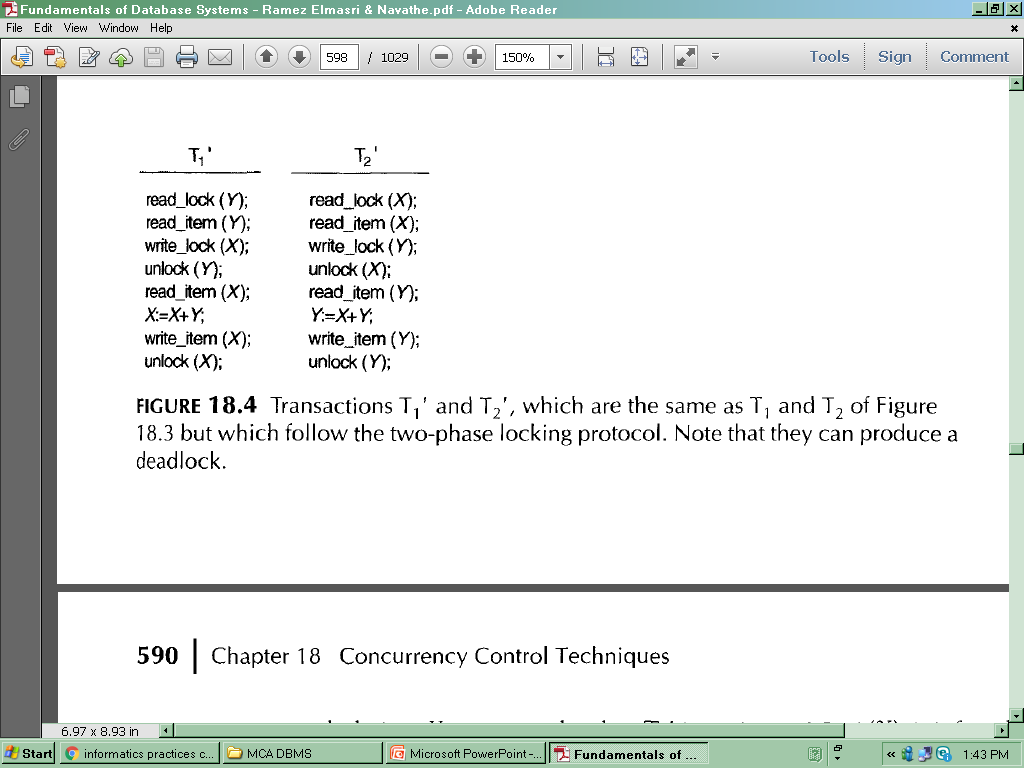
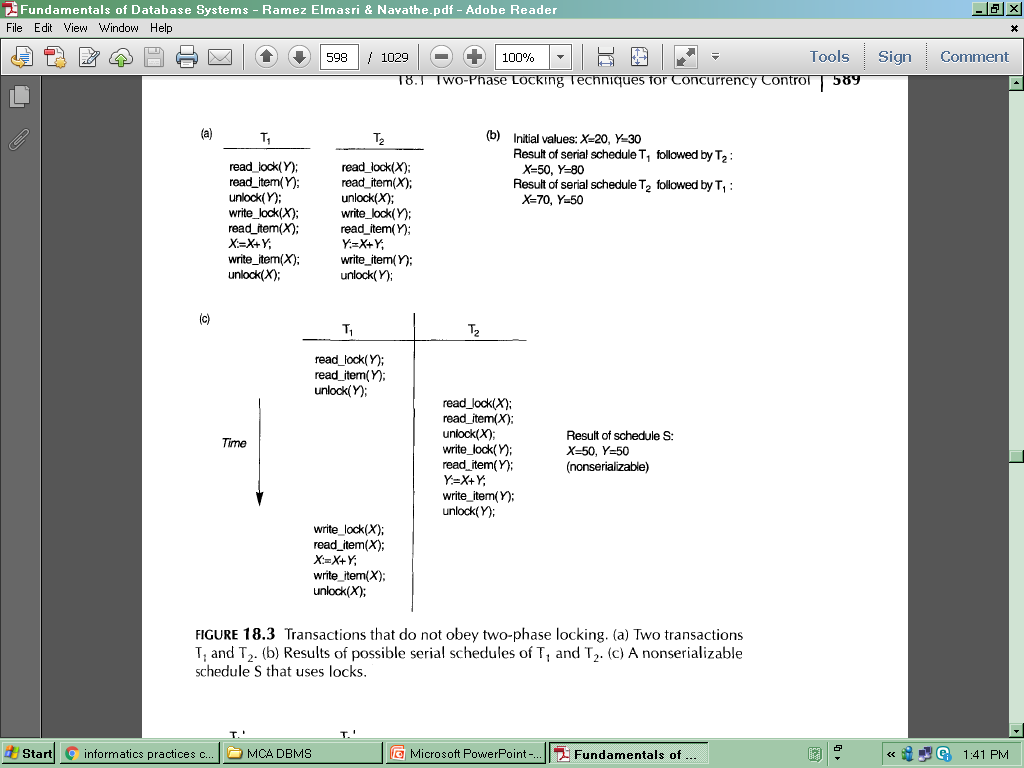
ii)  transaction may not obtain locks

 If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a read\_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points  (i.e. the point where a transaction acquired its final lock). Two-phase locking does not ensure freedom from deadlocks.

Types of 2PL:

* Basic 2PL – described above.
* Conservative 2PL (Static 2PL) - requires a transaction to lock all the items it accesses before the transaction begins execution, by pre declaring its read-set and write-set.
* Strict 2PL--Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking. Here a transaction must hold all its exclusive locks till it commits/aborts.
* Rigorous two-phase locking is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.



**CONCURRENCY CONTROL BASED ON TIMESTAMP ORDERING**

A timestamp is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time.* We will refer to the timestamp of transaction T as TS(T). Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur.* Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered **1,** 2, 3, ... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

**The Timestamp Ordering Algorithm**

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has the transactions in order of their timestamp values. This is called timestamp ordering (TO).

The algorithm must ensure that, for each item accessed by conflicting operations in the schedule, the order in which the item is accessed does not violate the serializability order. To do this, the algorithm associates with each database item X two timestamp (TS) values:

1. **Read\_TS(X):** The read timestamp of item Xi this is the largest timestamp among all the timestamps of transactions that have successfully read item X-that is, read\_TS(X) = TS(T), where T is the youngest transaction that has read X successfully.

2. **Write\_TS(X):** The write timestamp of item Xi this is the largest of all the timestamps of transactions that have successfully written item X-that is, write\_TS(X)= TS(T), where T is the youngest transaction that has written X successfully.

**Basic Timestamp Ordering.**

Whenever some transaction T tries to issue a read item(X) or a write\_item(X) operation, the basic TO algorithm compares the timestamp of T with read\_TS(X) and write\_TS(X) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a new timestamp. If T is aborted and rolled back, any transaction T 1 that may have used a value written by T must also be rolled back. Similarly, any transaction Tz that may have used a value written by T1 must also be rolled back, and so on. This effect is known as cascading rollback and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable.

1. Transaction T issues a write\_item(X) operation:

a. If read\_TS(X) > TS(T) or if write\_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with a timestamp greater than TS(T)-and hence after T in the timestamp ordering-has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.

b. If the condition in part (a) does not occur, then execute the wri te\_item(X)operation of T and set write\_TS(X) to TS(T).

2. Transaction T issues a read\_; tem(X) operation:

a. If write\_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)-and hence after T in the timestamp ordering-has already written the value of item X before T had a chance to read X.

b. If write\_TS(X) :s; TS(T), then execute the read\_item(X) operation of T and set read\_TS(X) to the larger of TS(T) and the current read\_TS(X).

Hence, whenever the basic TO algorithm detects two conflicting operations that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be conflict serializable.

**Ques: What are various recovery techniques in databases? Briefly describe.[GGSIPU2011]**

Answer:

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

* It should check the states of all the transactions, which were being executed.
* A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
* It should check whether the transaction can be completed now or it needs to be rolled back.
* No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

* Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
* Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

**Log-based Recovery**

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

Log-based recovery works as follows −

* The log file is kept on a stable storage media.
* When a transaction enters the system and starts execution, it writes a log about it.

<Tn, Start>

* When the transaction modifies an item X, it write logs as follows −

<Tn, X, V1, V2>

It reads Tn has changed the value of X, from V1 to V2.

* When the transaction finishes, it logs −

<Tn, commit>

The database can be modified using two approaches −

* **Deferred database modification** (No Undo/Redo)− All logs are written on to the stable storage and the database is updated when a transaction commits.(Updation in database is deferred till commit occurs)

The data update in **single user system** goes as follows:

* + A set of transactions records their updates in the log.
  + At commit point under WAL scheme these updates are saved on database disk.
  + After reboot from a failure the log is used to **Redo** all the transactions affected by this failure. **No undo** is required because no AFIM is flushed to the disk before a transaction commits.

The data update in **multi user system** goes as follows:

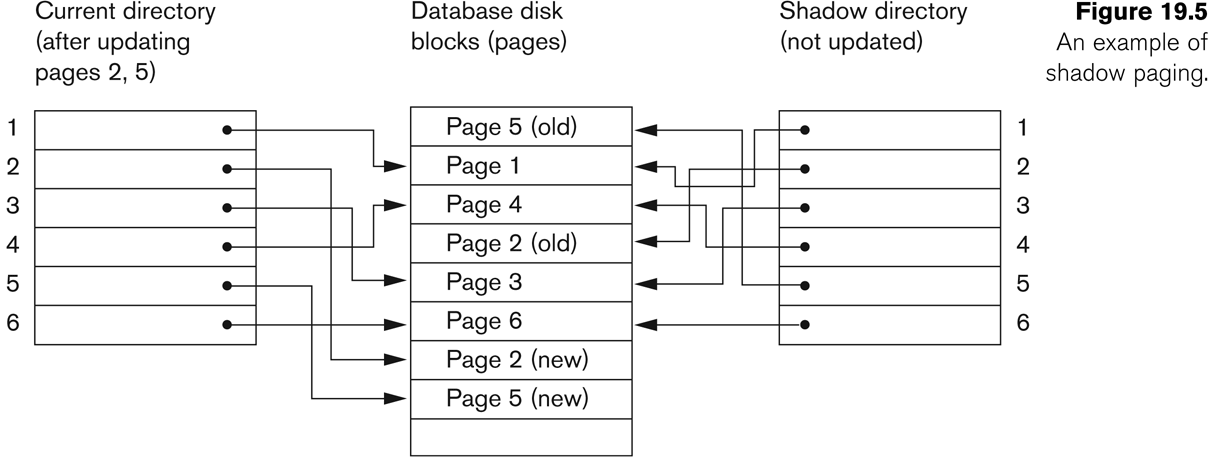
* Two tables are required for implementing this protocol:
  + - **Active table**: All active transactions are entered in this table.
    - **Commit table**: Transactions to be committed are entered in this table.
* During recovery, all transactions of the **commit** table are redone and all transactions of **active** tables are ignored since none of their AFIMs reached the database. It is possible that a **commit** table transaction may be **redone** twice but this does not create any inconsistency because of a redone is “**idempotent**”, that is, one redone for an AFIM is equivalent to multiple redone for the same AFIM.
* **Immediate database modification** (Undo/Redo)− Each log follows an actual database modification. That is, the database is modified immediately after every operation.
  + **Undo/Redo Algorithm** (**Single-user** environment)
    - Recovery schemes of this category apply **undo** and also **redo** for recovery.
    - In a single-user environment no concurrency control is required but a log is maintained under WAL.
    - Note that at any time there will be one transaction in the system and it will be either in the commit table or in the active table.
    - The recovery manager performs:
      * **Undo** of a transaction if it is in the **active** table.
      * **Redo** of a transaction if it is in the **commit** table.
* **Undo/Redo Algorithm** (**Concurrent** execution)
  + - Recovery schemes of this category applies **undo** and also **redo** to recover the database from failure.
    - In concurrent execution environment a concurrency control is required and log is maintained under WAL.
    - Commit table records transactions to be committed and active table records active transactions. To minimize the work of the recovery manager checkpoint is used.
    - The recovery performs:
      * **Undo** of a transaction if it is in the **active** table.
      * **Redo** of a transaction if it is in the **commit** table.

**Recovery via shadow paging**

Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)-say, n-for recovery purposes. A directory with n entries' is constructed, where the ith entry points to the ith database page on disk. The directory is kept in main memory if it is not too large, and all references-reads or writes-to database pages on disk go through it.

When a transaction begins executing, the current directory-whose entries point to the most recent or current database pages on disk-is copied into a shadow directory. The shadow directory is saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is *never modified. When a write\_*item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten. Instead, the new page is written elsewhere-on* some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure below illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory. To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory.



**An example of Shadow Paging**