Base de Data

JP Momo & Sttv & Brunoir

16 décembre 2014

Chapitre 1

Requêtes et résultats

1.1 Statements et ResultSet : définitions

Bon vu que vous êtes vraiment nuls, on va voir à quoi correspond les Statement en Java. Donc Statement est un interface de l'API JDBC qui définit les objets qui permettent d'exécuter des requêtes statiques SQL et ainsi retourner le résultat produit.
Généralement un Statement s'utilise avec un ResultSet (c'est tout le temps en fait). En effet, les instances de l'interface ResultSet contiennent les requêtes SQL. En gros, ils contiennent des tuples (#BogDonald, #3.33), lignes si vous préférez. Il faut savoir qu'un seul ResultSet ne peut être actif à la fois.
Pour assez blablater, getExemple() :

```
1 // creation d'un statement BASIC
2 Statement createStatement() throws SQLException // parce qu'on est des BG
3 // un autre statement
4 Statement createStatement(int rsType, int rsConcurrency) throws SQLException
5 // encore un autre
6 Statement createStatement(int rsTypen int rsConcurrency, int rsHoldability)
    throws SQLException
```

Vous me direz "Bordelle de merde pourquoi Pa..". Non en fait. Donc comme vous le voyez ci-dessus, il y plusieurs manières de créer un Statement.
La première, bon pas besoin de la commenter. A partie de la deuxième, bah y a des paramètres. On va les définir tout de suite.

- **int rsType** permet de définir le type du ResultSet, c'est-à-dire sa navigabilité (que je définirai plus tard).
- **int rsConcurrency** permet de définir le fait qu'on puisse faire mises à jour ou non.
- enfin **int rsHoldability** qui définit son comportement lors d'un commit.

Bon vous comprenez toujours pas et c'est bien parce que j'ai encore rien expliqué. Donc il y a quelques méthodes à connaître pour les ResultSet.

- pour créer un ResultSet contenant les résultats d'une requête c'est **ResultSet executeQuery(String sql) throws SQLException**.
- **void close() throws SQLException** (bon pas de commentaires)
- **int executeUpdate(String sql) throws SQLException** (c'est même pas définit dans la doc donc bon)
- **boolean execute(String sql) throws SQLException** qui renvoie vrai si le ResultSet s'est bien exécuté faux sinon ofc

1.2 Statements et ResultSet : utilisations

Bon vous vous rappelez des termes que j'avais définit auparavant ? Bien sûr que non parce que je l'ai ap fait GRO.
C'est bien beau de connaître quelques méthodes (parce qu'il y en a beaucoup) mais faut faire des choses avec notre ResultSet.
En ce qui concerne la navigabilité (int rsType) d'un ResultSet :

- **TYPE_FORWARD_ONLY** (par défaut) : donc navigation en avant.
- **TYPE_SCROLL_INSENSITIVE** : navigation dans tous les sens et où l'on veut (merde c'est dégueulasse). Cependant, on a pas d'accès aux modifications sur la source de données depuis l'ouverture du ResultSet.
- **TYPE_SCROLL_SENSITIVE** : navigation comme le précédent et en plus on a accès aux modifications.

En ce qui concerne la mise à jour (int rsConcurrency) :

- **CONCUR_READ_ONLY** (par défaut) : aucune modification.
- **CONCUR_UPDATABLE** : modifications possibles.

Enfin en ce qui concerne son comportement à la validation, à un commit (int rsHoldability) :

- **CLOSE_CURSORS_AT_COMMIT** (par défaut) : fermé après chaque validation ou commit pour les malins.
- **HOLD_CURSORS_OVER_COMMIT** : reste ouvert lors d'une validation (COMMIT ou ROLLBACK).

Bah maintenant qu'on connait tout sur les Statement et les ResultSet, il est grand temps de l'illustrer par un exemple.
Supposons que je veuille :

- parcourir le résultat de la requête dans n'importe quel sens sans avoir accès à d'éventuelles modifications dans les données sur la base.
- sans modifier les éléments au travers de ResultSet.
- et fermer le ResultSet si il y a un commit.

Voici the solution :

```
1 // on cree un Statement avec les differentes proprietes du ResultSet que l'on
      veut (voir au dessus pour les parametres)
2 Statement stm = co.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
      ResultSet.CONCUR_READ_ONLY, ResultSet.CLOSE_CURSOR_AT_COMMIT);
3 // apres avoir creer le Statement et donner les proprietes de notre ResulSet,
      on cree un ResultSet contenant notre requete ofc
4 ResultSet rs = stm.executeQuery("SELECT * FROM Film");
```

Les méthodes de mouvement d'un ResultSet dépendent des propriétés que l'on a définit. (il faut savoir qu'à la création, l'ouverture d'un ResultSet, le **ResultSet pointe avant la première ligne**)
Donc si il est de type TYPE_FORWARD_ONLY :
on ne pourra utiliser que **.next()** qui renvoie un booléen. En effet, **next()** passe à la ligne suivante du résultat de la requête si elle existe (vrai) et rien sinon (faux).
Sinon pour les reste on a :

- **previous()** : ligne précédente.
- **first()** : sur la première ligne, retourne vrai si elle existe, faux sinon (en gros un ResultSet vide).
- **last()** : sur la dernière ligne.
- **beforeFirst()** : avant la première (comme à l'ouverture).
- **afterLast()** : après la dernière.

- **relative (int rows (lignes pour les nuls))** : il fait rows ligne après la position courante du ResultSet et revient en arrière si rows est négatif.
- **absolute(int row)** : se place à la row-ième ligne. Si row = 1, il se place sur la première ligne et si row est négatif ou row = 0 se place à la dernière ligne.

Bon on va utiliser l'exemple précédent avec quelques modifications pour des raisons de commodité.

```
// comme d'hab un Statement (voir au-dessus pour les parametres)
Statement stm = co.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE);
ResultSet rs = stm.executeQuery("SELECT * FROM Film");
rs.absolute(3); // on se positionne sur la 3eme ligne
rs.relative(5); // on se positionne sur la 8eme ligne car on etait sur la 3eme
    ligne et on rajoute 5 lignes
rs.previous(); // on se positionne sur la 7eme ligne
rs.last(); // on se positionne sur la derniere ligne
```

En ce qui concerne les mises à jour de lignes, si le Statement est ouvert en mode CONCUR_UPDATABLE, alors on peut mettre à jour par le biais du ResultSet.
La mise à jour d'une ligne se fait en 2 étapes :

- mise à jour de la nouvelle valeur de la colonne : **update<type>()**.
- changement affectés à la ligne concernée **updateRow()**(et à ce-moment là la base sera mis à jour).

Exemples :

```
Statement stm = co.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_UPDATABLE);
ResultSet rs = stm.executeQuery("SELECT Titre FROM Film
                                 WHERE NumFilm = 123");
rs.next(); // on se positionne a la 1ere ligne
rs.updateString("Titre", "Django Unchained"); // on modifie le titre
rs.updateRow(); // effectue la modification de la ligne
```

Toujours avec le même mode (voir au-dessus), on peut supprimer une ligne dans la base avec **deleteRow()**. Elle se place, après la suppression, avant la première ligne valide suivant celle qui vient d'être supprimée. Son indice devient invalide et on peut tester son existence avec **rowDeleted()**.

```
Statement stm = co.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_UPDATABLE);
ResultSet rs = stm.executeQuery("SELECT * FROM Film");
rs.absolute(4); // on se positionne a la 4eme ligne
rs.deleteRow(); // supprime la ligne courante
bool = rs.rowDeleted(); // bool vaut vrai
```

Enfin qui dit suppression dit insertion (toujours avec le même mode). En gros, ça permet l'insertion de nouvelles lignes.
L'insertion se fait en 3 étapes :

- on se positionne sur la ligne d'insertion avec **moveToInsertRow()**.
- initialiser les valeurs des champs de la ligne à insérer avec **update<type>()**.
- enfin insérer la ligne avec **insertRow()**.

```
Statement stm = co.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.
    CONCUR_UPDATABLE);
ResultSet rs = stm.executeQuery("SELECT Prenom, Nom, NumIndividu FROM Individu
    ");
rs.moveToInsertRow(); // positionne sur une ligne d'insertion en gros vide
rs.updateString(1, "Bruno"); // cree le prenom
rs.updateString(2, "Benjamin Pierrot"); // cree le nom
rs.updateLong(3, 269); // cree le numero
rs.insertRow(); // insere la ligne qu'on vient de creer
rs.moveToCurrentRow(); // revient a l'ancienne position
```

1.3 PreparedStatement trop ez

Bon on pas vu le gros du cours. Normalement cette partie devrait être assez facile.
Tout d'abord, on va parler de requêtes pré-compilées. Si on doit répéter plusieurs fois la même requête en utilisant un objet Statement, à chaque fois le serveur devra interpréter la requête SQL et en particulier créer un plan de requête. En général, c'est très coûteux (le CAAASH).
C'est pour ça qu'on va utiliser l'interface PreparedStatement. En effet, le code SQL est transmis à la base une seule fois, dès que la méthode prepareStatement() délivre l'objet PreparedStatement correspondant. Vu que le principe revient à la même qu'un Statement, on va faire un multitude d'exemples.

```
// on cree un PreparedStatement avec la requete
PreparedStatement psm = co.prepareStatement("SELECT titre FROM Film
                                            WHERE numFilm = 50
                                            AND realisateur = 8");
// on execute la requete
ResultSet rs = psm.executeQuery();
// tant qu'il y a un suivant, donc une ligne contenant un resultat
while(rs.next()){
        // on recupere et on affiche le resultat
        System.out.println(rs.getString(1));
}
```

Bon ça c'est un truc assez simple.(faut savoir qu'on peut toujours utiliser les méthodes de ResultSet).
La principale différence avec un Statement c'est qu'on va pouvoir passer des paramètres à notre requête grâce au?. On va reprendre le même exemple.

```
PreparedStatement psm = co.prepareStatement("SELECT titre FROM Film
                                            WHERE numFilm = ?
                                            AND realisateur = ?");
psm.setInt(1, 50); // pour le 1er parametre, on lui attribut la valeur 50
psm.setInt(2, 8); // pour le second, on lui attribut 8
ResultSet rs = psm.executeQuery();
// tant qu'il y a un suivant, donc une ligne contenant un resultat
while(rs.next()){
        // on recupere et on affiche le resultat
        System.out.println(rs.getString(1));
}
```

1.4 ResultSetMetaData, double v T F

Bon cette partie va être assez courte. En gros, ResultSetMetaDate permet d'obtenir des informations sur la structure d'un ResultSet comme le nom de la colonne choisie, le nombre de colonnes du ResultSet, la table d'où provient la colonne, etc... ? En gros, plein d'informations très utiles(non je rigole).
Je vais résumer ça par l'exemple du cours parce que bon y a pas grand chose.

```
1  Connection co = DriverManager.getConnection(url);
2  Statement stm = co.createStatement();
3  ResultSet rs = stm.executeQuery("SELECT a, b, c FROM Table1");
4  ResultSetMetaData rsmd = rs.getMetaData();
5  int nbreColonnes = rsmd.getColumnCount();
6  for (int i = 1; i <= nbreColonnes; i++){
7          int jdbcType = rsmd.getColumnType(i);
8          String name = rsmd.getColumnTypeName(i);
9          System.out.println("L'attribut " + i + " est du type JDBC " +
                     jdbcType + " dont le nom est " + name);
10 }
```

Chapitre 2

Fonctions et procédures stockées

2.1 Partie PL/SQL

Donc comme vous êtes tous des GROSSES mer..., je vais vous expliquer ce que sont les fonctions et les procédures en PL/SQL. Tout d'abord, à quoi servent les fonctions et les procédures STOCKEES ?
Elle servent à **enregistrer des programmes dans le noyau Oracle**. Elles peuvent être utilisées par tout le monde (selon les droits) et sont stockées sous forme de *pseudo-code*, c'est-à-dire qu'elles ne sont compilées qu'UNE SEULE FOIS. C'est génial non ?
Trêve de bavardages, on va voir leurs syntaxes.
Une procédure en PL/SQL se déclare de la manière suivante :

```
1 -- declaration d'une procedure
2 CREATE [OR REPLACE] PROCEDURE nom_prodecure ([liste des parametres]) IS|AS
3         [declaration des varibles]
4 BEGIN
5         -- corps de la procedure
6 EXCEPTION -- facultatif
7         -- definition des exceptions
8 END;
```

Bon ce n'est pas très clair mais on va voir ça petit à petit.
Donc, la première ligne correspond à une déclaration de procédure basique avec son nom et ses différents paramètres (ne pas oublier le IS ou le AS).
La deuxième ligne correspond à la déclaration des variables.
A partir du **BEGIN**, on écrit ce que va faire notre procédure. **EXCEPTION** est facultatif mais correspond à la zone d'exception du code.
Enfin le **END**, bah THE END.
En ce qui concerne les paramètres dans une procédure (ou une fonction), il faut leur donner un nom (gicLo), spécifié si c'est un paramètre **d'entrée** (IN, donc non modifiable) ou **de sortie** (OUT, modifiable par la procédure) ou les deux et enfin spécifié son type(une procédure ou une fonction n'a pas obligatoirement de paramètres).

```
1 CREATE OR REPLACE PROCEDURE numActeur (nom IN VARCHAR2, prenom IN VARCHAR2,
      num OUT INTEGER)
```

Ici, la procédure numActeur qui donne le numéro d'un acteur, prend en paramètres **d'entrée**, de type VARCHAR 2, *nom* et *prenom*. En paramètre **de sortie**, de type INTEGER, on a *num*.

Voici un exemple de procédure stockée.

```
/* on definit la procedure unTitre qui renvoie le nom d'un film
en entree, on a le numero du film et du realisateur
en sortie, le titre du film */
CREATE OR REPLACE PROCEDURE unTitre (numFilm IN INTEGER, realisateur IN
    INTEGER, titreFilm OUT VARCHAR2) IS
BEGIN
        /* on selectionne le titre du film DANS titreFilm (le parametre de
            sortie)
        sinon ca ne marche pas */
        SELECT f.titre INTO titreFilm
        FROM ens2004.film f
        WHERE f.numFilm = numFilm
        AND f.realisateur = realisateur;
END;
```

Pas besoin d'expliquer ce qui différencie une fonction d'une procédure. Donc syntaxe.

```
-- declaration d'une fonction
CREATE [OR REPLACE] FUNCTION nom_fonction([liste des parametres])
        RETURN type_retour IS|AS
        [variable de retour]
        [declaration des varibles]
BEGIN
        -- corps de la fonction
        RETURN variable_retour;
EXCEPTION \\facultatif
        -- definition des exceptions
END;
```

La principale différence c'est que l'on spécifie la variable que l'on va renvoyé. En ce qui concerne les paramètres d'une fonction, c'est la même chose qu'une procédure **SAUF** que l'on préfère ne spécifier que des paramètres d'entrées (IN).
Vu que vous êtes trop fort, on peut passer directement à un exemple ofc.

```
/* on definit la fonction numFilm qui renvoie le numero d'un film
en entree, on a le titre du film et le numero du realisateur */
CREATE OR REPLACE FUNCTION numFilm (titreFilm IN VARCHAR2, realisateur IN
    INTEGER)
        RETURN INTEGER IS
          numFilm INTEGER;
BEGIN
        -- on selectionne le numero du film dans numFilm
        SELECT f.numFilm INTO numFilm
        FROM ens2004.film f
        WHERE f.titre = titreFilm
        AND f.realisateur = realisateur
        -- et on retourne numFilm qui contient le numero du film
        RETURN numFilm;
END;
```

2.2 Partie Java

C'est bien beau de définir des fonctions ou des procédures stockées mais faut quand même les utiliser un jour.
Je suppose que vous avez tous suivi le magnifique cours sur JDBC, donc on va directement voir le code Java. Donc comment appeler nos fonctions et procédures en Java. Il suffit d'utiliser ce que l'on appelle l'interface CallableStatement. Comme l'indique son nom, elle appelle les procédures et les fonctions stockées sur Oracle. Comment ça marche ?(pas le site ofc) Il suffit d'indiquer le nom de la fonction ou de la procédure lors de l'initialisation de l'objet **CallableStatement** grâce à la méthode **prepareCall()** de l'interface Connection (fallait suivre le cours sur JDBC).
Il existe deux cas : soit la procédure ou la fonction comporte des paramètres lors de l'appel soit elle n'en comporte pas. EXEMPLE!!!

```
1  // procedure sans parametres
2  CallableStatement cst = co.prepareCall("{call nom_procedure()}");
3  // procedure avec parametres (2 ici)
4  CallableStatement cst1 = co.prepareCall("{call nom_procedure(?, ?)}");
5  // fonction sans parametres
6  CallableStatement cst2 = co.prepareCall("{? = call nom_fonction()}");
7  // fonction avec parametres (2 ici)
8  CallableStatement cst3 = co.prepareCall("{? = call nom_fonction(?, ?)}");
```

Dans les cas, où il y a des paramètres, il faut qu'on les définissent pour Java (bah oui c'est pas de la gicMa). Pour les paramètres d'entrée (IN), il faut utiliser la méthode set<type>(<rang>, <valeur>). On va la définir pour les plus mauvais d'entre vous :

- <type> correspond au type que l'on va définir pour notre paramètre
- <rang> correspond au rang du paramètre à positionner (il faut respecter l'ordre de la procédure ou de la fonction stockée)
- <valeur> correspond à la valeur que va prendre notre paramètre

En ce qui concerne les paramètres de sortie, il faut spécifier, dans un premier temps, son type. Pour cela, on utilise la méthode registerOutParameter(<rang>, <type>) :

- <rang> même chose que les paramètres d'entrée
- <type> le type de notre paramètre

Si on reprend la procédure unTitre et la fonction numFilm, voilà ce que ça donne :

```
1  // pour la procedure unTitre
2  // on a 2 parametres en entree (INTEGER) et 1 en sortie (VARCHAR2)
3  CallableStatement cst = co.prepareCall("{call unTitre(?, ?, ?)}");
4  cst.setInt(1, '50'); // numFilm est a la premiere place
5  cst.setInt(2, '45'); // realisateur est a la deuxieme place
6  cst.registerOutParameter(3, java.sql.Types.VARCHAR); // titreFilm a la
       troisieme
7
8  // pour la fonction numFilm
9  // on a 2 parametres en entree (VARCHAR2 et INTEGER)
10 CallablaStatement cst1 = co.prepareCall("{? = call numFilm(?, ?)}");
11 // il faut respecter l'ordre des ? et dans la fonction SQL
12 cst1.setString(2, 'TITANIQUE');
13 cst1.setInt(3, '69');
14 cst1.registerOutParameter(1, java.sql.Types.INTEGER);
```

Vous avez compris ? Cependant, on n'a toujours pas exécuter notre procédure ou fonction. C'est très simple THREAD, il suffit d'utiliser la méthode **execute()**. (Vu que c'est un booléen, on préfère la stocker dans une

variable). Et enfin, il faut pouvoir récupérer les valeurs de sortie (si il y en a).
Pour cela, on utilise la méthode get<type>(<rang>) qui fonctionne un peu près comme set<type>(<rang>, <valeur>).
Si on reprend les exemples précédents :

```
// pour la procedure unTitre
CallableStatement cst = co.prepareCall("{call unTitre(?, ?, ?)}");
cst.setInt(1, '50');
cst.setInt(2, '45');
cst.registerOutParameter(3, java.sql.Types.VARCHAR);
boolean success = cst.execute(); // on execute notre procedure
String titreFilm = cst.getString(3); // on recupere dans titreFilm le titre
    renvoye
cst.close(); // on oublie pas de fermer (c'est pour etre propre)

// pour la fonction numFilm
CallablaStatement cst1 = co.prepareCall("{? = call numFilm(?, ?)}");
cst1.setString(2, 'TITANIQUE');
cst1.setInt(3, '69');
cst1.registerOutParameter(1, java.sql.Types.INTEGER);;
boolean success1 = cst1.execute(); // on execute notre procedure
int numFilm = cst1.getInt(1); // on recupere dans numFilm le numero du film
cst1.close();
```

Chapitre 3

Concurrence d'accès

La concurrence d'accès est un principe, en base de données, qui consiste en **la gestion de données**. Pour être plus clair, en reprenant le cours, si plusieurs utilisateurs manipulent les mêmes données en même temps, il faut arbitrer entre :

- la disponibilité de l'information : ne pas bloquer tout le monde parce que une personne travaille
- la cohérence de l'information : ne pas rendre les données incohérentes en tenant compte de plusieurs demandes en concurrence en même temps.

3.1 Accès concurrents

Pour cela, on va définir plusieurs termes importants pour la fiche et pour tout le reste.

Transaction : unité logique de traitement. Pour résumer grossièrement, ça correspond à la modification ou l'interrogation d'une base de données (requêtes, fonctions, procédures, etc...).
Le début d'une transaction se définit par un ordre SQL(SELECT, UPDATE, INSERT, etc...) ou la fin de la transaction précédente. La fin d'une transaction est définit par l'instruction COMMIT ou ROLLBACK.
L'instruction COMMIT consiste à valider la transaction en cours et donc rendre les modifications persistantes.
Quant à ROLLBACK, elle annule la transaction en cours et toutes les modifications effectués lors de cette dernière.
Accès concurrents : plusieurs utilisateurs accèdent, en même temps, à la même donnée dans la base.
Base cohérente : contraintes d'intégrités respectées(en gros les règles sur les tables). Si lors d'une transaction une BD passe d'un état cohérent à un autre état cohérent, l'intégrité des données est sauvegardée.
Consistance des données : Si SGBD (Système de gestion de base de données) garantit que les données utilisés par une transaction ne sont pas modifiées par des requêtes d'autres transactions pendant cette même transaction.

Bon c'est un peu trop théorique mais y aura sûrement ça au DS (non je rigole). Pour mieux comprendre la consistance des données, on va aller ouvrir les verrous.
Pour assurer la consistance de nos données, il faudra verrouiller les données durant la durée de la transaction. Il en existe 3 :

- verrous partagés (shared lock) : pour lire les données avec l'intention d'y faire des mises à jour.
- verrous exclusifs (exclusive lock) : pour modifier des données.
- verrous globaux (global lock) : pour bloquer un ensemble de données, généralement une table toute entière.

Il y a une autre notion à connaître c'est la collision. si 2 transactions accèdent à la même donnée en même temps, il y a collision avec un GROS CAMION et donc une perte de cohérence.

3.2 Utilisation et problèmes liés aux verrous

Quand est ce qu'un verrou est posé ?
Il est posé jusqu'à la fin de la transaction en cours. Le seul moyen de relâcher les verrous posés par une transaction est par le biais des instructions **COMMIT** et **ROLLBACK**(lors de la création d'une table aussi, il y a un COMMIT implicite).
Pour mieux comprendre cette histoire de verrous, on va reprendre les exemples du cours et les commenter.

Pour une collision de type : perte de mise à jour :
T_1 et T_2 modifient simultanément la quantité *qte*.

Temps	État de la base	Transaction T_1	Transaction T_2
t_0	qte=1000		
t_1		Lire qte	
t_2		qte←qte+3000	
t_3	qte=4000	Écrire qte COMMIT	
t_4			Lire qte
t_5			qte←qte+500
t_6			Écrire qte
	qte=4500		COMMIT

Donc dans cet exemple, 2 transactions veulent modifier simultanément la quantité qte. Le but étant d'avoir une quantité qte égale à 4500. Donc ici on a l'ordre des opérations à faire pour chacune des transactions. Cependant, il faut voir comment fonctionnent les verrous dans cet exemple.

Temps	Etat de la base	Transaction T_1	Transaction T_2
t_0	qte=1000		
t_1		Lire qte	
t_2			Lire qte
t_3		qte←qte+3000	
t_4	qte=4000	Écrire qte COMMIT	
t_5			qte←qte+500
t_6	**qte=1500**		Écrire qte COMMIT

À la fin, *qte* : 1000 ? ou 1500 ? ou 4500 ?

Dans ce tableau, à t0, qte = 1000. Ensuite à t1, la transaction T1 lit la valeur de qte (donc 1000), de même pour la transaction T2 à l'instant t2 (qte = 1000).
A l'instant t3, T1 modifie la valeur de qte (qte = qte + 3000) et à t4 enregistre cette modification par l'instruction COMMIT. Donc qte = 4000. A t5, T2 modifie aussi la valeur de qte et enregistre cette valeur. Cependant on s'attendrait à avoir qte = 4500 mais non bande de gogols. Cette erreur intervient à l'instant t2. En effet, T2 lit la valeur actuel de qte, c'est-à-dire 1000. Donc sa modification de la valeur qte ne prend pas en compte la modification de T1. En effet, il y a une collision et donc une mauvaise gestion des verrous. La solution pour avoir qte = 4500 est la suivante.

Temps	État base	Transaction T_1	Transaction T_2
t_0	qte=1000		
t_1		**Verrou exclusif** sur qte + Lire qte	
t_2			**Attente** pour poser **verrou exclusif** sur qte
t_3		qte←qte+3000	
t_4	qte=4000	Écrire qte COMMIT	
t_5			**Verrou exclusif** sur qte
t_6			Lire qte
t_7			qte←qte+500
t_8	**qte=4500**		Écrire qte COMMIT

Dans cette version, T1 pose un verrou exclusif lors de la lecture de qte, c'est-à-dire, que T1 va appliquer une modification sur cette valeur et que T2 doit attendre la fin de ce verrou (COMMIT ou ROLLBACK). Après que T1 ait modifier la valeur et fait un COMMIT (qte = 4000), c'est au tour de T2 d'appliquer un verrou exclusif et de modifier la valeur de qte. Et comme par gicMa, qte = 4500, car la valeur à laquelle accède T2 a été modifié et validé par T1.

Pour les collisions de type : lecture impropre et lecture non reproductible se référait au cours parce que j'ai trop la flemme. Si vous avez compris cette exemple, vous comprendrez sûrement les autres (je vous conseille d'aller voir le cours pour tester vos connaissances ofc).

Vu que vous êtes chimiques, on va faire une suite d'exemples :

```
-- l'utilisateur 1 change le nom d'un film (numero 15 et realisateur 58)
-- on a ici l'application d'un verrou exclusif
UPDATE ens2004.film
SET titre = 'MABITE'
WHERE numFilm = 15
AND realisateur = 58;

-- l'utilisateur 1 et 2 executent la meme requete d'affichage
SELECT * FROM ens2004.film;
-- A ce moment la, la modification du film n'etant pas valide, le titre du
    film affiche est toujours l'ancien

-- l'utilisateur 1 fait un COMMIT
COMMIT;

SELECT * FROM ens2004.film;
-- La valeur etant modifie, l'affichage contiendra la nouvelle valeur c'est-a-
    dire MA...
```

Il existe plusieurs niveaux de cohérence pour une transaction. Une donnée est dites salie si elle a été modifiée par une transaction non confirmée. En gros, par un COMMIT. Pour une transaction T, on peut exiger que T satisfasse une ou plusieurs des propriétés (c'est pas vraiment important mais c'est toujours utile) :

1. : T ne modifie pas des données salies par d'autres transactions.
2. : T ne confirme pas ses changements avant la fin de la transaction.

3. : T ne lit pas des données salies pas d'autres transactions.
4. : D'autres transactions ne salissent pas des données lues par T avant que T soit terminée.

Pour chacune de ces propriétés, un niveau de cohérence est attribué à T :
- niveau 0 si T vérifie 1 : donc pas de problèmes de perte de mise à jour.
- niveau 1 si T vérifie 1 et 2 : si une transaction est annulée (ROLLBACK), il n'est pas nécessaire de défaire explicitement les modifications antérieures à l'annulation.
- niveau 2 si T vérifie 1 et 2 et 3 : pas de problèmes de perte de mise à jour et de lecture impropre.
- niveau 3 si T vérifie 1 et 2 et 3 et 4 : pas de problèmes de perte de mise à jour, de lecture impropre et assure la reproductibilité des lectures, donc une isolation totale de la transaction. (je vous avais dit de voir le cours)

Sur Oracle les verrous sont invoqués automatiquement dans des cas spécifiques, mais on peut le faire manuellement aussi. Ce tableau résume bien les différents verrous que l'on peut mettre.

Opération SQL	Verrou ligne	Verrou table
SELECT	Non	Aucun
INSERT	Oui	RX
UPDATE	Oui	RX
DELETE	Oui	RX
SELECT... FOR UPDATE	Oui	RS
LOCK TABLE ... IN :		
ROW SHARE MODE	Non	RS
ROW EXCLUSIVE MODE	Non	RX
SHARE MODE	Non	S
SHARE EXCLUSIVEMODE	Non	SRX
EXCLUSIVE MODE	Non	X

- **RX** : Row exclusif, interdiction de modification et de lecture sur la ligne que l'on modifie pour tout autre utilisateur
- **RS** : Row share, interdiction de modification seulement sur la ligne que l'on lit
- **S** : Share, interdiction de modification sur la table que l'on lit
- **X** : Exclusif, interdiction de modification et lecture sur la table que l'on modifie

Problèmes possibles liés aux verrous

Deadlock : On rencontre ce problème lorsque un utilisateur attend la fin du verrou d'un autre alors que celui ci attend la fin du verrou de l'un.
Par Exemple je pose un verrou sur la ligne 1 d'une table puis un random pose un verrou sur la ligne 2. Je souhaite récupérer la valeur de la ligne 2 → je suis mis en attente par le random, sauf que ce putain de random attend que moi j'enlève mon verrou sur la ligne 1. Du coup on se retrouve en attente tout les deux comme des cons. → Pour éviter ce problème il faut que l'ordre dans laquelle le random et moi posons les verrous soit le même. Je prend la ligne 1, il prend la ligne 1, je prend la ligne 2, il prend la ligne 2 etc...
Cette solution est appelé **Règle de Havender**. Allez savoir pourquoi...
Prenons deux utilisateurs lambda :

```
-- l'utilisateur 1 change le nom du film (numero 15 et realisateur 58)
UPDATE ens2004.film
SET titre = 'TD32'
WHERE numFilm = 15
AND realisateur = 58;

-- l'utilisateur 2 change le nom d'un autre film (numero 32 et realisateur 96)
UPDATE ens2004.film
SET titre = 'BOGOSS'
WHERE numFilm = 32
AND realisateur = 96;

-- l'utilisateur 1 change le nom du film qui est en train d'etre modifie par l
    'utilisateur 2 (numero 32 et realisateur 96)
UPDATE ens2004.film
SET titre = 'TD321'
WHERE numFilm = 32
AND realisateur = 96;

-- l'utilisateur 2 change le nom du film qui est en train d'etre modifie par l
    'utilisateur 1 (numero 15 et realisateur 58)
UPDATE ens2004.film
SET titre = 'BOGOSS1'
WHERE numFilm = 15
AND realisateur = 58;
```

Dans cet exemple, on ne fait pas de COMMIT. A partir de la troisième instruction, Oracle va tourner en boucle (il attend le COMMIT de l'utilisateur 2) et va afficher sur l'utilisateur 1 blocage fatale de même à la quatrième instruction. Cette erreur est due au fait que deux utilisateurs veulent modifier des données qui sont modifiés par l'un et par l'autre mais n'ont fait aucune validation ou annulation.

Problème de la tasse à café.... : Il s'agit du fait que la saisie d'un utilisateur peut potentiellement mettre indéfiniment le programme en pause (Par exemple, je vais prendre une tasse de thé parce que j'aime le thé et faire chier les gens), ce qui provoque dans le cas où on a posé un verrou avant la demande de saisie, cela verrouille la cible indéfiniment et donc bloqué d'autre utilisateur. → Pour éviter ce problème, il faut poser le verrou après la saisie tout simplement.

Chapitre 4

Base de données Objet

Objet ? Wut ? C'est quoi encore cette partie ?
Ne vous inquiétez pas ! La partie objet de la base de données n'est juste qu'une simple révision de vos cours d'informatique objet mais avec ce merveilleux langage SQL.

4.1 Créer un type

A quoi sa sert ? Bah sa sert à définir un type objet qui est abstrait. Il est défini pas l'utilisateur pour décrire la structure (ses attributs) et le comportement (ses méthodes).
Sa fameuse syntaxe :

```
1 CREATE [ OR REPLACE ] TYPE nomType AS OBJECT
2 (nomAttribut type [, nomAttribut type ....]);
```

Prenons un exemple : Nous voulons créer un type adresse avec une rue, une ville, un code postal et le nom de la ville :

```
1 CREATE TYPE obAdresseTy AS OBJECT(
2 Rue VARCHAR2(50),
3 Ville VARCHAR2(30),
4 CodePostal CHAR(5),
5 Pays VARCHAR2(15));
```

Comme vous le constatez, nous créons un type comme si c'était une table avec des attributs, la nature de l'attribut et la taille de l'attribut. AS OBJECT permet de définir le type comme un objet ; c'est à dire comme un constructeur en Java qui va être utilisé dans d'autres types et tables quand c'est nécessaire.

Maintenant nous voulons créer un type personne qui possède un nom, un prénom et le type adresse qu'on a créé plus tôt.

```
1 CREATE TYPE obPersonneTy AS OBJECT(
2 Nom VARCHAR2(30),
3 Prenom VARCHAR2(30),
4 Adresse obAdresseTy);
```

Vous pouvez le remarquer aussi qu'un attribut est de nature un type. Cet attribut permet d'appeler le type qu'on souhaite pour éviter de réécrire tous les attributs d'adresse. En gros l'attribut adresse va appeler tous les attributs du type adresse. Plus simple non ?

Pour supprimer un type rien de spécial. Mais à faire si il n'est plus utilisé !

```
DROP TYPE nomObjetType;
```

Une erreur apparaitra si vous voulez supprimer un type qui est encore présent dans une table utilisée. Pour cela il faut supprimer la table puis le type.

4.2 Créer une table avec des types

Après avoir pris le temps de créer ces jolis types, nous voulons créer une table qui comporte la colonne personne à l'intérieur accompagné d'un login et de la date d'inscription.

```
CREATE TABLE obClient (
login CHAR(20) PRIMARY KEY ,
client obPersonneTy,
dateIns DATE NOT NULL);
```

Le code ci-dessus nous montre que la table peut appeler un type. A quoi sa sert ? Bah sa sert à raccourcir le code et d'éviter d'appeler tous les attributs qu'on a besoin. De manière plus abstraite, le table va appeler tous les attributs de type Personne qui va appeler tous les attributs de type Adresse. En gros nous avons économiser 7 lignes de création d'attributs qui va quand même apparaître dans la table. Elle n'est pas belle la vie ?
REMARQUE : Dans une table, il faut toujours avoir un attribut qui a le rôle de clé primaire (Genius gnégné) MAIS un type ne peut pas être une clé primaire (Je t'ai troué le cul là hein !).

Maintenant qu'on a perdu du temps pour créer des tables, il faut savoir les remplir (de sp... spécialité maison). Regardez bien le code :

```
INSERT INTO obClient
VALUES ('toto',
obPersonneTy ('Duval', 'Thomas',
obAdresseTy('rue des dunes', 'Trouville', '76666', 'France')),
TO DATE('01/05/07','DD/MM/YY'));
```

Comme vous le constatez, pour donner des valeurs à une table c'est la même procédure SQL MAIS nous avons mis un type dans notre table et pour compléter les attributs de ce type, il faut juste l'appeler.
ATTENTION : Si un type possède un autre type dans ses attributs, il faut faire la même opération : appeler le type et compléter ses attributs.

Après avoir bien remplis (hohoho), faut savoir rechercher les données d'un type. C'est la même procédure mais faut faire un appel spécial pour le type.

Pour chercher le nom d'un client :

```
SELECT p.client.Nom FROM obClient p;
```

Pour que le code marche à tous les coups, il faut donner une "abréviation" à notre table. Pourquoi ? Cela permet au code SQL de ne pas se confondre lors de l'exécution. Ici, nous avons mis comme abréviation la lettre p pour le client. Cela permettra de pouvoir cibler plus facilement ce qu'on cherche. Nous allons cibler dans ce cas l'attribut client qui est de type Personne dans la table et plus précisément nous voulons l'attribut nom du type Personne.

Pour chercher la ville d'un client ;

```
SELECT p.client.Adresse.Ville FROM obClient p;
```

C'est la même procédure mais pour ce cas, nous voulions adresse. Pour le chercher, il faut passer par l'attribut client de type Personne qui contient Adresse qui celui la contient l'attribut ville.

Maintenant pour modifier un type dans un table, c'est la même méthode mais faut spécifier l'appel tout simplement.
Modifions la rue d'un client tiens :

```
UPDATE obClient p
SET p.client.adresse.rue='15 rue des vagues'
WHERE UPPER (p.client.nom)='DUVAL';
```

Comme prévu, la méthode SQL pour modifier ne change pas mais faut juste faire la spécification comme dans la recherche.

4.3 Créer une table avec un seul type d'objet

La méthode ne change pas, juste spécifier le type d'objet à utiliser et donner une clé primaire.

```
CREATE TABLE nomTable OF nomTypeObjet;
```

ATTENTION : Si nous voulons créer une table avec un seul type d'objet MAIS le type concerné possède un attribut venant d'un autre type, là c'est la merde. Mais SQL a tout pensé, faut juste donner une référence à l'attribut qui possède le type.
Regardez le code ça sera plus simple à comprendre.

```
CREATE TYPE obIndividuTy AS OBJECT (
NumIndividu NUMBER(5),
Nom VARCHAR2(30),
Prenom VARCHAR2(30) );

CREATE TABLE obIndividu OF obIndividuTy
(NumIndividu PRIMARY KEY);

CREATE TYPE obFilmTy AS OBJECT (
NumFilm NUMBER(5),
Titre VARCHAR2(50),
realisateur REF obIndividuTy );

CREATE TABLE obFilm OF obFilmTy
(NumFilm PRIMARY KEY);
```

Comme vous le voyez, la table obFilm n'est que de type obFilmTy. Cela veut dire que cette table ne peut posséder que les attribut du type Film. Le problème c'est qu'il a le type Individu qui vient foutre la merde. Pourquoi il a un REF ? ? Le REF a pour fonctionnalité de faire référence a un autre type qui possède aussi une table (ici c'est Individu). Donc avec le REF l'affaire est réglée.

Maintenant pour donner des valeurs, c'est autre chose. Il faut d'abord donner les attributs du type. Ici il faut d'abord donner les attributs de Individu avant de donner les attributs Film.

Le code pour individu :

```
INSERT INTO obIndividu VALUES
(1996,'LECONTE', 'PATRICE');
INSERT INTO obIndividu VALUES
(2871,'SPIELBERG','STEVEN');
INSERT INTO obIndividu VALUES
(2987,'PAGNOL', 'MARCEL');
```

Ici c'est simple on ajoute des données dans la table obIndividu. Ensuite pour donner les attributs Film, c'est plus compliqué. Pour insérer les attributs, il faut donner la référence à l'individu qu'on souhaite.

```
INSERT INTO obFilm
SELECT 43, 'LE MARI DE LA COIFFEUSE',
REF(b) FROM obIndividu b
WHERE NumIndividu=1996;
INSERT INTO obFilm
SELECT 84,'LA LISTE DE SCHINDLER',
REF(a) FROM obIndividu a
WHERE NumIndividu=2871;
INSERT INTO obFilm
SELECT 18,'CESAR',
REF(c) FROM obIndividu c
WHERE NumIndividu=2987;
```

Ce code nous explique que lorsqu'on ajoute un film, il faut faire référence (avec REF) sur l'individu qu'on souhaite car sinon, le film ne contient aucun réalisateur, c'est similaire à NullPointeurException de mes c...

4.4 Création d'une spécification et d'un corps par des méthodes

Une méthode permet de manipuler des types et de définir leur comportement à n'importe quel moment. Une méthode ne peut cependant effectuer des modifications sur la base de données.

Pour créer un type avec sa spécification :

```
CREATE [OR REPLACE] TYPE nomType AS OBJECT
(nomAttribut type [, nomAttribut type ? ] ,
MEMBER specification de procedure | fonction
[MEMBER specification de procedure | fonction ? ]
[MAP MEMBER specification de fonction ]
[ , PRAGMA RESTRICT REFERENCES (
methode, mode [ , mode ? ] ] );
```

MAP : pour définir un critère de tri. mode : pour augmenter/restreindre les possibilités des méthodes (WNDS, WNPS, RNDS, RNPS).

Exemple avec le type Personne et une fonction qui détermine si la Personne habite en Ile de France ou en Province :

```
CREATE [OR REPLACE] TYPE obPersonneTy AS OBJECT
Nom VARCHAR2(30),
Prenom VARCHAR2(30),
Adresse obAdresseTy)
MEMBER FUNCTION parisProvince RETURN VARCHAR2,
[ , PRAGMA RESTRICT REFERENCES(
parisProvince,WNDS,RNDS));
```

WNDS : pas de modification de la base de données RNDS : ne doit pas consulter la base de données.

Maintenant faut définir le corps :

```
CREATE [OR REPLACE] TYPE BODY nomType IS|AS
MEMBER corps de la procedure | fonction
[MEMBER corps de la procedure | fonction ? ]
[MAP MEMBER corps de la fonction de tri ]
```

Dans notre exemple nous voulons définir la fonction qui détermine où la Personne habite :

```
CREATE OR REPLACE TYPE BODY obPersonneTy IS
MEMBER FUNCTION parisProvince RETURN VARCHAR2 IS
                pp VARCHAR2(8)
        BEGIN
                if (SUBSTR(adresse.CodePostal,1,2) IN
                                ('77','78','75', '91','92', '93','94','95'))
                        then pp:='idf';
                        else pp:='province';
                end if;
        return pp
        END parisProvince;
END;
```

Nous pouvons faire pareil avec une fonction de tri.
Spécification :

```
CREATE OR REPLACE TYPE obAdresseTy AS OBJECT
Rue VARCHAR2(50),
Ville VARCHAR2(30),
Code Postal CHAR(5)
Pays VARCHAR2(15),
MAP MEMBER FUNCTION triAdresse RETURN VARCHAR2);
```

Corps :

```
CREATE OR REPLACE TYPE BODY obAdresseTy IS
        MAP MEMBER FUNCTION triAdresse RETURN VARCHAR2 IS
                BEGIN
                RETURN CodePostal||Ville;
        END triAdresse;
END;
```

Recherche :

```
SELECT p.client.Nom,
        p.client.parisProvince situation ,
        p.client.adresse.codePostal code FROM obClient p
ORDER BY p.client.adresse;
```

4.5 Tableau dans une table

Oui un tableau dans une table vous avez bien entendu. Voyons d'abord comment créer un tableau en SQL. Il s'agit d'un type spécifique qui "hérite" de VARRAY dont il faut spécifier le type qu'il doit stocker. On déclare ce type de la façon suivante :

```
CREATE TYPE nomType AS VARRAY (limite) OF type;
```

Supposons qu'un film ai dix acteurs, je ne vais pas lui mettre 10 attributs de type ob_Individu_Ty, je vais plutôt créer le type acteur**s**Ty qui sera un tableau de 10 ob_Individu_Ty :

```
CREATE TYPE acteursTy AS VARRAY (10) OF ob_Individu_Ty;
```

Ce qui donnera donc pour le type obfilmty :

```
CREATE TYPE obFilmTy AS OBJECT (
NumFilm NUMBER(5),
Titre VARCHAR2(50),
listeActeurs acteursTy,
realisateur REF obIndividuTy );
```

Mais alors attention, il ne faut pas y mettre des valeurs bruts. Il faut y mettre des **références**, vu ?

```
INSERT INTO obFilm
SELECT 43, 'LE MARI DE LA COIFFEUSE',
  acteursTy(ref(a),ref(c)),
REF(b) FROM obIndividu a,obIndividu b, obIndividu c
WHERE b.NumIndividu=1996
and a.NumIndividu=1
and c.NumIndividu=1254;
```

Maintenant supposons que vous faites un SELECT sur la table obFilm, ce que vous allez avoir comme affichage c'est une reference (une adresse) vers le tableau que vous sélectionnée. Si vous souhaitez afficher le contenue d'un tableau, vous devez passez par une requete PL/SQL.

```
DECLARE
        stock acteursTy; // le tableau d'individu
        p REF obIndividuTy; // variable qui va contenir la reference vers un
            acteur
        reali obIndividuTy;
        nom varchar2(30);
        CURSOR c IS
                SELECT numfilm, titre, DEREF(realisateur) r, acteurs
                FROM obFilm;
BEGIN
        FOR s IN c LOOP
                stock:=s.acteurs; // on recupere la liste d'acteurs
                i:=1;
                LOOP
                        p:=stock(i);
                        SELECT a.nomindividu INTO nom FROM obIndividu a
                        WHERE REF(a)=p;
                        DBMS OUTPUT.PUT LINE(TO CHAR(s.numfilm) || ' ** ' || s
                            .titre || ' ** ' || s.r.nomindividu || ' ** ' ||
                            nom);
                        i:=i+1;
                        EXIT WHEN i>=stock.count;
                END LOOP;
        END LOOP;
END;
```

Voilà vous savez tout sur cette partie.