RMIT UNIVERSITY

School of Computing Technologies

ISYS1101/ 1102 Database Applications

Week 7: Transaction management and Concurrency Control

Semester 2 2022

1 Objective

The objective of this tute/lab session is to explore the transaction management and concurrency control in Oracle. More specifically, we explore:

- The impact of COMMIT and ROLLBACK of transactions;
- The impact of two isolation levels: READ COMMITTED and SERIALIZABLE levels. We execute two sets of SQL statements in two terminal windows (or two SQL Developer sessions.)

2 Preparation Tasks

This activity requires the access to your Oracle database account. Since you may not have used your Oracle account since the completion of the first assignment, you may need to refresh your account access credentials and how to access your Oracle account using Oracle SQL Developer. In the event you have forgotten your Oracle password, you can change it at: access.csit.rmit.edu.au
(For more information on your Oracle account, revisit Week 1 Tute/Lab sheet.)

For this activity, you will require Movies sample database. If it doesn't exist, or you have made changes to the data in tables, it is recommended that you re-build it from scratch. Download the Week07-Sample.sql script from the Week 07 Module on canvas, save it locally, and run it on SQL Developer to rebuild the Movies database.

3 Exercises

3.1 Commit or Rollback?

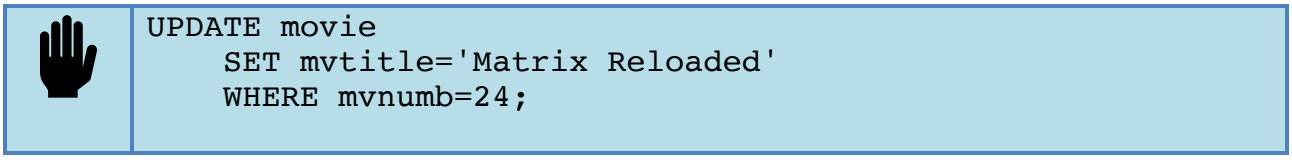
Activity 1:
In this activity, you experiment with the behaviour of the COMMIT and ROLLBACK statements. Basically, you finish the current transaction with one of these statements. (If you shut down your SQL session, that will also commit the current transaction).

First of all, check if you have successfully created and populated the movie table.

```
SELECT * FROM movie;
```

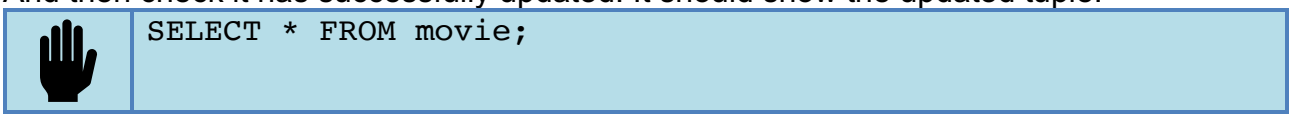
It should show 24 rows in it.

Now, do the following modification to the movie table.



```
UPDATE movie
    SET mvtitle='Matrix Reloaded'
    WHERE mvnumb=24;
```

And then check it has successfully updated. It should show the updated tuple.

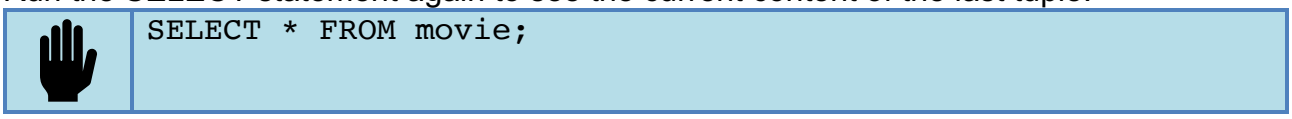


```
SELECT * FROM movie;
```

Now, rollback the transaction.

```
ROLLBACK;
```

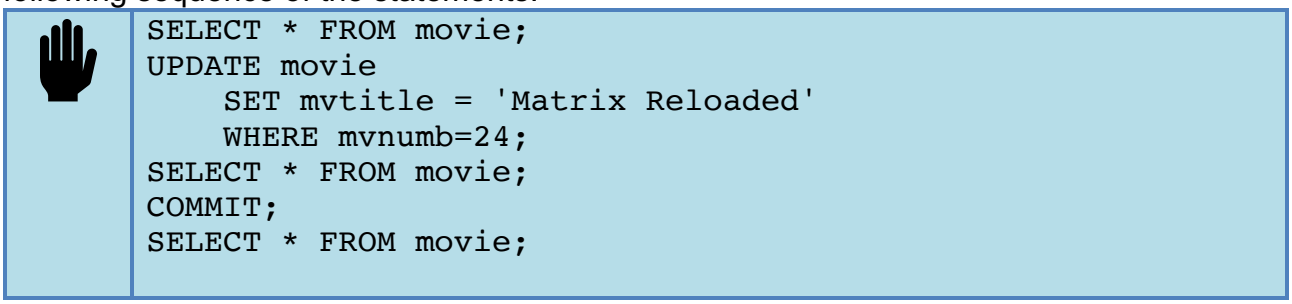
Run the SELECT statement again to see the current content of the last tuple.



```
SELECT * FROM movie;
```

Did you notice the movie title has gone back to its initial value -- "Grapes of Wrath"?

Start again the above exercise, replacing ROLLBACK with COMMIT statement. Try the following sequence of the statements.



```
SELECT * FROM movie;
UPDATE movie
    SET mvtitle = 'Matrix Reloaded'
    WHERE mvnumb=24;
SELECT * FROM movie;
COMMIT;
SELECT * FROM movie;
```

The last SELECT statement should show the updated movie title, since it reads it from the committed transaction.

3.2 Concurrency Control

Activity 2:

In this activity, you investigate the behaviour of concurrent (interleaved) transactions. To experiment with concurrent transactions, you should open two SQL sessions on two windows. Make sure both of these windows are visible (side-by-side), otherwise you may not notice the changes happening in both windows. In the following activities, enter statements in red colour in first window, and statements in blue colour in second window. The following schedule illustrates the order of the actions in two transactions you are going to execute.

Transaction 1 (T1)	Transaction 2 (T2)
SELECT * FROM movie; You see the old value.	

	UPDATE movie SET mvtitle = 'X-Men 2' WHERE mvnumb = 24;
SELECT * FROM movie; You see the old value.	
	Now, COMMIT the second transaction. COMMIT;
SELECT * FROM movie; You see the new value	

Activity 3: Avoiding Lost Update:

Oracle uses a row-level locking mechanism to control concurrent read and write operations. The following activity demonstrates how this locking mechanism controls the concurrency, in order to avoid lost update problem.

Note: Make sure that both sessions (in two windows) are fresh before the following activities. To ensure they are clean, start with running COMMIT statement in both sessions to guarantee that no uncommitted operations exist in each session.

As in the previous activity, enter red statements in window 01, and blue statements in window 02.

Transaction 1 (T1)	**Transaction 2 (T2)**
SELECT * FROM movie; You see the old value.	
	UPDATE movie SET mvtitle = 'X-Men 2' WHERE mvnumb = 24;
UPDATE movie SET mvtitle = 'Johny English' WHERE mvnumb = 24; Can you successfully continue the transaction 01? What happened when you ran the UPDATE statement in transaction 01?	
	Now, COMMIT the second transaction. COMMIT;
What happened to the transaction 01, after committing the transaction 02?	

3.3 Isolation Levels

In the Activity 02, you experimented with two transactions executing concurrently. The default behaviour of the Oracle concurrency control didn't allow dirty reads to happen. But, it has allowed non-repeatable reads to happen. However, you can change this behaviour in Oracle transactions.

Oracle has two isolation levels. You can define the required isolation level at the beginning of the transaction.
The available isolation levels are:

READ COMMITTED -- This is the default isolation level. When this isolation level is in use, each query executed by a transaction sees only data that was committed before the query (not the transaction) began. Therefore, the query will never read "dirty" data.
Since Oracle does not prevent other transactions from modifying the data read by a query, that data may be changed by other transactions between two executions of the query. (That's what happened in activity 02). Thus, a transaction that executes a given query twice may experience non-repeatable reads.

SERIALIZABLE -- SERIALIZABLE transactions see a fixed snapshot of the database, established as of the beginning of the transaction. This transaction mode prevents read/write and write/read and write/write conflicts that would cause serialisability failures. Once a SERIALIZABLE transaction is started, all reads within that transaction will see only committed data as of the start of that transaction. (The only new updates that the SERIALIZABLE transaction can see are the updates done by the transaction itself). All reads by a SERIALIZABLE transaction are therefore repeatable.

The SET TRANSACTION ISOLATION LEVEL statement is used to change the isolation level of a transaction. Note that, you cannot change this behaviour after a transaction is started. Therefore, it is only valid as the first statement of a transaction.

The syntax of this statement is:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
or
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Note: Oracle keywords cannot be localised. So, even if you use it within Australian context, you must use the keyword SERIALIZABLE, not SERIALISABLE.

The behaviour of the READ COMMITTED isolation has been reflected in the activity 02. It is the default behaviour.

Activity 4: Exploring different isolation levels

The following activity uses the SERIALIZABLE isolation level. Make sure you start two fresh transactions (by committing any previous actions in both windows.)

Transaction 1 (T1)	Transaction 2 (T2)
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	
SELECT * FROM movie; You see the old value.	
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
	UPDATE movie SET mvtitle = 'X-Men 2' WHERE mvnumb = 24;

SELECT * FROM movie; You see the old value.	
	Now, COMMIT the second transaction. COMMIT;
SELECT * FROM movie; You still see the old value	

Here, you will notice that all the SELECT statements in transaction 01 see the same snapshot of data, even if the transaction 02 has updated (and committed) them in half-way through.

If you commit the current transaction in first window and start a new transaction, you will see the updated mvtitle for movie 24 there.

Refer to the table provided in the lecture notes to compare the behaviour of different isolation levels. Oracle implements only READ COMMITTED and SERIALIZABLE isolation levels.

3.4 Transaction Management with PHP

In web database applications, it is extremely important to manage transactions. In most of the applications, one web page consists of a series of actions that are to be executed as one unit. If one action failed, we cannot proceed with the rest of the actions, as well as we have to rollback already-completed actions.

In this activity, we explore how to write a PHP program embedded with correct transaction management logic. We also use a set of oci functions that help us build "transaction-aware" PHP programs.

oci Functions:

oci_execute() – A transaction begins when the first oci_execute() function is invoked with **OCI_NO_AUTO_COMMIT** flag. **OCI_NO_AUTO_COMMIT** is important, the default execution mode is **OCI_COMMIT_ON_SUCCESS** which will automatically commit all outstanding changes for this connection when the statement has succeeded. However, when **OCI_NO_AUTO_COMMIT** flag is set, the transaction will stay in active mode until a COMMIT or ROLLBACK is executed.

e.g.

```
$query1 =
     'INSERT INTO movie (mvnumb, mvtitle)
             VALUES (25, \'Lion King\')';
$stid = oci_parse($conn, $query1);

$r = oci_execute ($stid, OCI_NO_AUTO_COMMIT);
```

oci_rollback()

This function explicitly rolls back the outstanding database transaction. It reverts all uncommitted changes for the Oracle connection and ends the transaction.

```
oci_rollback ($connection);
```

oci_commit()
Commits the outstanding transaction for the Oracle **connection**. A commit ends the current transaction and makes permanent all changes. It releases all locks held.

```
oci_commit ($connection);
```

A typical PHP program with transaction management will have the following structure:

```
$conn = oci_connect($username, $password, $connection);
$stid = oci_parse($conn, $query1);
$r = oci_execute($stid, OCI_NO_AUTO_COMMIT);
$stid = oci_parse($conn, $query2);
$r = oci_execute($stid, OCI_NO_AUTO_COMMIT);
...
if (!$r)
    oci_rollback($conn);
else
    oci_commit($conn);
```

Activity 5: Transaction Management with PHP

Download `test_transaction.php` file from the Module 12 of the Canvas.

This program will

1. insert one row for a new movie
2. update the new row with a director number who supposedly directed the new movie.

First query in the transaction:

```
$query1 =
     'INSERT INTO movie (mvnumb, mvtitle) VALUES (25, \'Lion King\')';
```

Second query in the transaction:

```
$query2 = 'UPDATE movie SET dirnumb = 15 WHERE mvnumb = 25';
```

If the director number doesn't exist in the Director table, the second action will fail due to referential integrity constraint. As a result, the transaction fails and `oci_rollback()` function is called.
On the other hand, if it was a valid director number, both actions will be successful and transaction ends with an `oci_commit()` function call.

1. Before you run the web script, open Oracle SQL Developer, and run the following SQL query to view the Movie table.

```
SELECT mvnumb, mvtitle, dirnumb
FROM movie;
```

2. Run this program from your web server on `titan.csit.rmit.edu.au/~s1234567/test_transaction.php`

3. After running the web script, run the above SQL query again.
   What is the output? Do you see the newly inserted Lion King movie?

4. Change the `test_transaction.php` script to change the director number from 15 (invalid) to 1 (which is a valid director number). Run it again.

5. After running the web script, run the above SQL query again.
   What is the output? Do you see the newly inserted Lion King movie?

6. Run the `test_transaction.php` again.
   What is the output?