TP 2 - Initiation aux transactions

L'objectif de ce TP est de se familiariser avec les commandes de Postgres relatives aux transactions, pour l'instant sur des bases de données de taille très réduite. Nous allons observer le comportement normal des transactions et les erreurs possibles liées aux interactions concurrentes de plusieurs transactions.

Dans **tous** les exercices, **avant** de tester les requêtes, essayez de prédire quel va être le résultat de chaque `SELECT` et quel sera l'état final de la base de données.

▸ Exercice 1 : Prise en main avec un seul client

Créez une petite table de travail `test(id,a,b)` permettant de stocker des paires d'entiers sans aucune contrainte. L'attribut `id` sera la clé primaire de la table et de type `serial`. Nous laisserons le système incrémenter automatiquement cet attribut.

Sans transactions pour l'instant :

1. Insérez une première paire d'entiers dans la table, et observez la valeur choisie par le système pour la clé primaire. *Rappel : le mot-clé* `returning` *permet de récupérer les attributs voulus après l'insertion, comme ci-dessous.*

```
INSERT INTO test(a,b) VALUES (1,2)
RETURNING id;
```

2. Insérez maintenant une deuxième paire d'entiers en réutilisant le même identifiant que pour la première ligne. Quelle erreur observez-vous ?

Observons à présent le comportement des transactions :

3. Initiez une transaction, et insérez quelques valeurs dans la table.

```
BEGIN;
INSERT INTO test(a,b) VALUES (1,3);
INSERT INTO test(a,b) VALUES (1,4);
INSERT INTO test(a,b) VALUES (1,5);
```

Inspectez le contenu de la table.

```
SELECT * FROM test;
```

Et terminez votre transaction par `ROLLBACK`. Quelle propriété des transactions observez-vous ?

```
ROLLBACK;
SELECT * FROM test;
```

4. Répétez les mêmes opérations, mais en simulant cette fois une panne, c'est-à-dire en fermant votre terminal avant la fin de la transaction. Relancez votre terminal et reconnectez vous à la base de données. Qu'observez-vous ?

5. Répétez les mêmes opérations, mais en terminant cette fois avec `COMMIT`.

6. Essayez à présent de produire une erreur lors d'une transaction. Par exemple, essayez d'insérer une valeur produisant une erreur de clef primaire.

```
BEGIN;
INSERT INTO test(a,b) VALUES (2,5);
INSERT INTO test VALUES (x,y,z);
-- choisissez x, y et z pour produire une erreur de clé primaire.
```

7. Essayez de taper une autre requête (par exemple une requête `SELECT`), et observez le comportement du système, puis terminez la transaction par `ROLLBACK`.

8. Rejouez le scénario précédent en terminant cette fois par `COMMIT`. Qu'observez-vous ?

9. Testez enfin le comportement des transactions lors d'une simple faute de frappe.

▸ Exercice 2 : Premiers tests concurrents

Utilisez le script `fundme.sql` pour recréer la base de données utilisée dans les exemples du cours sur les niveaux d'isolation. Nous allons essayer d'observer les différentes erreurs de sérialisation prédites par le standard SQL. Nous allons donc simuler la présence de deux clients accédant de manière concurrente à la base de données. Pour cela, vous devrez ouvrir deux terminaux, et effectuer deux connexions simultanées à la base. Chaque terminal représentera un client.

1. Quel est le niveau d'isolation par défaut de Postgres, lorsqu'une transaction est simplement commencée par `BEGIN;` ?

Pour initier une transaction avec le niveau d'isolation voulu :

```
BEGIN TRANSACTION ISOLATION LEVEL niveau;
-- où niveau est le niveau d'isolation voulu
```

2. Rejouez les scénarios d'erreurs du cours dans les différents niveaux d'isolation. Observez bien ce que voit chaque client à chaque étape.

3. Dans quels cas le comportement observé diffère-t-il du comportement attendu ? Dans quels cas Postgres est-il plus strict que le standard SQL ? Moins strict ?

4. Essayez le scénario d'anomalie de sérialisation (du cas `REPEATABLE READ`) dans le niveau d'isolation `SERIALIZABLE`. Qu'observez-vous ?

▸ Exercice 3 : Transactions et triggers

Le but de cet exercice est d'observer l'interaction des triggers et des transactions. Pour cela, nous allons reprendre la table `test(id,a,b)` de l'exercice 1.

1. Créez un trigger simple déclenché avant l'insertion d'une nouvelle ligne dans la table `test` et permettant de tester tous les cas de figure. Par exemple, votre trigger :
   - fera un **`RAISE NOTICE`** lorsque `a` est pair ;
   - annulera l'insertion par **`RETURN NULL`** lorsque `b` est pair ;
   - annulera l'insertion par **`RAISE EXCEPTION`** lorsque `a = b` ;

2. Dans une transaction, testez l'insertion de plusieurs lignes simultanément, dont certaines font un **`RAISE NOTICE`**. Observez l'état de la table. Essayez de terminer par un **`ROLLBACK`** puis par un **`COMMIT`**.

   Par exemple :

```
INSERT INTO test(a,b)
VALUES (1,3), (2,3), (5,9);
```

3. Dans une transaction, testez l'insertion de plusieurs lignes simultanément, dont certaines sont annulées par un **`RETURN NULL`**. Observez l'état de la table. Essayez de terminer par un **`ROLLBACK`** puis par un **`COMMIT`**.

4. Finalement, testez l'insertion de plusieurs lignes simultanément, dont certaines sont annulées par un **`RAISE EXCEPTION`**. Observez l'état du système et concluez.

▸ Exercice 4 : Transactions et opérations LDD

Certains systèmes forcent un **`COMMIT`** implicite lors d'opérations LDD (Langage de Définition de Données, c'est-à-dire création de table, modification du schéma d'une table, ajout de contrainte, etc.) C'est par exemple le cas d'Oracle, qui ne permet donc pas un **`ROLLBACK`** après ces opérations. Voyons ce qu'il en est pour Postgres.

1. Démarrez une transaction et créez une table à l'intérieur de la transaction.

   Annulez votre transaction par un **`ROLLBACK`** et observez le résultat.

   Faites de même en terminant par un **`COMMIT`** et concluez.

2. Essayez d'autres opérations LDD, par exemple essayez de renommer une colonne, de changer le type d'une colonne ou d'ajouter une contrainte.

▸ Exercice 5 : ♣ Pour les curieux...

Étudiez dans la documentation de Postgres le fonctionnement de la commande **`SAVEPOINT`**. Testez-en l'utilisation.