Base de Données : Compte-rendu du TP1

Line POUVARET, Mickaël TURNEL

2015-2016

Partie 1 : Atomicité

Exercice 1 : Annulation de transaction

1) SQL> **INSERT INTO** Comptes **VALUES**(1, 'Paul', 200);
1 row created.
SQL> **INSERT INTO** Comptes **VALUES**(2, 'Paul', 0);
1 row created.

2) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Paul	200
2	Paul	0

3) SQL> **ROLLBACK**;

4) SQL> **SELECT** * **FROM** Comptes;
no rows selected.

On constate que le rollback a annulé les requêtes d'insertion dans la base car nous n'avons pas effectué de commit.

Exercice 2 : Validation de transaction

1) SQL> **INSERT INTO** Comptes **VALUES**(1, 'Pierre', 200);
1 row created.
SQL> **INSERT INTO** Comptes **VALUES**(1, 'Pierre', 0);
1 row created.

2) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
1	Pierre	0

3) SQL> **COMMIT**;
Commit complete.

4) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
1	Pierre	0

5) SQL> **ROLLBACK**;
Rollback complete.

6) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
1	Pierre	0

On constate que le fait d'avoir effectué un commit après l'insertion nous a validé les changements sur la base. Donc un rollback ne permet pas de revenir avant l'insertion.

Exercice 4 : Points de sauvegarde

1) SQL> **INSERT INTO** Comptes **VALUES**(3, 'Paul', 40000);
1 row created.

2) SQL> SAVEPOINT UnInsert;
Savepoint created.

3) SQL> **INSERT INTO** Comptes **VALUES**(4, 'Paul', 6743647);
1 row created.

4) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
1	Pierre	0
3	Paul	40000
4	Paul	6743647

5) SQL> **ROLLBACK** TO UnInsert;
Rollback complete.

6) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
1	Pierre	0
3	Paul	40000

7) SQL> **ROLLBACK**;
Rollback complete.

8) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
1	Pierre	0

Le rollback vers le point de sauvegarde UnInsert ramène la base de données à l'état dans lequel elle était au moment du point de sauvegarde. Le rollback que nous effectuons après annule toutes les requêtes car nous ne les avons pas commit.

Conclusion sur la gestion de l'atomicité

La gestion des transactions par Oracle assure bien la propriété d'atomicité c'est à dire que lors de l'exécution d'une transaction, toutes ses actions sont effectuées ou bien aucune ne l'est. En effet, toutes les opérations englobées dans une seule transaction sont considérées comme une opération unique par le SGBD.

Partie 2 : Cohérence

Exercice 1

1) SQL> **INSERT INTO** Comptes **VALUES**(3, 'Claude', 100);
   1 row created.

2) SQL> **INSERT INTO** Comptes **VALUES**(4, 'Henri', 200);
   1 row created.

3) SQL> **COMMIT**;
   Commit complete.

4) SQL> **SET CONSTRAINT** SoldePositif **IMMEDIATE**;
   Constraint set.

5) SQL> **UPDATE** Comptes **SET** solde = solde + 50 **WHERE** nom='Henri';
   1 row updated.

6) SQL> **UPDATE** Comptes **SET** solde = solde − 150 **WHERE** nom='Claude';
   UPDATE Comptes **SET** solde = solde − 150 **WHERE** nom='Claude'
   *
   ERROR **at** line 1:
   ORA−02290: **check constraint** (POUVAREL.SOLDEPOSITIF) violated

   On voit qu'en décrémentant le solde de Claude, on aurait du obtenir un solde de -50 mais la contrainte SoldePositif est violée.

7) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
1	Pierre	0
3	Claude	100
4	Henri	250

   On constate que le solde de Claude n'a pas été modifié mais celui de Henri l'a bien été.

8) SQL> **ROLLBACK**;
   Rollback complete.

9) SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
1	Pierre	0
3	Claude	100
4	Henri	200

   On a annulé toutes les requêtes après le commit. Donc la modification des soldes n'a pas été prise en compte.

- **contrainte en mode IMMEDIATE (étape 4) et validation à la fin (étape 8)** : on constate que l'opération sur la décrémentation de Claude n'a pas été prise en compte.
- **contrainte en mode DEFERRED (étape 4) et annulation à la fin (étape 8)** : on constate que l'opération sur la décrémentation du solde de Claude n'affiche pas d'erreur au moment de la requête. Du moment qu'on ne commit pas, il n'affichera pas d'erreur. Le rollback nous ramène avant les incrémentations et décrémentations.
- **contrainte en mode DEFERRED (étape 4) et validation à la fin (étape 8)** : l'opération sur la décrémentation du solde de Claude provoque une erreur au moment du commit et annule toute la transaction (force un rollback). (Donc les incrémentations et décrémentations).

Conclusion sur la gestion des contraintes de cohérence

Pour assurer la cohérence de la base, il existe des contraintes d'intégrité qui doivent respecter les données de la base. Ces contraintes sont toujours vraies sur une partie ou bien la totalité des données. Si le schéma de la base de données respecte bien ces contraintes alors la base est cohérente.

Partie 3 : Isolation

Exercice 1 : Niveau d'isolation READ COMMITTED (niveau par défaut)

1) S1

```
SQL> SELECT * FROM Comptes;
```

NC	NOM	SOLDE
1	Pierre	200
2	Pierre	0
3	Claude	100
4	Henri	200

2) S2

```
SQL> SELECT * FROM Comptes;
```

NC	NOM	SOLDE
1	Pierre	200
2	Pierre	0
3	Claude	100
4	Henri	200

3) S1

```
SQL> UPDATE Comptes SET solde = solde + 1 WHERE nc=1;
1 row updated.
```

4) S1

```
SQL> SELECT * FROM Comptes;
```

NC	NOM	SOLDE
1	Pierre	1200
2	Pierre	0
3	Claude	100
4	Henri	200

5) S2

SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	200
2	Pierre	0
3	Claude	100
4	Henri	200

Au moment où S2 demande l'affichage de la table Comptes, la valeur du solde de Pierre n'a pas été modifié (par S1).

6) S1

SQL> **COMMIT**;
Commit complete.

7) S2

SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	1200
2	Pierre	0
3	Claude	100
4	Henri	200

Quand S2 demande l'affichage après le commit de S1, on constate que le solde de Pierre a bien été modifié donc la transaction de S1 a été validé.

Exercice 2 : Niveau d'isolation SERIALIZABLE

1) S1

SQL> **DELETE FROM** Comptes **WHERE** Nom='Paul';
0 **rows** deleted.

2) S1

SQL> **COMMIT**;
Commit complete.

3) S1

SQL> **INSERT INTO** Comptes **VALUES**(5, 'Paul', 200);
1 row created.

4) S2

SQL> **SET TRANSACTION ISOLATION LEVEL** serializable;
Transaction set.

5) S2

```
SQL> SELECT * FROM Comptes;
```

NC	NOM	SOLDE
1	Pierre	1200
2	Pierre	0
3	Claude	100
4	Henri	200

6) S1

```
SQL> COMMIT;
Commit complete.
```

7) S2

```
SQL> SELECT * FROM Comptes;
```

NC	NOM	SOLDE
1	Pierre	1200
2	Pierre	0
3	Claude	100
4	Henri	200

8) S2

```
SQL> COMMIT;
Commit complete.
```

9) S2

```
SQL> SELECT * FROM Comptes;
```

NC	NOM	SOLDE
1	Pierre	1200
2	Pierre	0
3	Claude	100
4	Henri	200
5	Paul	200

Même après la validation de la transaction de S1, les requêtes d'affichage de S2 ne voient pas les modifications du contenu de la table. Après le commit de S2, l'isolation est terminée et on peut voir les modifications apportées par S1 dans la table.

Avec READ COMMITTED la deuxième transaction ne voit les changements effectués par la première dès que celle-ci a commit. Avec SERIALIZABLE, même si la première transaction a effectué son commit, la deuxième transaction ne peut voir les changements effectués tant que celle-ci n'a pas été finie.

Exercice 3 : Verrouillage

1) S1

```
SQL> DELETE FROM Comptes WHERE Nom='Pierre';
2 rows deleted.
```

2) S2

SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
1	Pierre	1200
2	Pierre	0
3	Claude	100
4	Henri	200
5	Paul	200

3) S2

SQL> **UPDATE** Comptes **SET** Nom='Pierre';
...(attente)

La requête de la transaction S2 reste en attente de la validation de la transaction de S1 car les lignes avec le nom 'Pierre' sont verrouillés par S1.

4) S1

SQL> **COMMIT**;
Commit complete.

S2

3 **rows** updated.

La validation de S1 permet les modifications de S2 et la transaction S2 se retrouve débloquée.

5) S2

SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
3	Pierre	100
4	Pierre	200
5	Pierre	200

La transaction de S2 affiche toutes les lignes avec comme nom Pierre sauf les deux supprimées par la transaction S1 puisqu'elles ne sont plus dans la table.

En effet, S1 ayant validé sa transaction les anciens tuples avec comme nom Pierre ont été bel et bien supprimés.

6) S2

SQL> **COMMIT**;
Commit complete.

7) S1

SQL> **SELECT** * **FROM** Comptes;

NC	NOM	SOLDE
3	Pierre	100
4	Pierre	200
5	Pierre	200

Après validation de la transaction S2, la transaction S1 affiche bien les noms des comptes qui ont été changé par Pierre.

Le SGBD gère les modifications concurrentes sur les mêmes données grâce à des verrous sur ces données. En effet, si une transaction modifie une donnée de la base et une autre transaction désire modifier la même donnée, cette dernière devra attendre que la première libère le verrou sur cette donnée en validant sa transaction.

Exercice 4 : Interblocage

S1

```
SQL> DELETE FROM Comptes WHERE nc=4;
1 row deleted.

SQL> UPDATE comptes SET solde = solde + 100;
...(attente)
```

S2

```
SQL> UPDATE Comptes SET nom='Toto';
UPDATE Comptes SET nom='Toto'
...(attente)
       *
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
```

S2 a été tué par le SGBD et permet donc la dernière requête de S1.

Oracle détecte quelle transaction génère un interblocage (celle qui engendre le cycle dans le graphe d'attente) et va terminer cette transaction (en la "tuant") libérant ainsi les autres transactions bloquées.