TP2
Accès concurrents

Si vous ressentez le besoin de lire la référence **ORACLE** :
http://www.oracle.com/pls/db111/homepage

Pour faciliter le TP
*Pour les étudiants utilisant **ORACLE***
La notion de séquence et son utilisation :

```
CREATE SEQUENCE CptHOTEL -- Create a sequence titled CptHOTEL
                         --   (using in the table HOTEL)
       INCREMENT BY 1    -- by default, the incremental value is 1
       START WITH 1      -- by default the default value is 1
       NOCYCLE           -- ensure that nocycle are present
/
```

En PL/SQL directement dans l'interpréteur **sqlplus** ou **sqldeveloper** d'ORACLE.

```
BEGIN
FOR x in 1..&NbLoop LOOP
INSERT INTO Hotel VALUES ( cptHOTEL.nextval, 'Ibis',
CptHOTEL.currval || ' Rue I', '0404040404' );
COMMIT;                  -- commit here or after the loop or
                         --   after the block PL/SQL
END LOOP;
END;
/
```

Vous devez écrire avant les manipulations ce que vous pensez obtenir, ce que vous obtenez réellement et expliquer les différences. Certains points ne nécessitent aucune réponse (question 1 par exemple).

Rendu du tp « Concurrence et Transactions »

Par mail à noel.novelli@lis-lab.fr et « [M2ILD] GD TP 2 – Concurrence et Transactions » en sujet, un seul fichier **tp2_nom_CT.sql** (si vous êtes en binôme, mettre les deux noms) contenant tout votre script avec tous les commentaires qui me permettront de bien comprendre vos choix et votre démarche. Les mauvaises routes que vous aurez empruntées peuvent vous permettre d'expliquer vos choix.

Pour s'échauffer

Pour illustrer les accès concurrents sous ORACLE, nous allons travailler sur le schéma suivant qui représente une gestion simplifiée des réservations de chambre d'hôtel (clé primaire en **GRAS**, clé étrangère suivie du caractère #) :

```
Hotel( NumH, NomH, AdresseH, TelephoneH )
Chambre( NumC, NumH#, NbPlaceC, EtageC, PrixC )
PClient( NumP, NomP, AdresseP, TelephoneP )
Reservation( NumR, NumP#, NumH#, DateA, DateP )
Occupation( NumP#, NumH#, NumC#, NumR# )
```

ATTENTION : Les questions ou instructions suivantes ont pour but de vous faire manipuler la concurrence d'accès aux données. Vous devez bien comprendre ce qui s'est passé pour chaque question.
Les questions sont à faire dans l'ordre. Chaque réponse modifiera l'état de votre base de données. Si

vous sautez une question, les réponses suivantes ont une très grande probabilité d'être fausses. Afin d'éviter ce problème, il faut que vous identifiez parmi les questions déjà faites, la question vous permettra de repartir avec une base de données correcte.

1. Ouvrez deux sessions sous avec le même `<login>`. Nous avons donc la session 1 et la session 2.
2. Détruire toutes les tables du schéma ci-dessus pouvant exister puis créer uniquement la table `Hotel` (session 1).
3. Faites des insertions dans la table `Hotel` dans la session 1 et voyez si les modifications sont connues des deux sessions.
4. Faites un **`COMMIT`** des modifications dans la session 1 et voyez si les modifications sont connues de l'autre session.
5. Faites des modifications (du type LMD, **`INSERT`** ou **`UPDATE`** par exemple) dans la session 1 et voyez si les modifications sont connues de l'autre session.
6. Créer la table `Chambre` dans la session 1.
7. Toutes les modifications sont-elles connues de toutes les sessions ?
8. Faites des insertions dans la table `Chambre` puis faire **`COMMIT`**.
9. Modifiez le prix d'une même chambre dans les deux sessions (avec des valeurs différentes). Que se passe-t-il ?
10. Faites un **`COMMIT`** sur la session qui a fait les modifications en premier. Que se passe-t-il ? Faites un **`SELECT`** dans les 2 sessions pour voir les modifications.
11. Faites un **`COMMIT`** dans la deuxième session. Faites un **`SELECT`** dans les 2 sessions pour voir les modifications.
12. Utilisez un **`SELECT ... FOR UPDATE`** dans la session 1 et modifier les lignes restituées par le **`SELECT`** dans la session 2.
13. Que se passe-t-il ? Faites un **`COMMIT`** dans la session **`SELECT ... FOR UPADTE`**.
14. Mettre les deux sessions dans le même état.
15. Utilisez un **`SELECT ... FOR UPDATE`** dans la session 1 et insérer des lignes dans la session 2.
16. Que se passe-t-il ? Faites un **`COMMIT`** dans la session **`SELECT ... FOR UPADTE`**.
17. Mettre les deux sessions dans le même état.
18. Faites des modifications (du type LMD) dans la session 1 et puis réaliser un point de sauvegarde à l'aide de **`SAVEPOINT <SAVEPOINT NAME>`** puis faites encore des modifications de type LMD.
19. Que voient les sessions ?
20. Faites des modifications (du type LMD en évitant tout inter-blocage de transaction avec les modifications de la question 18) dans la session 2 puis **`COMMIT`**. Que voient les sessions ?
21. Faites un **`ROLLBACK TO <SAVEPOINT NAME>`** dans la session 1. Que voient les deux sessions ?
22. Faites un **`COMMIT`** dans la session 1. Que voient les deux sessions ?
23. Utiliser ce principe pour insérer 3 valeurs mais seules deux seront dans la base au final (la première et la troisième) !
24. Créer toutes les tables manquantes puis faites des insertions dans toutes les tables et **`COMMIT`** pour finir.

Modifications d'accès aux données pour une transaction

Ecrire les requêtes qui illustrent les comportements suivants :

1. Autorisez uniquement les lectures pour la transaction en cours puis vérifiez que cela fonctionne correctement.
2. Revenez au mode par défaut. Deux façons sont demandées !

Mode de fonctionnement d'ORACLE ou MySQL

Pour la suite et à l'aide de requête `SQL`, illustrer les phrases suivantes. Pour cela utiliser au moins deux sessions.

Utilisez uniquement la table Chambre

1. Les lectures ne bloquent ni les autres lectures ni les écritures.
2. Les lectures ne sont bloquées par rien, même pas par un blocage d'une table en mode exclusif.
3. Il n'y pas de lecture impropre.
4. Il n'y a pas de perte de mise à jour.
5. Il peut y avoir des lectures non reproductibles.
6. Il peut y avoir des lignes fantômes.

Lectures non reproductibles

Que pouvez-vous faire pour empêcher les lectures non reproductibles,

1. Dans le cas où la transaction ne modifie aucune donnée.
2. Dans le cas où elle modifie des données.

Lignes fantômes

Mêmes questions que l'exercice précédent, mais pour les lignes fantômes.

Interblocages

1. Ouvrez 2 sessions et provoquez un interblocage en commençant par faire des modifications, puis en bloquant des tables. Voyez comment `Oracle` réagit.
2. Provoquez un blocage en commençant par lancer un **`UPDATE`** que vous ne validez pas tout de suite et un blocage en mode **`SHARE`** dans l'autre session. Voyez comment `Oracle` réagit.

Read only

1. Ouvrez une nouvelle transaction en "**`READ ONLY`**".
2. Ouvrez en parallèle une deuxième transaction dans laquelle vous modifiez des prix.
3. Validez cette deuxième transaction. Voyez-vous les modifications dans la première transaction ?
4. Essayez de modifier des données dans la première transaction.
5. Que ce serait-il passé si la première transaction n'avait pas été en "**`READ ONLY`**" ? Vérifiez-le après avoir terminé la première transaction.

Mode serialisable

Essayez de faire cet exercice en devinant ce qui va se passer avant de lancer chaque commande.

1. Ouvrez 2 sessions de travail avec des transactions T1 et T2.
2. Passez T1 en mode sérialisé.
3. T1 affiche tous les noms et prix des chambres.
4. T2 modifie le prix d'une chambre.
5. T1 affiche à nouveau tous les prix. Quel prix voit-il pour cette chambre ? Pourquoi ?
6. T2 valide sa transaction.
7. T1 affiche à nouveau tous les prix. Quel prix voit-il pour cette chambre ? Pourquoi ?
8. T1 modifie le prix de cette chambre. Que se passe-t-il ? Pourquoi ?
9. T1 modifie le prix d'une autre chambre. Que se passe-t-il ?

Voyez-vous une différence avec le mode par défaut d'`Oracle` ? Explications ?

Comportement optimiste ou pessimiste

Dans cet exercice vous allez simuler un comportement optimiste d'un programme pour la modification des données dans la table `Chambres`. Vous allez augmenter le prix d'une des chambres.

1. Pour cela, faites d'abord afficher le prix d'une chambre (puisque vous êtes optimiste, vous ne bloquez pas ce prix lors de sa lecture). Calculez à la main une augmentation de 10 % (on simule ainsi le calcul complexe et long du nouveau prix) et enregistrez le nouveau prix. Qu'allez-vous faire pour vous prémunir contre une modification de ce prix par une autre transaction entre la lecture du prix et l'enregistrement du nouveau prix dans la base de données ? Il n'y pas de variable pour ranger les valeurs lues. Vous vérifiez donc de visu que les données n'ont pas été modifiées.
2. Même exercice, mais cette fois-ci, modifiez la table pour y ajouter une colonne qui est incrémentée à chaque modification de la ligne et utilisez cette colonne pour savoir si les données ont été modifiées entre la lecture des données et la fin de la transaction.
3. Faites le même traitement en étant cette fois-ci pessimiste.

ANNEXE

On peut paramétrer la gestion de la concurrence au niveau système et au niveau des transactions.

Le niveau d'isolation d'une transaction peut être indiqué par :

```
SET TRANSACTION ISOLATION LEVEL { SERIALIZABLE | READ COMMITTED } ;
SET TRANSACTION { READ ONLY | READ WRITE } ;
```

cette instruction doit être la première de la transaction.

read committed
: C'est le niveau par défaut. Toute modification validée avant le début de l'instruction en cours est visible par cette instruction. La sérialisabilité de la transaction n'est pas garantie.

read only
: Ce niveau est propre à Oracle. Une transaction **READ ONLY** ne voit que les modifications validées avant son début. Une telle transaction ne peut pas modifier la base et elle n'est jamais bloquée.

serializable
: Ce niveau est comme **READ ONLY**, mais en plus la transaction T peut modifier des nuplets. Si T tente de modifier (**DELETE** ou **UPDATE**) un nuplet dont la dernière version a été produite par une autre transaction validée après le démarrage de T, Oracle considère qu'il y a un problème potentiel de sérialisabilité et provoque donc l'annulation de l'instruction DML et la propagation d'une erreur Oracle (-08177).

L'exécution d'une instruction DDL ou DCL forme automatiquement une transaction : elle provoque une validation implicite en début et en fin d'exécution.

Pose implicite de verrous par Oracle

INSERT, **DELETE** ou **UPDATE** provoque un verrouillage intentionnel row exclusive de la table, puis verrouille les nuplets impliqués.

SELECT ... FOR UPDATE fait de même mais pose un verrou intentionnel **ROW SHARE** sur la table. Si une telle exécution tente de poser un verrou sur un nuplet déjà verrouillé par une autre transaction, elle est bloquée jusqu'à la terminaison de cette autre transaction. Lors du déblocage l'ensemble des nuplets sélectionnés est réévalué.

Pose explicite de verrous sous Oracle

Il est parfois nécessaire de verrouiller explicitement les données avec la commande **LOCK TABLE**.

```
LOCK TABLE <table> IN { EXCLUSIVE | SHARE } MODE [NOWAIT] ;
```

EXCLUSIVE
: Les autres transactions ne peuvent que lire la table (mises à jour et poses de verrou bloquées).

SHARE
: Bloque, pour les autres transactions, la pose du verrou intentionnel row exclusive, et la pose de verrou exclusive.

NOWAIT
: Plutôt que de bloquer l'exécution si la table est déjà verrouillée, Oracle abandonne l'instruction en signalant l'erreur -54 : la transaction peut alors faire autre chose avant de retenter le verrouillage.