# CMSC 4003
# Lab Assignment 10: Transactions and Isolation Levels

Name:

Due: See the due date in D2L calendar.

1. Preparation: Open two SQL*Plus windows.
   You need to open two SQL*Plus windows on your screen. We are going to pretend that there are two users, one window for each user. Place the two windows so that one window is at the top of the screen and the other is at the bottom of the screen. We name the top SQL*Plus window as TOP and the bottom window as BOT.

2. Create tables that keep track of an inventory of parts of a Doll and an assembled doll. Execute the following SQL statements in either TOP or BOT.

```
drop table DollParts;
create table DollParts (
name   varchar2(10) primary key,
cnt    number check (cnt >= 0));

insert into DollParts(name, cnt) values('HEAD', 17);
insert into DollParts(name, cnt) values('BODY', 17);
insert into DollParts(name, cnt) values('ARM', 17);
insert into DollParts(name, cnt) values('LEG', 17);
commit;

create table Dolls (
name   varchar2(20) primary key,
cnt    number);

insert into Dolls(name, cnt) values('Barbie', 0);
insert into Dolls(name, cnt) values('Ken', 0);
commit;
```

3. Setting the isolation levels.
   SQL standard defines four isolation levels: Read Uncommitted, Read Committed, Repeatable Read and Serializable. They provide the degree of independence between concurrently executing interacting transactions. At this part of the lab we use the default isolation level in Oracle, namely, Read Committed.

4. Using transactions.
   The transaction in which we are interested is to deduct from the inventory (DollParts) those parts that we need to make a doll and increment the count of that doll by 1. The

problem that we may get into is this: what if two users are doing this at the same time?

Now run the following SQL statements in the TOP and BOT windows accordingly.

```
TOP:  select * from DollParts;
BOT:  select * from DollParts;
TOP:  update DollParts set cnt=cnt-1 where name='HEAD';
TOP:  update DollParts set cnt=cnt-1 where name='BODY';
BOT:  select * from DollParts;
TOP:  select * from DollParts;
TOP:  commit;
BOT:  select * from DollParts;
```

What are the 'HEAD' count and 'BODY' count from the second BOT query?
Answer: In BOT window
The 'HEAD' count from the second BOT query is 17
The 'BODY' count from the second BOT query is 17

What are the 'HEAD' count and 'BODY' count from the second TOP query?
Answer: In TOP window
The 'HEAD' count from the second TOP query is 16
The 'BODY' count from the second TOP query is 17

What are the 'HEAD' count and 'BODY' count from the third BOT query?
Answer: In BOT window
The 'HEAD' count from the second TOP query is 16
The 'BODY' count from the second TOP query is 16

Note that, due to the use of MVCC, Oracle does not have any implicit read lock, that is, before a read operation Oracle will not try to put a shared read lock on the data item. At the Read Committed isolation level, MVCC guarantees that a read operation on a data item will only get <u>the latest committed</u> value of that data item before the read, unless the data item is modified by an update of the same transaction in which the read operation exists. This is why the results we get from the second and the third query in BOT are different, whereas the result we get from TOP is consistent with its update operation.

Why did Oracle choose MVCC, instead of using implicit read locks for its concurrency control.
Answer: MVCC allows non-blocking reads whereas implicit shared locks doesn't. MVCC make sure that the read operation on the data item will get the last committed

values of the data item before the read, this happens unless the user make a modification by an update in the same transaction in which read operation exists.

What we get from the second query in BOT is an inventory in a partial condition. BOT wants to check if there are enough parts for a doll, but it might get the wrong information because there is a doll-build in progress (TOP window) and one head and one body have already been used by TOP. If BOT uses the wrong information, database consistency may be affected. We will deal with this problem later in the lab.

5. Transactions with writes.

   Oracle does have implicit long duration write locks (LDW). Before an update operation on a data item, Oracle requires that an exclusive write lock should be obtained for that data item. If a write lock cannot be obtained, the transaction will hang and wait until the LDW of that data item is released. The following example illustrates this.

   Note the keyword "implicit" means that the transaction does not need to explicitly lock a data item. Oracle will request the lock automatically for the transaction. Transactions can also explicitly ask for a write lock as we will see later.

   Run the following SQL statements in TOP and BOT accordingly. Note that BOT will hang after the update statement. See explanations below when it happens.

```
TOP:  Update DollParts set cnt=cnt-1 where name='HEAD';
TOP:  Update DollParts set cnt=cnt-1 where name='BODY';
TOP:  Select * from DollParts;
BOT:  Update DollParts set cnt=cnt-1 where name='HEAD';
```

   The above statements reflect a situation when two doll-builds start at the same time. Notice that BOT has hung on its update statement. This is due to the LDW held by TOP on the HEAD and BODY rows of the table DollParts. BOT will be suspended until TOP commits, at which point the LDW is released by TOP.

   Now run the following statements in TOP to finish the transaction:

```
TOP:  Update DollParts set cnt=cnt-2 where name='ARM';
TOP:  Update DollParts set cnt=cnt-2 where name='LEG';
TOP:  commit;
```

   Now complete the transaction in BOT by using commit. Note that you can finish BOT successfully now.

```
BOT:  commit;
```

IMPORTANT: You have to complete the transaction in BOT, especially the commit statement. Otherwise, BOT will hold an LDW lock on the row and no other transactions can access it. If you accidentally close the BOT window before commit, SQL*Plus will commit it for you. However, in a program such as a PHP script, if the LDW is not released by a commit or rollback, it will only be released until the system timeouts the LDW. Before that, you cannot write to that row anymore. Of course if you have the administrator's priority, you can manually release that lock by killing the connection session. But if you are not an administrator, you want to be very careful on this issue. In our environment, if a lock cannot be released, you are allowed to kill the session. Refer to the article "Kill a session in SQL*Plus" in D2L under Lectures: "1. Relational Data Model".

6. Transactions that explicitly ask for a lock.
   Run the following statements in TOP and BOT accordingly:

```
TOP:  select * from Dolls where name='Barbie' for update;
BOT:  select cnt from Dolls where name='Barbie';
BOT:  update Dolls set cnt=cnt+1 where name='Barbie';
```

   Note that BOT blocks. The reason is that TOP uses a "SELECT ... FOR UPDATE" statement to explicitly obtain an LDW on row 'Barbie' of table Dolls. Although BOT is still allowed to read row 'Barbie' (MVCC, no read lock in Oracle), when BOT tries to update the row, it cannot obtain the LDW on 'Barbie'. Therefore, BOT has to block and wait. Now finish the transactions in TOP and BOT:

```
TOP: commit;
BOT: commit;
```

   Note that "SELECT ... FOR UPDATE" can be used to solve the problem that we proposed at step 4. The problem is that when we check the value of a table using SELECT, the value may not reflect the actual situation since it may be modified by some uncommitted transaction at the time of the check. To solve the problem, we use "SELECT ... FOR UPDATE" instead of "SELECT" when we want to check some value before an update, e.g., check the availability of doll parts. The idea is that if some transaction is modifying the value, "SELECT ... FOR UPDATE" will not be able to obtain the LDW—it has to wait. Once the "SELECT ... FOR UPDATE" statement obtains the LDW, the value it returns can be trusted since no other transaction can modify the data any more. This idea demonstrates the use of locks for concurrency control. An alternative is to use the isolation levels provided by the system for concurrency control. Isolation levels are less likely to cause deadlocks, which will be discussed in the next section.

7. Deadlock.
   Run the following statements in TOP and BOT accordingly:

```
TOP:  select * from Dolls where name='Barbie' for update;
BOT:  select * from Dolls where name='Ken' for update;
TOP:  select * from Dolls where name='Ken' for update;
BOT:  select * from Dolls where name='Barbie' for update;
```

   Notice that neither TOP nor BOT can proceed any more until Oracle detects the deadlock situation. Explain why the deadlock occurs.
   Answer: In the top table we are calling an update on the row 'BARBIE' and in the bottom table we are calling an update on the row 'KEN'. Which will create long duration write locks LDW on both the rows of Dolls table. So, in the 2nd TOP the user calls row 'KEN' then the TOP blocks. Similarly, in the 2nd BOT the user calls row 'BARBIE' then BOT blocks.
   Each session is holding a lock that other session needs, but no one can proceed further because they are waiting other locks to be released. This creates a situation of DEADLOCK.

   Now finish the transactions in TOP and BOT:

```
TOP: commit;
BOT: commit;
```

8. Rollback.
   There are two ways to end a transaction: commit or abort. The command to abort a transaction is ROLLBACK. In either TOP or BOT, run the following statements:

```
select * from Dolls;
update Dolls set cnt=cnt+1 where name='Barbie';
update Dolls set cnt=cnt+1 where name='Ken';
select * from Dolls;
rollback;
select * from Dolls;
```

   Note that, after rollback, the modified rows in Dolls are restored to the state before the transaction starts. Rollback can be used when we want to cancel a transaction.

For example, if one part for a doll is not available in the inventory, we can rollback all the other items we have taken. Like commit, rollback guarantees the atomicity of a transaction.

9. Different Isolation Levels.
   So far, we have run transactions under the Oracle default isolation level: Read Committed. Isolation levels provide an easy way for concurrency control (It is better than manipulating locks directly). Starting from this section, we will study more isolation levels. ANSI SQL defines four isolation levels, which are abbreviated as follows.

   - Read Uncommitted: RU
   - Read Committed: RC
   - Repeatable/Phantom Read: RR
   - Serializable: SR

   The four isolation levels allow different combinations of the following four phenomena: Dirty Read, Lost Update, Nonrepeatable Read and Phantom Read, as illustrated in the following table. Oracle only offers two levels among the four.

|  | Dirty Read | Lost Update | Non-repeatable Read | Phantom Read |
|---|---|---|---|---|
| RU | Allowed | Allowed | Allowed | Allowed |
| RC (Oracle) | Not Allowed | Allowed | Allowed | Allowed |
| RR | Not Allowed | Not Allowed | Not Allowed | Allowed |
| SR (Oracle) | Not Allowed | Not Allowed | Not Allowed | Not Allowed |

   We will run two concurrent transactions and observe their interactions (i.e., presence or absence of the four phenomena) at the two isolation levels that are supported by Oracle.

10. Run the following SQL statements in either TOP or BOT to create a table that will be used later.

```
create table Account (
AcctName    varchar2(30) primary key,
AcctBal     number);

insert into Account values('John', 100);
insert into Account values('David', 200);
insert into Account values('Mary', 300);
insert into Account values('Cathy', 100);
commit;
```

11. Two transactions are defined as follows, one for TOP and the other for BOT. Note that the step numbers are indicated in the transactions as comments. These step numbers are used for specifying schedules in the following pages. Also note that in Oracle, PL/SQL code segment can also be part of a transaction, as we will see.

```
TOP:

-- steps 1 & 2
var x number
begin select AcctBal into :x from Account where AcctName='John';
end;
/
print x

-- step 3
begin :x := :x + 10; end;
/
print x

-- step 4
begin update Account set AcctBal=:x where AcctName='John'; end;
/

-- step 5
commit; or rollback;

BOT:
-- steps 1' & 2'
var x number
begin select AcctBal into :x from Account where AcctName='John';
end;
/
print x

-- step 3'
begin :x := :x + 20; end;
/
print x

-- step 4'
begin update Account set AcctBal=:x where AcctName='John'; end;
/

-- step 5'
commit;
```

Note that the above two transactions contain a mixture of SQL, PL/SQL and SQL*Plus statements/commands. "var x number" is an SQL*Plus command that declares a (global) variable in the SQL*Plus session. The variable is used by those PL/SQL code segments as an external variable (note the colon before x). Remember that the concept of external variables was mentioned in the PHP lab assignment. "print x" is an SQL*Plus command to display the value of x. Certainly, x is not considered external to SQL*Plus itself. Therefore the colon is not needed.

"SELECT ... INTO" is a PL/SQL statement. It assigns an attribute value or several attribute values from a tuple into a variable or several variables. When using the "SELECT ... INTO" statement, you MUST guarantee that only one tuple is returned by the SELECT. If several tuples are returned by the SELECT statement, then "SELECT ... INTO" cannot be used. You need to use a cursor instead.

12. Complete tables for different isolation levels.
As we have mentioned earlier, Oracle uses MVCC and write-locks to implement only two of the four ANSI isolation levels, Read Committed and Serializable.

To fill out the tables, you will run in TOP and BOT windows the two transactions defined in the previous section (Section 11) in an interleaved fashion, step by step, according to the given schedule. The tables can be completed based on your observations. You will see that schedules for some phenomena are allowed under Oracle's Serializable level.

When you run the transactions in Oracle, <u>you need to set the isolation level at the beginning of each transaction by using the following SQL statements</u>. You need to set the isolation levels in <u>both</u> TOP and BOT.

```
set transaction isolation level read committed;
set transaction isolation level serializable;
```

Again, Oracle has only implemented the two isolation levels (Read Committed and Serializable). We have mentioned in Section 4 that, at the Read Committed level, Oracle uses MVCC to guarantee that a read operation on a data item gets <u>the latest committed value before the read operation</u>.

At the Serializable level, Oracle still does not use read locks. For read operations, Oracle, through MVCC, guarantees that a read operation on a data item only gets <u>the latest committed value before the transaction starts</u>. The write operations are controlled differently in the Serializable level. Well-formed write (WFW) and long duration write lock (LDW) are used. At the serializable level, Oracle offers lost update detection: a transaction T1 cannot write to a data item that has been written and committed by another transaction T2 after T1 starts, i.e., the w1[x] in schedule (r1[x], w2[x], c2, w1[x]) is not allowed at Oracle Serializable level. Even though the LDW on x is released by T2 after c2, Oracle will check the timestamp of x, if its timestamp is later than the starting time of T1, w1[x] is not allowed—an error "can't serialize access for this transaction" will occur. Thus Oracle provides more consistency at the Serializable isolation level although, as we know, Oracle's Serializable level is not truly serializable (i.e., the schedule is not guaranteed to be equivalent to a serial schedule).

(a) **Dirty Read** (i.e., w1[x], r2[x], a1[x])
The following schedule involves a dirty read. As we know (see the table on page 6) in standard ANSI isolation levels, neither Read Committed nor Serializable will allow the schedule to complete since it contains a dirty read.
1, 2, 3, 4, 1', 2', 5r, 3', 4', 5'c
Run the schedule in Oracle and fill out the table below. Note that you need to reset the table to its original value before trying the next isolation level. So run the following statement in one of the two windows before starting the schedule.

```
update Account set AcctBal=100 where AcctName='John';
commit;
```

Also do not forget to set the corresponding isolation levels at the beginning of each transaction.

| | What will BOT see if dirty read happens? | Is the schedule completed? | If no, explain why | Actual account value, if the schedule is completed |
|---|---|---|---|---|
| RC | Bot can see the uncommitted value of AcctBal | **YES** | | **120** |
| SR | | **YES** | | **120** |

You have seen that the schedule that contains a dirty read can actually be completed in both Read Committed and Serializable levels in Oracle. Why?
Answer: This schedule with a "dirty read" can complete under Oracle Read Committed and even Serializable level of isolation because Oracles MVCC isolates transactions from viewing only the committed data, hence true "dirty read" cannot occur, yet the schedules can go forward without conflict.

(b) **Lost Update** (i.e., r1[x], r2[x], w1[x], c1, w2[x], c2)
The following schedule involves a lost update. As we know (see the table on page 6) in standard ANSI isolation levels, Read Committed allows lost updates. However, Serializable will not allow the schedule to complete if it contains a lost update.

1, 2, 1', 2', 3, 4, 5c, 3', 4', 5'c

Run the schedule in Oracle and fill out the table below. Note that you need to reset the table to its original value before trying the next isolation level. So run the following statement in one of the two windows before starting the schedule.

```
update Account set AcctBal=100 where AcctName='John';
commit;
```

Also do not forget to set the corresponding isolation levels at the beginning of each transaction.

| | John's account value if TOP & BOT run sequentially | Is the schedule completed? | If no, explain why | Actual account value, if the schedule is completed |
|---|---|---|---|---|
| RC | 120 | **YES** | | **120** |
| SR | | **NO** | Because the AcctBal has been committed in the TOP. | **110** |

Complete the transaction in BOT:

BOT: commit;

(c) **Non-repeatable Read** (i.e., r1[x], w2[x], c2, r1[x])
Revise the BOT transaction as follows:

```
BOT:

-- step 1'
```

```
set serveroutput on
var x number

-- step 2'
begin select AcctBal into :x from Account where AcctName='John';
end;
/
print x

-- step 3'
declare
  bal number;
begin
  if :x=100  then
    select AcctBal into bal from Account where AcctName='John';
    dbms_output.put_line(bal);
  end if;
end;
/

-- step 4'
commit;
```

The following schedule involves a non-repeatable read. As we know (see the table on page 6) in standard ANSI isolation levels, Read Committed allows non-repeatable reads. However, Serializable will not allow the schedule to complete if it contains a non-repeatable read.

1', 2', 1, 2, 3, 4, 5c, 3', 4'c

Run the schedule in Oracle and fill out the table below. Note that you need to reset the table to its original value before trying the next isolation level. So run the following statement in one of the two windows before starting the schedule.

```
update Account set AcctBal=100 where AcctName='John';
commit;
```

Also do not forget to set the corresponding isolation levels at the beginning of each transaction.

|  | The value BOT expects to print | Is the schedule completed? | If no, explain why | The value BOT prints, if the schedule is completed |
|---|---|---|---|---|

|  |  |  |  |  |
|---|---|---|---|---|
| RC | **110** | **YES** |  | **110** |
| SR |  | **YES** |  | **100** |

You have seen that the schedule that contains a non-repeatable read can actually be completed in the Serializable level in Oracle. Why?
Answer: A non-repeatable read would allow completion of a schedule in Oracle's Serializable isolation, since transactions are executed on a consistent snapshot, and repeated reads will return the same result without reflecting changes after the transaction started execution.

(d) **Phantom Read** (i.e., r1[x], w2[x], c2, r1[x], in which r1[x] reads multiple tuples)
Revise the BOT transaction as follows:

```
BOT:

-- step 1'
set serveroutput on
var x number

-- step 2'
begin select AVG(AcctBal) into :x from Account where AcctBal>100;
end;
/
print x

-- step 3'
declare
  avgBal number;
begin
  if :x=250 then
    select AVG(AcctBal) into avgBal from Account where
AcctBal>100;
    dbms_output.put_line(avgBal);
  end if;
end;
/
print x

-- step 4'
commit;
```

The following schedule involves a phantom read. As we know (see the table on page 6) in standard ANSI isolation levels, Read Committed allows phantom reads. However, Serializable will not allow the schedule to complete if it contains a phantom read.

1’, 2’, 1, 2, 3, 4, 5c, 3’, 4’c

Run the schedule in Oracle and fill out the table below. Note that you need to reset the table to its original value before trying the next isolation level. So run the following statement in one of the two windows before starting the schedule.

```
update Account set AcctBal=100 where AcctName='John';
commit;
```

Also do not forget to set the corresponding isolation levels at the beginning of each transaction.

| | The value BOT expects to print | Is the schedule completed? | If no, explain why | The value BOT prints, if the schedule is completed |
|---|---|---|---|---|
| RC | **203.33** | **YES** | | **203.3** |
| SR | | **YES** | | **250** |

You have seen that the schedule that contains a phantom read can actually be completed in the Serializable level in Oracle. Why?
Answer: Phantom reads are avoided by Oracle's Serializable isolation, wherein the transaction views only the state of the database as of the beginning, with later conflicts detected and resolved for strict consistency, as if the execution is serial.

13. Long duration transactions
At step 6, we discussed how to use an explicit LDW (SELECT ... FOR UPDATE) to guarantee a status check is valid at the Read Committed level. However, there are still issues in the scheme. First of all, using the Serializable isolation level can solve

the problem without even using the explicit lock. For example, if you check the part counts in DollParts and there are parts available, you will use those parts by making changes to the table. If there is any change in the table by another transaction in between, Oracle will not allow your transaction to proceed at the Serializable level. However, the Serializable isolation level is usually too strict for your application in terms of concurrency. Therefore, you want to use the Read Committed level in most cases. Then, explicit LDW is still needed to guarantee the consistency of the database. How to use the LDW properly so that concurrency is not too badly affected by the lock is a problem now.

Consider an airline ticketing system. It uses the following table to keep track of the seats left for each flight.

```
drop table flight;
create table flight(
  flightno number primary key,
  seats number
);

insert into flight values (101, 1);
insert into flight values (102, 1);
commit;
```

Now consider the following PL/SQL code in a transaction:

```
set serveroutput on
declare
  seats101 number;
  seats102 number;
begin
  select seats into seats101 from flight where flightno=101 for
update;
  select seats into seats102 from flight where flightno=102 for
update;
  if seats101 > 0 and seats102 > 0 then
    -- let the customer know the availability of the seats
    -- get customer payment (this step takes a long time)
    update flight set seats=seats-1 where flightno=101;
    update flight set seats=seats-1 where flightno=102;
    commit;
    dbms_output.put_line('Committed');
  else
    rollback;
    dbms_output.put_line('No seats, rollback');
  end if;
end;
/
```

The above code can guarantee the consistency of the flight table.  Why?
Answer: The code will assure consistency in the flight table, whereby through SELECT... FOR UPDATE, it acquires exclusive row-level locks on flightno=101 and flightno=102 to avoid concurrent modifications. It checks seat availability under the lock and only proceeds with an update if conditions are met, which makes this atomic, as either both rows are updated together or neither of them. The locking mechanism eliminates race conditions that can result in incorrect or inconsistent seat availability during the course of the transaction.

However, the above code contains a concurrency problem: it usually takes a long time for a customer to provide/enter payment information.  The rows are kept locking for too long during the period, which significantly affects the concurrency of the whole system.  A **long duration transaction** is a transaction that involves human interactions.  To guarantee database concurrency, it is highly desirable for a transaction not to hold any locks during its human interaction phase.  Now check the following PL/SQL code:

```
set serveroutput on
declare
  seats101 number;
  seats102 number;
begin
  select seats into seats101 from flight where flightno=101;
  select seats into seats102 from flight where flightno=102;
  if seats101 > 0 and seats102 > 0 then
    -- let the customer know the availability of the seats
    -- get customer payment (this step takes a long time)
    select seats into seats101 from flight where flightno=101 for
update;
    select seats into seats102 from flight where flightno=102 for
update;
    if seats101 > 0 and seats102 > 0 then
      update flight set seats=seats-1 where flightno=101;
      update flight set seats=seats-1 where flightno=102;
      commit;
      dbms_output.put_line('Committed');
    else
      rollback;
      dbms_output.put_line('No seats, rollback');
    end if;
  else
    dbms_output.put_line('No seats');
  end if;
end;
/
```

The above code solves the problem of a long duration transaction.  Why?
Answer: This code addresses the long-duration transaction problem by checking seat availability without acquiring locks so that concurrent access can be allowed while the customer is making the payment. Confirmation of a payment-the transaction acquires LDW with the use of SELECT. FOR UPDATE in order not to allow any other transaction to modify the rows. Thus, it reduces the duration of locks and thereby improves concurrency with the system's performance and consistency.