



INF3710 –Fichiers et Bases de données

Hiver 2019

TP No. 4

Groupe 4

1340155 – Alassane Maiga

1862313 – Mohamed Esseddik BENYAHIA

Soumis à : Manel Grinchi.

24 Mars 2019

1 Partie 2 - Algèbre relationnelle et SQL

1.1 1.1

1. Retrouver les détails de tous les spectacles en 2010

$$\sigma_{\text{anne}=2010}(\text{Spectacle})$$

2. Retrouver le détail de tous les danseurs qui ne sont pas dans la vingtaine

$$\sigma_{\text{age} > 29 \vee \text{age} < 20}(\text{Danseur})$$

3. Retrouver le nom de tous les directeurs artistiques Canadiens nom (nationalite = Canadien (DirecteurArtistique))

$$\Pi_{\text{nom}}(\sigma_{\text{nationalite}=\text{Canada}}(\text{DirecteurArtistique}))$$

4. Retrouver le nom de chaque danseur ainsi que les titres des Spectacles dans lesquels il/elle s'est produit nom, titre (Danseur \bowtie Performance \bowtie Spectacle)

$$\Pi_{\text{nom, titre}}(\text{Danseur} \bowtie \text{Performance} \bowtie \text{Spectacle})$$

5. Trouver les noms de tous les danseurs qui ont dansé le rôle du 'cygne' ainsi que l'année du spectacle

$$\Pi_{\text{nom, annee}}(\text{Danseur} \bowtie (\sigma_{\text{role}=\text{'cygne'}}(\text{Performance})) \bowtie \text{Spectacle})$$

6. Retrouver toutes les informations des danseurs du Spectacle 'Opus Cactus' sans opération non nécessaire (indice : vous ne pouvez pas utiliser uniquement un join)

$$\text{OpusC} \leftarrow \text{Danseur} \bowtie \text{Performance} \bowtie \sigma_{\text{titre}=\text{'Opus Cactus'}}(\text{Spectable})$$

$$\text{R6} \leftarrow \Pi_{\text{DanseurId, nom, nationalite, age}}(\text{OpusC})$$

7. Retrouver les titres de tous les spectacles dans lesquels les danseurs Philippe et Kate ont dansé ensemble

$$\text{S_Phil} \leftarrow \sigma_{\text{nom}=\text{'Philippe'}}(\text{Danseur}) \bowtie \text{Performance} \bowtie \text{Spectable}$$

$$\text{S_Kat} \leftarrow \sigma_{\text{nom}=\text{'Kate'}}(\text{Danseur}) \bowtie \text{Performance} \bowtie \text{Spectable}$$

$$\text{R7} \leftarrow \Pi_{\text{titre}}(\text{S_Phil} \bowtie_{\text{S_Phil.SpectacleId} = \text{S_Kat.SpectacleId}} \text{S_Kat})$$

1.2 1.2

1. Quel est l'âge moyen des danseurs ? Stockez-le dans une colonne nommée AgeMoyen.

Algèbre :

$$\rho_R(\text{AgeMoyen}) \mathfrak{J}_{\text{AVG}(\text{age})}(\text{Danseur})$$

SQL :

```
1 SELECT AVG(age) AS AgeMoyen FROM Danseur;
```

2. Quels danseurs (Nom) ont dansé dans au moins un spectacle où la danseuse Lucie Tremblay n'a pas dansé ?

Algèbre :

$$\begin{aligned} S_Luci &\leftarrow \sigma_{\text{nom}='Lucie Tremblay'}(\text{Danseur}) \bowtie \text{Performance} \\ S_nLuci &\leftarrow \Pi_{\text{spectacleid}'}(\text{Performance}) - \Pi_{\text{spectacleid}'}(S_Luci) \\ R9 &\leftarrow \Pi_{\text{nom}}(S_nLuci \bowtie \text{Danseur}) \end{aligned}$$

SQL :

```
1 SELECT nom FROM Danseur WHERE DanseurId IN
2   (SELECT DanseurId FROM Performance WHERE SpectacleId NOT IN
3     (SELECT SpectacleId FROM Performance p, Danseur d
4       WHERE nom = 'Lucie Tremblay' AND p.DanseurId = d.DanseurId));
```

3. Quel est le nombre de spectacles du danseur dont l'id = 1 ? Stockez le résultat dans une colonne nommée nbSpectacle.

Algèbre :

$$\rho_R(\text{nbSpectacle}) \mathfrak{J}_{\text{COUNT}(\text{SpectacleId})}(\sigma_{\text{DanseurId}=1}(\text{Performance}))$$

SQL :

```
1 SELECT COUNT(*) AS nbSpectacle FROM Performance WHERE DanseurId = 1;
```

4. Affichez une liste des danseurs ainsi que les spectacles (ID) qui leur sont associés s'ils existent, sinon affichez null. L'attribut en commun ne doit pas être répété.

Algèbre :

$$\text{Danseur} \bowtie \Pi_{\text{DanseurId}, \text{SpectacleId}}(\text{Performance})$$

SQL :

```
1 SELECT d.*, p.SpectacleId
2 FROM Danseur d NATURAL LEFT OUTER JOIN Performance p;
```

5. Combien de spectacles existent par catégorie? Stockez le résultat en donnant un nom à la ou les colonnes correspondantes de la relation résultat.

Algèbre :

$$\rho_R(\text{Catégorie}, \text{nbSpectacle})_{\text{catégorie}} \mathfrak{J}_{\text{COUNT}(\text{spectacleid})}(\text{Spectacle})$$

SQL :

```
1 SELECT categorie, COUNT(*) AS nbSpectacle
2 FROM Spectacle GROUP BY categorie;
```

6. Quels danseurs (affichez leurs détails) n'ont participé à aucun spectacle

Algèbre :

$$(\Pi_{\text{danseurid}}(\text{Danseur}) - \Pi_{\text{danseurid}}(\text{Performance})) \bowtie \text{Danseur}$$

SQL :

```
1 SELECT * FROM Danseur
2 WHERE DanseurID NOT IN
3 (SELECT DanseurID FROM Performance);
```

2 Partie 3 - Transactions

1. .

- (a) Que se passe-t-il quand vous exécutez ces deux transactions concurrentes? Quel est le problème?

Ici, le problème est que la transaction A est complètement ignorée. Seule la valeur de la transaction B est prise en compte à la fin. En effet, la transaction A retire 200\$ du compte et la transaction b retire 500\$. On s'attendrait donc que le résultat final soit une réduction de 700\$ du compte. Mais on voit qu'on a juste le retrait de 500 de la transaction B dans l'état final. Ceci est dû au fait que la mise à jour du compte par la transaction B s'est faite sur la valeur lue précédemment par la transaction B. Une valeur qui avait été lue avant que la transaction A n'ait validé sa transaction. Ainsi, lorsque la transaction A écrit sur la table et valide, la valeur se fait écrasée par la transaction B.

```

TP4=# \set AUTOCOMMIT off
TP4=# begin transaction isolation level READ COMMITTED;
BEGIN
TP4=# SELECT balance -200 as bal
TP4=# into balancea
TP4=# FROM Accounts WHERE acctID = 101;
SELECT 1
TP4=# SELECT bal FROM balancea;
bal
-----
800
(1 row)

TP4=# UPDATE Accounts
TP4=# SET balance = (select bal from balancea)
TP4=# WHERE acctID = 101;
UPDATE 1
TP4=# SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctid | balance
-----+-----
101 | 800
(1 row)

TP4=# COMMIT;
TP4=# SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctid | balance
-----+-----
101 | 500
(1 row)

```

Fig.1 : Transaction A : Elle commence et termine avant la transaction B

```

TP4=# \set AUTOCOMMIT off
TP4=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
TP4=# SELECT balance - 500 as bal into balanceb
TP4=# FROM Accounts WHERE acctID = 101;
SELECT 1
TP4=# SELECT bal from balancea;
bal
-----
500
(1 row)

TP4=# UPDATE Accounts
TP4=# SET balance = (select bal from balanceb)
TP4=# WHERE acctID = 101;
UPDATE 1
TP4=# SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctid | balance
-----+-----
101 | 500
(1 row)

TP4=# COMMIT;
TP4=# SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctid | balance
-----+-----
101 | 500
(1 row)

```

Fig.2 : Transaction B : lorsqu'elle valide, elle écrase la valeur du compte de la transaction A

(b) Comment pourrions-nous nous assurer que les résultats soient cohérents ?

Pour éviter le problème, la transaction B doit attendre que la transaction A écrit la valeur sur le compte avant que la transaction B la lise. On doit donc verrouiller le tuple que la transaction A mettra à jour. On peut le faire avec un `SELECT ... FOR UPDATE`. Ainsi la transaction B ne lira pas tout de suite la valeur du compte et attendra la fin de la transaction A avant de le faire.

```

TP4=# \set AUTOCOMMIT off
TP4=# begin transaction isolation level READ COMMITTED;
BEGIN
TP4=# SELECT balance -200 as bal
TP4=# into balancea
TP4=# FROM Accounts WHERE acctID = 101 FOR UPDATE;
SELECT 1
TP4=# SELECT bal FROM balancea;
bal
-----
800
(1 row)

TP4=# UPDATE Accounts
TP4=# SET balance = (select bal from balancea)
TP4=# WHERE acctID = 101;
UPDATE 1
TP4=# SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctid | balance
-----+-----
101 | 800
(1 row)

TP4=# COMMIT;

```

Fig.3 : Transaction A : Elle commence et termine avant la transaction B

```

TP4=# \set AUTOCOMMIT off
TP4=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
TP4=# SELECT balance - 500 as bal into balanceb
TP4=# FROM Accounts WHERE acctID = 101 FOR UPDATE;
UPDATE Accounts
SET balance = (select bal from balancea)
WHERE acctID = 101;
SELECT 1
TP4=# SELECT bal from balancea;UPDATE Accounts
bal
-----
300
(1 row)

TP4=# SET balance = (select bal from balanceb)
TP4=# WHERE acctID = 101;
UPDATE 1
TP4=# SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctid | balance
-----+-----
101 | 300
(1 row)

TP4=# COMMIT;

```

Fig.4 : Transaction B : lorsqu'elle écrit, elle utilise, comme attendu, la valeur du compte résultant de A

2. .

(a) Quel problème constatez-vous avec `READ COMMITTED` ?

Quand la transaction A utilise l'isolation `READ COMMITTED`, si la transaction B fait un commit pendant que A est encore en cours, la transaction A lit des valeurs différentes

lorsqu'elle fait des SELECT sur les tuples que B écrit et valide. En effet : le deuxième SELECT utilise la nouvelle valeur des montants dans les comptes comme écrit par B.

```
TP4=# \set AUTOCOMMIT off
TP4=# BEGIN;
BEGIN
TP4=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
TP4=# SELECT * FROM Accounts
TP4=# WHERE balance > 500;
acctid | balance
-----+-----
 101 | 1000
 202 | 2000
(2 rows)

TP4=# SELECT * FROM Accounts
TP4=# WHERE balance > 500;
acctid | balance
-----+-----
 202 | 2500
(1 row)

TP4=# COMMIT;
COMMIT
```

Fig.5 : Transaction A : La valeur du select dépend des valeurs qui ont été validée concouramment par B

```
TP4=# \set AUTOCOMMIT off
TP4=# BEGIN;
BEGIN
TP4=# UPDATE Accounts
TP4=# SET balance = balance - 500
TP4=# WHERE acctID = 101;
UPDATE 1
TP4=#
TP4=# UPDATE Accounts
TP4=# SET balance = balance + 500
TP4=# WHERE acctID = 202;
UPDATE 1
TP4=#
TP4=# SELECT * FROM Accounts;
acctid | balance
-----+-----
 101 | 500
 202 | 2500
(2 rows)

TP4=# COMMIT;
COMMIT
```

Fig.6 : Transaction B : Transfert entre les deux comptes

- (b) Que se passe-t-il si changez le niveau d'isolation de la transaction A à REPEATABLE READ ?

Avec Repeatable READ par contre, lorsque la transaction B fait un commit, la transaction A utilise encore les anciennes valeurs de Accounts : celle qui ont été validées avant le début de la transaction A.

```
TP4=# \set AUTOCOMMIT off
TP4=# BEGIN;
BEGIN
TP4=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
TP4=# SELECT * FROM Accounts
TP4=# WHERE balance > 500;
acctid | balance
-----+-----
 101 | 1000
 202 | 2000
(2 rows)

TP4=# SELECT * FROM Accounts
TP4=# WHERE balance > 500;
acctid | balance
-----+-----
 101 | 1000
 202 | 2000
(2 rows)

TP4=# COMMIT;
COMMIT
```

Fig.7 : Transaction A : elle lit toujours les mêmes valeurs sur les même tuples

```
TP4=# \set AUTOCOMMIT off
TP4=# BEGIN;
BEGIN
TP4=# UPDATE Accounts
TP4=# SET balance = balance - 500
TP4=# WHERE acctID = 101;
UPDATE 1
TP4=#
TP4=# UPDATE Accounts
TP4=# SET balance = balance + 500
TP4=# WHERE acctID = 202;
UPDATE 1
TP4=#
TP4=# SELECT * FROM Accounts;
acctid | balance
-----+-----
 101 | 500
 202 | 2500
(2 rows)

TP4=# COMMIT;
COMMIT
```

Fig.8 : Transaction B : transfert entre les deux comptes

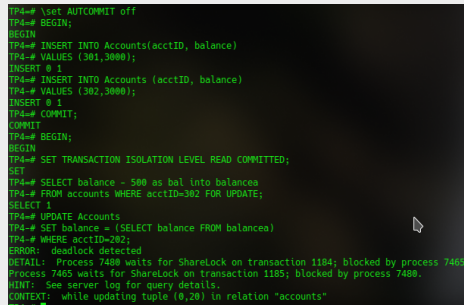
- (c) Quel problème constatez-vous avec REPEATABLE READ READ ONLY ?

Similairement lorsqu'on utilise l'isolation REPEATABLE READ en mode Lecture seule, La transaction A voit les valeurs de la table qui ont été validée avant que la transaction commence. Donc quand la transaction B valide sa transaction, ceci est toujours invisible

à la transaction A. Le mode “read only” n’affecte que la transaction A, l’empêchant de modifier les tables. La transaction B peut encore changer les données des tables.

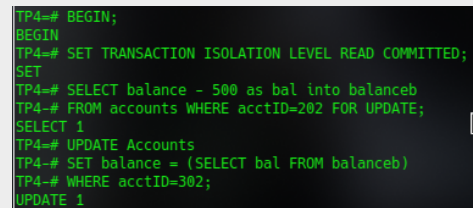
3. Interblocage :

Nous voyons dans les figures qui suivent une situation d’interblocage. Le système abandonne la transaction A pour débloquer les transactions.



```
TP4=# \set AUTOCOMMIT off
TP4=# BEGIN;
BEGIN
TP4=# INSERT INTO Accounts(acctID, balance)
TP4=# VALUES (301,3000);
INSERT 0 1
TP4=# INSERT INTO Accounts (acctID, balance)
TP4=# VALUES (302,3000);
INSERT 0 1
TP4=# COMMIT;
COMMIT
TP4=# BEGIN;
BEGIN
TP4=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
TP4=# SELECT balance - 500 as bal into balancea
TP4=# FROM accounts WHERE acctID=302 FOR UPDATE;
SELECT 1
TP4=# UPDATE Accounts
TP4=# SET balance = (SELECT balance FROM balancea)
TP4=# WHERE acctID=202;
UPDATE 1
ERROR: deadlock detected
DETAIL: Process 7480 waits for ShareLock on transaction 1184; blocked by process 7480.
Process 7465 waits for ShareLock on transaction 1185; blocked by process 7480.
HINT: See server log for query details.
CONTEXT: while updating tuple (0,20) in relation "accounts"
```

Fig.9 : Transaction A : Détection de l’interblocage



```
TP4=# BEGIN;
BEGIN
TP4=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
TP4=# SELECT balance - 500 as bal into balanceb
TP4=# FROM accounts WHERE acctID=202 FOR UPDATE;
SELECT 1
TP4=# UPDATE Accounts
TP4=# SET balance = (SELECT bal FROM balanceb)
TP4=# WHERE acctID=302;
UPDATE 1
ERROR: deadlock detected
DETAIL: Process 7480 waits for ShareLock on transaction 1184; blocked by process 7480.
Process 7465 waits for ShareLock on transaction 1185; blocked by process 7480.
HINT: See server log for query details.
CONTEXT: while updating tuple (0,20) in relation "accounts"
```

Fig.10 : Transaction B : la dernière mise à jour cause une situation d’interblocage

```
1  -- Transaction A
2  \set AUTOCOMMIT off BEGIN;
3  INSERT INTO Accounts(acctID, balance)
4  VALUES (301,3000);
5  INSERT INTO Accounts (acctID, balance)
6  VALUES (302,3000);
7  COMMIT;
8  -- Transaction B
9  \set AUTOCOMMIT off
10 BEGIN;
11 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
12 SELECT balance - 500 as bal into balanceb
13 FROM accounts WHERE acctID=202 FOR UPDATE;
14 -- Transaction A
15 BEGIN;
16 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
17 SELECT balance - 500 as bal into balancea
18 FROM accounts WHERE acctID=302 FOR UPDATE;
19 -- Transaction B
20 UPDATE Accounts
21 SET balance = (SELECT bal FROM balanceb)
22 WHERE acctID=302;
23 -- Transaction A
24 UPDATE Accounts
25 SET balance = (SELECT balance FROM balancea)
26 WHERE acctID=202;
27 -- INTERBLOCAGE DETECTE
```