Base de Data

JP Momo & Brunoir

13 décembre 2014

Chapitre 1

Fonctions et procédures stockées

1.1 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 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;
```

1.2 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 2

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.

2.1 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, INSEERT, 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.

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

Problème possible lié au verrou

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 :

```
1  -- l'utilisateur 1 change le nom du film (numero 15 et realisateur 58)
2  UPDATE ens2004.film
3  SET titre = 'TD32'
4  WHERE numFilm = 15
5  AND realisateur = 58;
6
7  -- l'utilisateur 2 change le nom d'un autre film (numero 32 et realisateur 96)
8  UPDATE ens2004.film
9  SET titre = 'BOGOSS'
10 WHERE numFilm = 32
11 AND realisateur = 96;
12
13 -- l'utilisateur 1 change le nom du film qui est en train d'etre modifie par l
       'utilisateur 2 (numero 32 et realisateur 96)
14 UPDATE ens2004.film
15 SET titre = 'TD321'
16 WHERE numFilm = 32
17 AND realisateur = 96;
18
19 -- l'utilisateur 2 change le nom du film qui est en train d'etre modifie par l
       'utilisateur 1 (numero 15 et realisateur 58)
20 UPDATE ens2004.film
21 SET titre = 'BOGOSS1'
22 WHERE numFilm = 15
23 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.