# Bases de données

Ecole Marocaine des Sciences de l'Ingénieur

© Yousra Lembachar

# Déclencheurs dans SQL

- Syntaxe des déclencheurs dans SQL
- Evénements BEFORE AFTER INSERT, UPDATE ou DELETE
- Comment renforcer les contraintes d'intégrité avec des déclencheurs
- Enchaînement des déclencheurs

# Déclencheurs (suite)

# Déclencheurs dans SQL

- Déclencheurs (suite)
  - Déclencheurs cycliques
  - Auto-déclencheurs
  - Conflits entre déclencheurs
- Transactions dans SQL

# Déclencheurs dans SQL

```
CREATE TRIGGER déclencheur
(événement)
BEFORE | AFTER

INSERT | DELETE | UPDATE ON nom_table

[REFERENCING variables]

[FOR EACH ROW]
(condition)
[WHEN] [condition]
 (action)
action
```

# Auto-déclencheurs

On considère les tables T1(A1), T2(A2) et T3(A3)

```
CREATE TRIGGER R1
AFTER INSERT ON T1
FOR EACH ROW
INSERT INTO T1
        VALUES (NEW.A1 + 1);
```

Déclencheur qui s'auto-déclenche infiniment

# Auto-déclencheurs

```
CREATE TRIGGER R1
AFTER INSERT ON T1
FOR EACH ROW
WHEN (SELECT COUNT(A1) FROM T1) <= 5
INSERT INTO T1
        VALUES (NEW.A1 + 1);
```

Déclencheur qui s'auto-déclenche au maximum 5 fois

# Déclencheurs cycliques

```
CREATE TRIGGER R1

AFTER INSERT ON T1

FOR EACH ROW

INSERT INTO T2 VALUES (NEW.A1 + 1);

CREATE TRIGGER R2

AFTER INSERT ON T2

FOR EACH ROW

INSERT INTO T3 VALUES (NEW.A2 + 1);

CREATE TRIGGER R3

AFTER INSERT ON T3

FOR EACH ROW

INSERT INTO T1 VALUES (NEW.A3 + 1);
```

Cycle infini:

R1 -> R2 -> R3 -> R1

# Déclencheurs cycliques

```
CREATE TRIGGER R1

AFTER INSERT ON T1

FOR EACH ROW

WHEN (SELECT COUNT(A1) FROM T1) <= 5

INSERT INTO T2 VALUES (NEW.A1 + 1);

CREATE TRIGGER R2

AFTER INSERT ON T2

FOR EACH ROW

INSERT INTO T3 VALUES (NEW.A2 + 1);

CREATE TRIGGER R3

AFTER INSERT ON T3

FOR EACH ROW

INSERT INTO T1 VALUES (NEW.A3 + 1);
```

Cycle qui se répète 5 fois:

R1 -> R2 -> R3 -> R1

# Conflits entre déclencheurs

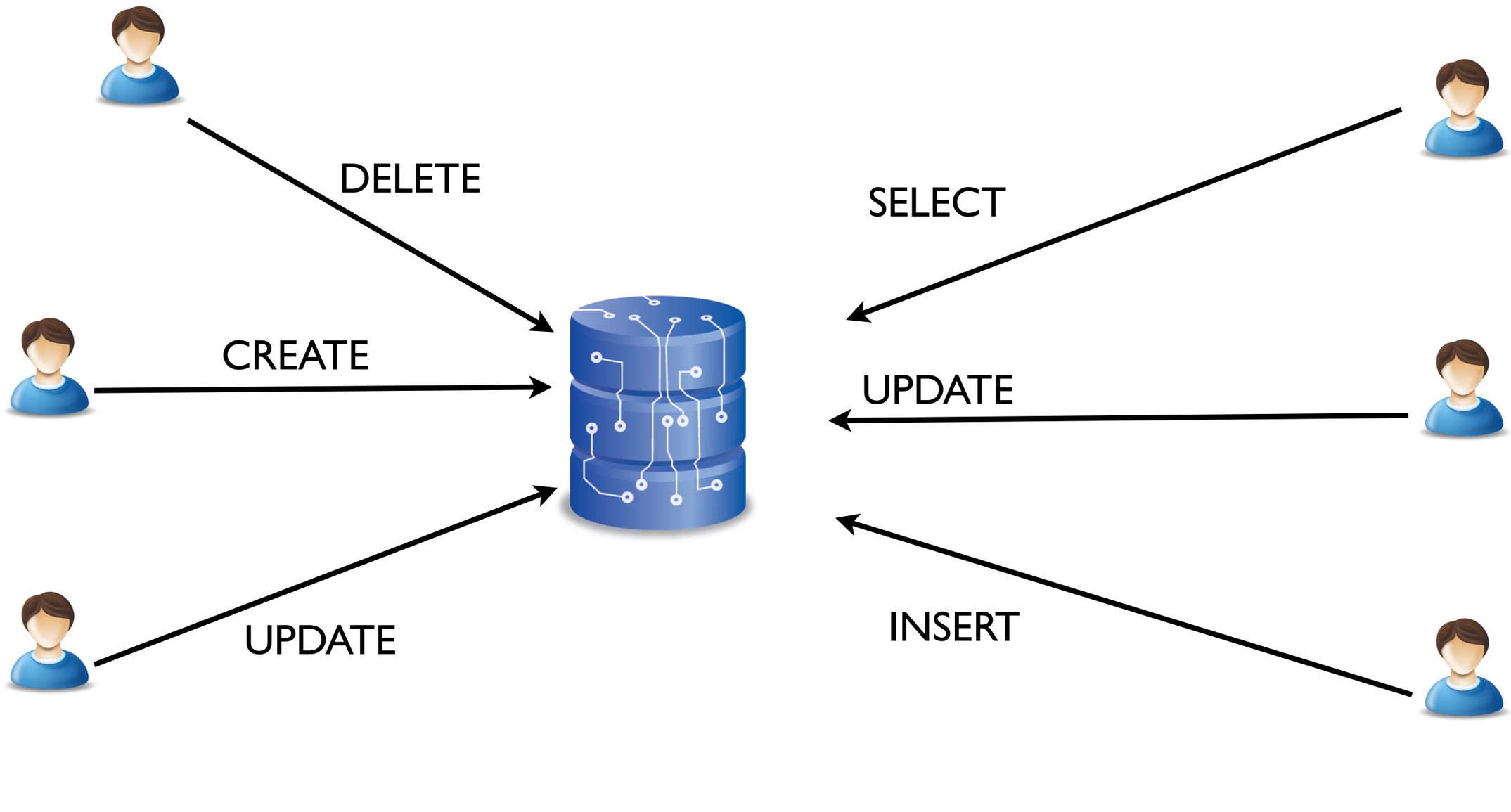
```
CREATE TRIGGER R1
AFTER INSERT ON T1
FOR EACH ROW
INSERT INTO T1 VALUES (2);

CREATE TRIGGER R2
AFTER INSERT ON T1
FOR EACH ROW
WHEN (EXISTS (SELECT A1 FROM T1 WHERE A1 = 2))
INSERT INTO T1 VALUES (3);
```

R1 -> R2 => insertion de 2,3

R2 -> R1 => insertion de 2

# Transactions dans SQL

# 1. Gestion des accès concurrents à la BD



# Accès concurrent à un attribut

Considérons les deux requêtes concurrentes:

UPDATE NOTE SET note = note + 1 WHERE idEtudiant = 1;

UPDATE NOTE SET note = note - 1 WHERE idEtudiant = 1;

Etat de la table NOTE avant la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 14 |

Etat de la table NOTE après la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 13 |

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 14 |

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 15 |

# Accès concurrent à une ligne

Considérons les deux requêtes concurrentes:

```
UPDATE NOTE SET note = 15 WHERE idEtudiant = 1;

UPDATE NOTE SET idMatiere= 4 WHERE idEtudiant = 1;
```

Etat de la table NOTE avant la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 0 |

Etats possibles de la table NOTE après la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 15 |

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 4 | 0 |

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 4 | 15 |

# Accès concurrent à une table

Considérons les deux requêtes concurrentes:

UPDATE NOTE SET note = note + 1 WHERE note >= 10;

UPDATE NOTE SET note = 10;

Etat de la table NOTE avant la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 0 |

Etats possibles de la table NOTE après la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 10 |

NOTE

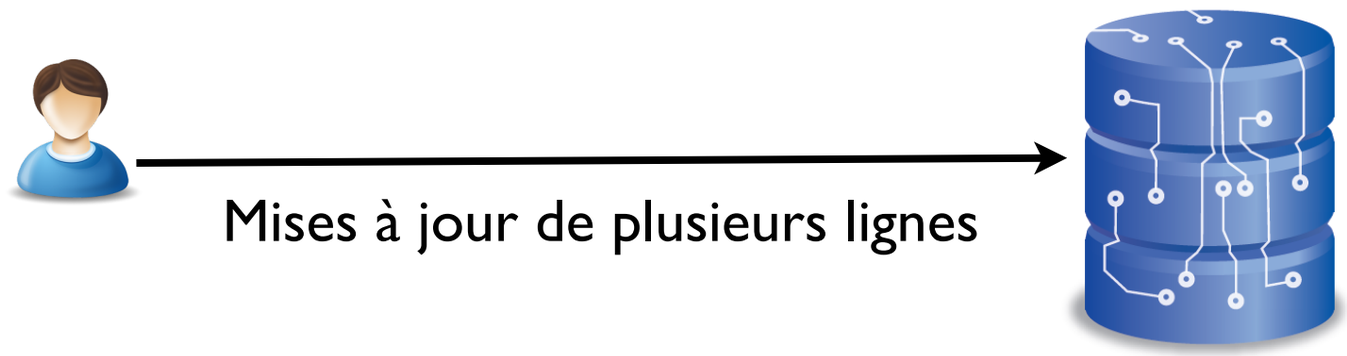
| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 11 |

# Pourquoi les transactions?

- Besoin de définir un mécanisme de contrôle de la concurrence
  - Permettre des appels concurrents
  - Exécuter les requêtes SQL d'un certain utilisateur de manière isolée
  - Eviter des résultats inattendus

## 2. Gestion de pannes

Exemple: Coupure de courant avant la mise à jour de toutes les lignes => Certaines lignes sont mises à jour, d'autres non.



# Pourquoi les transactions?

Besoin de définir un mécanisme qui va garantir soit l'exécution de toutes les requêtes, soit l'exécution d'aucune requête, même si le système tombe en panne.

# Transactions

Définition:

- Une transaction est une suite d'opérations qui sont exécutées indépendamment.
- Si une transaction est exécutée, ses opérations sont soit toutes exécutées, soit annulées.
- Une transaction doit être atomique, consistante, isolée et durable => Propriétés ACID.

# Transactions dans SQL

- Une transaction est une suite de requêtes SQL qui se termine avec la clause «COMMIT».
- Après le COMMIT, toutes les opérations sont exécutées.
- Quand «AUTO-COMMIT» est spécifié, chaque requête est considérée comme une transaction et est exécutée comme tel.
- Une transaction peut-être annulée par un «ROLLBACK»

# Transactions

## Exemple:

T1

```
INSERT INTO ETUDIANT VALUES (1,'Charles',40);
UPDATE ETUDIANT SET age = 20 WHERE idEtudiant = 1;
SELECT * FROM ETUDIANT;
COMMIT;
```

T2

```
DELETE FROM ETUDIANT WHERE idEtudiant = 1;
UPDATE ETUDIANT SET nom = 'Charlie' WHERE nom = 'Charles';
COMMIT;
```

T3

```
DELETE FROM ET
DELETE...
COMMIT;
```

# Propriétés ACID

Ensemble de propriétés qui garantissent qu'une transaction est exécutée de façon fiable

- A Atomicité
- C cohérence
- I isolation
- D durabilité

# Atomicité

- Les opérations d'une transaction sont soit toutes exécutées, soit toutes annulées.
- Avant le commit d'une transaction, si le système tombe en panne, les effets de la transaction ne sont pas pris en compte (Rollback)
- Le rollback peut être automatique ou initié par un utilisateur

# Cohérence

- Une transaction ne doit pas violer les contraintes d'intégrité

- Une transaction doit changer l'état de la bases de données, d'un état cohérent vers un état cohérent

# Isolation

Les transactions sont toujours exécutées selon un ordre équivalent à un ordre séquentiel des transactions

Exemple:

T1, T2 en concurrence T3, T4

Les ordres séquentiels possibles sont:

T1, T2, T3, T4 - T1, T3, T4, T2 - T3, T4, T1, T2

T1, T3, T2, T4 - T3, T1, T2, T4 - T3, T1, T4, T2

# Isolation - Accès concurrent à un attribut

Considérons les deux requêtes concurrentes:

(T1) UPDATE NOTE SET note = note + 1 WHERE idEtudiant = 1;

(T2) UPDATE NOTE SET note = note - 1 WHERE idEtudiant = 1;

Etat de la table NOTE avant la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 14 |

Etat de la table NOTE après la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 14 |

T1, T2 ou T2, T1 => Un seul état final

# Accès concurrent à une ligne

Considérons les deux requêtes concurrentes:

(T1) UPDATE NOTE SET note = 15 WHERE idEtudiant = 1;

(T2) UPDATE NOTE SET idMatiere= 4 WHERE idEtudiant = 1;

Etat de la table NOTE avant la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 0 |

Etat de la table NOTE après la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 4 | 15 |

T1, T2 ou T2, T1 => Un seul état final

# Accès concurrent à une table

Considérons les deux requêtes concurrentes:

(T1) UPDATE NOTE SET note = note + 1 WHERE note >= 10;

(T2) UPDATE NOTE SET note = 10;

Etat de la table NOTE avant la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 0 |

T1,T2

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 10 |

T2,T1

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 11 |

# Durabilité

- Après le commit d'une transaction, même si le système tombe en panne, les effets de la transaction sont enregistrées.

# Niveaux d’isolation

# Démo commit & rollback

Rappel:

Une transaction est une unité d'opérations qui s'exécutent en totalité (si commit), ou sont toutes annulées (si rollback)

# Pourquoi définir des niveaux d'isolation?

- Quand les transactions sont exécutées selon un ordre équivalent à un ordre séquentiel des transactions
  - Besoin de protocoles de verrouillage qui ralentit l'exécution
  - Réduction de la concurrence

# Niveaux d'isolation

- Relatif à une transaction
  - READ UNCOMMITTED (Lecture de données non validées)
  - READ COMMITTED (Lecture de données validées)
  - REPEATABLE READ (Lecture répétée)
  - SERIALIZABLE (Sérialisable)

# Lecture sale

Considérons les deux requêtes concurrentes:

(T1) UPDATE NOTE SET note = note + 1;

(T2) SELECT * FROM NOTE;

Etat de la table NOTE avant la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 0 |

Etat de la table NOTE avant le commit de T1:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 1 |

Les données sont sales «dirty»

Etat de la table NOTE si rollback de T1:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 0 |

Si T2 lit les informations de NOTE avant le commit de T1

=> Lecture sale

# READ UNCOMMITTED

«Une transaction peut effectuer une lecture sale»

(T1) UPDATE NOTE SET note = note + 1;

(T2) SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT * FROM NOTE;

Etat de la table NOTE avant la mise à jour:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 0 |

Etat de la table NOTE avant le commit de T1:

T2 peut lire les informations de la table NOTE

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 5 | 1 |

# READ COMMITTED

«Une transaction ne peut pas effectuer de lecture sale»

(T1) UPDATE NOTE SET note = note + 1;

(T2) SET TRANSACTION ISOLATION LEVEL READ COMMITTED

SELECT avg(note) FROM NOTE;

SELECT max(note) FROM NOTE;

Etat de la table NOTE avant l'exécution de T1:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 2 | 14 |
| 1 | 5 | 10 |

Etat de la table NOTE après le commit de T1:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 2 | 15 |
| 1 | 5 | 11 |

Exécution possible non équivalente à T1;T2 ou T2;T1 => Inconsistance

SELECT avg(note) FROM NOTE; T1; SELECT max(note) FROM NOTE

=> Lecture non reproductible

# REPEATABLE READ

« 1. Une transaction ne peut pas effectuer de lecture sale

2. Les données lues à plusieurs reprises ne peuvent pas être modifiées»

(T1) UPDATE NOTE SET note = note + 1;

(T2) SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

SELECT avg(note) FROM NOTE;

SELECT max(note) FROM NOTE;

Etat de la table NOTE avant l'exécution de T1:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 2 | 14 |
| 1 | 5 | 10 |

Etat de la table NOTE après le commit de T1:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 2 | 15 |
| 1 | 5 | 11 |

Exécutions possibles: T1;T2 ou T2;T1

# REPEATABLE READ

(T1) UPDATE NOTE SET note = note + 1; UPDATE NOTE SET idMatiere = idMatiere + 1;

(T2) SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

SELECT avg(note) FROM NOTE; SELECT min(idMatiere) FROM NOTE;

Etat de la table NOTE avant l'exécution de T1:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 2 | 14 |
| 1 | 5 | 10 |

Etat de la table NOTE après le commit de T1:

NOTE

| idEtudiant | idMatiere | note |
|---|---|---|
| 1 | 3 | 15 |
| 1 | 6 | 11 |

Exécution possible non équivalente à T1;T2 ou T2;T1 => Inconsistance

SELECT avg(note) FROM NOTE; T1; SELECT min(idMatiere) FROM NOTE

# REPEATABLE READ (Lecture fantôme)

(T1) INSERT INTO NOTE VALUES(...); (Données fantômes)

(T2) SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

SELECT avg(note) FROM NOTE; SELECT avg(note) FROM NOTE;

Même si les données lues ne sont pas modifiées, l'insertion de nouvelles lignes est possible et peut engendrer des inconsistances.

SELECT avg(note) FROM NOTE; T1; SELECT avg(note) FROM NOTE

=> Différent de T1;T2 ou T2;T1

# Niveaux d'isolation

- Concurrence élevée mais possibilité d'inconsistances
- Par défaut, serialisable ou repeatable read (MySQL & Oracle)