MAHATMA GANDHI MISSION'S
COLLEGE OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY

(Affiliated to University Of Mumbai)
Kamothe, Navi-Mumbai – 410209
Batch: -2015-2018

Certificate

This is to certify that, Mr./Ms._____
of _____ Class, Examination Seat No. _____ has performed the experiments in subject - _____ of T.Y.B.Sc (CS) SEM - VI , as per guidelines of University of Mumbai.

_____ **_____**
Subject in Charge **Principal**

_____
External Examiner

INDEX

No	Experiments	Date	Sign
1	Study of transaction and locks	3/2/18	
2	Creating and handling deadlocks	10/2/18	
3	Working with packages part 1	17/2/18	
4	Working with packages part 2	17/2/18	
5	Working with data dictionary	24/2/18	
6	Working Dynamic SQL part 1	5/3/18	
7	Working Dynamic SQL part 2	5/3/18	
8	Working with triggers	7/3/18	
9	Working with indexes	10/3/18	

Practical 1

Study of transaction and locks.

In this will demonstrate the use of lock during the transaction on a table;

Step1: Login to scott/tiger and create a table as follows,

```
SQL> create table emp_tab(
  2  emp_id number,
  3  emp_name varchar2(20),
  4  emp_sal number
  5  );

Table created.

SQL> insert into emp_tab values(100,'Jack',20000);

1 row created.

SQL> insert into emp_tab values(200,'Jill',25000);

1 row created.

SQL> insert into emp_tab values(300,'John',20000);

1 row created.

SQL> select * from emp_tab;

    EMP_ID EMP_NAME               EMP_SAL
---------- -------------------- ----------
       100 Jack                      20000
       200 Jill                      25000
       300 John                      20000

SQL> commit;

Commit complete.
```

Step 2: Next connect to sysdba (Conn /as sysdba) in another sql window;

And further try to update the value of it as follows

```
SQL> select * from scott.emp_tab;

    EMP_ID EMP_NAME               EMP_SAL
---------- -------------------- ----------
       100 Jack                      20000
       200 Jill                      25000
       300 John                      20000

SQL> update scott.emp_tab set emp_sal=35000 where emp_id=100;

1 row updated.

SQL> select * from scott.emp_tab;

    EMP_ID EMP_NAME               EMP_SAL
---------- -------------------- ----------
       100 Jack                      35000
       200 Jill                      25000
       300 John                      20000
```

Step 3: Now try to update the table in already running sysdba user session

```
SQL> update scott.emp_tab set emp_sal=50000 where emp_id=100;
1 row updated.
```

Step 4: Similarly try the same in scott user with the same emp_id,

```
SQL> update emp_tab set emp_sal=35000 where emp_id=100;
```

The record won't get updated unless you have committed in sysdba session

```
SQL> commit;
Commit complete.
```

Now you can observe that record is updated in the scott session

```
SQL> update emp_tab set emp_sal=35000 where emp_id=100;
1 row updated.
```

This is demonstration of row level locking in a transaction.

Practical 2

Creating and handling deadlock situation.

Run 2 scott/tiger session

In session1

```
SQL> CREATE TABLE deadlock_1 (
  2    id  NUMBER
  3  );

Table created.

SQL>
SQL> CREATE TABLE deadlock_2 (
  2    id  NUMBER
  3  );

Table created.

SQL>
SQL> INSERT INTO deadlock_1 (id) VALUES (1);

1 row created.

SQL> INSERT INTO deadlock_2 (id) VALUES (1);

1 row created.

SQL> COMMIT;

Commit complete.
```

Before running the pl/sql block make sure the user has permission to dbms_lock,

```
SQL> conn /as sysdba;
Connected.
SQL> grant execute on dbms_lock to scott;

Grant succeeded.

SQL> connect scott/tiger;
Connected.
SQL> _
```

Now run the two blocks simultaneously ,

```
SQL> -- Run in session 1.
SQL> DECLARE
  2    l_deadlock_1_id  deadlock_1.id%TYPE;
  3    l_deadlock_2_id  deadlock_2.id%TYPE;
  4  BEGIN
  5    -- Lock row in first table.
  6    SELECT id
  7    INTO   l_deadlock_1_id
  8    FROM   deadlock_1
  9    WHERE  id = 1
 10    FOR UPDATE;
 11
 12    -- Pause.
 13    DBMS_LOCK.sleep(30);
 14
 15    -- Lock row in second table.
 16    SELECT id
 17    INTO   l_deadlock_2_id
 18    FROM   deadlock_2
 19    WHERE  id = 1
 20    FOR UPDATE;
 21
 22    -- Release locks.
 23    ROLLBACK;
 24  END;
 25  /
DECLARE
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
ORA-06512: at line 16



SQL> DECLARE
  2    l_deadlock_1_id  deadlock_1.id%TYPE;
  3    l_deadlock_2_id  deadlock_2.id%TYPE;
  4  BEGIN
  5    -- Lock row in second table.
  6    SELECT id
  7    INTO   l_deadlock_2_id
  8    FROM   deadlock_2
  9    WHERE  id = 1
 10    FOR UPDATE;
 11
 12    -- Pause.
 13    DBMS_LOCK.sleep(30);
 14
 15    -- Lock row in first table.
 16    SELECT id
 17    INTO   l_deadlock_1_id
 18    FROM   deadlock_1
 19    WHERE  id = 1
 20    FOR UPDATE;
 21
 22    -- Release locks.
 23    ROLLBACK;
 24  END;
 25  /

PL/SQL procedure successfully completed.
```

Only one of them would get completed that is how create a deadlock and handle it using the rollback.

Practical 3

Using packages,

Firstly create a table,

```
SQL> create table pack_emp(
  2  emp_id number,
  3  emp_name varchar2(20)
  4  );

Table created.

SQL> insert into pack_emp values(100,'Mack');

1 row created.

SQL> insert into pack_emp values(101,'Mike');

1 row created.
```

Next create package specification,

```
SQL> create or replace package HR
  2  as
  3  procedure hire (e in number, n in varchar2);
  4  procedure fire (e in number);
  5  end;
  6  /

Package created.
```

Next create package body,

```
SQL> create or replace package body HR
  2  as
  3  procedure hire(e in number, n in varchar2)
  4  is
  5  begin
  6     insert into pack_emp values(e,n);
  7     commit;
  8  end hire;
  9  procedure fire(e in number)
 10  is
 11  begin
 12     delete from pack_emp where emp_id=e;
 13     commit;
 14  end fire;
 15  end;
 16  /
```

Now call and check whether it works

```
SQL> begin
  2  HR.hire(104,'Jimmy');
  3  dbms_output.put_line('Hired employee ');
  4  end;
  5  /

PL/SQL procedure successfully completed.

SQL> select * from pack_emp;

    EMP_ID EMP_NAME
---------- --------------------
       100 Mack
       101 Mike
       104 Jimmy



SQL> begin
  2  HR.fire(104);
  3  dbms_output.put_line('Fired employee ');
  4  end;
  5  /

PL/SQL procedure successfully completed.

SQL> select * from pack_emp;

    EMP_ID EMP_NAME
---------- --------------------
       100 Mack
       101 Mike
```

b. Forward declaration (simply using before defining)

```
SQL> create or replace package forward_decl
  2  as
  3  procedure proc1;
  4  procedure proc2;
  5  end;
  6  /

Package created.



SQL> create or replace package body forward_decl
  2  as
  3  procedure proc1
  4  is
  5  begin
  6     dbms_output.put_line('Calling from proc1');
  7     proc2;
  8
  9  end;
 10  procedure proc2
 11  is
 12  begin
 13     dbms_output.put_line('Calling from proc2');
 14  end;
 15
 16  end;
 17  /

Package body created.

SQL> begin
  2  forward_decl.proc1;
  3  end;
  4  /
Calling from proc1
Calling from proc2

PL/SQL procedure successfully completed.
```

Practical 4

Package using function and public, private constructs.

```
PL/SQL procedure successfully completed.

SQL> create or replace package pack_func
  2  as
  3  temp1 varchar2(20);
  4  function add_digits(a number, b number) return number;
  5  function sub_digits(a number, b number) return number;
  6  end;
  7  /

Package created.

SQL> create or replace package body pack_func
  2  as
  3  temp1 varchar2(20):='Hello';
  4  temp2 varchar2(20);
  5  function add_digits(a number, b number) return number
  6  is
  7  begin
  8
  9     dbms_output.put_line('Addition is');
 10     return (a+b);
 11
 12  end;
 13  function sub_digits(a number, b number) return number
 14  is
 15  begin
 16     dbms_output.put_line('Substraction is');
 17     return (a-b);
 18  end;
 19
 20  end;
 21  /

Package body created.


SQL> declare
  2  val number;
  3  var varchar2(20);
  4  begin
  5  val:=pack_func.add_digits(10,20);
  6  dbms_output.put_line(val);
  7  val:=pack_func.sub_digits(10,20);
  8  dbms_output.put_line(val);
  9  end;
 10
 11  /
Addition is
30
Substraction is
-10

PL/SQL procedure successfully completed.
```

Practical 5

Use of data dictionary user_tables ,all_tables ,dba_tables.

```
SQL> select table_name,owner from all_tables where rownum <10;

TABLE_NAME                     OWNER
------------------------------ ------------------------------
ICOL$                          SYS
CON$                           SYS
UNDO$                          SYS
PROXY_ROLE_DATA$               SYS
FILE$                          SYS
UET$                           SYS
IND$                           SYS
SEG$                           SYS
COL$                           SYS

9 rows selected.

SQL> select table_name from dba_tables where rownum <10;

TABLE_NAME
------------------------------
ICOL$
CON$
UNDO$
PROXY_ROLE_DATA$
FILE$
UET$
IND$
SEG$
COL$

9 rows selected.

SQL>

SQL> select table_name from user_tables;

TABLE_NAME
------------------------------
DEPT
EMP
BONUS
SALGRADE
SHIP
EMP_TAB
DEADLOCK_1
DEADLOCK_2
PACK_EMP

9 rows selected.

SQL> _
```

Practical 6

a) Dynamic sql : use of dbms_sql package

First create a table as follows,

```
SQL> create table employees_tab2(
  2  emp_id number,
  3  emp_name varchar2(20),
  4  salary number);

Table created.

SQL> insert into employees_tab2 values(100,'James',20000);

1 row created.

SQL> insert into employees_tab2 values(100,'James',30000);

1 row created.

SQL> insert into employees_tab2 values(101,'Jimmy',40000);

1 row created.
```

Next create a procedure using dbms_sql

```
SQL> create or replace procedure dynamic(n varchar2)
  2  is
  3      c  number;
  4           stmt  varchar2(500);
  5           r  integer;
  6           var_emp_id integer;
  7           var_emp_name varchar2(50);
  8      var_emp_sal number;
  9
 10  begin
 11           c := dbms_sql.open_cursor;
 12           stmt := 'select * from employees_tab2 where emp_name= :name';
 13           dbms_sql.parse(c,stmt,dbms_sql.native);
 14           dbms_sql.bind_variable(c,':name',n);
 15      dbms_sql.define_column(c,1,var_emp_id);
 16           dbms_sql.define_column(c,2,var_emp_name,50);
 17      dbms_sql.define_column(c,3,var_emp_sal);
 18           r := dbms_sql.execute(c);
 19
 20     loop
 21
 22             if dbms_sql.fetch_rows(c) = 0 THEN
 23                     exit;
 24             end if;
 25
 26           dbms_sql.column_value(c,1,var_emp_id);
 27           dbms_sql.column_value(c,2,var_emp_name);
 28      dbms_sql.column_value(c,3,var_emp_sal);
 29           dbms_output.put_line(var_emp_id|| ' ' || var_emp_name ||' '||var_e
mp_sal);
 30
 31     end loop;
 32
 33           dbms_sql.close_cursor(c);
 34  end;
 35  /

Procedure created.
```

Now call it to execute ,

```
SQL> begin
  2  dynamic('James');
  3  end;
  4  /
100 James 20000
100 James 30000

PL/SQL procedure successfully completed.
```

b) Execute immediate working,

```
SQL> create table temp1(id number,name varchar2(10));

Table created.

SQL> create or replace procedure drop_tab_proc (s in varchar2, n in varchar2)
  2  is
  3  begin
  4     Execute immediate 'Drop '||s||' '||n;
  5  end;
  6  /

Procedure created.

SQL> begin
  2  drop_tab_proc('table','temp1');
  3  end;
  4  /

PL/SQL procedure successfully completed.

SQL> select * from temp1;
select * from temp1
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

As output our drop table procedure works as table does not exists.

Practical 7

Implementing dbms_sql with a parameterized dml statement

```
SQL> create table employees_tab1
  2  (emp_id number,
  3  name varchar2(20),
  4  salary number);

Table created.

SQL> insert into employees_tab1 values(100,'John',5000);

1 row created.

SQL> insert into employees_tab1 values(101,'Mack',6000);

1 row created.

SQL> insert into employees_tab1 values(100,'Jane',8000);

1 row created.

SQL> select * from employees_tab1;

    EMP_ID NAME                      SALARY
---------- -------------------- ----------
       100 John                       5000
       101 Mack                       6000
       100 Jane                       8000
```

```
SQL> create or replace procedure param_dml(n varchar2,s number)
  2  is
  3      stmt varchar2(200);
  4      c number;
  5      dummy number;
  6  begin
  7      stmt:='update employees_tab1 set salary = :salary where name = :emp_nam
e';
  8      c:= dbms_sql.open_cursor;
  9      dbms_sql.parse(c,stmt,dbms_sql.native);
 10      dbms_sql.bind_variable(c,':emp_name',n);
 11      dbms_sql.bind_variable(c,':salary',s);
 12      dummy:=dbms_sql.execute(c);
 13      dbms_sql.close_cursor(c);
 14  end;
 15  /

Procedure created.

SQL> begin
  2  param_dml('John',10000);
  3  end;
  4  /

PL/SQL procedure successfully completed.

SQL> select * from employees_tab1;

    EMP_ID NAME                      SALARY
---------- -------------------- ----------
       100 John                      10000
       101 Mack                       6000
       100 Jane                       8000
```

Practical 8

Creating and working with triggers

First we will create a table of product and then perform a backup of it using triggers;

```
SQL> create table products(
  2  product_id number,
  3  product_name varchar2(20)
  4  );

Table created.
```

Now creating a backup table

```
SQL> create table products_backup(
  2  product_id number,
  3  product_name varchar2(20)
  4  );

Table created.
```

Next we create a trigger which will insert record in products_backup after products table is filled.

```
SQL> create or replace trigger product_history
  2  after insert on products
  3  for each row
  4  begin
  5     insert into products_backup values(:new.product_id,:new.product_name);
  6  end;
  7  /

Trigger created.
```

Next lets insert records to check if the triggers work

```
SQL> insert into products values(1001,'kenley');

1 row created.

SQL> insert into products values(1002,'Cola');

1 row created.

SQL> insert into products values(1003,'Pepsi');

1 row created.

SQL> insert into products values(1004,'Bisleri');

1 row created.

SQL> select * from products;

PRODUCT_ID PRODUCT_NAME
---------- --------------------
      1001 kenley
      1002 Cola
      1003 Pepsi
      1004 Bisleri

SQL> select * from products_backup;

PRODUCT_ID PRODUCT_NAME
---------- --------------------
      1001 kenley
      1002 Cola
      1003 Pepsi
      1004 Bisleri
```

As shown in the output both the tables have records , where products_backup is filled with a row –level trigger.

For updating the backup create a new backup trigger

```
SQL> create or replace trigger trigger_for_update
  2  after update on products
  3  for each row
  4  begin
  5     update products_backup set product_name=:new.product_name
  6     where product_id=1004;
  7  end;
  8  /

Trigger created.
```

Next perform update operation

```
SQL> update products set product_name='PaperBoat' where product_id=1004;

1 row updated.
```

Now lets check both the tables

```
SQL> select * from products
  2  ;

PRODUCT_ID PRODUCT_NAME
---------- --------------------
      1001 kenley
      1002 Cola
      1003 Pepsi
      1004 PaperBoat

SQL> select * from products_backup;

PRODUCT_ID PRODUCT_NAME
---------- --------------------
      1001 kenley
      1002 Cola
      1003 Pepsi
      1004 PaperBoat
```

As shown product_backup has the updated value of products table;

Now we'll create trigger for deleting records;

```
SQL> create or replace trigger trigger_for_delete
  2  after delete on products
  3  for each row
  4  begin
  5     delete from products_backup where product_id=:old.product_id;
  6  end;
  7  /

Trigger created.
```

Next execute an delete query

```
SQL> delete from products where product_id=1004;

1 row deleted.
```

Now lets check from both the tables

```
SQL> select * from products;

PRODUCT_ID PRODUCT_NAME
---------- --------------------
      1001 kenley
      1002 Cola
      1003 Pepsi
```

```
SQL> select * from products_backup;

PRODUCT_ID PRODUCT_NAME
---------- --------------------
      1001 kenley
      1002 Cola
      1003 Pepsi
```

As shown the trigger works for deleting the record;

Statement and row level trigger

```
SQL> create table employees(
  2  employee_id number,
  3  first_name varchar2(20),
  4  salary number
  5  );

Table created.

SQL> insert into employees values(100,'Mack',5000);

1 row created.

SQL> insert into employees values(101,'John',6550);

1 row created.

SQL> insert into employees values(102,'Jane',7800);

1 row created.
```

```
SQL> create or replace trigger statement_level_trig
  2  before update on employees
  3  begin
  4     dbms_output.put_line('Before update statement trigger');
  5  end;
  6  /

Trigger created.

SQL> create or replace trigger row_level_trig
  2  before update on employees
  3  for each row
  4  begin
  5     dbms_output.put_line('Before update row level trigger');
  6  end;
  7
  8  /

Trigger created.

SQL> update employees set salary=10000 where employee_id>100;
Before update statement trigger
Before update row level trigger
Before update row level trigger

2 rows updated.
```

```
SQL> drop trigger row_level_trig;

Trigger dropped.

SQL> drop trigger statement_level_trig;

Trigger dropped.
```

Practical 9

Creating and maintaing indexes

```
SQL> create table supplier(
  2  supplier_name varchar2(20),
  3  city varchar2(20)
  4  );

Table created.

SQL> create index supplier_indx on supplier(supplier_name);

Index created.
```

```
SQL> drop index supplier_indx;

Index dropped.

SQL> create index supplier_indx on supplier(supplier_name,city);

Index created.

SQL> alter index supplier_indx rename to supp_indx;

Index altered.
```