TP 3 - Niveaux d'isolation

L'objectif de ce TP est d'observer plus finement le comportement des différents niveaux d'isolation et de comprendre les garanties que chaque niveau apporte. Cela nous amènera notamment à découvrir certains détails d'implémentation de Postgres et à utiliser les outils mis à votre disposition pour contrôler l'état du système.

▸ Exercice 1 : Quelques interactions particulières

Le but de cet exercice est d'observer quelques comportements particuliers du système en présence de transactions concurrentes. Pour cela, nous allons utiliser une petite table de travail `test(a,b)` permettant de stocker des paires d'entiers sans contrainte.

1. Écrivez un script pour créer la table et y insérer les valeurs $(1, 1)$ et $(2, 2)$. Vous utiliserez ce script au début de chaque question pour ramener la base de données dans son état de départ.

2. Prédisez le comportement du système lors de l'exécution ci-dessous puis faites le test. Qu'observez-vous ? Comment se nomme cette erreur ? Survient-elle aussi dans le niveau d'isolation **`REPEATABLE READ`** ?

Transaction 1	Transaction 2
`BEGIN;`	`BEGIN;`
`UPDATE test SET a = 3 WHERE b = 1;`	
	`UPDATE test SET a = 4 WHERE b = 2;` `UPDATE test SET a = 4 WHERE b = 1;`
`UPDATE test SET a = 3 WHERE b = 2;`	
`COMMIT;`	`COMMIT;`

3. Prédisez le comportement du système lors de l'exécution ci-dessous puis faites le test. Que se passe-t-il dans le niveau **`REPEATABLE READ`** ? Pourquoi ?

Transaction 1	Transaction 2
`BEGIN;`	`BEGIN;`
`UPDATE test SET a = a + 1;`	
	`UPDATE test SET a = a + 1;`
`COMMIT;`	`COMMIT;`

4. Prédisez quelles lignes seront supprimées par la transaction 2. Faites le test. Êtes-vous surpris du résultat ?

   Ce comportement ne correspond pas à une des erreurs classiques dues à la concurrence. Il s'agit d'une **spécificité d'implémentation** de Postgres. Quel autre comportement raisonnable aurait-on pu imaginer ?

Transaction 1	Transaction 2
`BEGIN;`	`BEGIN;`
`UPDATE test SET a = a + 1;`	
	`DELETE FROM test WHERE a = 2;`
`COMMIT;`	`COMMIT;`

▸ Exercice 2 : Scénarios d'erreur

Un club de jeux de plateau se sert d'une base de données pour permettre à ses membres d'organiser des parties sur les différents jeux possédés par le club. Le schéma de la base est donné ci-dessous et implémenté avec quelques valeurs d'essai dans le script `boardgame.sql`.

```
partie(pid, jeu, nbmin, nbmax, etat)
joueur(jid, pseudo)
demande(jid, pid, date, statut)
```

FK : `jid` dans `demande` fait référence à `jid` dans `joueur`.
FK : `pid` dans `demande` fait référence à `pid` dans `partie`.

Dans la base de données, `jeu` représente le nom du jeu sur lequel sera joué la partie et `nbmin` et `nbmax` sont les nombres de joueurs minimum et maximum souhaités.

L'`etat` d'une partie peut être 'ouvert' (la partie accepte encore des demandes), 'fermé' (la partie n'accepte plus de nouvelles demandes et est prête à être jouée) et 'annulé' (la partie ne sera finalement pas jouée).

Le `statut` d'une demande peut être 'en attente' (la demande n'a pas encore été traitée), 'validé' (le joueur participera à la partie) ou 'refusé' (le joueur ne jouera pas).

Les actions possibles des utilisateurs sur la base de données dépendent de leur rôle.

- Les **joueurs** peuvent :
  - **Demander à rejoindre une partie** dont l'`etat` est 'ouvert'. La date de la demande devra être `now()` et le `statut` 'en attente'.
  - **Se désister d'une partie**, c'est-à-dire supprimer leur demande. Le désistement n'est autorisé que pour les parties dont l'`etat` est 'ouvert'.

— Les **organisateurs** peuvent :
  — **Créer une nouvelle partie**, dont l'`etat` sera 'ouvert'. L'attribut `nbmin` doit être inférieur ou égal à l'attribut `nbmax`.
  — **Clôturer une partie** dont l'`etat` est 'ouvert'. Le nombre de demandes pour la partie doit être supérieur ou égal à `nbmin`. L'`etat` de la partie doit alors être changé en 'fermé'. Le `statut` des `nbmax` premières demandes doit être changé en 'validé', et celui des autres demandes en 'refusé'.
  — **Annuler une partie** dont l'`etat` est 'ouvert'. L'`etat` de la partie doit être changé en 'annulé', et le `statut` de toutes les demandes encore en attente pour la partie en 'refusé'.

On se place dans le niveau d'isolation `READ COMMITTED`. Proposez des transactions qui respectent la cohérence de la base de données lorsqu'elles sont exécutées **en isolation**, mais dont les exécutions concurrentes produisent les erreurs ci-dessous. L'erreur persiste-t-elle dans le niveau d'isolation `REPEATABLE READ` ? Que se passe-t-il dans le niveau d'isolation `SERIALIZABLE` ?

1. Une partie est fermée alors qu'elle n'a pas assez de joueurs inscrits.
2. Un joueur a une demande en attente pour une partie annulée.
3. Une partie annulée a des demandes validées.
4. Une demande est refusée alors qu'elle a une date plus ancienne qu'une demande acceptée pour la même partie.
5. Une partie contient plus de demandes validées que le maximum souhaité.

▸ Exercice 3 : Verrous et estampilles

La table `pg_locks` de Postgres fournit des informations sur les verrous actuellement détenus par les différentes transactions qui opèrent sur le système. Nous allons faire quelques tests pour observer et interpréter son évolution au cours de scénarios de transactions concurrentes.

Nous allons à nouveau travailler avec la table `test(a,b)` de l'exercice 1.

1. Affichez le contenu de `pg_locks`. Qu'observez-vous ?

Nous allons nous concentrer sur un nombre réduit d'informations : `locktype` (le type de verrou), `relation` (la table concernée par le verrou), `tuple` (l'enregistrement concerné par le verrou), `virtualtransaction` (l'identifiant de la transaction qui détient le verrou) et `mode` (le niveau de contrôle du verrou).

2. Dans `pg_locks`, les tables ne sont pas référencées par leur nom, mais pas leur identifiant interne. Allez chercher dans la table `pg_class` l'identifiant (`relfilenode`) de la table dont le nom (`relname`) est `'test'`.
3. Y a-t-il un verrou actif sur la table `test` ?

4. Exécutez les transactions ci-dessous et observez dans un troisième terminal l'évolution à chaque étape des verrous sur la table `test`. Quand sont-ils pris et par quelles transactions ? Sont-ils bloquants ? Quand sont-ils relâchés ? Quels sont les enregistrements concernés ?

Transaction 1	Transaction 2
`BEGIN;`	`BEGIN;`
`SELECT * FROM test;`	
	`SELECT * FROM test;`
`UPDATE test SET a = 4 WHERE b = 1;`	
	`UPDATE test SET a = b;`
`COMMIT;`	`COMMIT;`

5. Nous allons maintenant observer l'implémentation du niveau `SERIALIZABLE` de Postgres. Comme dans la question précédente, exécutez les transactions ci-dessous et observez dans un troisième terminal l'évolution des verrous.

Transaction 1	Transaction 2
`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;`	`BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;`
`SELECT * FROM test;`	
	`SELECT * FROM test;`
`UPDATE test SET a = 4 WHERE b = 1;`	
	`INSERT INTO test VALUES (3,3);`
`COMMIT;`	`COMMIT;`

6. Quels nouveaux verrous avez-vous observé ? Sont-ils bloquants ? Quand sont-ils relâchés ?

7. Quelle semble être l'utilité de ces verrous ? À quel autre outil de contrôle de la concurrence pouvez-vous les comparer ?

8. Cherchez dans la table `pg_class` l'identifiant de la table `pg_locks`. Y a-t-il un verrou actif sur la table `pg_locks` elle-même ? Faites quelques tests pour déterminer quelle transaction détient ce verrou et pourquoi !

Note : il y a encore de très nombreux détails et niveaux de verrouillage sous Postgres dont nous n'avons pas parlé. Si vous êtes curieux, n'hésitez pas à consulter les pages *pg_locks* et *explicit locking* de la documentation.

▸ Exercice 4 : Encore des triggers !

1. Proposez une implémentation avec des triggers des actions des joueurs et des organisateurs de l'exercice 2.

2. ♣ En supposant que les joueurs et organisateurs ont uniquement le droit d'agir (en écriture) au travers de ces triggers, et donc sans faire de transations explicites, parvenez-vous à reproduire les erreurs liées à la concurrence de l'exercice 2 ?