# Administration de Bases de Données

## Les Transactions

SKLAB Youcef

May 7, 2017

# Les Transactions

# Les Transactions

# Les **Transactions**

## Definition - **État cohérent**

Une base de données est cohérente (ou dans un état cohérent) lorsque toutes les données qu'elle contient sont en accord avec **les contraintes d'intégrité** du schéma conceptuel.

## Definition - **État correct**

Une base de données est dans un état correct lorsqu'elle est dans un **état cohérent** et que les valeurs des **données reflètent exactement le résultat attendu des modifications effectuées**.

# Cohérence et correction - Propriétés

## Propriété 1

Dans un SGBD relationnel, toute requête conserve la cohérence d'une base de données. Si elle s'exécute avec succès, elle transforme l'état cohérent en un nouvel état cohérent; si elle échoue (erreur de syntaxe, violation de contrainte, bug SGBD, arrêt machine . . . ), elle laisse la base dans son état cohérent.

## Propriété 2

Dans un SGBD relationnel, les requêtes (même en cas de succès) ne conservent pas forcément la correction d'une base de données.

# Cohérence et correction - Exemple

⇒ Deux requêtes correctes:

## Requête 1

```
CREATE TABLE Client (
nom char (30),
compte1 integer,
compte2 integer,
PRIMARY KEY (nom)
);
```

## Requête 2

```
CREATE TABLE Compte (
numero integer,
valeur integer,
PRIMARY KEY ( numero )
);
```

# Cohérence et correction - Manipulation des données

## Requêtes d'insertion

**INSERT INTO** Compte **VALUES** (1 , 4000);
**INSERT INTO** Compte **VALUES** (2 , 300);

## Requêtes: *Transférer du compte 1 vers le compte 2 la somme 2000 DA*

- **UPDATE** Compte **SET** valeur = valeur + 2000 **WHERE** numero = 2;

1. Est ce que la base de données est dans un état cohérent?
2. Est ce que la base de données est dans un état correct?

# Cohérence et correction - Manipulation des données

## Requêtes d'insertion

**INSERT INTO** Compte **VALUES** (1 , 4000);
**INSERT INTO** Compte **VALUES** (2 , 300);

## Requêtes: *Transférer du compte 1 vers le compte 2 la somme 2000 DA*

- **UPDATE** Compte **SET** valeur = valeur + 2000 **WHERE** numero = 2;

1. Est ce que la base de données est dans un état cohérent?
2. Est ce que la base de données est dans un état correct?

- **UPDATE** Compte **SET** valeur = valeur - 2000 **WHERE** numero = 1;

# Cohérence et correction - Manipulation des données

## Requêtes d'insertion

**INSERT INTO** Compte **VALUES** (1 , 4000);
**INSERT INTO** Compte **VALUES** (2 , 300);

## Requêtes: *Transférer du compte 1 vers le compte 2 la somme 2000 DA*

- **UPDATE** Compte **SET** valeur = valeur + 2000 **WHERE** numero = 2;

1. Est ce que la base de données est dans un état cohérent?
2. Est ce que la base de données est dans un état correct?

- **UPDATE** Compte **SET** valeur = valeur - 2000 **WHERE** numero = 1;

⇒ **Entre les deux mises à jour, la base de données est cohérente mais non correcte.**

# Cohérence et correction -Manipulation des données

■Deux Problèmes:

⇒ De multiples utilisateurs doivent pouvoir accéder à la base de donnée en même temps → **problème d'accès concurrents.**

⇒ De nombreuses et diverses pannes peuvent apparaître. Il ne faut pas perdre les données → **problème de perte de données suite à une panne**

# Cohérence et correction -Manipulation des données

■Deux Problèmes:

⇒ De multiples utilisateurs doivent pouvoir accéder à la base de donnée en même temps → **problème d'accès concurrents.**

⇒ De nombreuses et diverses pannes peuvent apparaître. Il ne faut pas perdre les données → **problème de perte de données suite à une panne**

⇒ **La Gestion de transactions répond à ces problèmes**

# Transactions

## Définition

Une transaction est une suite ordonnée de requêtes qui se termine de deux façons possibles:

1. **Succès**: Toutes les requêtes de la liste se sont exécutées avec succès. Le SGDB exécute une instruction **COMMIT** qui valide la transaction.
2. **Échec**: Au moins une requête de la liste ne s'est pas exécutée avec succès. Le SGDB exécute une instruction **ROLLBACK** qui invalide la transaction et remet la base dans l'état du début de la transaction.

# Transactions - **Propriétés**

Un système de gestion de transactions doit garantir les propriétés suivantes (**ACID**)

1. **Atomicité**: Une transaction doit effectuer toutes ses mises à jour avec succès; sinon ne rien faire du tout.

# Transactions - **Propriétés**

Un système de gestion de transactions doit garantir les propriétés suivantes (**ACID**)

1. **Atomicité**: Une transaction doit effectuer toutes ses mises à jour avec succès; sinon ne rien faire du tout.
2. **Cohérence**: La transaction doit faire passer la base de données d'un état cohérent à un autre.

# Transactions - **Propriétés**

Un système de gestion de transactions doit garantir les propriétés suivantes (**ACID**)

1. **Atomicité**: Une transaction doit effectuer toutes ses mises à jour avec succès; sinon ne rien faire du tout.
2. **Cohérence**: La transaction doit faire passer la base de données d'un état cohérent à un autre.
3. **Isolation**: Les résultats d'une transaction ne doivent être visibles aux autres transactions qu'une fois la transaction validée.

# Transactions - **Propriétés**

Un système de gestion de transactions doit garantir les propriétés suivantes (**ACID**)

1. **Atomicité**: Une transaction doit effectuer toutes ses mises à jour avec succès; sinon ne rien faire du tout.
2. **Cohérence**: La transaction doit faire passer la base de données d'un état cohérent à un autre.
3. **Isolation**: Les résultats d'une transaction ne doivent être visibles aux autres transactions qu'une fois la transaction validée.
4. **Durabilité**: Dès qu'une transaction valide ses modifications, le système doit garantir que ces modifications seront conservées en cas de panne.

# Les Transactions - **Les ordres** Transact SQL

1. **BEGIN TRAN**: début de la transaction
2. **SAVE TRAN**: définition des points d'arrêt.
3. **COMMIT TRAN**: fin avec succès,
4. **ROLLBACK TRAN**: fin avec échec,

# Les ordres Transact SQL - BEGIN TRAN

## BEGIN TRAN[SACTION]

1. Démarrer de façon explicite une transaction.
2. En l'absence de cette commande, toute instruction SQL est une transaction implicite qui est validée aussitôt la modification effectuée sur les données.
3. Lors du début de la transaction, il est possible de nommer la transaction, mais aussi de marquer le début de la transaction dans le journal de la base de données.
4. Cette marque pourra être exploitée lors d'un processus de restauration des données

# Les ordres Transact SQL - **BEGIN TRAN**

## BEGIN TRAN[SACTION]

BEGIN { TRAN | TRANSACTION } [nomTransaction]

## Exemple 1

**SELECT COUNT(*) AS** NbreClients **FROM** Clients

$\Rightarrow$ Dans cet exemple, une nouvelle transaction est démarrée.

## Exemple 2

**BEGIN TRAN** Tr1
**SELECT COUNT(*) AS** NbreClients **FROM** Clients

$\Longrightarrow$ Résultat: ***NbreClients = 200***

# Les ordres Transact SQL - **SAVE TRAN**

## SAVE TRAN

1. Cette instruction permet de définir des points d'arrêt et donc donne la possibilité d'annuler une partie de la transaction en cours.
2. Un point d'arrêt est identifié par un nom.
3. Il est possible de définir plusieurs points d'arrêt sur une même transaction.
4. Il est possible d'annuler les modifications d'une transaction jusqu'à un point d'arrêt précis.

# Les ordres Transact SQL - **SAVE TRAN**

## SAVE TRAN

**SAVE** { **TRAN** | **TRANSACTION** } nomPointArret[;]

## Exemple

**BEGIN TRAN** Tr1
–Ajouter un client
**INSERT INTO** clients(nom, prenom, telephone) **VALUES** ('Ali', 'Bal', '021364897');
– Poser un point d'arrêt
**SAVE TRAN** P1;
– Compter les clients
**SELECT COUNT(*) AS** NbreClients **FROM** clients;

$\Longrightarrow$ Résultat: ***NbreClients = 201***

# Les ordres Transact SQL - **ROLLBACK TRAN**

## ROLLBACK TRAN[SACTION]

1. **ROLLBACK** permet d'annuler une partie ou la totalité de la transaction (les modifications intervenues sur les données).
2. L'annulation partielle d'une transaction n'est possible que si des points d'arrêt ont été définis à l'aide de l'instruction **SAVE TRAN**.
3. Il n'est pas possible d'arrêter l'annulation entre deux points d'arrêt.

## ROLLBACK - Syntaxe

**ROLLBACK** { **TRAN** | **TRANSACTION** }
[nomTransaction|nomPointArret][;]

# Les ordres Transact SQL - ROLLBACK TRAN

## ROLLBACK- Exemple (suite Tr1)

**BEGIN TRAN** Tr1
**INSERT INTO** clients **VALUES** ('Ali', 'Bal', '021364897');
**SAVE TRAN** P1; - - *Poser un point d'arrêt*
**SELECT COUNT(*) AS** NbreClients **FROM** clients; $\Longrightarrow$ Résultat: ***NbreClients = 201***
**DELETE FROM** clients **WHERE** numero>=0;
**SELECT COUNT(*) AS** NbreClients **FROM** clients; $\Longrightarrow$ Résultat: ***NbreClients = 0***
**ROLLBACK TRAN** P1; - - *Annuler la suppression*
**SELECT COUNT(*)** NbreClients **FROM** clients; $\Longrightarrow$ Résultat: ***NbreClients = 201***

$\Rightarrow$ Tous les clients sont supprimés, puis une requête compte le nombre de clients dans la base. En fin, la suppression est annulée par l'instruction **ROLLBACK** qui annule toutes les modifications effectuées depuis le point d'arrêt P1.

# Les ordres Transact SQL - **COMMIT**

## COMMIT

1. Cette instruction permet de mettre fin avec succès à une transaction (de conserver l'ensemble des modifications effectuées dans la transaction)
2. Les modifications sont visibles par les autres utilisateurs de la base de données à l'issue de la transaction.

# Les ordres Transact SQL - **COMMIT**

## COMMIT

**COMMIT** { **TRAN** | **TRANSACTION** } [nomTransaction] [;]

## Exemple

– Valider le reste de la transaction

**COMMIT TRAN** Tr1;

–Compter les clients

**SELECT COUNT(*) AS** NbreClients **FROM** clients;

$\Longrightarrow$ Les modifications sont validées et la transaction prend fin.

$\Longrightarrow$ Résultat: ***NbreClients = 201***

# Les ordres Transact SQL - **L'exemple Complet**

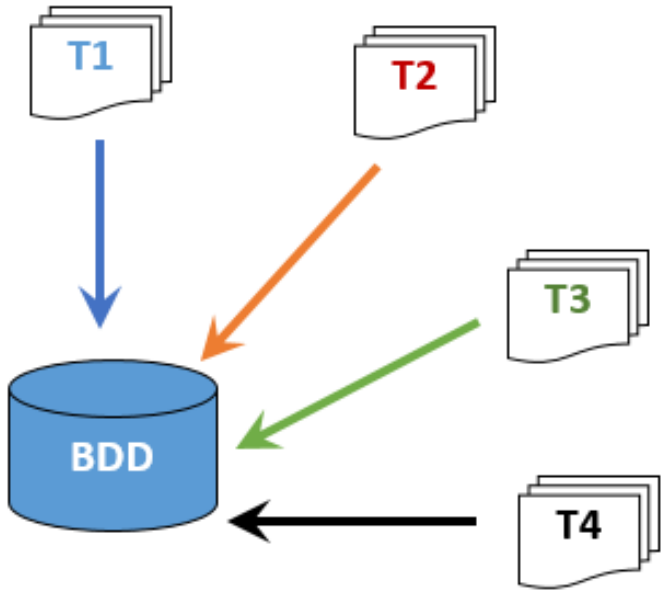
## Exemple - Transaction

**BEGIN TRAN** Tr1
–Ajouter un client
**INSERT INTO** clients(nom, prenom, telephone) **VALUES** ('Ali', 'Bal', '021364897');
– Poser un point d'arrêt
**SAVE TRAN** P1;
– Compter les clients
**SELECT COUNT(*) AS** NbreClients **FROM** clients; – $\Rightarrow$ *NbreClients = 201*
**DELETE FROM** clients **WHERE** numero>=0;
**SELECT COUNT(*) AS** NbreClients **FROM** clients; – $\Rightarrow$ *NbreClients = 0*
–Annuler la suppression
**ROLLBACK TRAN** P1;
**SELECT COUNT(*)** NbreClients **FROM** clients; – $\Rightarrow$ *NbreClients = 201*
– Valider le reste de la transaction
**COMMIT TRAN** Tr1;
–Compter les clients
**SELECT COUNT(*) AS** NbreClients **FROM** clients; – $\Rightarrow$ *NbreClients = 201*

# Les Transactions - Problème des **accès concurrents**

## Concurrence d'accès

- Plusieurs utilisateurs peuvent lancer des transactions en même temps $\Rightarrow$ **Concurrence d'accès**.
- Des transactions exécutées concurremment peuvent interférer et mettre la base de données dans un état incorrect.



# Gestion des Accès concurrents: **Granule, Action et Exécution**

1. **Granule de concurrence**: unité de données dont les accès sont contrôlés individuellement par le SGBD (Exemple: un tuple ou bien une table).
2. **Action**: Un accès élémentaire à un granule. C'est une unité indivisible par le SGBD sur un granule pour un utilisateur (lire, écrire, modifier, supprimer et insérer).
3. **Exécution**: Une suite d'actions. C'est une séquence d'actions obtenues en intercalant les diverses actions des transactions tout en respectant l'ordre interne des actions de chaque transaction.
   - **Exemple**: $E_1 = Lire(A), Lire(B), Ecrire(A), Ecrire(B)$

# Gestion des Accès concurrents: **Succession et sérialisation**

1. **Succession**: Une succession de transactions est une exécution E d'un ensemble de transactions (T1,T2,..,Tn) tel qu'il existe une permutation p de (1,2,..n) tel que :
   - E = <Tp(1); Tp(2); ... ;Tp(n)>
2. **Sérialisation**: Exécution sérialisable: une exécution E d'un ensemble de transactions (T1,T2,..,Tn) donnant globalement et pour chaque transaction participante le même résultat qu'une succession de (T1,T2,..,Tn).

# Accès concurrents: **Succession et sérialisation** - Exemple

⇒ Succession: T1,T2

- T1:
  - (1): A ← A+1
  - (2): A ← A*2
- T2:
  - (3): B ← B+1
  - (4): B ← B*2

⇒ (2) et (3) sont permutables car elles n'agissent pas sur le même granule.

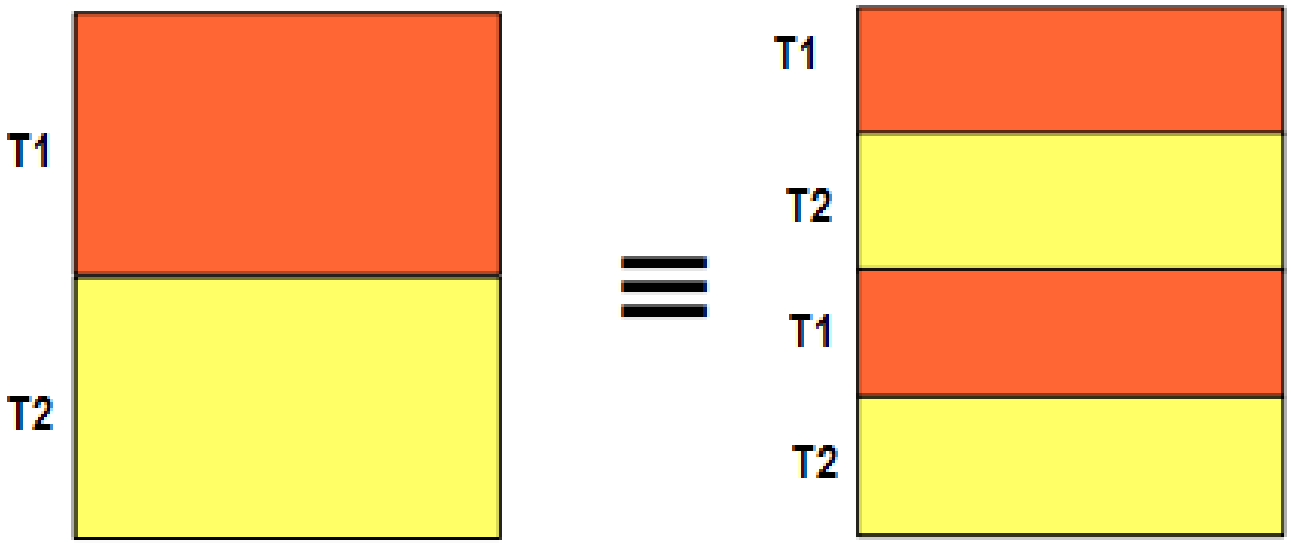
On peut transformer en:

1. T1: A ← A+1
2. T2: B ← B+1
3. T1: A ← A*2
4. T2: B ← B*2

⇒ Cette exécution est sérialisable.

# Accès concurrents: **Succession et sérialisation** - Exemple

1. Sérialisation des transactions: L'état final d'une BD après exécutions en parallèle de transactions doit être identique à une exécution en série des transactions.




⇒ Le problème du contrôle de concurrence consiste pour le système de ne générer que des exécutions sérialisables.

# Les Transactions - Problème des **accès concurrents**

## Exemple - Problème des **accès concurrents**

- Considérons deux transactions T1 et T2 qui s'intéressent à un même objet A.
- Les deux seules opérations possibles sur A, sont:
  1. **Lecture**
  2. **Ecriture**.
- 04 cas possibles:
  1. **Lecture - Lecture**
  2. **Ecriture - Ecriture**
  3. **Ecriture - Lecture**
  4. **Lecture - Ecriture**

# Problème des accès concurrents: **Lecture - Lecture**

✓ **Lecture - Lecture**

- Aucun conflit.
- Un même objet peut toujours être partagé en lecture

# Problème des accès concurrents: **Ecriture - Ecriture**

## ⊗ **Perte de données**: Ecriture - Ecriture

- T2 peut **écraser** la valeur de A, par une autre écriture, celle effectuée par T1 (perte de données).

| Temps | Transaction T1 | Etat de la base | Transaction T2 |
|---|---|---|---|
| t1 | lire A | A = 10 | - |
| t2 | - | | lire A |
| t3 | A := A + 10 | | - |
| t4 | - | | - |
| t5 | - | | A := A + 50 |
| t6 | écrire A | A = 20 | - |
| t7 | - | A = 60 | écrire A |

# Problème des accès concurrents: **Ecriture - Lecture**

## ⊗ **Lecture impropre**: Ecriture - Lecture

- Une transaction lit des données écrites par une transaction concurrente non validée.
- T2 lit une valeur modifiée par T1 et ensuite T1 est annulée.

| Temps | Transaction T1 | Etat de la base | Transaction T2 |
|---|---|---|---|
| t1 | lire A | A=10 | - |
| t2 | A := A+ 20 | | - |
| t3 | écrire A | A = 30 | - |
| t4 | - | | lire A |
| t5 | **annulation** | | |
| t6 | - | | - |

# Problème des accès concurrents: **Lecture - Ecriture**

## ⊗ **Lecture non reproductible**: Lecture - Ecriture

- Une transaction relit des données qu'elle a lu précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale).
- T1 modifie la valeur de A entre deux lectures de T2.

| Temps | Transaction T1 | Etat de la base | Transaction T2 |
|---|---|---|---|
| t1 | lire A | A=10 | - |
| t2 | - | | lire A |
| t3 | A := A + 10 | | - |
| t4 | écrire A | A = 20 | - |
| t5 | - | | - |
| t6 | - | | lire A |

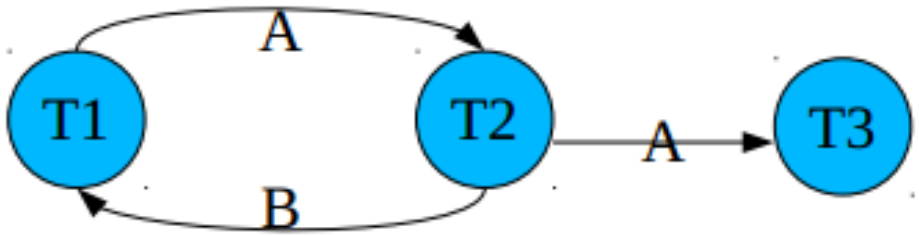
# Accès concurrents: **Précédence**

1. Une exécution sérialisable est correcte car elle donne le même résultat qu'une exécution en série des transactions.
2. Mais, il n'est pas toujours possible d'avoir une exécution sérialisable (ex: non commutativité de l'addition et multiplication).
3. **Précédence** : propriété indiquant qu'une transaction a accompli une opération Oi sur une donnée avant qu'une autre transaction réalise une opération Oj. Oi et Oj n'étant pas commutatives (Oi;Oj$\neq$Oj;Oi).

# Accès concurrents: **Graphe de précédence**

1. La notion de précédence de transactions peut être représentée par un graphe.
2. Un graphe dont les noeuds représentent les transactions et dans lequel il existe un arc Ti vers Tj si Ti précède Tj dans l'exécution analysée.

{T1:Lire A;
T2: Ecrire A;
T2: Lire B;
T3: Lire A;
T1: Ecrire B}




⇒ Condition suffisante de sérialisabilité: le graphe de précédence est sans circuit.

# Accès concurrents: **Solution**

## Gestion des accès concurrents

- De nombreuses solutions ont été proposées pour traiter le problème des accès concurrents.
- 2 principales techniques pour garantir la sérialisabilité des transactions:
  - Prévention de conflits, basée sur le **verrouillage**
  - Détection des conflits, basée sur estampillage

# Les Transactions - **Verrouillage**

## Principes

1. Le verrouillage est la technique la plus classique pour résoudre les problèmes dus à la concurrence:

# Les Transactions - **Verrouillage**

## Principes

1. Le verrouillage est la technique la plus classique pour résoudre les problèmes dus à la concurrence:
   - Avant de lire ou écrire une donnée une transaction peut demander un verrou sur cette donnée pour interdire aux autres transactions d'y accéder.

# Les Transactions - **Verrouillage**

## Principes

1. Le verrouillage est la technique la plus classique pour résoudre les problèmes dus à la concurrence:
   - Avant de lire ou écrire une donnée une transaction peut demander un verrou sur cette donnée pour interdire aux autres transactions d'y accéder.
   - Le verrouillage est effectué automatiquement.

# Les Transactions - **Verrouillage**

## Principes

1. Le verrouillage est la technique la plus classique pour résoudre les problèmes dus à la concurrence:
   - Avant de lire ou écrire une donnée une transaction peut demander un verrou sur cette donnée pour interdire aux autres transactions d'y accéder.
   - Le verrouillage est effectué automatiquement.
   - Si ce verrou ne peut être obtenu, parce qu'une autre transaction en possède un, la transaction demandeuse est mise en attente.

# Les Transactions - **Verrouillage**

## Principes

1. Le verrouillage est la technique la plus classique pour résoudre les problèmes dus à la concurrence:
   - Avant de lire ou écrire une donnée une transaction peut demander un verrou sur cette donnée pour interdire aux autres transactions d'y accéder.
   - Le verrouillage est effectué automatiquement.
   - Si ce verrou ne peut être obtenu, parce qu'une autre transaction en possède un, la transaction demandeuse est mise en attente.
2. Afin de limiter les temps d'attente, on peut jouer sur:
   - La granularité du verrouillage : pour restreindre la taille de la donnée verrouillée (n-uplet, une table)

# Les Transactions - **Verrouillage**

## Principes

1. Le verrouillage est la technique la plus classique pour résoudre les problèmes dus à la concurrence:
   - Avant de lire ou écrire une donnée une transaction peut demander un verrou sur cette donnée pour interdire aux autres transactions d'y accéder.
   - Le verrouillage est effectué automatiquement.
   - Si ce verrou ne peut être obtenu, parce qu'une autre transaction en possède un, la transaction demandeuse est mise en attente.
2. Afin de limiter les temps d'attente, on peut jouer sur:
   - La granularité du verrouillage : pour restreindre la taille de la donnée verrouillée (n-uplet, une table)
   - Le mode de verrouillage: pour restreindre les opérations interdites sur la donnée verrouillée

# Les Transactions - **Verrouillage**

## Actions

1. Il repose sur les deux actions:
   - **verrouiller (A)**: acquérir un contrôle de l'objet A
   - **libérer (A)**: libérer l'objet A
2. Deux types de verrous:
   - Verrous exclusifs **(X locks)** ou verrous d'écriture
   - Verrous partagés **(S locks)** ou verrous de lecture

# Verrouillage - Exemple

| T1 | T2 | Résultat A=10 |
|---|---|---|
| Read A avec verrou | | |
| A=A+10 | Read A avec verrou | |
| | attente | A=20 |
| Write A<br>Commit; | | A=20 |
| | Read A avec verrou | 20 |
| | A=A+50 | 50 |
| | Write A<br>commit | A=70 |

# Les Transactions -**S-lock** et **X-lock**

- **S-lock**: Lecture partagée.
- **X-lock**: Écriture exclusive.

| Verrou demandé | Verrou déjà accordé pour une autre transaction | |
|---|---|---|
| | **S-lock** | **X-lock** |
| **S-lock** | Accordé | Attente de libération |
| **X-lock** | Attente de libération | Attente de libération |

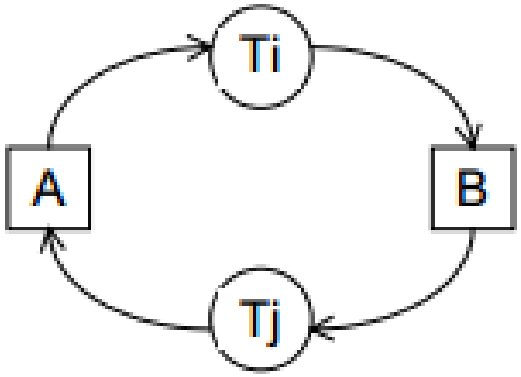
# Les Transactions – **Verrouillage**

## Protocole d'accès aux données

1. Aucune transaction ne peut effectuer une lecture ou une mise à jour d'un objet si elle n'a pas acquis au préalable un verrou S ou X sur cet objet
2. Si une transaction a un verrou S sur un granule, elle peut demander le verrou X (upgrade).
3. Si une transaction ne peut obtenir un verrou déjà détenu par une autre transaction T2, alors elle doit attendre jusqu'à ce que le verrou soit libéré par T2.
4. Les verrous X sont conservés jusqu'à la fin de la transaction (**COMMIT** ou **ROLLBACK**)
5. En général les verrous S sont également conservés jusqu'à cette date

# Les Transactions - **Phénomèmes indésirables**

1. **La privation**:
   - Une transaction risque d'attendre un objet indéfiniment si à chaque fois que cet objet est libéré, il est pris par une autre transaction.
   - Pour traiter ce problème, on peut organiser sur chaque verrou une file d'attente avec une politique "première arrivée", "première servie".
2. **L'interblocage (ou verrou mortel)**: Ti attend Tj , Tj attend Ti : il y a interblocage.



# Les Transactions - **Interblocage**

## Traiter le problème d'interblocage

1. **Prévention des interblocages**:
   - Lorsqu'une demande d'acquisition de verrou ne peut être satisfaite on fait passer un test aux deux transactions impliquées, à savoir celle qui demande le verrou, Ti, et celle qui le posséde déjà, Tj.
   - Si Ti et Tj passent le test alors Ti est autorisé à attendre Tj, sinon l'une des deux transactions est annulée pour être relancée par la suite.
2. **Détection des interblocages**:
   - Les interblocages sont détectés en construisant le graphe "qui attend quoi" et en y recherchant les cycles.
   - Lorsqu'un cycle est découvert l'une des transactions est choisie comme victime, elle est annulée de manière à faire disparaître le cycle.

# Les Transactions - **Verrouillage à deux phases**

⇒ Technique de contrôle des accès concurrents consistant à verrouiller les objets au fur et à mesure des accès par une transaction et à relâcher les verrous seulement après obtention de tous les verrous.

## Protocole

1. Pour chaque transaction, tous les verrouillages doivent précéder toutes les libérations de verrous.
2. Après l'abandon d'un verrou, une transaction ne doit plus jamais pouvoir obtenir de verrous
3. On distingue deux phases:
   1. acquisition des verrous
   2. libération des verrous

# Les Transactions - **Verrouillage à deux phases** - Exercice

## Exercice

- Soit l'exécution: L1(y), L2(z), E1(z), R1, E2(y), R2. Avec L: Lire, E: Écrire, R: Relâcher Verrou.
- Faire le déroulement avec un ordonnanceur utilisant un système de verrouillage à deux phases (S-Lock et X-Lock)

# Les Transactions - **Verrouillage à deux phases** - Exercice

## Exercice

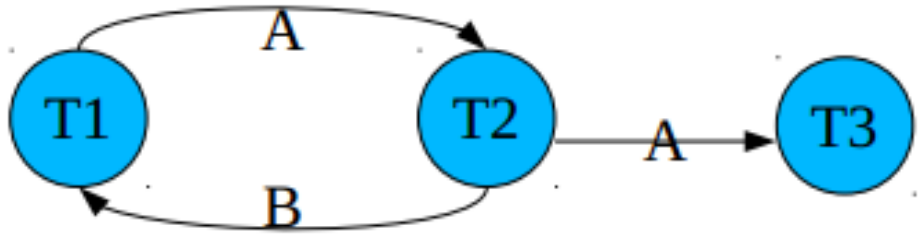
- Soit l'exécution: L1(y), L2(z), E1(z), R1, E2(y), R2. Avec L: Lire, E: Écrire, R: Relâcher Verrou.
- Faire le déroulement avec un ordonnanceur utilisant un système de verrouillage à deux phases (S-Lock et X-Lock)

| Temps | T1 | T2 |
|---|---|---|
| $t_1$ | L1(y) s'exécute avec un verrou S(y) | |
| $t_2$ | | L2(z) s'exécute avec un verrou S(z) |
| $t_3$ | E1(z) est bloquée par L2(z) pour avoir X(z) | |
| $t_4$ | | E2(y) est bloquée par L1(y) pour avoir X(y) |

# Verrouillage à deux phases: Exercice

Est-ce que l'exécution ci-dessous est sérialisable? Justifier (déroulement)?
(*L'ordonnanceur utilise un système de verrouillage à deux phases*)

{T1:Lire A;
T2: Ecrire A;
T2: Lire B;
T3: Lire A;
T1: Ecrire B}



# Fin...!