La gestion des transactions

La gestion des transactions

On ne doit appliquer à une base de données que des suites d'opérations qui en maintiennent la *cohérence* (*consistency*). On introduit dans ce but le concept de *transaction*.

Définition : Une *transaction* est une suite d'opérations (lectures - écritures) effectuées sur une base de données

Le concept de transaction s'utilise comme suit.

1. L'interaction avec la base de données se fait exclusivement par l'intermédiaire de transactions qui en maintiennent la cohérence.

2. Le système de base de données gère les transactions de façon à en garantir l'*atomicité*. Si une transaction est exécutée, elle doit l'être entièrement. En cas de parallélisme entre transactions, chacune doit être exécutée de façon en apparence indivisible.

Nous allons étudier comment garantir l'atomicité de transactions.

Exemple : Réservation de sièges.

read(VOL(#123,sièges))
sièges := *sièges* − 1
write(VOL(#123,sièges))
write(RESERV(#123,Jean))

Contrainte de cohérence : nb. de sièges + nb. de réservations = capacité du vol.

Transactions "ACID"

Les caractéristiques souhaitées pour l'exécution d'une transaction sont souvent résumées par l'acronyme ACID.

Atomicity : l'exécution des transactions doit être atomique.

Consistency : l'exécution des transactions doit respecter les contraintes d'intégrité définies pour la base de données.

Isolation : il n'y a pas d'interaction entre transactions exécutées en parallèle.

Durability : une fois une transaction exécutée, son résultat ne peut pas être perdu.

Atomicité et parallélisme

Définition : Un *ordonnancement* (*schedule*) d'un ensemble de transactions $\{T_1, \ldots, T_k\}$ est un ordre d'exécution des opérations a_i^j des transactions T_i.

Exemples :

T_0	T_1
r(VOL(#123,sièges))	
sièges := *sièges* $-$ 1	
	r(VOL(#123,sièges))
	sièges := *sièges* $-$ 1
w(VOL(#123,sièges))	
w(RESERV(#123,Jean))	
	w(VOL(#123,sièges))
	w(RESERV(#123,Georges))

T_0	T_1
r(VOL(#123,sièges))	
sièges $:=$ *sièges* $- 1$	
w(VOL(#123,sièges))	
w(RESERV(#123,Jean))	
	r(VOL(#123,sièges))
	sièges $:=$ *sièges* $- 1$
	w(VOL(#123,sièges))
	w(RESERV(#123,Georges))

Note : On suppose la variable *sièges* locale à chaque transaction.

Ordonnancements séquentiels et séquentialisables

Définition : Un *ordonnancement* (*schedule*) d'un ensemble de transactions $\{T_1, \ldots, T_k\}$ est *séquentiel* (*serial*) lorsqu'il existe une permutation π de $\{1, \ldots, k\}$ telle que l'ordonnancement est $T_{\pi(1)}, T_{\pi(2)}, \ldots, T_{\pi(k)}$.

Définition : Un *ordonnancement* (*schedule*) d'un ensemble de transactions $\{T_1, \ldots, T_k\}$ est *séquentialisable* (*serializable*) s'il existe un ordonnancement séquentiel de $\{T_1, \ldots, T_k\}$ qui "donne le même résultat".

Il nous faut :

1. un critère précis pour déterminer quand un ordonnancement est séquentialisable ;
2. un protocole d'exécution des transactions qui garantisse la sequentialisabilité.

Critère de séquentialisabilité basé sur les conflits

Définition : Deux actions a_1 et a_2 sont en *conflit* si l'exécution de $a_1; a_2$ peut donner un résultat différent de l'exécution de $a_2; a_1$.

Exemple : relation de conflit pour les opérations *read* et *write* appliquées au même élément.

	r	w
r	$-$	$+$
w	$+$	$+$

Critère : Un ordonnancement est *séquentialisable par rapport aux conflits* s'il peut être transformé en un ordonnancement séquentiel par permutations successives d'actions qui ne sont pas en conflit.

Note : Un ordonnancement peut être séquentialisable sans être sequentialisable par rapport aux conflits. L'intérêt du critère "par rapport aux conflits" est qu'il est plus facile à exploiter que la notion générale.

Exemples :

T_0	T_1
r(VOL(#123,sièges))	
sièges $:=$ *sièges* $-\ 1$	
	r(VOL(#123,sièges))
	sièges $:=$ *sièges* $-\ 1$
w(VOL(#123,sièges))	
w(RESERV(#123,Jean))	
	w(VOL(#123,sièges))
	w(RESERV(#123,Georges))

N'est pas séquentialisable.

T_0	T_1
r(VOL(#123,sièges))	
sièges $:=$ *sièges* -1	
w(VOL(#123,sièges))	
	r(VOL(#123,sièges))
	sièges $:=$ *sièges* -1
	w(VOL(#123,sièges))
w(RESERV(#123,Jean))	
	w(RESERV(#123,Georges))

Est séquentialisable

Comment déterminer la séquentialisabilité ?

Définition : Une transaction T_i *précède* une transaction T_j dans un ordonnancement donné, s'il existe une action a_i de T_i qui précède une action a_j de T_j avec laquelle elle est en conflit.

Le *graphe de précédence* d'un ordonnancement d'un ensemble de transactions $\{T_1, \ldots, T_k\}$ est le graphe (V, E) tel que

- L'ensemble des sommets V est l'ensemble des transactions $\{T_1, \ldots, T_k\}$.
- L'ensemble des arcs E est l'ensemble des paires (T_i, T_j) telles que T_i précède T_j dans l'ordonnancement donné.

Théorème : Un ordonnancement est séquentialisable par rapport aux conflits si et seulement si son graphe de précédence est sans cycle.

Démonstration :

Si : 1. Si le graphe est sans cycle, il contient au moins un noeud dans lequel n'entre aucun arc.

2. Les actions de la transaction correspondant à ce noeud ne sont donc en conflit avec aucune action qui les précède dans l'ordonnancement.

3. Ces actions peuvent donc être déplacées vers le début de l'ordonnancement en ne les permutant qu'avec des actions avec lesquelles elles ne sont pas en conflit.

4. En répétant le même raisonnement avec le graphe dont le noeud traité a été éliminé, on peut construire de proche en proche un ordonnancement séquentiel correspondant à l'ordonnancement de départ.

Seulement si :

1. Supposons donc que dans le graphe de précédence, il y ait un cycle :

   $$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1.$$

2. Vu que la permutation d'actions qui ne sont pas en conflit ne change pas le graphe de précédence, dans un ordonnancement séquentiel correspondant, les actions de T_1, doivent précéder celles de T_2, . . .qui précèdent celles de T_n, qui précèdent celles de T_1, ce qui est impossible.

Algorithmes d'ordonnancement : les verrous

On peut assurer la séquentialisabilité des ordonnancements par l'exclusion mutuelle des transactions (par exemple à l'aide d'un sémaphore). Toutefois, cette contrainte est souvent trop restrictive.

On va donc supposer que l'on n'assure l'exclusion mutuelle qu'au niveau des opérations de lecture/écriture élémentaires sur la base de données (accès à un bloc). Pour cela on utilise deux opérations :

- $lock(D, mode)$ où mode est r ou w ;
- $unlock(D)$.

Problème : comment utiliser ces opérations pour garantir la séquentialisabilité ?

Exemple :

T_0	T_1
r(VOL(#123,sièges))	
sièges $:=$ *sièges* $-$ 1	
	r(VOL(#123,sièges))
	sièges $:=$ *sièges* $-$ 1
w(VOL(#123,sièges))	
w(RESERV(#123,Jean))	
	w(VOL(#123,sièges))
	w(RESERV(#123,Georges))

Peut être évité par *lock*, *unlock* :

T_0	T_1
lock(VOL(#123), w)	
r(VOL(#123,sièges))	
sièges $:=$ *sièges* -1	
w(VOL(#123,sièges))	
unlock(VOL(#123))	
	lock(VOL(#123), w)
	r(VOL(#123,sièges))
	sièges $:=$ *sièges* -1
	w(VOL(#123,sièges))
	unlock(VOL(#123))
lock(RESERV(Jean), w)	
w(RESERV(#123,Jean))	
unlock(RESERV(Jean))	
	lock(RESERV(Georges), w)
	w(RESERV(#123,Georges))
	unlock(RESERV(Georges))

Règle ?

– Verrouiller une donnée lors de sa première utilisation.

– La déverrouiller après sa dernière utilisation.

Pas toujours suffisant.

Exemple :

T_0	T_1
lock(VOL(#123), w)	
r(VOL(#123,sièges))	
sièges := *sièges* − 1	
w(VOL(#123,sièges))	
unlock(VOL(#123))	
repas := *182* − *sièges*	
	lock(VOL(#123), w)
	r(VOL(#123,sièges))
	sièges := *sièges* − 1
	w(VOL(#123,sièges))
	unlock(VOL(#123))
	repas := *182* − *sièges*
	lock(REPAS(#123), w)
	w(REPAS(#123,repas))
	unlock(REPAS(#123))
lock(REPAS(#123), w)	
w(REPAS(#123,repas))	
unlock(REPAS(#123))	

Règle des 2 phases
(2 phase locking)

- Chaque donnée est verrouillée avant sa première utilisation et déverrouillée après sa dernière utilisation.
- Aucune opération de verrouillage ne peut suivre une opération de déverrouillage de la même transaction.
- Il y a donc deux phases : verrouillage et déverrouillage.

Théorème : Si un ordonnancement est obtenu en respectant la règle des deux phases, il est séquentialisable.

Démonstration :

1. Supposons que ce ne soit pas le cas. Le graphe de précédence de l'ordonnancement possède donc un cycle
   $$T_{i_1} \rightarrow T_{i_2} \rightarrow T_{i_3} \rightarrow \cdots \rightarrow T_{i_n} \rightarrow T_{i_1}$$
2. Le fait qu'il existe un arc dans la graphe de précédence entre T_{i_1} et T_{i_2} indique qu'il y a une ressource pour laquelle les deux transactions sont en conflit et qui est utilisée par T_{i_1} avant T_{i_2}.
3. Cette ressource doit donc être déverrouillée par T_{i_1} avant d'être verrouillée par T_{i_2}. Par conséquent, la phase "lock" de T_{i_2} s'étend au delà la phase "lock" de T_{i_1} (il y a au moins un "unlock" T_{i_1} avant la fin de la phase "lock" de T_{i_2}.
4. Similairement, la phase "lock" de T_{i_3} s'étend au delà de la phase "lock" de T_{i_2} (et donc de T_{i_1}), . . ., la phase lock de T_{i_n} s'étend au delà de la phase "lock" de T_{i_1}.
5. Par conséquent la phase "lock" de T_{i_1} s'étend au delà de la phase "lock" de T_{i_1}, ce qui signifie que la règle des deux phases n'est pas respectée.

Problème : la règle des 2 phases garantit la séquentialisabilité, mais elle n'exclut pas la présence de blocages (*deadlocks*) ou d'exclusion d'un processus (*starvation*)

Exemples :

Blocage :

T_0	T_1
lock(A,w)	
	lock(B,w)
lock(B,w) - fail	
	lock(A,w) - fail

Exclusion :

T_{0i}	T_{1i}	T_2
lock(A,r)		
	lock(A,r)	
		lock(A,w) - fail
unlock(A)		
lock(A,r)		
	unlock(A)	
		lock(A,w) - fail
	lock(A,r)	
unlock(A)		
lock(A,r)		
⋮	⋮	

Prévention des blocages

Imposer un ordre sur les transactions : $T_i < T_j$ si T_i a démarré avant T_j.

On applique alors la politique suivante :

- Si une transaction T_i doit avoir accès à une donnée verrouillée par T_j, alors
  - si $T_i > T_j$ (T_i a demarré après T_j),
    T_i attend ;
  - si $T_i < T_j$ (T_i a demarré avant T_j), la transaction T_j est annulée (*rolled back*) et T_i poursuit son exécution.

La transaction la plus ancienne poursuit donc toujours son exécution et il n'y a pas de blocage possible.

Question : Comment gérer des transactions qui peuvent être interrompues ?

Récupération en cas d'erreurs

Types d'erreurs :

- Annulation d'une transaction ;
- Arrêt du système ;
- Perte d'une partie du contenu du disque.

Modèle utilisé :

- Les données sont transférées de et vers le disque bloc par bloc (page par page) par des opérations atomiques.
- Chaque transaction T se termine
  - soit par $commit(T)$: les modifications peuvent être incorporées dans la base de données ;
  - soit par $abort(T)$: la transaction est annulée, les modifications ne doivent pas être incorporées dans la base de données.

Fichier historique
(*Log file*)

Pour permettre la récupération en cas d'erreurs, on utilise un fichier historique dans lequel on sauve les informations suivantes.

- Le début de chaque transaction.
- Chaque modification effectuée par une transaction : l'identification de la transaction, la page telle qu'elle est modifée par la transaction.
- Les opérations $commit(T)$ effectuées.
- Les points de synchronisation.

Gestion du fichier historique

1. Lorsqu'une transaction modifie une page, cela est directement répercuté dans le fichier historique, mais les pages ne sont pas modifiées (on retarde donc les opérations *unlock*).
2. Si la transaction se termine normalement ($commit(T)$), $commit(T)$ est ajouté au fichier historique. Les dernières additions au fichier historique sont effectivement sauvées sur disque.
3. Les modifications effectuées par la transaction sont effectivement réalisées (+ *unlock*). Les pages correspondantes peuvent être sauvées sur disque ou non.
4. On impose périodiquement un *point de synchronisation* (*checkpoint*) : les nouvelles transactions sont interdites ; on attend que toutes les transactions en cours soient terminées ; on sauve sur le disque toutes les pages modifiées.

Méthode de récupération

- **Arrêt d'une transaction** : rien à faire (les modifications ne sont pas effectuées).
- **Arrêt du système** : à l'aide du fichier historique, on réexécute toutes les transactions dont le $commit$ se trouve après le dernier point de synchronisation.
- **Perte de contenu disque** : idem, mais à partir du dernier point de synchronisation correspondant à une sauvegarde que l'on peut restaurer. **Note** : Le fichier historique ne peut pas être perdu ! On en garde donc plusieurs copies.

Transactions en SQL

Les systèmes de bases de données implémentent plusieurs mécanismes de gestion de la concurrence et des erreurs.

- **Gestion complète des transactions :** c'est le mécanisme que nous venons de décrire avec possibilité d'arrêt d'une transaction et mécanisme automatique de prévention des blocages (uniquement sur tables InnoDB en MySQL).
  La fin d'une transaction est signalée par la commande **commit** ou **rollback**. Le début d'une transaction par **start transaction** ou par la fin de la transaction précédente.
- **Verrouillage de tables et sauvetage immédiat sur disque des modifications :** Soit implicitement lors des opérations de modification, ce qui garantit leur atomicité, soit explicitement.
  Le verrouillage explicite se fait à l'aide des commandes **lock tables** et **unlock tables**. Si l'on verrouille plusieurs tables, l'ordre de verrouillage des tables est implicitement fixé, ce qui évite les deadlocks.

Modes de gestion des transactions en SQL

Le standard SQL prévoit plusieurs modes de gestion des transaction.

set transaction isolation level ....

- **serializable :** c'est le le mode le plus contraignant, les transactions sont exécutées de façon séquentialisables.
- **read uncommitted :** la lecture de données modifiées par d'autres transactions qui ne sont pas encore achevées (**commit** exécuté) est possible.
- **read committed :** on ne lit que des données modifiées par des transactions terminées, mais des lectures multiples des même données peuvent produire un résultat différent.
- **repeatable read :** On ne voit que des données modifiées par des transactions terminées et les lectures multiples donnent le même résultat, si ce n'est que des nouveaux tuples peuvent apparaître entre deux lectures.

Exemples de transactions

- Création d'une table utilisée pour chaque exemple

tbl

Key	Value
A	1
B	2
C	3
D	4

Verrouillage des tables

- LOCK TABLE <nom de la table> (, <autres tables)*
  - Mode READ : verrou en lecture. Plusieurs sessions peuvent lire, mais personne ne peut écrire tant que le verrou est en place.
  - Mode WRITE : verrou en écriture (exclusif). Une seule personne peut lire et écrire
- Mode READ:

T1

```
mysql> LOCK TABLE tbl READ;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
+-----+-------+
4 rows in set (0.00 sec)
```

T2

```
mysql> SELECT * FROM tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
+-----+-------+
4 rows in set (0.00 sec)
```

LOCK TABLE READ (suite)

T1

T2

```
mysql> INSERT INTO tbl VALUES('E',5);
```

(la commande ne fait rien et attend)

```
mysql> INSERT INTO tbl VALUES('F',6);
ERROR 1099 (HY000): Table 'tbl' was locked with a READ lock and can't be updated
```

```
mysql> UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)
```

```
Query OK, 1 row affected (30.81 sec)
```

```
mysql> SELECT * FROM tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
| E   |     5 |
+-----+-------+
5 rows in set (0.00 sec)
```

LOCK TABLE WRITE

T1

```
mysql> LOCK TABLE tbl WRITE;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)
```

T2

```
mysql> SELECT * FROM tbl;
```

La commande attend

```
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
+-----+-------+
4 rows in set (24.19 sec)
```

Sélection partielle des tuples

- **LOCK TABLES** permet d'assurer l'exclusion mutuelle, mais l'entièreté de la table est verrouillée.

- Dans certains cas, c'est trop restrictif
  - Ex : Applications bancaires

- On préfèrera alors verrouiller un ensemble de tuples plutôt que l'ensemble de la table
  - Chaque transaction est alors libre de travailler sur ses tuples, même à l'intérieur de la même table.

- **SELECT … LOCK IN SHARE MODE** et **SELECT … FOR UPDATE**

Exemple de verrou partiel

- Lock in share mode (lecture)

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tbl WHERE tbl.KEY = 'A' LOCK IN SHARE MODE;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
+-----+-------+
1 row in set (0.00 sec)
```

Début de la transaction

Un tuple verrouillé

Rien n'empêche de modifier les autres tuples

```
mysql> UPDATE tbl SET Value=5 WHERE tbl.Key  ='C';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE tbl SET Value=5 WHERE tbl.Key  ='A';
```

La commande attend

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

Verrou libéré

La commande est alors exécutée

```
Query OK, 1 row affected (16.44 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Exemple de verrou partiel (suite)

- For update (écriture)

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tbl WHERE tbl.KEY = 'A' FOR UPDATE;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
+-----+-------+
1 row in set (0.01 sec)
```

Début de la transaction

Un tuple verrouillé

Rien n'empêche de verrouiller les autres tuples

```
mysql> SELECT * FROM tbl WHERE tbl.KEY = 'B' FOR UPDATE;
+-----+-------+
| Key | Value |
+-----+-------+
| B   |     2 |
+-----+-------+
1 row in set (0.01 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tbl WHERE tbl.KEY = 'A' FOR UPDATE;
```

La commande attend

```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

La commande est alors exécutée

```
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
+-----+-------+
1 row in set (5.59 sec)
```

Les deadlocks

- L'utilisation de SELECT FOR UPDATE est susceptible de générer des deadlocks comme l'illustre l'exemple suivant.

```
mysql> SELECT * FROM tbl WHERE tbl.KEY = 'A' LOCK IN SHARE MODE;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
+-----+-------+
1 row in set (0.00 sec)
```

```
mysql> SELECT * FROM tbl WHERE tbl.KEY = 'A' FOR UPDATE;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
+-----+-------+
1 row in set (0.00 sec)
```

```
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting trans
action
```

- Situation classique où l'escalade de privilèges fait que T1 attend T2 qui attend T1 à son tour → deadlock.
- D'autres cas sont moins reproductibles, mais les deadlocks sont une réalité à accepter, et il faut parfois redémarrer la transaction.

Les différents niveaux d'isolation

- Permet d'effectuer des transactions avec certaines propriétés sans s'inquiéter de l'utilisation explicite de verrous.

- 4 modes
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

- Désactivation de la validation automatique
  - Set autocommit=0;

- On termine une transaction par
  - Commit : validation
  - Rollback : annulation

READ UNCOMMITTED

```
mysql> set session transaction isolation level read uncommitted;
Query OK, 0 rows affected (0.01 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
+-----+-------+
4 rows in set (0.00 sec)
```

```
mysql> update tbl set Value = 5 WHERE tbl.Key = 'C';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     5 |
| D   |     4 |
+-----+-------+
4 rows in set (0.00 sec)
```

- Aucune protection

READ COMMITTED

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update tbl set value = 5 WHERE tbl.Key = 'C';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
+-----+-------+
4 rows in set (0.00 sec)
```

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     5 |
| D   |     4 |
+-----+-------+
4 rows in set (0.01 sec)
```

- Lecture uniquement des données validées. Lectures différentes possibles.

REPEATABLE READ

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update tbl set value = 5 WHERE tbl.Key = 'C';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> select * from tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
+-----+-------+
4 rows in set (0.00 sec)
```

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
+-----+-------+
4 rows in set (0.00 sec)
```

- Lectures identiques dans toute la transaction (par défaut en MySQL)

Ecritures fantômes

- Le niveau REPEATABLE READ permet en théorie les lectures fantômes (insertion d'un nouveau tuple qui apparait dans l'autre transaction).
- L'implémentation de MySQL est différente : ce niveau empêche les lectures fantômes.
- Par contre, elle n'empêche pas les écritures fantômes:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
+-----+-------+
4 rows in set (0.00 sec)
```

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into tbl values ('E',5);
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> update tbl set value = 10 where tbl.key = 'E';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from tbl;
+-----+-------+
| Key | Value |
+-----+-------+
| A   |     1 |
| B   |     2 |
| C   |     3 |
| D   |     4 |
| E   |    10 |
+-----+-------+
```

SERIALIZABLE

- C'est le niveau le plus contraignant. Ici, une insertion sera bloquante, donc plus de problème de lecture/écriture fantôme.
- Attention toutefois à la possibilité de deadlock:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update tbl set Value = 1 WHERE tbl.Key = 'A';
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0
```

```
mysql> start transaction;
Query OK, 0 rows affected (0.01 sec)

mysql> update tbl set Value = 2 WHERE tbl.Key = 'B';
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql> update tbl set Value = 1 WHERE tbl.Key = 'A';
```

(attend la libération du verrou sur le premier tuple)

```
mysql> update tbl set Value = 2 WHERE tbl.Key = 'B';
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting trans
action
```

Le choix de niveau d'isolation est un compromis entre fiabilité, facilité d'utilisation et performances.

La définition exacte des niveaux d'isolation dépend du système utilisé.