

# Transactions and Concurrency Control (Oracle)

## Practical Assignment

This practical assignment will use three separate sessions connected to the same database, using the same Oracle user. This makes it possible to observe the interactions between several client processes concurrently accessing the same data.

### Starting the Oracle DBMS and connecting to it with SQL Developer (reminder)

Recall: I can launch the Oracle server (its Docker image) by running the following command we have seen before with PA1:

```
docker run -d --name oracle-ee -v
/Users/Bogdan/data/OracleDBData:/opt/oracle/oradata -p 1521:1521 -p
5500:5500 -e ORACLE_PWD='BoGDdanC1!e!' container-
registry.oracle.com/database/enterprise:21.3.0.0
```

Recall: in your command, you must replace

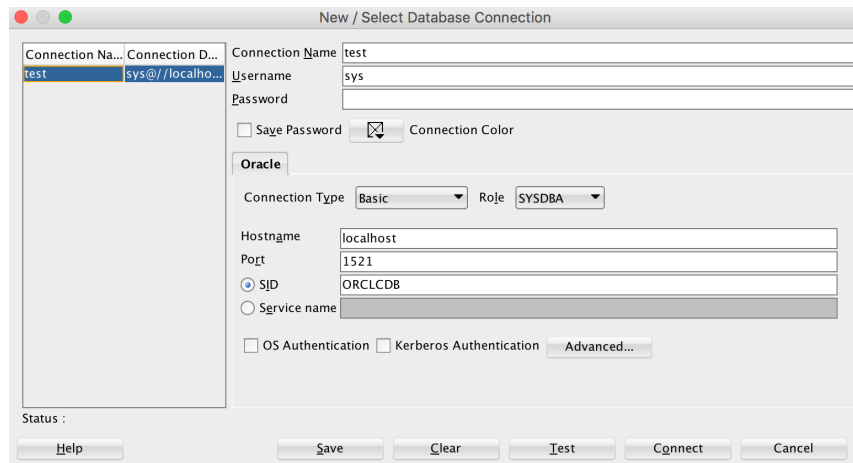
- 1) /Users/Bogdan/data/OracleDBData with a directory that exists on your machine. This is the directory where the Oracle database will be saved whenever you turn off the Oracle server (its Docker image); in this way, after each restart you can reuse the database.
- 2) 'BoGDdanC1!e!' by your own password (feel free to use mine too...), recall that you need a not-too-simple password, at least 8 characters long, with at least one upper-case, at least one digit, and at least one special character.

The -v option does the directory “sharing” between your machine the Docker image, and it works in both directions; your directory will be mapped to the ORCL directory on the Docker image.

The -p option indicates that the container’s port 1521 (the one usually used by the Oracle server) is mapped to the same port 1521 of the Docker (the one we will use); same for 5500.

Connecting to the Oracle server is done via an Oracle client, such as SQL Developer, as instructed in the Docker setup document.

You can launch SQL Developer and connect to the Oracle server by clicking on the + icon, then filling the connection form as follows (using your chosen password):



Recall you can restart the Oracle database system, if and only if required, using the maintenance scripts: `/home/oracle/shutDown.sh` and `/home/oracle/startUp.sh`.


## Creating a new user, opening the three sessions

In the “test” connexion (as SYSDBA, so administrator) run the following statements:

```
CREATE USER c##concurrentUser IDENTIFIED BY im_portant1;
GRANT CONNECT TO c##concurrentUser;
GRANT CREATE SESSION TO c##concurrentUser;
GRANT UNLIMITED TABLESPACE TO c##concurrentUser;
GRANT CREATE TABLE to c##concurrentUser;
```

You will then open **three different sessions/connections**, two called *booking1* and *booking2*, one called *control*, for all three using this newly created user, the password “im\_portant1”, the role “default”, etc (see screenshots below).

Créer / Sélectionner une connexion de base de données

Name   Color

Type de base de données

**Informations utilisateur** Utilisateur proxy

Type d'authentification

Nom utilisateur  Rôle

Mot de passe  ☒ Enregistrer le mot de passe

Type de connexion

**Détails** Avancé


Nom d'hôte

Port

☒ SID

☐ Nom de service

Créer / Sélectionner une connexion de base de données

Name   Color

Type de base de données

**Informations utilisateur** Utilisateur proxy

Type d'authentification

Nom utilisateur  Rôle

Mot de passe  ☒ Enregistrer le mot de passe

Type de connexion

**Détails** Avancé

Nom d'hôte

Port

☒ SID

☐ Nom de service

Créer / Sélectionner une connexion de base de données

Name   Color

Type de base de données Oracle ▾

**Informations utilisateur** Utilisateur proxy

Type d'authentification Par défaut ▾

Nom utilisateur  Rôle par défaut ▾

Mot de passe  ☒ Enregistrer le mot de passe

Type de connexion De base ▾

**Détails** Avancé

Nom d'hôte

Port

☒ SiD

☐ Nom de service

## Creating the database

The database we will use represents clients buying seats to a show and has one consistency constraint, as follows: the sum of the number of seats reserved by the clients (individually) must be equal to the overall number of reserved seats for the show (out of 50 available seats, initially). This constraint will be implicit (not specified explicitly in Oracle).

Start by executing the following in the control session. At a first run, the drop commands will raise errors, at a second run (you can do it) no errors should be raised.

---**DB creation script** – to be executed in the control session

```
drop table client;
drop table show;
CREATE TABLE Client (id_client INT PRIMARY KEY,
                     nb_reserved_seats INT NOT NULL,
                     balance INT NOT NULL);
CREATE TABLE Show (id_show INT PRIMARY KEY,
                    nb_offered_seats INT NOT NULL,
                    nb_available_seats INT NOT NULL,
                    price DECIMAL(10,2) NOT NULL);
commit;
```

## Collection of SQL commands (to be executed later)

The SQL commands in this section should just be copied to the control session (for later usage), but **should not be executed for now**. Whenever you are asked to run a piece of code in the sessions booking1 / booking2, you will copy / paste that code from control into booking1 / booking2 and execute it there (**but only that piece of code**); **after each lesson, you will delete everything from booking1 / booking2**. Whenever you are asked to run a piece of code in control you will execute it there (**but only that piece of code**).

Recall: To execute just one command, you have to place the cursor (mouse) on it and click on green triangle (execute). Alternatively, you can select one or several commands with the mouse and then click on the green triangle to execute all of them in the order they appear.

To execute **all** the commands in a session (as a script), you have to click on the white page with a little green triangle on it (script execution). But you will not need to do this normally.

**--database reset/initialization script**, to be executed in the control session when requested  
--leads to initial database state: 2 clients w/o reservations, 1 show, 50 available seats initially

```
DELETE FROM Client;
DELETE FROM Show;
INSERT INTO Client VALUES (1, 0, 2000);
INSERT INTO Client VALUES (2, 0, 350);
INSERT INTO Show VALUES (1, 250, 50, 10);
commit;
```

You can notice that, at initialization, the database is in a consistent state. By that we mean that the two clients have no reserved seats yet (0), and that the total number of available seats to be sold is 50 (50 is considered here a hardcoded value, the overall capacity of seats on sale). Besides the 50 seats on sale, there are 250 seats that are always offered to guests (not sold).

Queries allowing to test the behavior of concurrent transactions (isolation, conflicting operations, commit and rollback):

**-- selections (reading) queries**

```
SELECT * FROM Client WHERE id_client=1;
SELECT * FROM Client WHERE id_client=2;
SELECT * FROM Show;
```

**-- booking of 5 seats for client 1**

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 5 WHERE id_client=1;
```

```
UPDATE Show SET nb_available_seats = nb_available_seats - 5;
```

----OR

```
UPDATE Show SET nb_available_seats = 50 - 5;
```

-- note the "hardcoded" 50 value for initial number of available seats

-- **booking of 2 seats for client 2**

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 2 WHERE id_client=2;
```

```
UPDATE Show SET nb_available_seats = nb_available_seats - 2;
```

----OR

```
UPDATE Show SET nb_available_seats = 50 - 2;
```

-- note the "hardcoded" 50 value for initial number of available seats

-- **control queries / control script**

```
SELECT * FROM Client WHERE id_client=1;
```

```
SELECT * FROM Client WHERE id_client=2;
```

```
SELECT * FROM Show;
```

Oracle provides two isolation levels among the ones discussed in class, namely Read Committed (default) and what it calls "Serializable" (whose name is misleading, as it is in fact a version of Snapshot Isolation). **Important note: the "commit" command resets the isolation level in Oracle back to default.**

In any of the three sessions, one can set the isolation level (but only if not in an ongoing transaction, so only after a commit or rollback) as follows:

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

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

## **First database initialization**

Execute the database insertions in the control session:

-- **Database initialization script**

```
INSERT INTO Client VALUES (1, 0, 2000);
```

```
INSERT INTO Client VALUES (2, 0, 350);
```

```
INSERT INTO Show VALUES (1, 250, 50, 10);
```

```
commit;
```

## **Concurrency control lessons**

**In what follows, you must execute SQL instructions one by one, not several at a time. In each lesson, execute only what is listed there, in the listed order, and nothing else.**

**Whenever asked what you observe and what you can conclude, apply the following 5 second rule:**

*Observe what happens in the next 5 seconds, then explain, then pass to the next step/command/group of commands in the scenario. Do not wait much longer than 5 seconds (or indefinitely) and **do not stop anything by brute force**.*

### **Lesson 1. Transaction isolation**

(Recall to execute instructions one at a time)

Make a reservation transaction in booking1, for client 1 booking 5 seats (two updates), by the two instructions below:

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 5 WHERE id_client=1;  
UPDATE Show SET nb_available_seats = nb_available_seats - 5;
```

Then, perform the two queries below in the control session and in booking1:

```
SELECT * FROM Client WHERE id_client=1;  
SELECT * FROM Show;
```

What do you see ? What can you conclude ?

### **Lesson 2. Commit and rollback**

(Recall to execute instructions one at a time)

Rollback the current transaction (in booking1), by executing

```
rollback;
```

Perform the two queries below in the control session and in booking1:

```
SELECT * FROM Client WHERE id_client=1;  
SELECT * FROM Show;
```

What do you see ? What can you conclude ?

Now, repeat the transaction in booking1, and commit, by executing

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 5 WHERE id_client=1;  
UPDATE Show SET nb_available_seats = nb_available_seats - 5;  
commit;
```

Perform the two queries below in the control session:

```
SELECT * FROM Client WHERE id_client=1;  
SELECT * FROM Show;
```

What do you see ? What can you conclude ?

### Lesson 3. Atomicity

(Recall to execute instructions one at a time)

Reset the database to the initial state in the control session.

Repeat the transaction in booking1, and then crash the system, by executing

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 5 WHERE id_client=1;  
UPDATE Show SET nb_available_seats = nb_available_seats - 5;
```

and then restarting the Oracle system with the shutdown/restart scripts.

After restart is complete (5-10 seconds), perform the two queries below in the control session:

```
SELECT * FROM Client WHERE id_client=1;  
SELECT * FROM Show;
```

What do you see ? What can you conclude ?

Issue a commit command in booking1, and then re-execute the two queries below in the control session:

```
SELECT * FROM Client WHERE id_client=1;  
SELECT * FROM Show;
```

What do you see ? What can you conclude ?

### Lesson 4. Transaction on hold

(Recall to execute instructions one at a time)

Reset the database to the initial state (the reset script) in the control session, and then perform two reservations in parallel, according to the following schedule:

- Perform the following two queries in **booking1**  

```
SELECT * FROM Client WHERE id_client=1;  
SELECT * FROM Show;
```
- Perform the queries in **booking2**  

```
SELECT * FROM Client WHERE id_client=1;  
SELECT * FROM Client WHERE id_client=2;  
SELECT * FROM Show;
```



- Make a reservation in **booking1**, for client 1 booking 5 seats (two updates, execute them one by one) but do not validate (do not commit)

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 5 WHERE id_client=1;
UPDATE Show SET nb_available_seats = nb_available_seats - 5;
```

- Make a reservation in **booking2**, for client 1 booking 5 seats and for client 2 booking 2 seats (three updates, execute them one by one) but do not validate (do not commit)

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 5 WHERE id_client=1;
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 2 WHERE id_client=2;
UPDATE Show SET nb_available_seats = nb_available_seats - 7;
```

What do you see ? What can you conclude ?

Now **commit in the booking1** session.

What do you see ? What can you conclude ?

At this point in the booking2 session what should we do ? Is “commit” a good idea ?

Now **rollback in the booking2** session and **redo the control queries** (control script) in the control session.

What do you see ? What can you conclude ?

## **Lesson 5. Incomplete isolation = possible inconsistency**

(Recall to execute instructions one at a time)

Reset the database to the initial state in the control session and then perform two reservations in parallel, according to the following schedule:

- Perform the two queries below in booking1  
 SELECT \* FROM Client WHERE id\_client=1;  
 SELECT \* FROM Show;  
 -- we'll remember the value 50 and use it directly in a query later
- Perform the two queries below in booking2  
 SELECT \* FROM Client WHERE id\_client=2;  
 SELECT \* FROM Show;  
 --we'll remember the value 50 and use it directly in a query later
- Make the two booking updates booking1 and validate (commit)

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 5 WHERE id_client=1;
UPDATE Show SET nb_available_seats = 50 - 5; --using value 50 directly
commit;
```

- Make the two booking updates in booking2 and validate (commit)

```
UPDATE Client SET nb_reserved_seats = nb_reserved_seats + 2 WHERE id_client=2;
UPDATE Show SET nb_available_seats = 50 - 2; --using value 50 directly
commit;
```

- Perform again the control queries in the control session
 

```
SELECT * FROM Client WHERE id_client=1;
SELECT * FROM Client WHERE id_client=2;
SELECT * FROM Show;
```

What do you see? What can you conclude?

## Lesson 6. Full isolation

(Recall to execute instructions one at a time)

If we want to ensure consistency, we must choose the maximum isolation level, that is to say serializable. Reset the database and set the isolation mode to serializable in all three sessions.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

(note: if this command gives an error message just run a commit before it)

Now repeat the previous concurrent execution.

What do you see? What can you conclude?

## Lesson 7. Deadlocks

(Recall to execute instructions one at a time)

Reset the database and set the isolation mode to serializable in all sessions.

Perform two reservations in parallel, according to the following schedule:

- Perform the two queries below in booking1
 

```
SELECT * FROM Show where id_show=1;
SELECT * FROM Client WHERE id_client=1;
--we'll remember the value 50 and use it directly in a query later
```
- Perform the two queries below in booking2
 

```
SELECT * FROM Client WHERE id_client=1;
SELECT * FROM Show where id_show=1;
--we'll remember the value 50 and use it directly in a query later
```
- Make the following booking update in booking2

UPDATE Show SET nb\_available\_seats = 50 - 5 where id\_show=1; --using 50 directly

What do you see at this point ?

- Make the following booking updates (one by one) in booking1

UPDATE Client SET nb\_reserved\_seats = nb\_reserved\_seats + 5 WHERE id\_client=1;

UPDATE Show SET nb\_available\_seats = 50 - 5 where id\_show=1; --using 50 directly

What do you see at this point ?

- Make the following booking update in booking2

UPDATE Client SET nb\_reserved\_seats = nb\_reserved\_seats + 5 WHERE id\_client=1;

What do you see at this point?

What would happen if the *wait-die* deadlock avoidance approach was in place (priority by age), right after the booking2 update that caused the deadlock was issued? How would you mimic that approach in Oracle / SQL Developer ?

What would happen if the *wound-wait* deadlock avoidance approach was in place (priority by age)? How would you mimic that approach in Oracle / SQL Developer ?

## **Lesson 8. Explicit locking**

(Recall to execute instructions one at a time)

Reset the database and set the isolation mode to serializable in all sessions.

Redo the schedule from Lesson 7 by adding the clause "**FOR UPDATE**" at the end of each SELECT-FROM-WHERE statement.

What do you see? What can you conclude? What happens when you then commit in booking1 ?

## **Lesson 9. Read-only transactions**

(Recall to execute instructions one at a time)

Reset the database and set the transaction mode to READ ONLY in the booking sessions.

SET TRANSACTION READ ONLY;

Redo the schedule from Lesson 7.

### **Lesson 10. Another look at Oracle's SERIALIZABLE isolation level**

(Recall to execute instructions one at a time)

Reset the database (in control) and set the isolation mode to serializable in all three sessions.

Clients 1 and 2 decide to go together to the show, but do not explicitly agree whom among them should handle the booking of the two seats.

Unsure if it is up to him/her to do the booking, Client 1 will run the following transaction in booking1 (do not execute anything yet):

```
-- check if client 2 booked already or not:
SELECT nb_reserved_seats FROM Client WHERE id_client=2;
-- if the obtained value is 0, hence client 2 did not yet book:
UPDATE Client SET nb_reserved_seats = 2 WHERE id_client=1 ;
commit;
```

Unsure if it is up to him/her to do the booking, Client 2 will run also the following transaction in booking2 (do not execute anything yet):

```
-- check if client 1 booked already or not:
SELECT nb_reserved_seats FROM Client WHERE id_client=1;
-- if the obtained value is 0, hence client 1 did not yet book:
UPDATE Client SET nb_reserved_seats = 2 WHERE id_client=2 ;
commit;
```

These operations are executed in booking1 (red operations) or booking2 (blue operations) in the following order (execute them one by one, in either booking1 or booking2, depending on color):

```
SELECT nb_reserved_seats FROM Client WHERE id_client=2;
SELECT nb_reserved_seats FROM Client WHERE id_client=1;
UPDATE Client SET nb_reserved_seats = 2 WHERE id_client=1 ;
UPDATE Client SET nb_reserved_seats = 2 WHERE id_client=2 ;
commit;
commit;
```

Perform now the following in the control session  
commit;

```
SELECT * FROM Client;
```

What do you see? What can you conclude about Oracle's serializable level?