le cnam

Travaux Pratiques : PL/SQL - Triggers - Concurrence

Bases de Données

FIP1 Informatique

CNAM Paris
Nicolas.Travers (at) cnam.fr

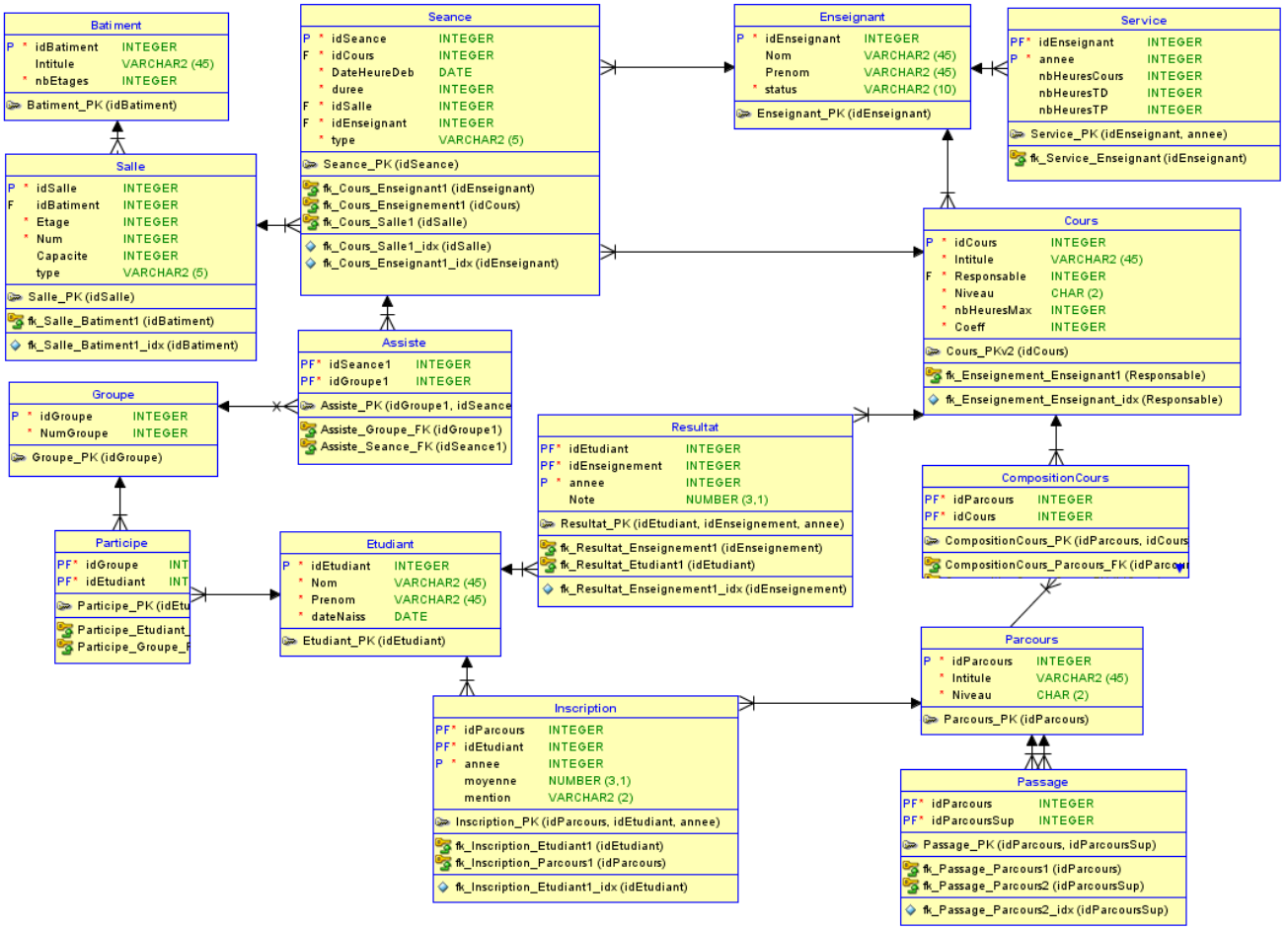
Table des matières

Chapitre 1
Procédures Stockées - Travaux Pratiques

Dans le cadre de ces travaux pratiques, nous utiliserons SQLDeveloper pour nous connecter à la base de données Oracle (Serveur 11gR2). Pour cela, veuillez vous référer au guide d'utilisation de SQLDeveloper disponible avec ce support d'exercices (seule la section 2.3 nous intéresse actuellement).

L'espace de travail est le même pour chacun d'entre vous. Pour faciliter la création des procédures et des fonctions, chacune doit être suffixée par votre login. Exemple : `affEtudiants_`**traversn**.

La base de données utilisée gère l'emploi du temps d'une école d'enseignement supérieur. Elle intègre les étudiants, les matières, les enseignants et les différents cursus. La base est stockée dans le TABLESPACE '**NFP107**', dont le schéma est représenté ci-dessous :



1.1 Procédures simples

(1.1.1) Créer une procédure 'affEtudiants_xxxx' qui affiche la liste des noms et prénoms d'étudiants inscrits d'une année donnée en paramètre ;

(1.1.2) Créer une procédure 'affNotes_xxxx' qui affiche les notes avec l'intitulé du cours correspondant pour un étudiant (nom et prénom) et année donnés ;

(1.1.3) Créer une procédure 'affMatieres_xxxx' qui affiche les intitulés et coefficients des matières qu'un étudiant (nom et prénom) suit durant une année donnée. Trier les matières par coefficients décroissants ;

1.2 Fonctions

(1.2.1) Créer une fonction 'retEtudiant_xxxx' qui retourne l'identifiant d'un étudiant pour un nom et prénom donné ;

(1.2.2) Créer une fonction 'moyennePonderee_xxxx' qui retourne la moyenne pondérée des résultats d'un étudiant (utiliser la fonction 'retEtudiant') et année donnés ;

(1.2.3) Donner la procédure qui affiche pour le parcours M1 'Informatique' de 2014, le nom des étudiants inscrits et leur moyenne pondérée en utilisant la fonction 'moyennePonderee' ;

1.3 Exceptions

(1.3.1) Modifier la fonction 'retEtudiant' pour qu'elle puisse lever une exception si aucun étudiant n'est trouvé ;

(1.3.2) Modifier la procédure 'affMatieres' en utilisant la fonction 'retEtudiant' et afficher un message d'erreur lorsqu'une exception est levé ;

1.4 Curseurs

(1.4.1) Créer une procédure 'affBulletins' qui pour un étudiant donné, affiche l'intitulé de chaque parcours (avec l'année d'inscription croissante) qu'il a suivi et pour chaque parcours les résultats obtenus pour chaque matière (avec intitulé) ;

(1.4.2) Créer une procédure 'affInterventions' qui affiche pour chaque niveau de parcours croissant les intitulés de cours et de parcours dans lequel intervient un enseignant donné (nom et prénom) ;

(1.4.3) Créer une procédure 'majBulletins' qui met à jour toutes les moyennes des inscriptions (utiliser la fonction 'moyennePondere') des étudiants d'un parcours donné (id) ;

1.5 Procédure complexe

(1.5.1) Créer une procédure 'affCours' qui affiche les informations d'un cours à une date donnée selon le schéma suivant :
HeureDeb, HeureFin, Salle/Batiment, Groupe1/Groupe2/Groupe3...

(1.5.2) Créer une procédure 'affPlanning' qui affiche le planning d'un parcours donné (niveau, intitulé, annee), organisé par cours ;

Chapitre 2
Déclencheurs - Travaux Pratiques

La création d'un Trigger est décrit dans le guide disponible sous claroline.

On souhaite ajouter à la base de données les contraintes qui ne sont pas exprimable sous formes de clés primaires, clés étrangères, types, domaines de valeurs ou CHECK. Créer les Trigger correspondant aux contraintes suivantes :

2.1 Le responsable d'un enseignement est un enseignant titulaire

Jeu de test :

```
DELETE FROM Cours WHERE idCours = 10;
DELETE FROM Cours WHERE idCours = 11;

SELECT * FROM Cours;

call insert_into_cours (10,'pratique de mysql_Boisson', 'Boisson','M1',20,2);
show errors;
call insert_into_cours (11,'pratique de mysql_Mourier', 'Mourier','M1',20,2);
show errors;

select * from cours;
```

la procédure 'insert_into_cours' est définie ci-dessous :

```
CREATE OR REPLACE PROCEDURE insert_into_cours
        (id Integer, intitule varchar, nomResp varchar, niveau varchar, duree integer, coeff integer)
IS
        idResp INTEGER;
BEGIN
        SELECT idEnseignant INTO idResp FROM Enseignant WHERE Nom = nomResp ;
        INSERT INTO Cours VALUES (id,intitule,idResp,niveau,duree,coeff) ;
        DBMS_OUTPUT.PUT_LINE('insertion du cours <' || intitule || '> réussie ');
EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('insertion du cours <' || intitule || '> refusée');
            DBMS_OUTPUT.PUT_LINE('ERROR : '||SQLERRM);
END;
/
```

2.2 On ne peut avoir de séances de 'Cours' dans une salle de 'TP', et on ne peut avoir de 'TP' que dans des salles de 'TP'

Jeu de test :

```
/* Requêtes de test */
delete from seance where TRUNC(dateHeureDeb) = TO_DATE('2015-10-15', 'YYYY-MM-DD');
select * from seance where TRUNC(dateHeureDeb) = TO_DATE('2015-10-15', 'YYYY-MM-DD');

-- cours CM en salle de TP
insert into Seance values (200,1,TO_DATE('2015-10-15 09:00:00', 'YYYY-MM-DD HH24:MI:SS'),90,5,1,'Cours');

-- cours TP en salle de amphi
insert into Seance values (201,1,TO_DATE('2015-10-15 09:00:00', 'YYYY-MM-DD HH24:MI:SS'),90,2,1,'TP');

-- cours CM en salle de cours
```

```
insert into Seance values (201,1,TO_DATE('2015-10-15 09:00:00', 'YYYY-MM-DD HH24:MI:SS'),90,2,1,'Cours');

select * from Seance where TRUNC(dateHeureDeb) = TO_DATE('2015-10-15', 'YYYY-MM-DD');
```

2.3 L'étage d'une salle ne peut être supérieur au nombre d'étages du batiment

Requête de test :

```
INSERT INTO Salle VALUES (100, 1, 10, 1, 50, 'TP');
```

2.4 Deux séances de cours ne peuvent avoir lieu dans la même salle en même temps

Jeu de test :

```
delete from seance where TRUNC(dateHeureDeb) = TO_DATE('2015-10-15', 'YYYY-MM-DD');
SELECT idSeance, idCours, TO_CHAR(dateHeureDeb, 'YYYY-MM-DD HH24:MI:SS') as Debut,
        TO_CHAR(dateHeureDeb + Duree/24/60, 'YYYY-MM-DD HH24:MI:SS') as Fin
FROM seance WHERE TRUNC(dateHeureDeb) = TO_DATE('2015-10-15', 'YYYY-MM-DD');

insert into Seance values (201,1,TO_DATE('2015-10-15 09:00:00', 'YYYY-MM-DD HH24:MI:SS'),90,2,1,'Cours');
insert into Seance values (201,1,TO_DATE('2015-10-15 10:00:00', 'YYYY-MM-DD HH24:MI:SS'),90,2,1,'Cours');
insert into Seance values (201,1,TO_DATE('2015-10-15 11:00:00', 'YYYY-MM-DD HH24:MI:SS'),90,2,1,'Cours');
insert into Seance values (201,1,TO_DATE('2015-10-15 10:00:00', 'YYYY-MM-DD HH24:MI:SS'),90,2,1,'Cours');
```

2.5 Pour une année civile, la somme des durées des séances ne peut dépasser le nombre d'heures maximum du cours associé

Jeu de test :

```
/* restauration de l'état initial*/
UPDATE Cours set nbHeuresMax = 30 WHERE idCours = 1;
DELETE FROM Seance WHERE TRUNC(dateHeureDeb) = TO_DATE('2015-10-15', 'YYYY-MM-DD');

/* Pour tester le Trigger */
UPDATE Cours set nbHeuresMax = 6 WHERE idCours = 1;

insert into Seance values (201,1,TO_DATE('2015-10-15 10:00:00', 'YYYY-MM-DD HH24:MI:SS'),180,2,1,'Cours');
insert into Seance values (202,1,TO_DATE('2015-10-16 10:00:00', 'YYYY-MM-DD HH24:MI:SS'),180,2,1,'Cours');
insert into Seance values (203,1,TO_DATE('2015-10-17 10:00:00', 'YYYY-MM-DD HH24:MI:SS'),180,2,1,'Cours');
```

2.6 Triggers supplémentaires

(2.6.1) Le service des enseignants vacataires ne peut dépasser 60 heures d'enseignement équivalent TD (1h de cours = 1.5h TD, 1.5h de TP = 1h TD) ;

(2.6.2) Le service des enseignants moniteurs ne peut dépasser 96h de TD ;

(2.6.3) La moyenne associée à une inscription est égale à la moyenne pondérée des résultats de l'élève obtenus à ce parcours cette année là ;

(2.6.4) Pour une séance de cours, le nombre d'étudiants présents (cf. groupe) ne peut dépasser la capacité de la salle ;

Chapitre 3

Devoir à rendre : Généalogie

Ce devoir est à effectuer par groupes de 2 et à retourner par email : nicolas . travers (at) cnam . fr

Un fichier composé de vos deux noms doit y figurer. Les numéros des questions doivent être associées aux réponses.

Si vous souhaitez terminer ce travail à la maison, vous pouvez installer la base `Oracle Express` et `SQLDeveloper`.

3.1 Base de données

On souhaite créer une table qui permette de stocker la **généalogie** de différentes personnes. Créer la table `Personne_xxxx` avec le schéma suivant :

```
Personne (id INTEGER,
          prenom VARCHAR2(30),
          nom VARCHAR2(30),
          naissance DATE,
          genre INTEGER,
          pere INTEGER,
          mere INTEGER)
```

Les attributs Pere et mere peuvent avoir la valeur `NULL`.

Nous poserons une clé étrangère sur pere et mere sur la clé primaire personne_name.id avec comme contrainte `ON DELETE SET NULL` ;

3.2 Fonctions et Exception

(3.2.1) Créer une fonction `id_personne_name` qui retourne l'identifiant d'une personne grâce à son nom et prénom en paramètre. Si aucun identifiant n'est trouvé (exception `NO_DATA_FOUND`), retourner 0 ;

(3.2.2) Créer une procédure `insert_personne_name` d'ajout d'une personne avec en paramètre son nom, prénom et date de naissance, ainsi que les noms et prénoms de ses parents.
L'identifiant ajouté est le maximum de tous les identifiants de la base.
Si pour un parent, le nom et le prénom sont `null`, cela veut dire que le parent est inconnu (les valeurs nulles sont possibles dans notre schéma).
Si l'identifiant d'un parent retourné par `id_personne_name` est 0, afficher le message d'erreur correspondant et ne pas insérer la personne [1] ;

(3.2.3) Créer un script 'insert_personne.sql' qui fait appel à la procédure avec 6 appels de type ACCEPT pour insérer une personne et ses parents. Si la chaîne vide est sélectionnée, la remplacer par une valeur null ;

(3.2.4) Créer votre propre arbre généalogique avec au minimum 3 générations ;

(3.2.5) Créer une procédure `affiche_généalogie_name` qui affiche la généalogie d'une personne (nom et prénom) donnée. Une indentation des résultats sera nécessaire, ainsi que l'affichage de la parenté de la personne (père/mère) ;

(3.2.6) Créer une fonction `meme_famille_name` qui renvoie TRUE si deux personnes (nom, prénom) sont de la même famille. Même famille veut dire affiliation directe (père/mère, grand-père/grand/mère. . . , enfant, petit-endant, arrière. . .). Si aucune affiliation n'est trouvée, retourner l'exception `NOT_SAME_FAMILY_ERROR` ;

(3.2.7) Créer une procédure qui affiche le nombre de générations de différence entre deux personnes, à condition qu'il y ait une affiliation (dans ce cas, afficher l'erreur correspondante). Vous pourrez modifier la fonction précédente pour y intégrer une variable globale (ne pas modifier le retour de la fonction) ;

1. On pourra créer une seconde procédure `insert_personne2` avec les identifiants des parents

3.3 Paquetage

Pour cet exercice, vous écrirez l'intégralité du code dans un fichier `Genealogie_name.sql`

(3.3.1) Supprimer toutes les procédures et fonctions que vous avez créé HORS paquetage ;

(3.3.2) Créer un paquetage `Genealogie` dont les spécifications contiennent toutes les procédures de la section précédente (qui à priori sont publiques), ainsi que la variable globale ;

(3.3.3) Créer le corps (BODY) de ce paquetage avec toutes les procédures et les fonctions de la section précédente ;

(3.3.4) Modifier votre script d'insertion pour y intégrer le paquetage ;

(3.3.5) Faites appel à votre script d'insertion sur le paquetage pour insérer une nouvelle personne ;

(3.3.6) Ajouter une procédure publique au paquetage qui permet de modifier les informations d'une personne ;

3.4 Trigger

(3.4.1) Créer un Trigger qui empêche l'insertion si un enfant est né avant ses parents (voire moins de 12 ans après) ;

(3.4.2) Créer un Trigger qui empêche l'insertion si le sexe d'un des parents ne respecte pas : `père = homme et mère = femme` ;

4.1 Graphe de sérialisabilité et équivalence des exécutions

Construisez les graphes de sérialisabilité pour les exécutions (histoires) suivantes. Indiquez les exécutions sérialisables et vérifiez s'il y a des exécutions équivalentes.

(4.1.1) $\mathcal{H}_1 : w_2[x]w_3[z]w_2[y]c_2r_1[x]w_1[z]c_1r_3[y]c_3$

(4.1.2) $\mathcal{H}_2 : r_1[x]w_2[y]r_3[y]w_3[z]c_3w_1[z]c_1w_2[x]c_2$

(4.1.3) $\mathcal{H}_3 : w_3[z]w_1[z]w_2[y]w_2[x]c_2r_3[y]c_3r_1[x]c_1$

4.2 Verrouillage à 2 phases

Un scheduler avec verrouillage à 2 phases reçoit la séquence d'opérations ci-dessous :

$$\mathcal{H} : r_1[x]r_2[y]w_3[x]w_1[y]w_1[x]w_2[y]c_2r_3[y]r_1[y]c_1w_3[y]c_3$$

Indiquez l'ordre d'exécution établi par le scheduler, en considérant qu'une opération bloquée en attente d'un verrou est exécutée en priorité dès que le verrou devient disponible. On suppose que les verrous d'une transaction sont relâchés au moment du Commit.

4.3 Estampillage

Etant donnée la séquence d'opérations suivante, donner l'exécution établie par un scheduler avec estampillage simple. Le scheduler avec estampillage n'utilise que le test des estampilles, sans retarder ensuite l'exécution des opérations. Considérez qu'une transaction rejetée est relancée tout de suite avec une nouvelle estampille et que ses opérations déjà exécutées sont traitées avec priorité.

$$\mathcal{H} : r_1[x]r_2[y]w_3[x]w_1[x]w_3[y]w_2[y]$$

Chapitre 5
Concurrence - Travaux Pratiques

5.1 L'environnement de travail

Ce TP de concurrence devra être réalisé sous la base de données **MySQL** avec le moteur de stockage InnoDB (le seul qui soit capable de supporter tous les niveaux d'isolation).

Pour ce faire, il vous faut vous connecter sous l'environnement *WampServer* avec la console (bouton gauche sur l'icone de WampServer, puis 'MySQL', enfin 'Console').

(5.1.1) Pour pouvoir constater des problèmes de concurrences, il vous faut deux sessions différentes. Il vous faudra donc ouvrir 2 terminaux sous mysql. Pour les différencier, nous utiliserons la commande : `'PROMPT SESSION1> '` (resp SESSION2).

(5.1.2) Pour chaque session, il faut enlever le mode de validation automatique : `SET AUTOCOMMIT = 0;`

(5.1.3) Lorsque vous devrez changer de mode d'isolation :

- `SET SESSION TRANSACTION ISOLATION LEVEL` **READ UNCOMMITTED**;
- `SET SESSION TRANSACTION ISOLATION LEVEL` **READ COMMITTED**;
- `SET SESSION TRANSACTION ISOLATION LEVEL` **REPEATABLE READ**;
- `SET SESSION TRANSACTION ISOLATION LEVEL` **SERIALIZABLE**;

5.2 Schéma de données et Données

Schéma Pour tester la concurrence entre deux transactions, nous allons utiliser les deux tables suivantes :

```
USE TEST; -- Sélectionne la base de données
DROP TABLE Spectacle; -- supprime les tables si elles existaient déjà
DROP TABLE Client;
DROP TABLE Reservation;

CREATE TABLE Spectacle (id_spectacle INT NOT NULL PRIMARY KEY,
                        places_offertes INT NOT NULL,
                        places_libres INT NOT NULL,
                        tarif DECIMAL(10,2) NOT NULL
            ) ENGINE=InnoDB;

CREATE TABLE Client (id_client INT NOT NULL PRIMARY KEY,
                     Solde FLOAT NOT NULL
            ) ENGINE=InnoDB;

CREATE TABLE Reservation (id_client INT NOT NULL,
                          id_spectacle INT NOT NULL,
                          places_reservees INT NOT NULL,
                          PRIMARY KEY (id_client, id_spectacle),
                          KEY spectacle (id_spectacle)
            ) ENGINE=InnoDB;
```

Pour que la base de données reste cohérente, il faut pour cela que pour une spectacle donné : $sum(places\_reservees) = (places\_offertes - places\_libres)$.

Données L'état d'origine de notre base de données est donné par les requêtes suivantes :

```
DELETE FROM Reservation;
DELETE FROM Client;
DELETE FROM Spectacle;
```

```
INSERT INTO Client VALUES (1, 50);
INSERT INTO Client VALUES (2, 50);
INSERT INTO Spectacle VALUES (1, 250, 250, 20);
COMMIT;
SET SESSION TRANSACTION ISOLATION LEVEL ... ;
```

Entre chaque test de concurrence sur notre schéma, la base de données doit avoir cet état pour être cohérent. Ainsi, vous pourrez mettre à zéro la base en exécutant ce script.

5.3 Transactions

Pour chaque transactions ci-dessous, une **séquence de requêtes n'est pas interchangeable**. Les séquences ne sont pas intégrées dans des procédures, pour permettre de décomposer les opérations dans le temps et favoriser la concurrence. Vous devrez donc vérifier vous-même les conditions (T_1 et T_2) avec les valeurs LOCALES '@' à la session, et le cas échéant, faire un ROLLBACK et arrêter la transaction.

(5.3.1) Donner pour chaque transaction les séquences d'opérations possibles. Noter, le cas échéant, les conditions d'application de cette transaction (exemple : @nb_places < 2) ;

(5.3.2) Présenter par la suite, chaque histoire sous forme de suite d'opérations grâce aux transactions que vous venez de produire [1].

T_1 Réservation de 2 places pour le client 1

$R_1(sp_1)$	SELECT places_libres, tarif INTO @nb_libres, @tarif FROM Spectacle WHERE id_spectacle = 1 ;
r_1	*Vérifier si "SELECT @nb_libres - 2 ;" < 0 alors ROLLBACK ;*
$W_1(sp_1)$	UPDATE Spectacle SET places_libres = @nb_libres - 2 WHERE id_spectacle = 1 ;
$W_1(re_1)$	INSERT INTO Reservation VALUES (1, 1, 2) ;
$R_1(so_1)$	SELECT Solde INTO @solde FROM Client WHERE id_client = 1 ;
r_1	*Vérifier si "SELECT @solde - 2 * @tarif ;" < 0 alors ROLLBACK ;*
$W_1(cl_1)$	UPDATE Client SET Solde = @solde - 2 * @tarif WHERE id_client = 1 ;
c_1	COMMIT ;

T_2 Réservation de 5 places pour le client 2

$R_2(sp_1)$	SELECT places_libres, tarif INTO @nb_libres, @tarif FROM Spectacle WHERE id_spectacle = 1 ;
r_2	*Vérifier si "SELECT @nb_libres - 5 ;" < 0 alors ROLLBACK ;*
$W_2(sp_1)$	UPDATE Spectacle SET places_libres = @nb_libres - 5 WHERE id_spectacle = 1 ;
$W_2(re_2)$	INSERT INTO Reservation VALUES (2, 1, 5) ;
$R_2(so_2)$	SELECT Solde INTO @solde FROM Client WHERE id_client = 2 ;
r_1	*Vérifier si "SELECT @solde - 5 * @tarif ;" < 0 alors ROLLBACK ;*
$W_2(cl_2)$	UPDATE Client SET Solde = @solde - 5 * @tarif WHERE id_client = 2 ;
c_2	COMMIT ;

T_3 Vérification du nombre de places réservées

$R_3(re_{1,2})$	SELECT SUM(places_reservees) AS places_reservation FROM Reservation WHERE id_spectacle = 1 ;
$R_3(sp_1)$	SELECT (places_offertes - places_libres) AS places_spectacle FROM Spectacle WHERE id_spectacle = 1 ;
c_3	COMMIT ;

T_4 Mise à jour des places pour spectacle et client

$W_4(sp_1)$	UPDATE Reservation SET places_reservees = places_reservees + 10 WHERE id_client = 2 AND id_spectacle = 1 ;
$W_4(cl_1)$	UPDATE Spectacle SET places_offertes = places_offertes + 10, places_libres = places_libres + 10 WHERE id_spectacle = 1 ;
c_4	COMMIT ;

1. Tester sous MySQL pour vous assurer du bon fonctionnement

5.4 Concurrence

Trouver pour chaquer niveau d'isolation, un enchainement d'opérations (une histoire) dans 2 sessions différentes (deux terminaux) entre deux transactions [2], pour mettre en valeur l'erreur/le cas demandé.

(5.4.1) Mode `READ UNCOMMITTED` : Lecture sale

(5.4.2) Mode `READ COMMITTED` [3] : Ecriture sale

(5.4.3) Mode `REPEATABLE READ` : Tuple fantôme

(5.4.4) Mode `SERIALISABLE` [4] : *Dead lock*

2. Ne pas oublier pas de supprimer le mode auto-commit et ni de vérifier le niveau d'isolation dans CHAQUE session
3. Sous MySQL, la lecture sale est résolue grâce au *versioning*, qui permet de garder l'image de la donnée avant la transaction
4. Comme MySQL est en versionning, le deadlock ne peut arriver que sur une histoire de la sorte $w_1(x)w_2(y)w_1(y)w_2(x)$