

# Traîtement distribué des données

Master 1 ISD

## Contrôle de concurrence

Kim.Nguyen@lri.fr



Comprendre le monde,  
construire l'avenir





## 1 Contrôle de concurrence des SGBD

### 1.1 Transactions

### 1.2 Verrous

### 1.3 Niveaux d'isolation

### 1.4 Rappels JDBC



On se place dans le contexte : *1 source de donnée ↔ plusieurs utilisateurs.*

On considère les bases de données relationnelles classiques.

Comment gèrent elles les accès concurrent ?

- ◆ Transactions, modèle ACID
- ◆ Conflit et sérialisabilité
- ◆ Verrouillage (*locking*)
- ◆ Stratégies d'isolation

# Qu'est-ce qu'une transaction ?



Une *transaction* est une *suite d'opérations* exécutée sur la base de données de manière complète :

- ◆ Soit la transaction arrive a son terme et la base est modifiée
- ◆ Soit la transaction est interrompue et la base est dans son état initial (i.e. avant la première opération de la transaction)

# Modèle ACID



Les transactions d'une base de données relationnelle suivent le modèle ACID :

- ◆ **A**tomicity : une transaction est validée totalement ou pas du tout
- ◆ **C**onsistency : une transaction mène d'un état cohérent à un nouvel état cohérent (les contraintes de la base sont respectées)
- ◆ **I**solation : plusieurs transaction doivent donner l'illusion de s'exécuter sur des bases distinctes sans interférer
- ◆ **D**urability : les effets d'une transaction réussie (i.e. l'état dans lequel elle a mis la base) ne doivent *jamais* être perdus

On s'intéresse uniquement à *atomicité et isolation*.

La cohérence est une conséquence de l'atomicité et de l'isolation et du fait que les opérations de base (INSERT, DELETE, ...) vérifient les contraintes.

La durabilité repose sur des aspects systèmes et bas niveau (journal d'opération, re-jeu du journal en cas de défaillance, copies multiples, ...)

# Isolation et *sérialisabilité*



- ◆ On souhaite que les transactions du SGBD se comportent de manière sérielle (l'état de la base doit être celui qu'il aurait si on exécutait les transactions les unes à la suite des autres)
- ◆ L'ordonnateur de la base de donnée peut choisir l'ordre d'exécution

# Ordonnanceur naïf



Un ordonnanceur naïf (utilise une file de priorité pour exécuter les transactions en séquence) remplit ces contraintes. Mais :

- ◆ Les transactions doivent attendre même si elles travaillent sur des données disjointes
- ◆  $\Rightarrow$  performances mauvaises

# Ordonnanceur avancé



- ◆ les SGBD modernes possèdent un ordonnanceur permettant d'entremêler les transactions
- ◆ mais, il faut garantir que le résultat est équivalent à une évaluation séquentielle
- ◆ Ces ordonnancement sont dit *sériels* (*serializable*)
- ◆ En pratique les ordonnanceurs ne peuvent pas reconnaître tous les exécutions sérielles, ils vont juste en accepter certains (et refuser les autres)

# Notations (un peu de théorie)



On abstrait une transaction comme une suite de lecture et écriture d'objets

On note  $r_T(X)$  pour « la transaction  $T$  lit (*read*) l'objet  $X$ .

On note  $w_T(X)$  pour « la transaction  $T$  écrit (*write*) l'objet  $X$ .

Intuitivement il ne faut pas voir  $X$  comme une valeur mais plutôt comme un emplacement sur le disque (ou une adresse).

# Conflits



Certaines opérations peuvent être ré-ordonnées sans incidence sur le résultat final.

Par exemple : on peut changer l'ordre de deux lectures (entre elles), sans modifier le résultat final d'une transaction.

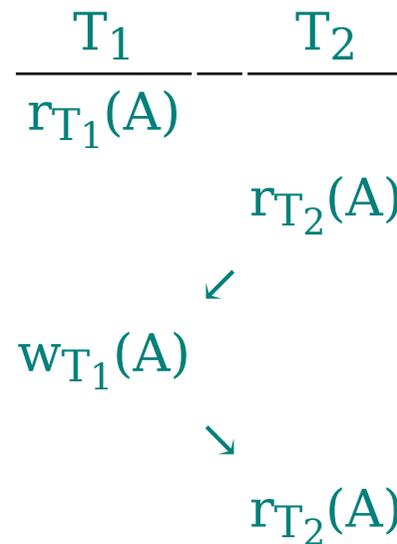
Si ré-ordonner deux opérations change le résultat final, ces deux opérations sont *en conflit*.

# Quelles opérations peuvent causer un conflit ?



Deux opérations sont en conflit, si :

- ◆ Elles portent sur *le même objet*
- ◆ L'une des deux au moins est une *écriture*



# Conflit de mise en séquence



- ◆ Soit un ordonnancement de plusieurs transactions
- ◆ On peut obtenir un ordonnancement *conflit-équivalent* en permutant les opérations adjacentes sans conflit.
- ◆ Si un ordonnancement est conflit-équivalent à un ordonnancement sériel, il est « sérialisable »

# Détection de conflit



Étant données  $n$  transactions  $T_1, \dots, T_n$ , comment détecter si elles sont sérialisables ?

- ◆ On crée un graphe dont les sommets sont les  $T_i$
- ◆ Pour toutes paires de transactions  $T_i, T_j$  :
  - ◆ s'il existe  $r_{T_j}(X)$  après  $w_{T_i}(X)$ , on ajoute une arrête de  $T_i$  vers  $T_j$
  - ◆ s'il existe  $w_{T_j}(X)$  après  $r_{T_i}(X)$ , on ajoute une arrête de  $T_i$  vers  $T_j$
  - ◆ s'il existe  $w_{T_j}(X)$  après  $w_{T_i}(X)$ , on ajoute une arrête de  $T_i$  vers  $T_j$
- ◆ Si le graphe formé est *acyclique* l'ordonnement est sérialisable



## 1 Contrôle de concurrence des SGBD

1.1 Transactions ✓

1.2 Verrous

1.3 Niveaux d'isolation

1.4 Rappels JDBC

# Transactions sérialisables en pratique



Le SGBD va faire en sorte de n'autoriser que les transactions sérialisables, et refuser les autres

Plusieurs techniques possibles

- ◆ Utilisation de verrous (*locking*)
- ◆ Utilisation d'étiquettes de temps (*timestamp*)

# Verrouillage en deux phases (2-Phase Locking ou 2PL)

Protocole de verrouillage simple *qui garanti la sérialisabilité*

Utilisé par tous les SGBD modernes (mais ils vont plus loin)

- ◆ chaque objet possède un verrou
- ◆ Les verrous nécessaires à une transaction sont acquis au fur et à mesure
- ◆ Dès qu'un verrou est relâché, alors plus moyen d'en acquérir un autre (dans la même transaction)

Sûr, mais trop restrictif (à pleine mieux que mise en séquence)

Solution : avoir plusieurs types de verrous :

- ◆ Verrou d'écriture (exclusif)
- ◆ Verrou de lecture (partagé)
- ◆ Verrou de mise à jour (version spéciale du verrou d'écriture)
- ◆ Verrou sur toute la table ou uniquement sur la ligne



## 1 Contrôle de concurrence des SGBD

1.1 Transactions ✓

1.2 Verrous ✓

1.3 Niveaux d'isolation

1.4 Rappels JDBC

# SQL dans tout ça



La sérialisabilité totale n'est pas toujours nécessaire et est couteuse en ressources

SQL définit 4 niveau d'isolation (i.e. manière dont sont cloisonnées les transaction). On présente les deux principaux:

- ◆ READ COMMITTED
- ◆ SERIALIZABLE

# READ COMMITTED



Dans ce mode *les verrous en lecture* sont relâché immédiatement

- ◆ Lire une valeur dans une transaction  $T_1$  n'empêche pas son écriture dans une transaction  $T_2$
- ◆ Bonnes performances
- ◆ Un même objet peut avoir deux valeurs différentes au sein de la même transaction, même s'il n'a pas été écrit par cette transaction

```
BEGIN; | BEGIN;
SELECT age FROM T WHERE id=1 |
-- age vaut 42 |
| UPDATE T SET age = age + 1 WHERE id = 1
| COMMIT;
SELECT age FROM T WHERE id=1 |
-- age vaut 43 |
```

C'est souvent ce que l'on veut.

# SERIALIZABLE



Utilisation du protocole 2PL les verrous de lecture sont relaché en fin de transaction, un snapshot est fait au début de la transaction.

- ◆ Les valeurs ne sont pas modifiées par les autres transactions
- ◆ Commitée à la fin
- ◆ Performances moins bonnes (plus de verrous)

```
BEGIN; | BEGIN;
SET TRANSACTION ISOLATION | SET TRANSACTION ISOLATION LEVEL
LEVEL SERIALIZABLE; | SERIALIZABLE;
SELECT age FROM T WHERE id=1 |
-- age vaut 42 | UPDATE T SET age = age + 1 WHERE id = 1
| COMMIT;
SELECT age FROM T WHERE id=1 |
-- age vaut 42 |
```

# Deadlocks



La présence de verrous implique la possibilité de *deadlock*

Souvent causé par des verrous de granularité fine ( $T_1$  verrouille la ligne 'a' et 'b',  $T_2$  verrouille la ligne 2 et 1)

Le SGBD *détecte les deadlocks* et annule arbitrairement l'une des transaction (avec une erreur).

# Verrous et SQL (simplifié)



Chaque ordre SQL manipule un type de verrou différent :

- ◆ DELETE, DROP : verrou en écriture sur la table
- ◆ SELECT ... WHERE  $\varphi$  : verrou en lecture sur toutes les lignes nécessaires au calcul de  $\varphi$
- ◆ UPDATE ... WHERE  $\varphi$  : verrou en lecture sur toutes les lignes nécessaires au calcul de  $\varphi$   
*et* verrou en écriture sur les lignes sélectionnées
- ◆ INSERT : pas de verrou mais risque de n'être pris en compte qu'à la transaction suivante.



## 1 Contrôle de concurrence des SGBD

1.1 Transactions ✓

1.2 Verrous ✓

1.3 Niveaux d'isolation ✓

1.4 Rappels JDBC

# Connexion à une base



```
Class.forName("org.postgresql.Driver");  
connection =  
DriverManager.getConnection("jdbc:postgresql://host:port/" + base,  
    username, password);
```

- ◆ On importe dans la JVM courante la classe qui code le driver vers une base de donnée (ici Postgresql)
- ◆ Le code de cette classe doit se trouver dans un *.jar* ou *.class* accessible depuis le CLASSPATH
- ◆ La classe *DriverManager* maintient une *Map* entre chaîne de caractères ("jdbc:postgresql") et classe (org.postgresql.Driver)
- ◆ La méthode `getConnection` utilise le préfixe de l'URL de connexion pour savoir quel driver utiliser.

# Exécution d'ordres SQL (version simple) (1/2)



```
Statement stmt = connection.createStatement();  
stmt.executeUpdate("DROP TABLE MOVIE CASCADE ");
```

```
stmt.addBatch("CREATE TABLE MOVIE (...)" );  
stmt.addBatch("CREATE TABLE PEOPLE (...)" );  
stmt.addBatch("CREATE TABLE ACTOR (...)" );  
stmt.addBatch("CREATE TABLE DIRECTOR (...)" );
```

```
stmt.executeBatch();
```

- ◆ On crée un objet *Statement*
- ◆ Les mises à jour (DROP, INSERT,...) se font via *executeUpdate*. Renvoie un entier qui est le nombre de mise à jours effectuées.
- ◆ On peut préparer plusieurs mises à jours avec (*addBatch*) et les envoyer d'un coup à la base (économie d'aller-retour via le réseau)
- ◆ Toutes ces fonctions peuvent lever des exceptions

# Exécution d'ordres SQL (version simple) (2/2)



```
Statement stmt = connection.createStatement();
ResultSet res = stmt.executeQuery("SELECT * FROM ...");
while (res.next()) {

    String name = res.getString(1);
    int age = res.getInt("age_col");
    ...
}
```

- ◆ L'évaluation d'une requête se fait via *executeQuery*.
- ◆ Un *ResultSet* implémente une interface d'itérateur, initialement positionné *avant* la première ligne de résultats.
- ◆ La méthode *next* avance dans l'itérateur et renvoie vrai tant qu'on est sur un résultat.
- ◆ On accède à la colonne voulue avec *getType*. On doit donner le type Java correspondant au type SQL de la colonne. On peut accéder aux colonnes par numéro (à partir de 1) ou par nom.

# Exécution d'ordres SQL (version avancée)



Il existe des sous-interfaces à l'interface *Statement*:

◆ *CallableStatement* permet d'appeler une procédure stockée dans la base (généralement en PL/SQL).

◆ *PreparedStatement* permet de créer un ordre paramétrable qui peut être réutilisé plusieurs fois :

```
String query = "SELECT * FROM T WHERE name LIKE ?"  
PreparedStatement stmt = connection.prepareStatement(query);  
stmt.setString(1, "Toto");  
ResultSet res1 = stmt.executeQuery();  
  
stmt.setString(1, "Titi");  
ResultSet res2 = stmt.executeQuery();
```

Attention ce n'est pas syntaxique, on ne peut pas faire "SELECT \* ?", suivi de `.setString(1, "FROM T");`

## ... bien d'autres choses



- ◆ On ne met pas de « ; » dans les chaînes de caractère des ordres
- ◆ Les indices commencent à 1 (pas 0)
- ◆ La plupart de ces méthode lèvent des exceptions, il faut les propager ou les rattraper
- ◆ Lire la doc, pour tout ce qui n'est pas présenté (en particulier l'utilisation de `.setAutocommit(false)/.commit()/.rollback()` pour faire des transactions).