Topic 1 and 2: Explicit Cursors and Managing Dependencies

Assignment 1: Demonstrate the Usage of CURSOR FOR UPDATE (declare cursor cursor\_name is …. for update and DML ending with WHERE CURRENT of cursor\_name ) with an appropriate example.

Ans :

CREATE OR REPLACE Function FindCourse

( name\_in IN varchar2 )

RETURN number

IS

cnumber number;

CURSOR c1

IS

SELECT course\_number

FROM courses\_tbl

WHERE course\_name = name\_in

FOR UPDATE of instructor;

BEGIN

OPEN c1;

FETCH c1 INTO cnumber;

if c1%notfound then

cnumber := 9999;

else

UPDATE courses\_tbl

SET instructor = 'SMITH'

WHERE CURRENT OF c1;

COMMIT;

end if;

CLOSE c1;

RETURN cnumber;

END;

==================================================================================

ASSIGNMENT2 : Create a View – CREATE VIEW EMP\_VIEW AS SELECT FIST\_NAME, SALARY FROM EMPLOYEES; Writer query to check the status of the view created Add a new column to the EMPLOYEES table. Writer query to check the status of the view created Drop SALARY column from EMPLOYEES table Writer query to check the status of the view created.

Answer :

CREATE TABLE employees

( employee\_number number(10) NOT NULL,

employee\_name varchar2(50) NOT NULL,

department\_id number(10),

salary number(6),

CONSTRAINT employees\_pk PRIMARY KEY (employee\_number)

);

CREATE VIEW EMP\_VIEW AS SELECT employee\_name, salary FROM EMPLOYEES;

SELECT \* from EMP\_VIEW;

ALTER TABLE employees ADD email\_id varchar2(200);

SELECT \* from EMP\_VIEW;

ALTER TABLE EMPLOYEES DROP COLUMN SALARY;

SELECT \* from EMP\_VIEW;

=========================================================================

Topic No 3: Adv Triggers

Assignment 2: Demonstrate the usage of COMPUND TRIGGER with an appropriate example

Answer:

create or replace trigger emp\_ct

for insert on employees compound trigger

type emp\_t is table of employees.employee\_number%type

index by binary\_integer;

emps emp\_t;

before statement is

begin

null;

end before statement;

before each row is

begin

null;

end before each row;

after each row is

begin

emps(emps.count + 1) := :new.employee\_number;

end after each row;

after statement is

begin

forall e in 1 .. emps.count

insert into employees (employee\_number, employee\_name, department\_id, email\_id)

values (5, 'I', emps(e), 'zora@gmail.com');

end after statement;

end;

Assignment 3: Create DDL Trigger that Prevents DROPPING of DB Objects on Saturday and Sunday

Answer :

CREATE TRIGGER TR\_DROPNOT

ON SYS FOR DROP\_TABLE AS

DECLARE

@edate , varchar2(100);

@edate:= select TO\_CHAR(sysdate, 'DAY') from dual;

IF @edate IN ('Saturday', 'Sunday')

BEGIN

PRINT 'Table can't be dropped on Saturday and Sunday' ;

ROLLBACK;

END;

GO

Assignment 5: Write a trigger that prevents decreasing/reducing the Salary of employees

Answer :

CREATE OR REPLACE TRIGGER Salary\_Not\_Decrease

BEFORE INSERT OR UPDATE

ON employees

FOR EACH ROW

BEGIN

if :new.salary < :old.salary then

RAISE\_APPLICATION\_ERROR(-20001,'Salary should not be decreased ') ;

end if;

END;

==================================================================================

Assignment 6: Write a Trigger to track/audit all UPDATE operations done on SALARY column of EMPLOYEES table by DB users – capture the data into a table called AUD\_EMP with columns – USERNAME, UPDATE\_DATE, EMP\_ID, OLD\_SALRY and NEW\_SALARY

Answer :

create or replace trigger TRG\_AUDIT\_EMPLOYEES

after update on EMPLOYEES for each row

DECLARE

UPDATE\_DATE DATE;

BEGIN

UPDATE\_DATE := SYSDATE;

IF UPDATING ('SALARY') THEN

INSERT INTO AUDIT\_EMPLOYEES

VALUES (Audit\_seq.CURRVAL, 'SALARY', :old\_salry, :new\_salary);

END IF;

END;

Topic No 4: Adv. PL-SQL Programming

Assignment 1: Demonstrate the usage and benefits of BULK BINDING and BULK COLLECT with appropriate examples

Answer:

Generally to process large number of records in pl/sql we use Cursors . But cursors process records sequentially. It increases number of context switches between SQL and PL/SQL there by hampers performance. We may overcome this problem using bulk binding.  
  
FOR ALL and BULK COLLECT together are called Bulk Binding.  
  
In Bulk binding instead of handling  records sequentially  the operations are carried out at once in bulk.  There by Bulk binding improves performance by minimizing no of context switches between SQL and PL/SQL.

DECLARE

CURSOR covid IS SELECT employee\_name FROM employee;

TYPE lv\_employee\_name\_tbl is TABLE OF VARCHAR2(50);

lv\_employee\_name lv\_employee\_name\_tbl;

BEGIN

OPEN covid;

FETCH covid BULK COLLECT INTO lv\_employee\_name LIMIT 5000;

FOR c\_employee\_name IN lv\_employee\_name.FIRST..lv\_employee\_name.LAST

LOOP

Dbms\_output.put\_line('Employee :' || c\_employee\_name);

END LOOP;

FORALL i in lv\_employee\_name.FIRST..lv\_employee\_name.LAST

UPDATE employee SET salary = salary + 5000 WHERE employee\_name = lv\_employee\_name(i);

COMMIT;

DBMS\_output.put\_line('Salary Updated');

CLOSE COVID;

END;

/

Assignment 4: Demonstrate the usage on PRAGMA INLINE and NOCOPY

Answer :

***PRAGMA INLINE:***

The INLINE pragma specifies whether a subprogram invocation is to be inlined. Inlining replaces a subprogram invocation with a copy of the invoked subprogram (if the invoked and invoking subprograms are in the same program unit).

Every call to a procedure or function causes a slight, but measurable, performance overhead, which is especially noticeable when the subprogram is called within a loop. Avoiding procedures and functions is not an option, as it goes against the concept of modular programming, making programs bulky and difficult to manage. Automatic subprogram inlining can reduce the overheads associated with calling subprograms, whilst leaving your original source code in its normal modular state. This is done by replacing the subprogram calls with a copy of the code in the subprogram at compile time.

The process of subprogram inlining is controlled by the PLSQL\_OPTIMIZE\_LEVEL parameter and the INLINE pragma. When PLSQL\_OPTIMIZE\_LEVEL=2 (the default), the INLINE pragma determines whether the following statement or declaration should be inlined or not. When PLSQL\_OPTIMIZE\_LEVEL=3, the optimizer may inline code automatically. In this case the INLINE pragma can turn it off inlining for a statement, or increase the likelihood that the optimizer will choose to inline a statement.

***NOCOPY :***

When a parameter is passed by reference the runtime engine sets up the procedure call so that both the actual and the formal parameters point (reference) the same memory location that holds the value of the parameter.  
  
By default OUT and IN OUT parameters are passed by value and IN parameters are passed by reference. When an OUT or IN OUT parameter is modified inside the procedure the procedure actually only modifies a copy of the parameter value. Only when the procedure has finished without exception is the result value copied back to the formal parameter.  
  
Now, if you pass a large collection as an OUT or an IN OUT parameter then it will be passed by value, in other words the entire collection will be copied to the formal parameter when entering the procedure and back again when exiting the procedure. If the collection is large this can lead to unnecessary CPU and memory consumption.  
  
The NOCOPY hint alleviates this problem because you can use it to instruct the runtime engine to try to pass OUT or IN OUT parameters by reference instead of by value.