Duquennoy Antoine
Duverney Thomas

Compte Rendu de Base de donnée « Application zoo »

I - Fonctionnalités et transactions

Les réponses à la « question 1 » sont présentent dans le programme « squellette_appli.java ». Chacune des réponses a été implémenté séparement dans les fonctions suivantes. Le but du TP étant de comprendre comment utiliser une base de donnée à travers un programme java, nous n'avons pas réalisé de vérification pousser sur les données entrées par l'utilisateur afin de garder une cohérence sur la pertinence des données dans la base.

La fonction « updateLesCages » permet de changer la fonction d'une cage. Elle fait appelle à la fonction « AfficherCages » qui liste les cages grâce à une requête SELECT. Après avoir afficher la liste des cages, elle demande à l'utilisateur de saisir la cage et de saisir la nouvelle fonction. Ensuite elle fait appel à une requête UPDATE prennant en compte les informations saisies précédement.

La fonction « ajouterAnimal » permet d'insérer un nouvel animal. Les informations sur l'animal sont saisies par l'utilisateur puis une requête INSERT est effectuée sur la table animal.

La fonction « deplacerAnimalCage » permet de déplacer un animal dans une autre cage. Le nom de l'animal est séléctionné par l'utilisateur. Ensuite le programme affiche les cages compatibles et disponibles dans lequelles l'animal pourra être déplacé. L'utilisateur choisi le numéro de cage parmi cette liste de cage puis l'animal est déplacé. Ici on utilise une requête UPDATE pour mettre à jour le numéro de cage de l'animal.

Pour répondre à la « question 2 », dans chacune des transactions précédentes, nous avons choisi le niveau d'isolation SERIALIZABLE pour plusieurs raisons. Dans ce niveau d'isolation, aucune autre transaction peut modifier les données qui ont été lu par la transaction courante jusqu'à qu'elle soit terminée. De plus, la transaction courante ne peut pas lire les données qui ont été modfié mais pas encore commité par d'autre transactions.
Ce mode d'isolation était nécessaire pour les transactions précédente. En effet si une transaction T1 souhaite mettre à jour la fonction d'une cage mais qu'au même moment une transaction T2 supprime cette cage alors la transaction T1 ne pourra aboutir. Voir ci dessous le schéma d'exécution des transactions T1 et T2.

T1	T2
Begin	Begin
Lecture table LesCages	Lecture table LesCages
	Suppression noCage = 2
Mise à jour fonction de la cage 2	Commit

Grâce au tableau, on comprend bien le problème pouvant survenir et pour le corriger il faut utiliser le niveau d'isolation sérialisable.
Voici un autre exemple justifiant l'utilisation du niveau d'isolation sérialisable, on met en concurrence les transactions de la question 1 et de la question 2 .

T1	T2
Begin	Begin
Lecture des cages compatibles avec l'animal (Lion)	
	Mise à jour fonction de la cage 2 (Fauve → aquarium)
	Commit
Mise à jour de la cage du lion, cage1 → cage2	
Commit	

Ici on est victime d'une lecture fantôme, T1 travaille sur des données qui ont été modifié par T2. De plus cette lecture fantôme provoque une violation des contraites d'intégritées, la fonction de la cage numéro 2 n'est pas compatible avec le type de l'animal.

II – Gestion des contraintes

Pour gérer les contraintes d'une base de données, il faut faire appel aux triggers. Ci-dessous le code des triggers répondant à la « question 3 ».

Question 1

```
CREATE OR REPLACE TRIGGER trigger_nb_maladie
BEFORE INSERT OR DELETE ON LesMaladies
FOR EACH ROW
BEGIN
IF INSERTING THEN
UPDATE LesAnimaux SET nb_maladies = nb_maladies + 1 WHERE nomA = :NEW.nomA;
END IF;
IF DELETING THEN
UPDATE LesAnimaux SET nb_maladies = nb_maladies - 1 WHERE nomA = :old.nomA;
END IF;
END;
/
```

Question 2

```
CREATE OR REPLACE TRIGGER trigger_cage_fonction
BEFORE INSERT OR UPDATE ON lesAnimaux
FOR EACH ROW
DECLARE
func varchar2(20);
BEGIN
SELECT fonction INTO func from lesCages where noCage = :new.noCage;
IF func!=:new.fonction_cage THEN
raise_application_error(-20001, 'ERREUR: Vous ne pouvez pas insérer un animal dans cette cage : fonction incompatible');
END IF;
END;
/
```

```
CREATE OR REPLACE TRIGGER
trigger_cage_non_gardee
BEFORE INSERT OR UPDATE ON lesAnimaux
FOR EACH ROW
DECLARE
nbGardien NUMBER;
BEGIN
SELECT count(*) INTO nbGardien from LesGardiens where noCage = :new.noCage;
IF nbGardien<1 THEN
raise_application_error(-20001, 'ERREUR: impossible d'insérer, il n'y a pas de gardien pour cette cage ');
END IF;
END;
/
```

III – Les tests

Pour vérifier le fonctionnement des triggers précédents, nous avons réalisé un jeu de test par trigger, ce sont les fichiers « test_trigger_nbMaladie »,« test_trigger_cage_fonction.sql », « test_trigger_noGarde.sql ». De plus pour plus de simplicité

Pour vérifier le focntionnement des triggers précédents, nous avons réalisé un jeu de test par trigger, ce sont les fichiers suivant présent dans l'archive «test_trigger_nbMaladie.sql » « test_trigger_cage_fonction.sql », « test_trigger_noGarde.sql » . De plus pour pouvoir créer les triggers plus facilement, nous avons ajouté un fichier « triggers.sql ».Les tests sont réalisés avec des données ajoutés pour chacun, ainsi vous pouvez lancer les tests dans l'ordre que vous le souhaitez.

Pour lancer les tests, il faut réalisé cette enchainement de commandes :

- start script.sql
- start triggers.sql
- start test_trigger_nbMaladie.sql
- start test_trigger_cage_fonction.sql
- start test_trigger_noGarde.sql