cour 21 : les Transactions

1. Définition d'une Transaction :

- **Définition :**
  - Une transaction est un ensemble d'opérations effectuées sur une BD de manière cohérente et fiable. Une transaction est une séquence d'instructions SQL qui doit être exécutée comme une seule entité indivisible. Elle doit respecter les propriétés ACID.
  - une transacton est un ensemble d'instructions qui fait passer la BD d'un état cohérant t1 à un autre état cohérant t2 telle que entre t1 et t2 la BD peut contenir des données incohérantes .
- **Instrcuton :**

1. **Début de Transaction** :

- Pour commencer une transaction , utilisez la commande `BEGIN` :

```
BEGIN;
```

2. **Validation de Transaction (Commit)** :

- Pour valider (commit) une transaction utilisez la commande `COMMIT` :

```
COMMIT;
```

3. **Annulation de Transaction (Rollback)** :

- Pour annuler (rollback) une transaction , utilisez la commande `ROLLBACK` :

```
ROLLBACK;
```

RQ : `autocommit :`

- **`autocommit`** est une propriété qui détermine si chaque instruction SQL individuelle est automatiquement validée (commit) après son exécution ou si elle doit être explicitement validée par le programme ou l'utilisateur.
- Lorsque l'`autocommit` est activé, chaque instruction SQL, telle qu'une insertion (`INSERT`), une mise à jour (`UPDATE`), ou une suppression (`DELETE`), est automatiquement validée et les changements sont permanents dans la base de données.
- Lorsque l'`autocommit` est désactivé, les instructions SQL ne sont pas automatiquement validées après leur exécution. Cela signifie que plusieurs instructions peuvent être regroupées au sein d'une transaction, et ces changements ne deviendront permanents que si une instruction `COMMIT` est

explicitement exécutée. Si une erreur survient au cours de la transaction, vous pouvez exécuter une instruction `ROLLBACK` pour annuler les changements effectués depuis le dernier `COMMIT`.

- comment vous pouvez activer ou désactiver l'`autocommit` :
- **Activer l'autocommit :**

```
SET autocommit = 1; -- ou SET autocommit = ON;
```

- **Désactiver l'autocommit :**

```
SET autocommit = 0; -- ou SET autocommit = OFF;
```

2. Les Régles `ACID`:

1. **Atomicité (A - Atomicity)** :

- Une transaction est atomique, ce qui signifie qu'elle est tout ou rien. Toutes les opérations à l'intérieur d'une transaction doivent être exécutées avec succès pour que la transaction soit validée. Si une seule opération échoue, la transaction entière est annulée, et aucune modification n'est effectuée dans la base de données.
- Dans la pratique, l'atomicité est mise en œuvre en utilisant des techniques telles que l'utilisation d'un journal d'image et en effectuant des opérations sur une copie temporaire des données.

    1. **Journal d'Image** : Avant d'apporter des modifications à la base de données, chaque opération de transaction est enregistrée dans un journal d'image. Ce journal enregistre l'état actuel de la base de données avant la modification. Si la transaction est validée, les modifications sont appliquées à la base de données. Si la transaction échoue, les opérations sont annulées en utilisant les informations du journal pour revenir à l'état précédent.
    2. **Opérations sur une Copie Temporaire** : Plutôt que de modifier directement les données dans la base de données principale, les opérations de la transaction sont effectuées sur une copie temporaire (appelée espace de travail). Cette copie contient une version temporaire des données et des modifications. Si la transaction est validée, les modifications sont fusionnées avec la base de données principale. Si la transaction échoue, la copie temporaire est abandonnée, ce qui annule toutes les modifications.

2. **Cohérence (C - Consistency)** :

- Une transaction doit maintenir la cohérence de la base de données. Cela signifie que la base de données doit passer d'un état valide à un autre état valide. Toutes les contraintes d'intégrité doivent être respectées avant et après l'exécution de la transaction.
- **Syntaxe SQL pour vérifier les contraintes dans une transaction :**

```

```

```
-- Début de la transaction
BEGIN;

-- Définir la maniére de verifier les contraintes :

-- Définir que toutes les contraintes doivent être vérifiées à la fin de la
transaction:

SET CONSTRAINTS ALL DEFERRED;


-- les contraintes doivent être vérifiée immédiatement
SET CONSTRAINTS nom_de_la_contarinte IMMEDIATE ;


-- Syntaxe génerale :

SET CONSTRAINT[S]
    (ALL | <constraint_name>)
    (IMMEDIATE | DEFERRED) ;

/*

--
*/


COMMIT;
```

- `SET CONSTRAINTS` : Il s'agit de la clause principale qui indique que vous allez spécifier le comportement des contraintes de dans une transaction.
- `(ALL | <constraint_name>)` : Cette partie de la clause permet de spécifier quelles contraintes de clé étrangère vous souhaitez affecter. Vous pouvez soit choisir de définir le comportement pour toutes les contraintes (en utilisant "ALL"), soit spécifier une contrainte spécifique en utilisant son nom (par exemple, le nom de la contrainte).
- `(IMMEDIATE | DEFERRED)` : Cette partie de la clause permet de spécifier quand les contraintes doivent être vérifiées dans le contexte de la transaction.
    - `IMMEDIATE` : Les contraintes sont vérifiées immédiatement, c'est-à-dire chaque fois qu'une modification de données est effectuée dans la transaction. Si une violation de contrainte est détectée, elle provoque une erreur et la transaction est annulée.
    - `DEFERRED` : Les contraintes de clé étrangère ne sont vérifiées qu'à la fin de la transaction.

RQ : le mot clé `DEFERRABLE`

L'ajout du mot-clé `DEFERRABLE` après la définition d'une contrainte d'intégrité dans une base de données relationnelle permet de spécifier que la contrainte peut être différée et vérifiée à la fin de la transaction,

au lieu d'être vérifiée immédiatement après chaque opération de modification de données. Cela offre une flexibilité supplémentaire dans la gestion des contraintes de la base de données.

Voici un exemple de la syntaxe pour ajouter le mot-clé `DEFERRABLE` lors de la création d'une contrainte d'intégrité :

```
-- Création d'une contrainte UNIQUE DEFERRABLE
CREATE TABLE Exemple (
    ID INT PRIMARY KEY,
    Valeur INT UNIQUE DEFERRABLE
);
```

Dans cet exemple, nous créons une table nommée "Exemple" avec une contrainte `UNIQUE` sur la colonne "Valeur". En ajoutant le mot-clé `DEFERRABLE`, nous indiquons que cette contrainte `UNIQUE` peut être différée, ce qui signifie qu'elle ne sera pas vérifiée immédiatement après chaque insertion ou mise à jour, mais à la fin de la transaction.

3. **Isolation (I - Isolation)** :

- Les transactions s'exécutent de manière isolée les unes des autres. Cela signifie que les modifications apportées par une transaction ne sont généralement pas visibles pour d'autres transactions tant que la première n'est pas validée. L'isolation garantit que les transactions concurrentes ne se gênent pas mutuellement .
- Principe :
    - L'utilisateur doit avoir l'impression d'être le seul connecté à la BD
    - Une transaction ne doit pas pouvoir voir les résultats intermédiaires des autres transactions

4. **Durabilité (D - Durability)** :

Une fois qu'une transaction est validée (c'est-à-dire qu'elle a réussi avec succès), ses modifications doivent être permanentes et survivre à des pannes du système. Même en cas de panne du système, les modifications de la transaction validée doivent être préservées.

3. Problèmes d'accents concurrent :

1. **Lecture Sale (Dirty Read)** :
    - Problème : Une transaction lit des données modifiées par une autre transaction qui n'a pas encore été validée (commit).
    - Exemple :
        - Transaction T1 commence à modifier un enregistrement de compte bancaire.
        - T2 lit le même enregistrement pendant que T1 travaille dessus.
        - T1 annule ses modifications, ce qui signifie que les données lues par T2 étaient incorrectes.

2. **Lecture Non Répétable (Non-Repeatable Read)** :
    - Problème : Une transaction lit plusieurs fois les mêmes données, mais entre les lectures, une autre transaction effectue des modifications, ce qui fait que les lectures suivantes renvoient des résultats différents.
    - Exemple :
        - Transaction T1 lit le solde d'un compte bancaire (par exemple, 1000 euros).
        - Transaction T2 effectue un dépôt, augmentant le solde à 1500 euros.
        - Transaction T1 relit le solde du même compte et obtient un résultat différent (1500 euros).
3. **Lecture Fantôme (Phantom Read)** :
    - Problème : Une transaction exécute une requête qui renvoie un ensemble de lignes satisfaisant une condition, mais entre deux exécutions de la requête, une autre transaction effectue des insertions ou des suppressions qui modifient les résultats de la requête.
    - Exemple :
        - Transaction T1 exécute SELECT * FROM Produits WHERE Catégorie = 'Électronique'.
        - Transaction T2 insère de nouveaux produits dans la catégorie 'Électronique'.
        - Transaction T1 réexécute la même requête et obtient un ensemble de produits différent avec les nouveaux produits inclus.

4. Les niveaux d'isolation :

Les niveaux d'isolation dans les bases de données définissent la manière dont les transactions concurrentes interagissent les unes avec les autres.

Voici une explication des niveaux d'isolation les plus couramment utilisés avec des exemples pour illustrer leur comportement. Nous allons utiliser le scénario d'une base de données contenant une table "Comptes" avec les colonnes "ID" et "Solde" pour les exemples.

1. **Niveau de Lecture Non Répétable (Read Uncommitted)** :
    - Dans ce niveau, une transaction peut lire des données non validées par d'autres transactions. Cela signifie qu'une transaction peut voir les modifications d'autres transactions avant qu'elles ne soient validées.
    - Exemple :
        - Transaction 1 lit le solde du compte A : 1000.
        - Transaction 2 met à jour le solde du compte A : 1500.
        - Transaction 1 lit à nouveau le solde du compte A : 1500 (avant validation de Transaction 2).
    - Probléme : lecture impropre (lectures sales) ie : lecture de données non validés.
2. **Niveau de Lecture Répétable (Read Committed)** :

- Ce niveau garantit que les transactions ne voient que les données validées par d'autres transactions. Il évite les lectures de données non validées.
- Probléme : **lecture non reproductible :**

  **Exmeple* : considérons une table "Comptes" avec une colonne "Solde" et deux transactions :

  - Transaction T1 :
    - Lecture du solde du compte A (solde initial : 1000 euros).
    - Attente.
  - Transaction T2 :
    - Modification du solde du compte A (nouveau solde : 1500 euros).
    - Validation.
  - Transaction T1 (suite) :
    - Deuxième lecture du solde du compte A (le solde est maintenant de 1500 euros).
  - **Pb** :
    - la transaction T1 a effectué deux lectures du solde du compte A et a obtenu des résultats différents.

3. **Niveau de Répétabilité (Repeatable Read)** :
   - Ce niveau garantit que les données lues par une transaction restent inchangées tant que la transaction est active. Il évite les lectures de données non validées et les modifications par d'autres transactions.
   - Exemple :
     - Transaction 1 lit le solde du compte A : 1000.
     - Transaction 2 met à jour le solde du compte A : 1500.
     - Transaction 1 lit à nouveau le solde du compte A : 1000 (indépendamment de Transaction 2).
   - Probléme : **Lecture Fantôme**.
4. **Niveau de Sérialisabilité (Serializable)** :
   - C'est le niveau le plus strict. Il garantit que les transactions se comportent comme si elles s'exécutaient séquentiellement, même si elles s'exécutent simultanément.
   - Exemple :
     - Transaction 1 lit le solde du compte A : 1000.
     - Transaction 2 met à jour le solde du compte A : 1500.
     - Transaction 1 lit à nouveau le solde du compte A : 1000 (comme si Transaction 2 n'avait pas encore eu lieu).
   - Problème : Performance réduite.

RQ :

Il est important de noter que plus le niveau d'isolation est élevé, plus il y a de verrous et moins de transactions concurrentes peuvent s'exécuter en parallèle. Cela peut affecter les performances de la base de données.

Le choix du niveau d'isolation dépend des exigences de cohérence des données de votre application et des performances que vous pouvez vous permettre.

5. Syntaxe SQL : choix de l'issoaltion

- **Syntaxe génerale :**

```
-- Début de la transaction
BEGIN

    -- Syntaxe générique pour définir le niveau d'isolation
    SET TRANSACTION ISOLATION LEVEL isolation_level;

   -- D'autres opérations de la transaction vont ici...

   -- Valider la transaction
   COMMIT;
END;
```

- `isolation_level` est le niveau d'isolation que vous souhaitez définir. Il peut s'agir de l'un des niveaux d'isolation pris en charge par votre SGBD, tels que :
  - **READ UNCOMMITTED**
  - **READ COMMITTED**
  - **REPEATABLE READ**
  - **SERIALIZABLE**

6. `point de sauvegarde `savepoint`:

- un point de sauvegarde `savepoint` est une fonctionnalité qui permet de définir un point dans une transaction à partir duquel la transaction peut être partiellement annulée (`rollback`) sans affecter l'ensemble de la transaction.

- **utilisation d'un savepoint dans SQL :**

1. **Définir un Savepoint :**
   - Avant d'effectuer une série d'opérations au sein d'une transaction, vous pouvez définir un savepoint.

- Cela se fait avec la commande `SAVEPOINT` suivie d'un nom que vous attribuez au point de sauvegarde.

```
SAVEPOINT nom_du_savepoint;
```

2. **Effectuer des Opérations :**
   - Vous effectuez ensuite vos opérations au sein de la transaction, qui peuvent inclure des mises à jour, des insertions, des suppressions, etc.

```
UPDATE table SET colonne = 'nouvelle_valeur' WHERE condition;
```

3. **Rollback jusqu'au Savepoint :**
   - Si une partie de la transaction doit être annulée, vous pouvez effectuer un rollback jusqu'au savepoint spécifié.
   - Cela annulera toutes les opérations effectuées après la définition du savepoint.

```
ROLLBACK TO nom_du_savepoint;
```

4. **Commit :** Si toutes les opérations dans la transaction sont satisfaisantes, vous pouvez effectuer un commit pour valider définitivement ces changements.

```
COMMIT;
```

L'utilisation de savepoints est pratique dans les situations où vous souhaitez annuler uniquement une partie des changements effectués au sein d'une transaction sans annuler l'ensemble de la transaction. Cela peut être utile dans des scénarios complexes où des erreurs peuvent survenir à différentes étapes de la transaction, et vous souhaitez gérer ces erreurs de manière sélective.