Lab 10: Transactions and Concurrency

Objective

This follow-along lab exercise aims to familiarize students with the fundamental concepts of transactions and concurrency.

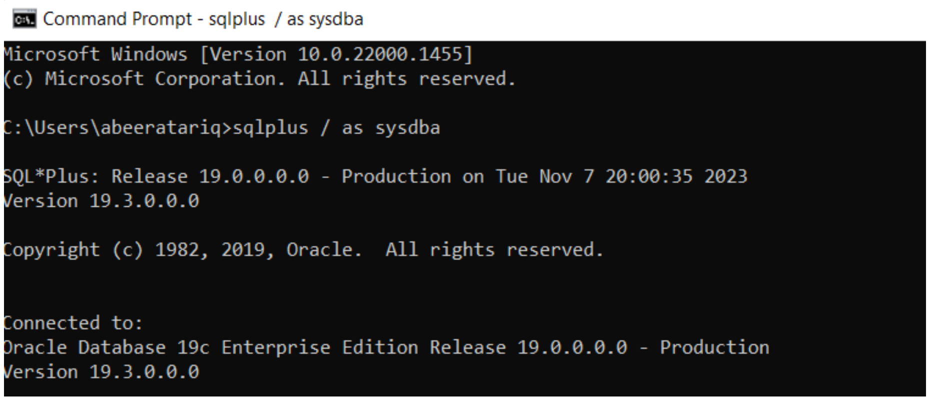
To complete this lab, you'll need to install Oracle Database, as the tasks involved in this exercise cannot be executed on LiveSQL. This is essential for simulating concurrent access through multiple SQLplus sessions.

Submission Requirements

During this lab exercise, document your progress with screenshots and text explanations where required. Compile your work in a Word document and upload on LMS.

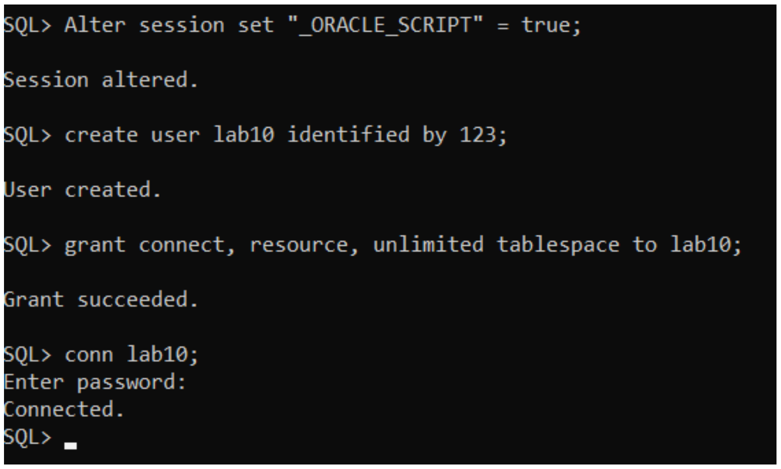
Simulating Concurrent Users using SQLplus

Begin by accessing the command line and logon to SQL plus as sysdba.



```
Microsoft Windows [Version 10.0.22000.1455]
(c) Microsoft Corporation. All rights reserved.

C:\Users\abeeratariq>sqlplus / as sysdba

SQL*Plus: Release 19.0.0.0.0 - Production on Tue Nov 7 20:00:35 2023
Version 19.3.0.0.0

Copyright (c) 1982, 2019, Oracle.  All rights reserved.


Connected to:
Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production
Version 19.3.0.0.0
```

Create a new user and connect to it. This window will be treated as session 01.



```
SQL> Alter session set "_ORACLE_SCRIPT" = true;

Session altered.

SQL> create user lab10 identified by 123;

User created.

SQL> grant connect, resource, unlimited tablespace to lab10;

Grant succeeded.

SQL> conn lab10;
Enter password:
Connected.
SQL>
```

Transactions

Follow through the lab exercises to strengthen your understanding of transaction management and concurrency control.

Note: if the session is not specified, you may execute in session 01.

1. Setting up the tables:

a) Create tables as below:

```
create table toys (
  toy_id   integer,
  toy_name varchar2(100),
  colour   varchar2(10)
);

create table bricks (
  brick_id          integer,
  colour            varchar2(10),
  shape             varchar2(10),
  unit_weight       integer DEFAULT 0,
  quantity          integer DEFAULT 0
      );

Desc toys;
Desc bricks;
```

b) Insert records

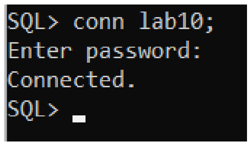
```
insert into toys values ( 1, 'Fluffy', 'pink' );
insert into toys values ( 2, 'Baby Turtle', 'green' );
insert into bricks (brick_id, colour, shape) values ( 1, 'green', 'cube' );
insert into bricks (brick_id, colour, shape) values ( 2, 'green', 'sphere' );
insert into bricks (brick_id, colour, shape) values ( 3, 'blue', 'cube' );
insert into bricks (brick_id, colour, shape) values ( 4, 'red', 'cube' );

select * from toys;
select * from bricks;
```

2. Committing Changes:

By default, Oracle operates in auto-commit mode, meaning each SQL statement is treated as a separate transaction and is automatically committed. To explicitly disable auto-commit, run the following command:

```
SET AUTOCOMMIT OFF;
```

Open another session of SQLplus on another instance of cmd prompt and connect with the same user, we will call this session 02. This way we will simulate concurrent access to the same schema. We now have 2 windows, session 01 and session 02.

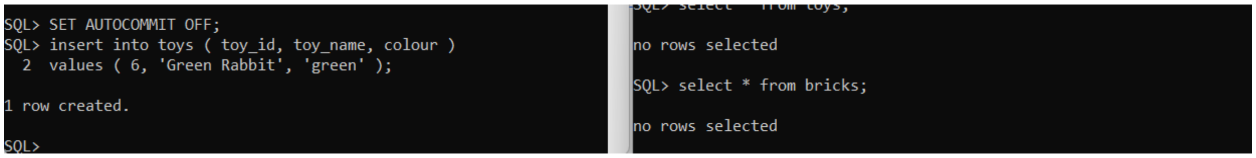


```
SQL> conn lab10;
Enter password:
Connected.
SQL>
```

Insert new record in session 01, but do not commit.

```
insert into toys ( toy_id, toy_name, colour )
values ( 6, 'Green Rabbit', 'green' );
```

In session 02, write a suitable query to retrieve all records inserted. Are the records visible?



```
SQL> SET AUTOCOMMIT OFF;
SQL> insert into toys ( toy_id, toy_name, colour )
  2  values ( 6, 'Green Rabbit', 'green' );

1 row created.

SQL>
```

```
SQL> select * from toys;

no rows selected

SQL> select * from bricks;

no rows selected
```

(Session 01 and Session 02 side by side)

In session 01, commit the transaction and repeat above. Can you now see the record?

```
Commit;
```

Between inserting a row and committing it, your code may throw an exception. So you may wish to undo the change. You can do this with **rollback**. Rollback reverts all the changes since your last commit. Say you added a row for Pink Rabbit. But realize you made a mistake, so want to remove it. The following does that:

```
insert into toys ( toy_id, toy_name, colour ) values ( 7, 'Pink Rabbit', 'pink' );

select * from toys where  toy_id = 7;

rollback;

select * from toys where  toy_id = 7;
```

When you issue a rollback, the database will undo all changes you made since your last commit. If you commit between insert and rollback, rollback does nothing. And you need to run a delete to remove the row. You may try this to validate.

Clicking one button in your application may fire many DML statements. You should defer committing until after processing them all. This allows you to rollback the changes if an error happens part way through. This ensures all other users of the application either see all the changes or none of them.

Test the above code in session 01. Make sure to execute the 4 lines of code one after the other to see how the changes happen and record your progress.

3. Savepoints

Your code will often add rows to many tables in one action. If there is an error part-way through, you may want to undo some - but not all - the changes since the last commit.

To help with this you can create savepoints. These are checkpoints. You can undo all changes made after it. And preserve those made beforehand.

To do this, first create the checkpoint with the savepoint command. Give it a name to refer to later for example:

```
savepoint save_this;
```

To undo the changes after the savepoint, add the clause "to savepoint" after rollback. Give the name of the savepoint you want to revert to. Note that savepoints do NOT commit! If you issue an unqualified rollback, you'll still reverse all changes since the last commit. Even those made before the savepoint.

For example, the code below:

- Adds a row for toy_id 8
- Creates the savepoint after_eight
- Then inserts toy_id 9

The rollback to savepoint only removes the row for toy_id 9. The final rollback at the end also removes toy_id 8. Execute to see how this happens and record your output and understanding.

```
insert into toys ( toy_id, toy_name, colour )
       values ( 8, 'Pink Rabbit', 'pink' );

savepoint after_eight;

insert into toys ( toy_id, toy_name, colour )
       values ( 9, 'Purple Ninja', 'purple' );

select * from toys
       where  toy_id in ( 8, 9 );

rollback to savepoint after_eight;

select * from toys
       where  toy_id in ( 8, 9 );

rollback;

select * from toys
       where  toy_id in ( 8, 9 );
```

4. Updating table

For example, say you paint ten of the green cubes blue. So you need to increase the quantity of blue cubes by 10. And decrease the green bricks by the same amount. If one update completes but not the other the total quantity will be out by 10. This can happen if you place a commit between the updates, as in this code:

```
update bricks
set    quantity = quantity - 10
where  colour = 'green'
and    shape = 'cube';

commit;

update bricks
set    quantity = quantity + 10
where  colour = 'blue'
and    shape = 'cube';

commit;


select * from bricks;
```

If the update of the blue rows fails for some reason, you've removed ten green cubes, but not added ten blue. So these bricks are "missing". Resolving this in code is hard. To fix the error, it's likely you'll need to run a one-off update. To avoid this problem, move the commit to the end:

```
update bricks
set    quantity = quantity - 10
where  colour = 'green'
and    shape = 'cube';

update bricks
set    quantity = quantity + 10
where  colour = 'blue'
and    shape = 'cube';

commit;

select * from bricks;
```

Now both updates are in the same transaction. If either fails, you can rollback the whole transaction and try again.

Placing commits at the end of your transaction makes it hard to reuse code. If you have two transactions that both include a commit, you can't combine them into one, larger transaction.

5. Simulating a Deadlock

When a transaction runs many updates, you need to take care. If two or more people try to change the same rows at the same time, you can get into deadlock situations. This is where sessions form a circle, each waiting on the other.

In the example below, both transactions start by updating different rows. So the first session has locked the rows with the colour red. And the second locks rows with the colour blue.

The first session then tries to update rows storing blue. But transaction two has these locked! So it's blocked, waiting for transaction 2 to complete.

But transaction two, instead of committing or rolling back, tries to update rows storing red. But transaction one has these rows locked! So it's stuck. At this point both sessions are waiting for the other to release a lock.

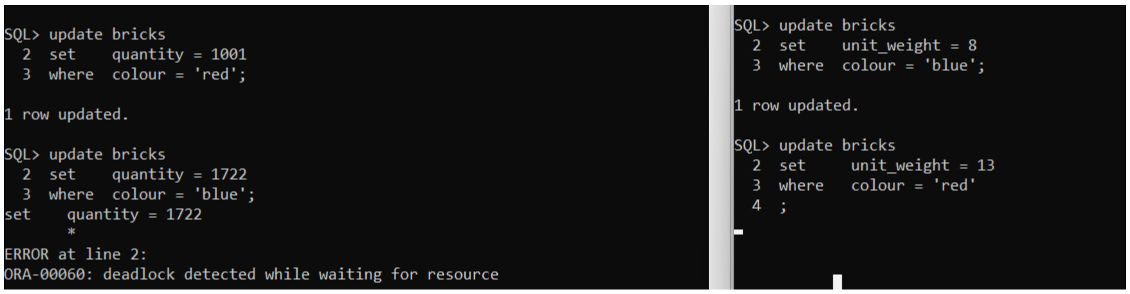
Transaction 1	Transaction 2

`update bricks`
`set    quantity = 1001`
`where  colour = 'red';`	

	`update bricks`
	`set    unit_weight = 8`
	`where  colour = 'blue';`

`update bricks`
`set    quantity = 1722`
`where  colour = 'blue';`
 	
`-- this session is now blocked,`
`-- waiting for transaction 2`	

	`update bricks`
	`set     unit_weight = 13`
	`where   colour = 'red';`
	`-- at this point both sessions`
	`-- are waiting on each other`
	`-- the database will stop one`
	`-- of them, raising an ORA-60`



```
SQL> update bricks
  2  set    quantity = 1001
  3  where  colour = 'red';

1 row updated.

SQL> update bricks
  2  set    quantity = 1722
  3  where  colour = 'blue';
set    quantity = 1722
       *
ERROR at line 2:
ORA-00060: deadlock detected while waiting for resource
```

```
SQL> update bricks
  2  set    unit_weight = 8
  3  where  colour = 'blue';

1 row updated.

SQL> update bricks
  2  set     unit_weight = 13
  3  where   colour = 'red'
  4  ;
```

When deadlock happens, Oracle Database will detect it. The database will then stop one of the statements, raising an ORA-00060. Note that an update locks the whole row. Even though the two updates set different columns, the second is still blocked. Leading to deadlock.
Try executing 'commit' command where the transaction gets aborted by Oracle. Record in your document what happens.

6. Select FOR UPDATE

You're at risk of deadlock if a transaction runs two or more update statements on the same table. And two people running the transaction may update the same rows, but in a different order. In these cases you should lock the rows before running any updates. You can do this with select for update. Like update itself, this locks rows matching the where clause. To use this, place the "for update" clause after your query.

So the following locks all the rows with the colour red:

```
select * from bricks
where  colour = 'red'
for    update;
```

No one else can update, delete or select for update these rows until you commit or rollback. For example, to avoid the deadlock described in the previous module, run select for update at the start. This selects rows for the colours red and blue and then locks it for updates:

```
select * from bricks
where  colour in ( 'red', 'blue' )
for    update;

update bricks
set    quantity = 1001
where  colour = 'red';

update bricks
set    quantity = 1722
where  colour = 'blue';
```

Close the previous SQLplus instances and open 2 new instances for lab10 user to execute the following and record the output in your document.

Transaction 1	Transaction 2

`select * from bricks`
`where colour in ( 'red', 'blue' )`
`for update;`	

	`select * from bricks`
	`where colour in ( 'red', 'blue' )`
	`for update;`
	`-- this session is now blocked,`
	`-- waiting for transaction 1`

`update bricks`
`set quantity = 1001`
`where colour = 'red';`
 	
`update bricks`
`set quantity = 1722`
`where colour = 'blue';`
`commit;`
`-- transaction 2 can now continue`	

	`update bricks`
	`set unit_weight = 8`
	`where colour = 'blue';`
	
	`update bricks`
	`set unit_weight = 13`
	`where colour = 'red';`
	
	`commit;`

7. Lost Update

A lost update happens when two people change the same row and one overwrites another's changes. This is a common problem if you set values for all columns that are not in the where clause regardless of whether the user changed the value.

For example, say currently there are 60 red cylinders in stock, with a unit weight of 13. Run the following to set these values:

```
insert into bricks (brick_id, colour, shape) values (5, 'red', 'cylinder');

update bricks
set    quantity = 60,
       unit_weight = 13
where  colour = 'red'
and    shape  = 'cylinder';
commit;
```

You have two people managing brick stock. One handles quantities, the other weights. They both load the current details for red cylinders to the edit form using this query:

```
select *
from   bricks
where  colour = 'red'
and    shape  = 'cylinder';
```

At this point they both see a quantity of 60 and weight of 13.
Then one person sets the weight to 8. And the other sets the quantity to 1,001. But they leave the value for the other attribute the same. They then both save their changes. The application updates both columns each time. For each person, the update runs with the original value for the unchanged column. Execute following queries in the 2 sessions to see the output.

The update to change the weight runs with these values: (Session 01)

```
update bricks
set    quantity = 60,   -- original quantity
       unit_weight = 8 -- new weight
where  colour = 'red'
and    shape  = 'cylinder';
commit;

select *
from   bricks
where  colour = 'red'
and    shape  = 'cylinder';
```

And the quantity change runs with these values: (Session 02)

```
update bricks
set    quantity = 1001, -- new quantity
       unit_weight = 13 -- original weight
where  colour = 'red'
and    shape  = 'cylinder';
commit;

select *
from   bricks
where  colour = 'red'
and    shape  = 'cylinder';
```

The second update sets the unit_weight from the new value of 8 back to the original, 13. So the update setting the weight to 8 is lost.

This problem can be hard to spot. Immediately after the person setting the weight saves their changes, everything looks fine. So they go on to do something else. Same for the stock level manager. So it may take some time for staff to realize there's a data error.

8. Isolation Levels

To demonstrate the Read Committed isolation level in Oracle using an example, we will once again use 2 sessions of lab10 user to simulate concurrent transactions.

Consider the following case where we set the isolation level at the transaction level to **read committed**. At this level, it will ensure a transaction only reads the committed data. If a transaction has not been completed, which means either committed or rolled back, the current transaction will wait.

Suppose 60 units of red cylinder-shaped bricks have been made while 100 have been sold. To ensure the net reduction of 40 units happen, we require the 2 changes to happen when one change is committed.

Session 01: Add 60 units. At this stage we do not commit. Make sure the auto commit is set to off.

```
select *
from   bricks
where  colour = 'red'
and    shape  = 'cylinder';

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

update bricks
set    quantity = quantity + 60
where  colour = 'red'
and    shape  = 'cylinder';

select *
from   bricks
where  colour = 'red'
and    shape  = 'cylinder';
```

Session 02: Sell 100 bricks.

```
select *
from   bricks
where  colour = 'red'
and    shape  = 'cylinder';


SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

update bricks
set    quantity = quantity -100
where  colour = 'red'
and    shape  = 'cylinder';
```

Since the isolation level is set to read committed, the update action will wait for Session 01 to complete the transaction or rollback.

Execute commit in Session 01. Notice how the waiting ends and the action is completed. Verify the update in both sessions and record in your document. Is there still a problem? Do we require a commit in Session 02, why or why not? What is the final value for quantity?

```
select *
from   bricks
where  colour = 'red'
and    shape  = 'cylinder';
```