

BDay-MI

Bases de données avancées

Cours de Cristina Sirangelo

IRIF, Université Paris Diderot

Assuré en 2021-2022 par Amélie Gheerbrant

amelie@irif.fr

Transactions et concurrence

Sources (quelques slides empruntés et réadaptés) :

- MOOC DB, B. Nguyen, U. d'Orléans
- Cours *database systems principles* - V. Vianu, UCSD, Californie
- cours *Database systems principles* - H.G. Molina, Stanford Univ.
- slides du livre *Database systems concepts* -
A. Silberschatz, Yale U. & H. Korth, Lehigh U. & S.Sudarshan, IIT Bombay

Transactions concurrentes

- Hypothèse initiale : absence de pannes
- Comment garantir une exécution concurrente de transactions
 - ▶ qui préserve la cohérence de la base de données - **cohérence**
 - ▶ qui permet à chaque transaction de s'exécuter correctement (i.e comme si elle était seule) - **isolation**

Un modèle simplifié de transaction

- Une transaction comporte
 - ▶ une suite d'opérations de lecture / modification de la BD
 - ▶ intercalées avec des opérations qui manipulent uniquement la mémoire locale de la transaction

Transaction de fermeture du compte de Bob avec transfert sur le compte de Alice

```
SELECT solde INTO x FROM Compte  
WHERE client="Alice"
```

```
SELECT solde INTO y FROM Compte  
WHERE client="Bob"
```

```
z := x+y
```

```
UPDATE Compte  
SET solde = z  
WHERE client="Alice"
```

```
UPDATE Compte  
SET solde = 0  
WHERE client="Bob";
```

Un modèle simplifié de transaction

- Abstractions des opérations d'une transaction:
 - ▶ **Read(A, x)**
lit l'élément A de la BD et le copie dans la mémoire locale de la transaction (variable locale x)
 - ▶ **Write (A,x)**
copie la valeur de la variable locale x dans l'élément A de la BD
 - ▶ **z := expression**
opération dans la mémoire locale de la transaction
- “Élément” de la BD : l'unité sur laquelle l'accès est contrôlé
 - ▶ typiquement : n-uplet
 - ▶ mais possiblement : table, block du disque, ...
 - ▶ ce qu'on va dire s'applique n'importe quelle notion de “element”

Un modèle simplifié de transaction

Abstraction de la transaction

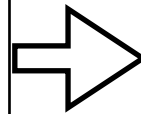
```
SELECT solde INTO x FROM Compte  
WHERE client="Alice"
```

```
SELECT solde INTO y FROM Compte  
WHERE client="Bob"
```

```
z := x+y
```

```
UPDATE Compte  
SET solde = t  
WHERE client="Alice"
```

```
UPDATE Compte  
SET solde = 0  
WHERE client="Bob";
```



```
READ(A, a)
```

```
READ(B, b)
```

```
z:= a.solde + b.solde
```

```
a.solde := z  
WRITE (A, a)
```

```
b.solde := 0  
WRITE (B, b)
```

A : n-uplet pour le compte de "Alice"

B : n-uplet pour le compte de "Bob"

a, b : variables locales

- pour simplifier :
dans la suite un n-uplet sera supposé constitué d'une seule valeur (entière)

Exécution concurrente de transactions

- Exemple d'exécution concurrente : considérer les deux transactions

T1: Read(A, a)
 a := a+100
 Write(A, a)
 Read(B, b)
 b := b+100
 Write(B, b)

T2: Read(A, a)
 a := a×2
 Write(A, a)
 Read(B, b)
 b := b×2
 Write(B, b)

Contrainte dans la BD : A=B

- On suppose que T1 et T2 sont exécutées indépendamment en parallèle (e.g. sur deux sites différents)
- Toutefois T1 et T2 opèrent sur la même base de données : les demandes de Read et Write de T1 et T2 arrivent au SGBD dans un certain ordre
- Le SGBD n'a aucun contrôle sur cet ordre, en revanche il peut contrôler l'ordre dans lequel il accepte ces demandes
- L'ordre dans lequel ces demandes sont exécutées est appelé *ordonnancement*

Exécution concurrente de transactions

- Ordonnancement (*schedule*) de T1 et T2

T1

Read(A, a); a := a+100

Write(A, a);

Read(B, b); b := b+100;

Write(B, b);

T2

Read(A, a); a := a×2;

Write(A, a);

Read(B, b); b := b×2;

Write(B, b);

Exécution concurrente de transactions

- Rappel : avec un **ordonnancement arbitraire**
 - ▶ **problèmes de cohérence** : la BD peut être laissée dans un état incohérent (e.g solde négatif, plusieurs réservations pour une même place dans l'avion, ...)
 - ▶ **problèmes d'isolation** : comportement incorrect des transactions (e.g des mises à jour perdues, lectures non-reproductibles etc.)

Idée principale du contrôle de la concurrence

1. Un ordonnancement *sériel* garantit cohérence et isolation (et donc correction)

Sériel : T1 ; T2 ou T2 ; T1

2. Un ordonnancement qui est équivalent à une exécution sérielle a aussi les mêmes propriétés

deux ordonnancements sont **équivalents** si pour tout état initial de la BD, ils produisent le même état final de la BD

Un ordonnancement équivalent à une exécution sérielle est dit *Sérialisable*

Objectif principal du contrôle de la concurrence :
garantir la sérialisabilité

Ordonnancement sériel SI (“ok” par définition)

► contrainte : $A=B$

| T1 | T2 | A | B dans la BD |
|-------------------------|-----------------------|-----|--------------|
| Read(A, a); a := a+100 | | 25 | 25 |
| Write(A, a); | | 125 | |
| Read(B, b); b := b+100; | | | 125 |
| Write(B, b); | | | |
| | Read(A, a); a := a×2; | | |
| | Write(A, a); | 250 | |
| | Read(B, b); b := b×2; | | 250 |
| | Write(B, b); | | |
| | | 250 | 250 |

Ordonnancement sériel S2 (également OK)

► contrainte : A=B

| T1 | T2 | A | B dans la BD |
|-------------------------|-----------------------|-----|--------------|
| | | 25 | 25 |
| | Read(A, a); a := a×2; | | |
| | Write(A, a); | 50 | |
| | Read(B, b); b := b×2; | | |
| | Write(B, b); | | 50 |
| Read(A, a); a := a+100 | | | |
| Write(A, a); | | 150 | |
| Read(B, b); b := b+100; | | | |
| Write(B, b); | | | 150 |
| | | 150 | 150 |

Ordonnancement S3 (sérialisable parce que même effet que S1)

► contrainte : $A=B$

| T1 | T2 | A | B dans la BD |
|-------------------------|-----------------------|-----|--------------|
| | | 25 | 25 |
| Read(A, a); a := a+100 | | | |
| Write(A, a); | | 125 | |
| | Read(A, a); a := a×2; | | |
| | Write(A, a); | 250 | |
| Read(B, b); b := b+100; | | | |
| Write(B, b); | | | 125 |
| | Read(B, b); b := b×2; | | |
| | Write(B, b); | | 250 |
| | | 250 | 250 |

► Remarque : même résultat que S1 pour n'importe quelle état initial de la BD

Ordonnancement S3 (sérialisable parce que même effet que S1)

► contrainte : $A=B$

| T1 | T2 | A | B dans la BD |
|-------------------------|-----------------------|----------|--------------|
| | | x | x |
| Read(A, a); a := a+100 | | | |
| Write(A, a); | | x+100 | |
| | Read(A, a); a := a×2; | | |
| | Write(A, a); | 2(x+100) | |
| Read(B, b); b := b+100; | | | |
| Write(B, b); | | | x+100 |
| | Read(B, b); b := b×2; | | |
| | Write(B, b); | | 2(x+100) |
| | | 2(x+100) | 2(x+100) |

► Remarque : même résultat que S1 pour n'importe quelle état initial de la BD

Ordonnancement S4 (non-sérialisable : il ne préserve pas la cohérence)

‣ contrainte : A=B

| T1 | T2 | A | B dans la BD |
|-------------------------|-----------------------|-----|--------------|
| | | 25 | 25 |
| Read(A, a); a := a+100 | | | |
| Write(A, a); | | 125 | |
| | Read(A, a); a := a×2; | | |
| | Write(A, a); | 250 | |
| | Read(B, b); b := b×2; | | |
| | Write(B, b); | | 50 |
| Read(B, b); b := b+100; | | | |
| Write(B, b); | | | 150 |
| | | 250 | 150 |

Ordonnancement S4 (sérialisable par “coïncidence”)

Comme S3 mais une nouvelle T2'

► contrainte :A=B

| T1 | T2' | A | B dans la BD |
|-------------------------|-----------------------|-----|--------------|
| | | 25 | 25 |
| Read(A, a); a := a+100 | | | |
| Write(A, a); | | 125 | |
| | Read(A, a); a := a×1; | | |
| | Write(A, a); | 125 | |
| | Read(B, b); b := b×1; | | |
| | Write(B, b); | | 25 |
| Read(B, b); b := b+100; | | | |
| Write(B, b); | | | 125 |
| | | 125 | 125 |

Ordonnancement S4 (sérialisable par “coïncidence”)

Comme S3 mais une nouvelle T2'

► contrainte :A=B

| T1 | T2' | A | B dans la BD |
|-------------------------|-----------------------|-----|--------------|
| | | 25 | 25 |
| Read(A, a); a := a+100 | | | |
| Write(A, a); | | 125 | |
| | Read(A, a); a := a×1; | | |
| | Write(A, a); | 125 | |
| | Read(B, b); b := b×1; | | |
| | Write(B, b); | | 25 |
| Read(B, b); b := b+100; | | | |
| Write(B, b); | | | 125 |
| | | 125 | 125 |

► équivalent à T1 T2' uniquement grâce à la sémantique de T2' (l'identité)

Notion “forte” de sérialisabilité

- Le SGBG (i.e. le *scheduler*) ne peut pas connaître la sémantique des transactions *
- Il voit uniquement une séquence de demandes venant de plusieurs transactions

Read(A) Write(B), Read(B),

- Solution : imposer un ordonnancement qui soit sérialisable quelle que soit la sémantique des transactions
- \Rightarrow dorénavant :

T: Read(A, a)
 a := a+100
 Write(A, a)
 Read(B, b)
 b := b+100
 Write(B, b)

| | |
|------------------------------|----------|
| T pour le <i>scheduler</i> : | Read(A) |
| | Write(A) |
| | Read(B) |
| | Write(B) |

* quand il la connaît elle est complexe à analyser,...

Un exemple d'ordonnancement sérialisable au sens fort

T1

Read(A)

Write(A);

Read(B);

Write(B)

T2

Read(A)

Write(A)

Read(B)

Write(B)

- Équivalent à T1;T2 quelles que soient les opérations que T1 et T2 font en local sur les valeurs lues

Syntaxe alternative :

R1(A) W1(A) R2(A) W2(A) R1(B) W1(B) R2(B) W2(B)

Notions de serialisabilité

- Plusieurs notions de sérialisabilité garantissent sérialisabilité au sens fort
 - ▶ sérialisabilité par conflit (*conflict serialisability*)
 - ▶ sérialisabilité de vue (*view serialisability*) - pas abordé

Sérialisabilité par conflit

Définition

Deux ordonnancements $S1$, $S2$ sont équivalents par conflit si $S1$ peut être transformé en $S2$ par une séquence d'échanges d'actions non-conflictuelles consécutives.

action : une Read ou Write

actions non-conflictuelles: couple d'actions de transactions distinctes $i \neq j$ qui soient :

- sur des éléments distincts, i.e.
 - $R_i(A) R_j(B)$
 - $R_i(A) W_j(B)$
 - $W_i(A) W_j(B)$
- ou deux Read sur le même élément, i.e.
 - $R_i(A) R_j(A)$

Sérialisabilité par conflit

Remarque

Échanger deux actions non-conflictuelles consécutives dans un ordonnancement

S1 R1(A) R2(A) W1(A) W2(A) R1(B) W1(B) W1(A) W2(B)



S2 R1(A) R2(A) W1(A) R1(B) W2(A) W1(B) W1(A) W2(B)

produit un ordonnancement équivalent,

en effet pour tout état initial de la BD :

- S1 et S2 produisent le même état final de la BD
- chaque transaction a la même exécution dans S1 et S2 (lit et écrit les mêmes valeurs)

Sérialisabilité par conflit

Remarque

Échanger deux actions non-conflictuelles consécutives dans un ordonnancement

S1 R1(A) R2(A) W1(A) W2(A) R1(B) W1(B) **W1(A) W2(B)**



S2 R1(A) R2(A) W1(A) W2(A) R1(B) W1(B) **W2(B) W1(A)**

produit un ordonnancement équivalent,

en effet pour tout état initial de la BD :


- S1 et S2 produisent le même état final de la BD
- chaque transaction a la même exécution dans S1 et S2 (lit et écrit les mêmes valeurs)

Sérialisabilité par conflit

Remarque

Échanger deux actions non-conflictuelles consécutives dans un ordonnancement

S1 R1(A) R2(A) W1(A) W2(A) R1(B) W1(B) W1(A) W2(B)



S2 R2(A) R1(A) W1(A) W2(A) R1(B) W1(B) W1(A) W2(B)

produit un ordonnancement équivalent,

en effet pour tout état initial de la BD :

- S1 et S2 produisent le même état final de la BD
- chaque transaction a la même exécution dans S1 et S2 (lit et écrit les mêmes valeurs)

Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**

Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

S: R1(A) W1(A) R2(A) W2(A) R1(B) W1(B) R2(B) W2(B)

Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**


Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

S: R1(A) W1(A) R2(A) W2(A) R1(B) W1(B) R2(B) W2(B)



Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**


Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

S: R1(A) W1(A) R2(A) R1(B) W2(A) W1(B) R2(B) W2(B)



Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**


Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

R1(A) W1(A) R2(A) R1(B) W2(A) W1(B) R2(B) W2(B)



Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**


Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

R1(A) W1(A) R1(B) R2(A) W2(A) W1(B) R2(B) W2(B)



Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**


Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

R1(A) W1(A) R1(B) R2(A) W2(A) W1(B) R2(B) W2(B)



Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**


Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

R1(A) W1(A) R1(B) R2(A) W1(B) W2(A) R2(B) W2(B)



Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**


Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

R1(A) W1(A) R1(B) R2(A) W1(B) W2(A) R2(B) W2(B)



Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**

Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

⇒ S sérialisable par conflit implique S sérialisable

Exemple :

R1(A) W1(A) R1(B) W1(B) R2(A) W2(A) R2(B) W2(B)



Sérialisabilité par conflit

⇒ L'ordonnancement produit d'un ordonnancement S par une suite d'échanges d'actions non-conflictuelles est équivalent à S

On appelle un tel ordonnancement **équivalent à S par conflit**

Définition

Un ordonnancement S est sérialisable par conflit si S est équivalent par conflit à un ordonnancement sériel

\Rightarrow S sérialisable par conflit implique S sérialisable

Exemple : S' sériel $\Rightarrow S$ sérialisable par conflit

S': R1(A) W1(A) R1(B) W1(B) R2(A) W2(A) R2(B) W2(B)

T1 T2

Sérialisabilité par conflit

Un autre exemple : S : R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

Deux possibilités pour obtenir un ordonnancement sériel par échanges successifs :

R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

Sérialisabilité par conflit

Un autre exemple : S : R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

Deux possibilités pour obtenir un ordonnancement sériel par échanges successifs :

I)

R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

The diagram shows a sequence of operations: R1(A), W1(A), R2(A), W2(A), R2(B), W2(B), R1(B), W1(B). A horizontal line with an upward-pointing arrow at its left end spans from the start of R2(A) to the end of W1(B). A curly bracket is positioned above the line, spanning from the start of R1(B) to the end of W1(B). This indicates a swap between the operations R1(B) and W1(B) to achieve serializability.

Sérialisabilité par conflit

Un autre exemple : S : R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

Deux possibilités pour obtenir un ordonnancement sériel par échanges successifs :

I)

R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

The diagram shows a sequence of operations: R1(A), W1(A), R2(A), W2(A), R2(B), W2(B), R1(B), W1(B). A horizontal line with an upward-pointing arrow at the left end and a red 'X' in the middle is positioned below the sequence. A curly bracket is placed under the last two operations, R1(B) and W1(B), indicating a swap between them to achieve serializability.


Sérialisabilité par conflit

Un autre exemple : $S : R1(A) \ W1(A) \ R2(A) \ W2(A) \ R2(B) \ W2(B) \ R1(B) \ W1(B)$

Deux possibilités pour obtenir un ordonnancement sériel par échanges successifs :

2)

$R1(A) \ W1(A) \ R2(A) \ W2(A) \ R2(B) \ W2(B) \ R1(B) \ W1(B)$



The diagram shows a sequence of operations: R1(A), W1(A), R2(A), W2(A), R2(B), W2(B), R1(B), W1(B). A horizontal line with an upward-pointing arrow at its right end is positioned below the sequence. A curly bracket is placed under the first two operations, R1(A) and W1(A), and the arrow points from the right side of this bracket to the operation W2(B), indicating a swap between W1(A) and W2(B).

Sérialisabilité par conflit

Un autre exemple : S : R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

Deux possibilités pour obtenir un ordonnancement sériel par échanges successifs :

2)

R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

The diagram shows a sequence of operations: R1(A), W1(A), R2(A), W2(A), R2(B), W2(B), R1(B), W1(B). A horizontal line with an upward-pointing arrow at the end spans from the start of R1(A) to the end of W2(B). A red 'X' is placed on this line between W1(A) and R2(A), indicating a conflict between these two operations.

Sérialisabilité par conflit



S : R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)

S n'est pas sérialisable par conflit

Comment tester si un ordonnancement est sérialisable par conflit ?

(sans faire une énumération exhaustive de tous les ordonnancements sériels possibles)

test fondé sur le graphe de précedence

Graphe de précédence $P(S)$ pour un ordonnancement S

Sommets: transactions de S

Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et
 S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Graphe de précédence $P(S)$ pour un ordonnancement S

Sommets: transactions de S

Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et
 S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Exemple. $P(S)$ pour $S = W_3(A) \ W_2(C) \ R_1(A) \ R_1(B) \ R_1(C) \ W_2(A) \ R_4(A) \ W_4(D)$

Graphe de précédence $P(S)$ pour un ordonnancement S

Sommets: transactions de S

Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et

S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Exemple. $P(S)$ pour $S = W_3(A) \ W_2(C) \ R_1(A) \ R_1(B) \ R_1(C) \ W_2(A) \ R_4(A) \ W_4(D)$

T1

T2

T3

T4

Graphe de précédence $P(S)$ pour un ordonnancement S

Sommets: transactions de S

Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et
 S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Exemple. $P(S)$ pour $S = W3(A) \ W2(C) \ R1(A) \ R1(B) \ R1(C) \ W2(A) \ R4(A) \ W4(D)$



Graphe de précédence $P(S)$ pour un ordonnancement S

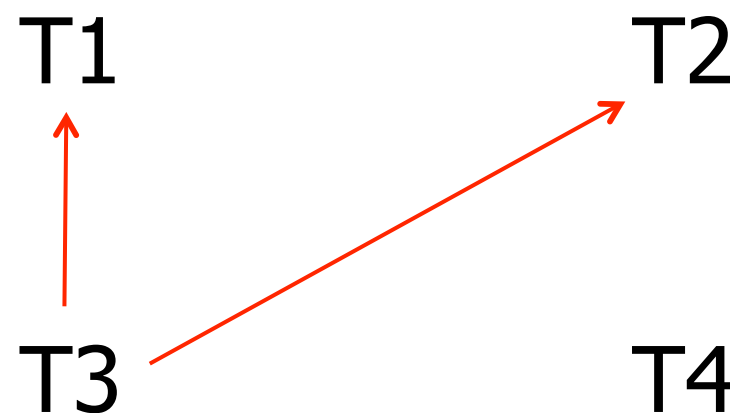
Sommets: transactions de S

Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et
 S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Exemple. $P(S)$ pour $S = W3(A) \ W2(C) \ R1(A) \ R1(B) \ R1(C) \ W2(A) \ R4(A) \ W4(D)$



Graphe de précédence $P(S)$ pour un ordonnancement S

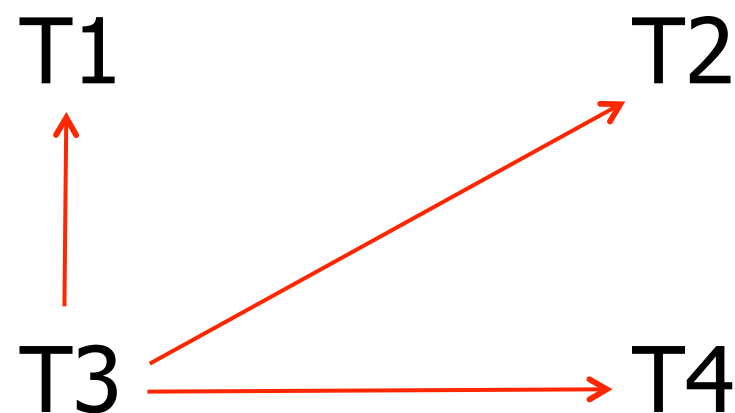
Sommets: transactions de S

Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et
 S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Exemple. $P(S)$ pour $S = W3(A) \ W2(C) \ R1(A) \ R1(B) \ R1(C) \ W2(A) \ R4(A) \ W4(D)$



Graphe de précédence $P(S)$ pour un ordonnancement S

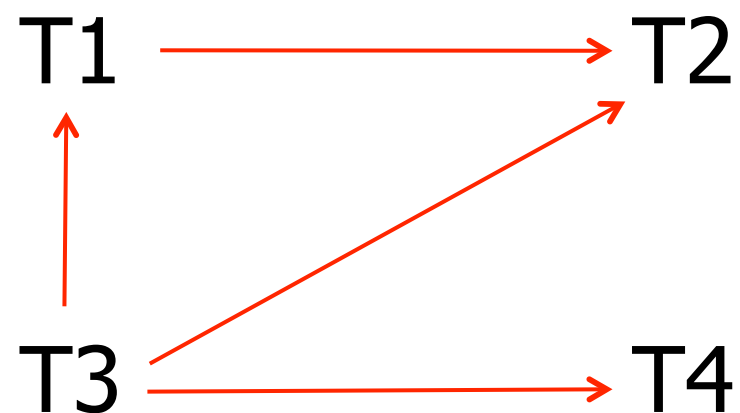
Sommets: transactions de S

Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et
 S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Exemple. $P(S)$ pour $S = W3(A) \ W2(C) \ R1(A) \ R1(B) \ R1(C) \ W2(A) \ R4(A) \ W4(D)$



Graphe de précédence $P(S)$ pour un ordonnancement S

Sommets: transactions de S

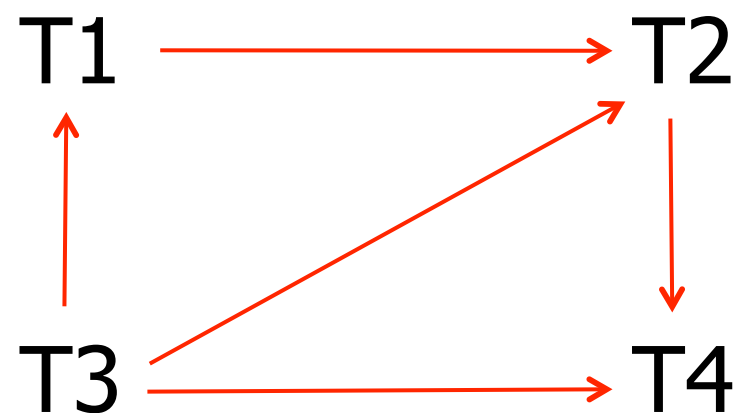
Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et

S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Exemple. $P(S)$ pour $S = W3(A) \ W2(C) \ R1(A) \ R1(B) \ R1(C) \ W2(A) \ R4(A) \ W4(D)$



Graphe de précédence $P(S)$ pour un ordonnancement S

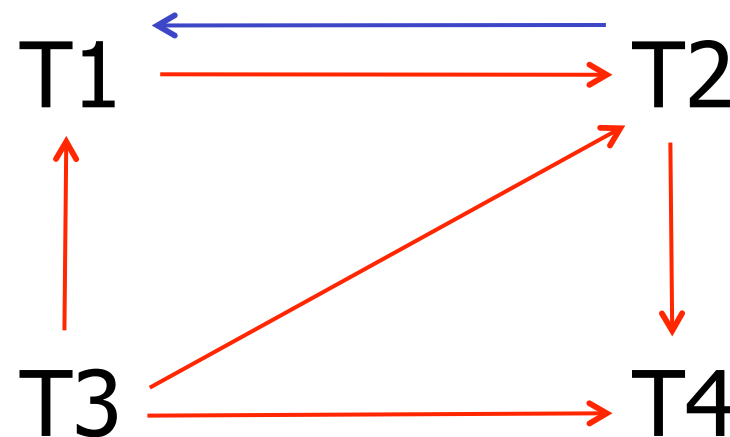
Sommets: transactions de S

Arcs: $T_i \rightarrow T_j$ ssi $i \neq j$ et
 S contient une action de T_i qui précède et est conflictuelle avec une action de T_j

Rappel :

actions conflictuelles : un couple d'actions sur le même élément dont une est Write

Exemple. $P(S)$ pour $S = W3(A) \ W2(C) \ R1(A) \ R1(B) \ R1(C) \ W2(A) \ R4(A) \ W4(D)$



Graphe de précédence $P(S)$ pour un ordonnancement S

- **Lemma.** Si $T_i \rightarrow T_j$ est dans $P(S)$
alors T_i précède T_j dans tout ordonnancement sériel équivalent à S par conflit

Graphe de précédence $P(S)$ pour un ordonnancement S

- **Lemma.** Si $T_i \rightarrow T_j$ est dans $P(S)$
alors T_i précède T_j dans tout ordonnancement sériel équivalent à S par conflit
- **Preuve:** Supposer par contradiction
 - ▶ $T_i \rightarrow T_j$ dans $P(S)$
 - ▶ S' sériel et équivalent à S par conflit
 - ▶ T_j précède T_i dans S'

alors

$$\begin{array}{ll} S = \dots \text{Pi(A)} \dots \text{Qj(A)} \dots & (\text{Pi(A) et Qj(A)} \\ S' = \dots \text{Qj(A)} \dots \text{Pi(A)} \dots & \text{conflictuelles}) \end{array}$$

$\Rightarrow S$ et S' ne peuvent pas être équivalents par conflit :

toute transformation de S en S' doit à un moment échanger Pi(A) et Qj(A)

Test de sérialisabilité par conflit

Théorème S sérialisable par conflit $\Leftrightarrow P(S)$ n'a pas de circuit

Preuve. (\Rightarrow)

Si $P(S)$ a un circuit $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$

alors par le lemme, dans tout ordonnancement sériel équivalent à S par conflit

$T_1 < T_2 < \dots < T_n < T_1 \Rightarrow$ un tel ordonnancement n'existe pas

Test de sérialisabilité par conflit

Théorème S sérialisable par conflit $\Leftrightarrow P(S)$ n'a pas de circuit

Preuve. (\Leftarrow)

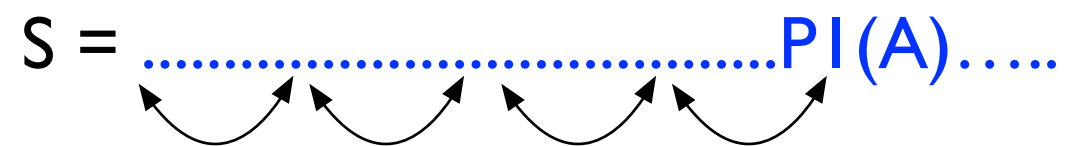
Si $P(S)$ n'a pas de circuit, transformer S de cette façon :

(1) Soit $T1$ une transaction sans arcs entrants

\Rightarrow les actions de $T1$ n'ont aucune action conflictuelle précédente en S (d'autres Ti)

(2) Déplacer par échange successifs toutes les actions de $T1$ en tête, dans l'ordre

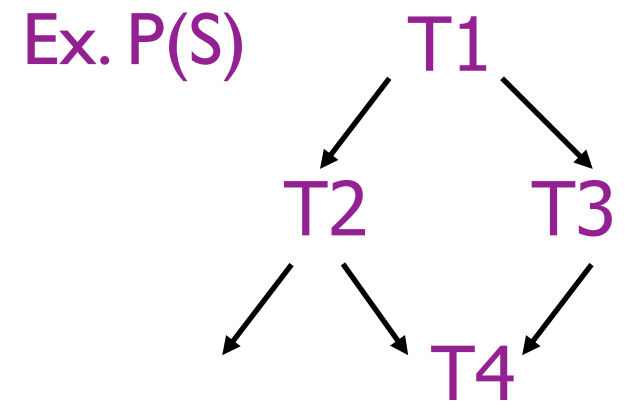
$S = \dots\dots\dots PI(A) \dots\dots$



(3) On obtient $S' = \langle \text{actions de } T1 \rangle \langle \dots \text{ le reste } \dots \rangle$

remarque : le graphe de précédence de $\langle \dots \text{ le reste } \dots \rangle$ est : $P(S)$ sans $T1$

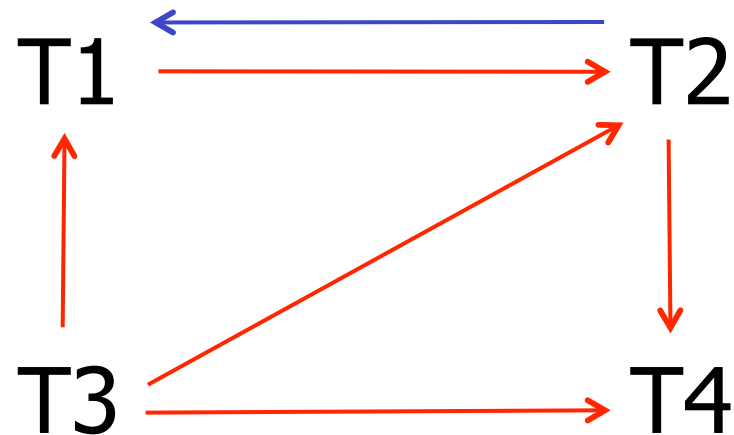
(4) Répéter les pas précédents pour sérialiser le reste!



Exemples de test de sérialisabilité par conflit

Exemple 2 (rappel) $S2 = W3(A) \ W2(C) \ R1(A) \ R1(B) \ R1(C) \ W2(A) \ R4(A) \ W4(D)$

$P(S2)$



$P(S2)$ a un circuit $\Rightarrow S$ non sérialisable par conflit

Exemple 1 (rappel) $S1 = R1(A) \ W1(A) \ R2(A) \ W2(A) \ R1(B) \ W1(B) \ R2(B) \ W2(B)$

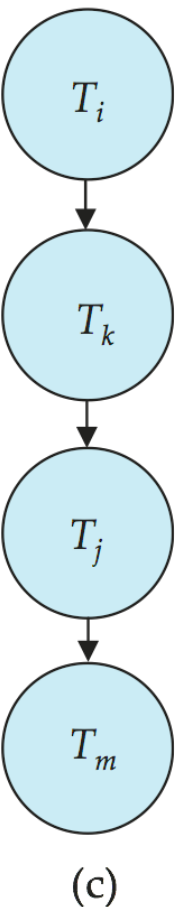
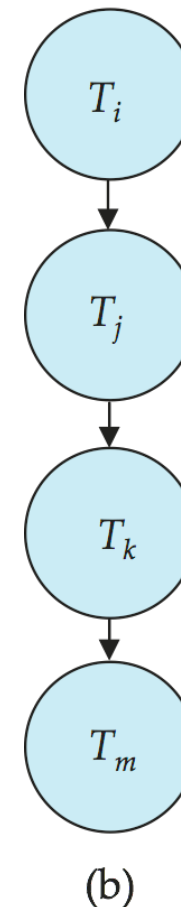
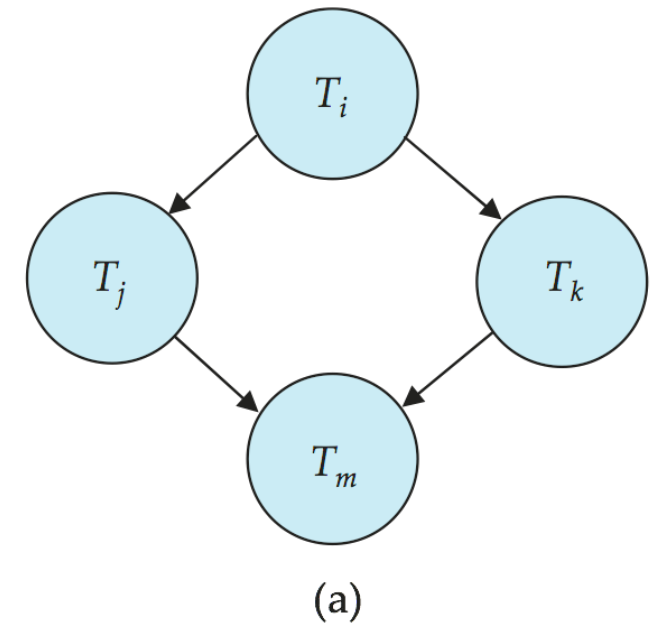


$P(S1)$ acyclique $\Rightarrow S1$ sérialisable

(équivalent au tri topologique $T1;T2$)

Tester la sérialisabilité par conflit

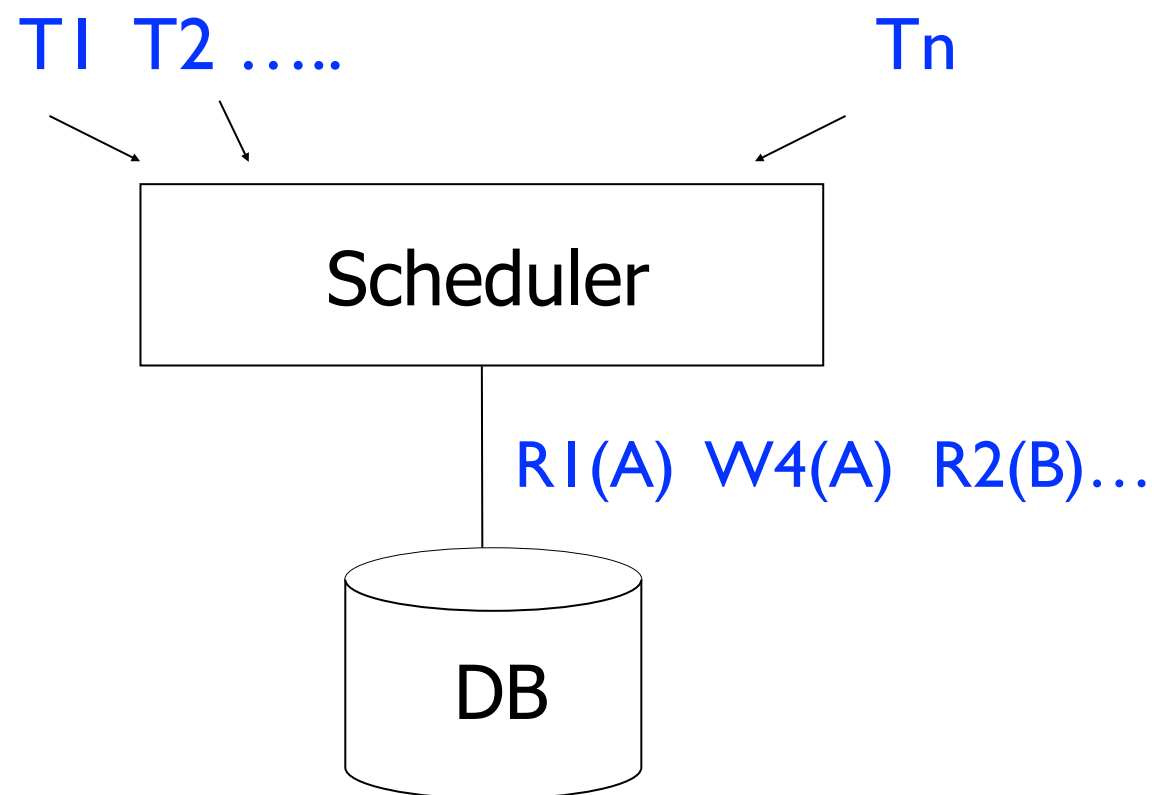
- Détecter les cycles dans un graphe : $O(n^2)$
(n nombre de sommets , i.e. de transactions)
 - ▶ (ou bien $O(n + e)$ ou e est le nombre d'arêtes)
- Si le graphe de précédence est acyclique, un ordre de sérialisabilité est donné par un *tri topologique* du graphe.
 - ▶ Il s'agit d'un ordre linéaire cohérent avec l'ordre partiel défini par le graphe



Contrôle de la concurrence

Comment garantir des ordonnancements sérialisables?

- Le SGBD n'a aucun contrôle sur l'ordre d'arrivée des demandes de read/write
- mais il peut contrôler l'ordre dans lequel il accorde l'exécution de ces demandes
- Idée : forcer l'ordonnancement pour garantir sa serialisabilité (e.g. pour empêcher les cycles dans P(S))



Le module du SGBD qui s'occupe de coordonner les demandes des transactions est appelé *Scheduler*

Contrôle de la concurrence

Comment garantir des ordonnancements sérialisables?

- **Approche curative** (estampillage)
- **Approche préventive** (verrouillage)
- D'autres mécanismes de contrôle de la concurrence
 - ▶ *Multi-version*
 - *Snapshot isolation*
 - ▶ ...

Estampillage : principe

- On associe à chaque transaction T_i un numéro distinct, appelé **estampille**, qu'on note $E(T_i)$
- Ce numéro est donné de manière croissante aux transactions :

$$E(T_i) < E(T_j) \Leftrightarrow T_i \text{ a débuté avant } T_j$$

- But du protocole d'estampillage : gérer l'exécution concurrente de telle sorte que les estampilles déterminent l'ordre de sérialisabilité.
- À tel fin :
 - s'assurer que les actions (Read/Write) conflictuelles aient lieu toujours dans l'ordre des estampilles

Estampillage : principe

Chaque élément A (e.g. n -uplet) de la BD va mémoriser :

1. La plus grande estampille des transactions qui ont lu A , notée $E_R(A)$
2. La plus grande estampille des transactions qui ont écrit A , notée $E_W(A)$

Règle d'estampillage :

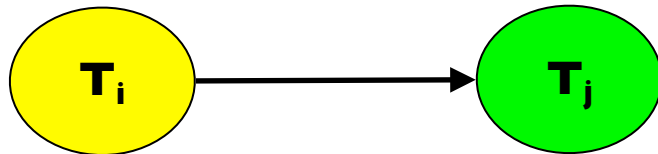
- Seule une transaction T_j telle que $E(T_j) \geq E_W(A)$ pourra accéder A en lecture.
- Seule une transaction T_j telle que $E(T_j) \geq E_R(A)$ et $E(T_j) \geq E_W(A)$ pourra accéder A en écriture.

Propriété : Tout ordonnancement respectant la règle d'estampillage partiel est sérialisable (par conflit).

Preuve de la propriété de sérialisabilité

- La règle d'estampillage garantit que :
deux actions conflictuelles dans l'ordonnancement ont lieu toujours dans l'ordre des estampilles

- \Rightarrow S'il y a un arc dans le graphe de précédence de l'ordonnancement :



T_i a accédé à A puis T_j a accédé à A
avec une action conflictuelle
 $\Rightarrow E(T_i) < E(T_j)$

- \Rightarrow les arcs sont compatibles avec l'ordre $<$ des estampilles
- \Rightarrow l'ordre des estampilles est un tri topologique du graphe de précédence
- \Rightarrow le graphe est acyclique et l'ordre des estampilles donne un ordre sériel équivalent

Si une transaction viole la règle ?

Si une transaction T_j tente de lire / écrire A ,
mais son estampille ne respecte pas la règle d'estampillage par rapport à
 $E_R(A)$ ou $E_W(A)$, alors T_j est annulée (rollback)!

1. On doit défaire toutes les opérations d'écriture de T_j
2. Ainsi que de toutes les transactions ayant lu des valeurs écrits par T_j
3. T_j (comme les autres transactions annulées) peut être relancée immédiatement

Estampillage bilan

- Un protocole *curatif* :
 - ▶ on exécute les demandes en vérifiant qu'elles respectent un ordre de serialisabilité pre-établi
 - ▶ si ce n'est pas le cas, on annule et on recommence
- Plusieurs inconvénients
 - ▶ Les abandons sont lourds à gérer! (abandon en cascade)
 - ▶ Une même transaction peut être annulée plusieurs fois et ne pas arriver à aboutir

Estampillage bilan

- Un protocole *curatif* :
 - ▶ on exécute les demandes en vérifiant qu'elles respectent un ordre de serialisabilité pre-établi
 - ▶ si ce n'est pas le cas, on annule et on recommence
- Plusieurs inconvénients
 - ▶ Les abandons sont lourds à gérer! (abandon en cascade)
 - ▶ Une même transaction peut être annulée plusieurs fois et ne pas arriver à aboutir

**L'estampillage n'est pas utilisé
dans les SGBD**

Ils utilisent tous des variantes du verrouillage

Verrouillage

Un mécanisme de contrôle de la concurrence de type préventif

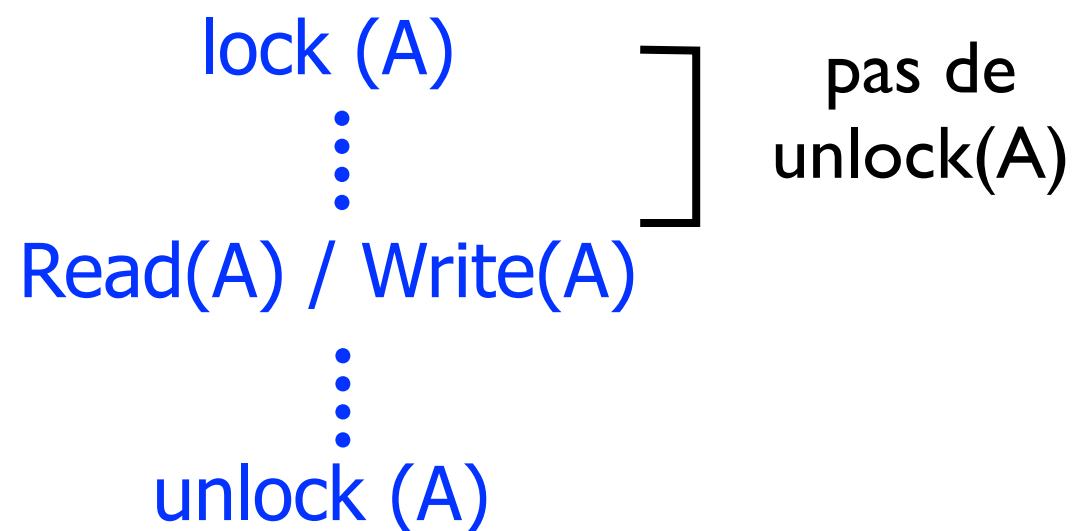
- Rappel : l'ordre des lectures/écritures de différentes transactions est critique uniquement quand elles concernent la même donnée (n-uplet)

Idée du verrouillage :

- On pose des **verrous** (*locks*) sur les éléments de données (n-uplets typiquement)
- Un verrou posé par une transaction bloque l'accès à la donnée de la part d'autres transactions
- Les transactions qui veulent accéder à un élément verrouillé devront attendre que le verrou soit lâché
- Cela force un certain ordonnancement des actions conflictuelles

Les verrous et les transactions

- Les transactions sont censées demander les verrous
 - ▶ deux instructions supplémentaires dans une transaction:
 - ▶ **lock(A)** : demande un verrou sur la donnée A
 - ▶ **unlock(A)** : lâche le verrou sur la donnée A
- On suppose que les transactions soient ``disciplinées``:
 - ▶ une transaction est *bien formée* si pour tout accès à un élément A



Ordonnancements possibles en présence de verrous

- Seulement les ordonnancements qui respectent les verrous sont possibles
- Un ordonnancement est **possible** si aucun `lock(A)` est exécuté par une transaction pendant qu'une autre transaction détient un verrou sur A

| T1 | T2 |
|----------------------------------|---|
| lock(A) Read(A) unlock(A) | |
| | lock(A) Read(A) Write(A) unlock(A) |
| lock(A) Write(A) unlock(A) | |

possible

| T1 | T2 |
|----------------------------------|-----------------------|
| lock(A) Read(A) | |
| | lock(A) Read(A) |
| unlock(A) | |
| | Write(A) unlock(A) |
| lock(A) Write(A) unlock(A) | |

pas possible

Poser des verrous ne suffit pas!

- Même si toutes le transactions utilisent les verrous correctement (i.e sont bien formées)

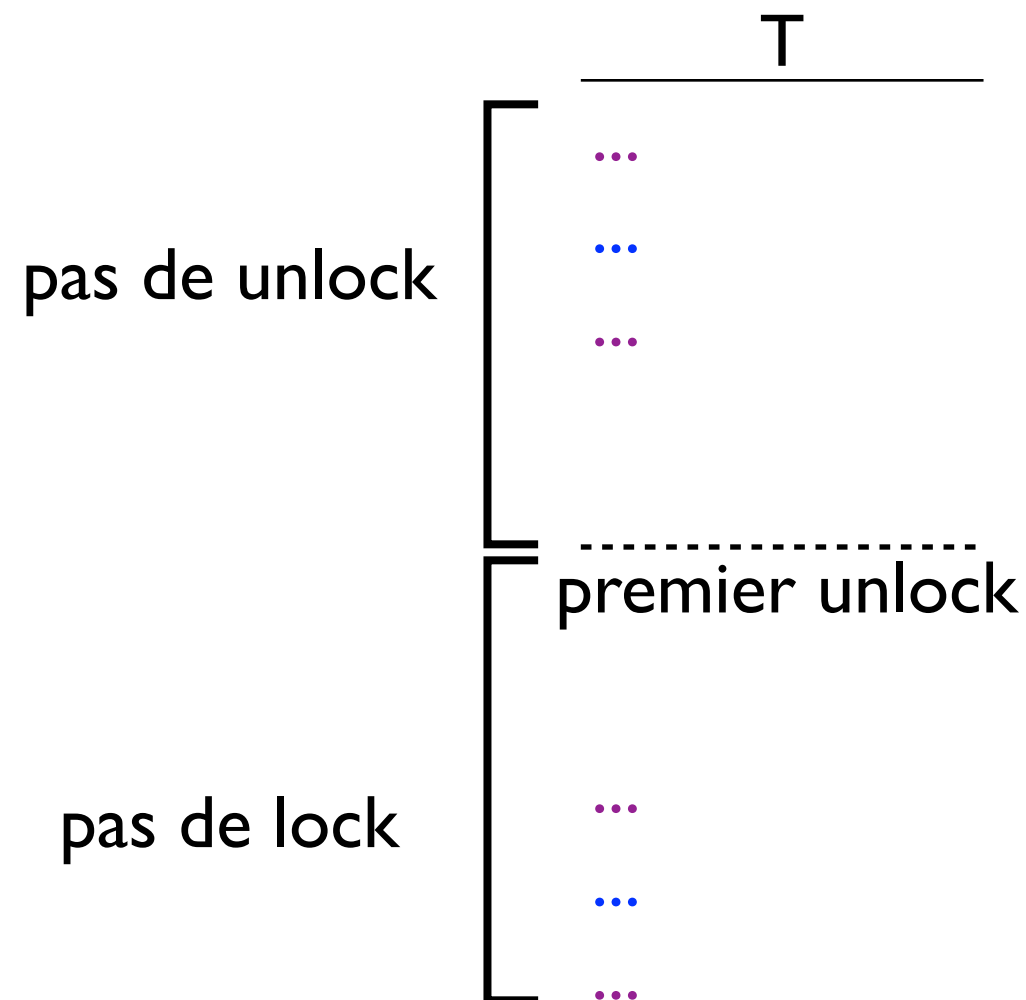
des ordonnancements non-sérialisables sont possibles :

| T1 | T2 |
|--|--|
| lock(A); Read(A) Write(A) ; unlock(A) | lock(A); Read(A) Write(A) ; unlock(A) |
| lock(B); Read(B) Write(B) ; unlock(B) | lock(B); Read(B) Write(B) ; unlock(B) |

pas sérialisable par conflit (et avec des opérations bien choisies, pas sérialisable)

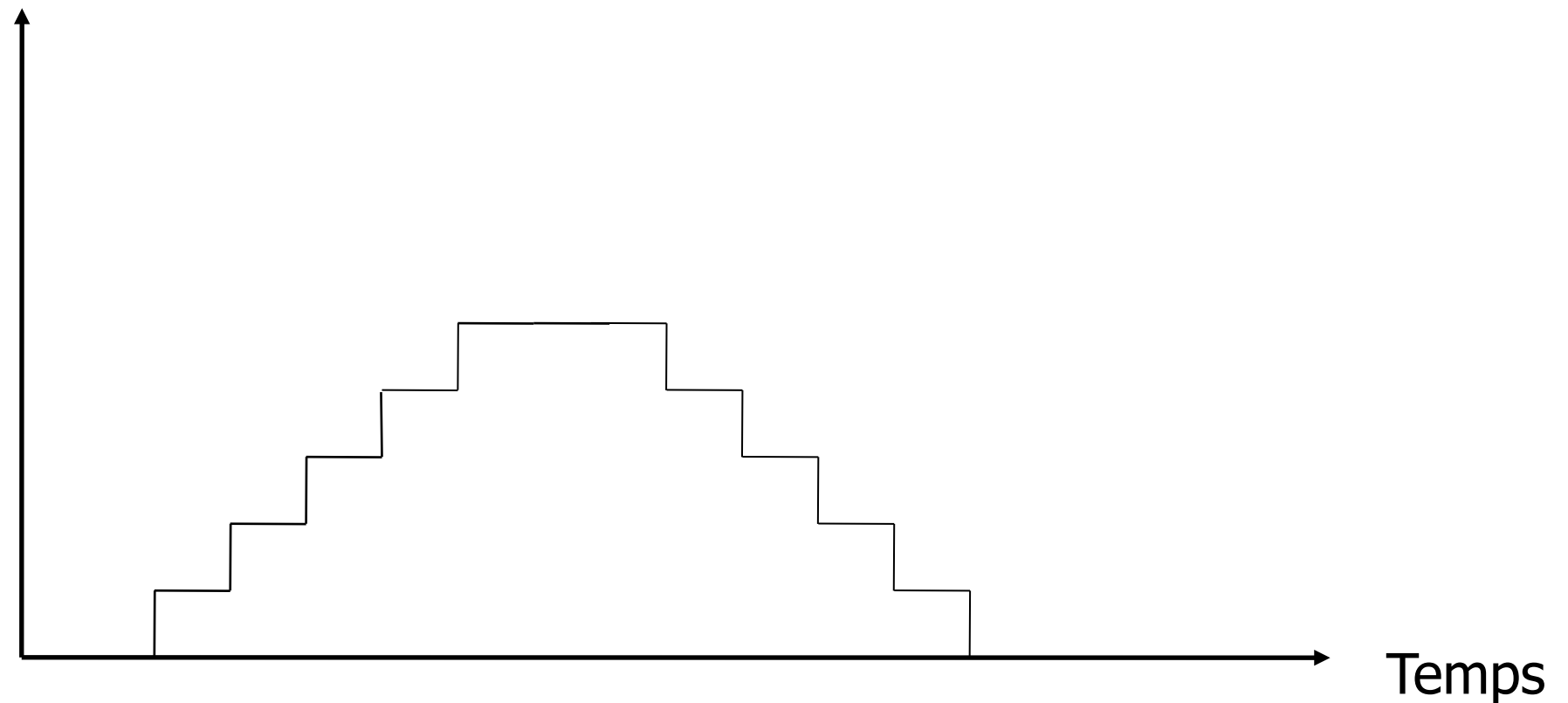
Verrouillage à deux phases (2PL)

- Une simple règle additionnelle sur l'utilisation des verrous par les transactions garantit sérialisabilité de tout ordonnancement possible :
- Règle de verrouillage à deux phases :
Dans chaque transaction, aucun lock n'est précédé par un unlock

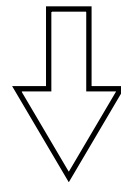


Les deux phases du verrouillage

de verrous
détenus par T

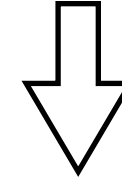


Phase
croissante



T prends
tous les verrous
dont elle a besoin

Phase
décroissante



T lâche tous
ses verrous

Exemple de 2PL

- T1 et T2 vues tout'à l'heure re-écrites en respectant le verrouillage à deux phases:

T1

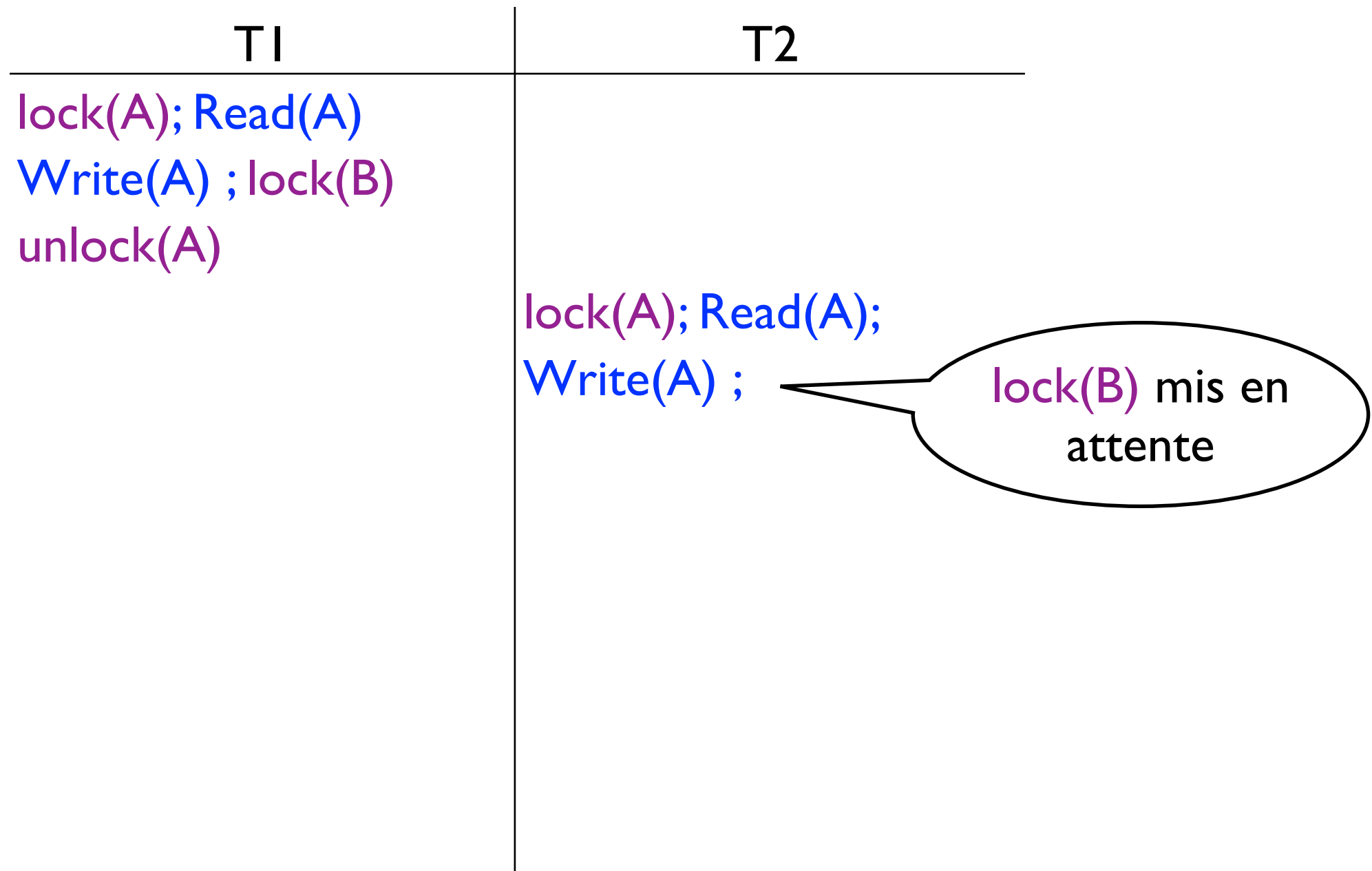
lock(A)
Read(A)
Write(A)
lock(B)
unlock(A)
Read(B)
Write(B)
unlock(B)

T2

lock(A)
Read(A)
Write(A)
lock(B)
unlock(A)
Read(B)
Write(B)
unlock(B)

Exemple de 2PL

- 2PL empêche l'ordonnancement non-sérialisable de tout à l'heure



Exemple de 2PL

- 2PL empêche l'ordonnancement non-sérialisable de tout à l'heure

| T1 | T2 |
|---|---------------------------------|
| lock(A); Read(A) Write(A) ; lock(B) unlock(A) Read(B) Write(B); unlock(B) | lock(A); Read(A); Write(A) ; |

Exemple de 2PL

- 2PL empêche l'ordonnancement non-sérialisable de tout'à l'heure

| T1 | T2 |
|---|--|
| lock(A); Read(A) Write(A) ; lock(B) unlock(A) | lock(A); Read(A); Write(A) ; |
| Read(B) Write(B); unlock(B) | lock(B) ; unlock(A) Read(B) Write(B); unlock(B) |

Sérialisable (par conflit) - équivalent à T1;T2

Correction de 2PL

Théorème

Tout ordonnancement possible de transactions bien formées conformes à 2PL est sérialisable par conflit

Pour le démontrer, on utilise la propriété suivante de tout ordonnancement possible S conforme à 2PL :

Si $T_i \rightarrow T_j$ est dans $P(S)$ alors

le premier *unlock* de T_i précède le premier *unlock* de T_j dans S

Correction de 2PL

Si $T_i \rightarrow T_j$ est dans $P(S)$ alors

le premier *unlock* de T_i précède le premier *unlock* de T_j dans S

En effet $T_i \rightarrow T_j$ implique

$S = \dots\dots\dots P_i(A) \dots\dots\dots Q_j(A) \dots\dots\dots$

P, Q conflictuelles

L : lock

U : unlock

Correction de 2PL

Si $T_i \rightarrow T_j$ est dans $P(S)$ alors

le premier *unlock* de T_i précède le premier *unlock* de T_j dans S

En effet $T_i \rightarrow T_j$ implique

$S = \dots\dots\dots P_i(A) \dots\dots U_i(A) \dots\dots L_j(A) \dots\dots Q_j(A) \dots\dots\dots$

P, Q conflictuelles

L : lock

U : unlock

Correction de 2PL

Si $T_i \rightarrow T_j$ est dans $P(S)$ alors

le premier *unlock* de T_i précède le premier *unlock* de T_j dans S

En effet $T_i \rightarrow T_j$ implique

$S = \dots\dots\dots P_i(A) \dots\dots U_i(A) \dots\dots\dots L_j(A) \dots\dots\dots Q_j(A) \dots\dots\dots$



premier U_i



premier U_j

P, Q conflictuelles

L : lock

U : unlock

Correction de 2PL

Preuve du théorème

Tout ordonnancement possible de transactions bien formées conformes à 2PL est sérialisable par conflit

- l'ordre des premiers *unlock* dans l'ordonnancement est un ordre totale $<$ entre les transactions
- S'il y avait un cycle $T_1 \rightarrow T_2 \rightarrow \dots T_n \rightarrow T_1$ dans $P(S) \Rightarrow T_1 < T_2 < \dots T_n < T_1$, absurde
- \Rightarrow
 - $P(S)$ acyclique et donc S sérialisable par conflit
 - l'ordre des premiers *unlock* donne l'ordonnancement sériel équivalent à S

Correction de 2PL : exemple

| T1 | T2 |
|--|---|
| lock(A); Read(A) Write(A) ; lock(B) unlock(A) | lock(A); Read(A); Write(A) ; |
| Read(B) Write(B); unlock(B) | lock(B) ; unlock(A) Read(B) Write(B); unlock(B) |

Sérialisable (par conflit) - équivalent à T1;T2

Ordre des premiers unlock : T1;T2

Inconvénient de 2PL

- 2PL peut être cause de *Deadlock*
 - ▶ i.e chaque transaction attend que l'autre libère un verrou

T3

lock(A)

Write(A)

lock(B)

unlock(A)

Read(B)

unlock(B)

T4

lock(B)

Write(B)

lock(A)

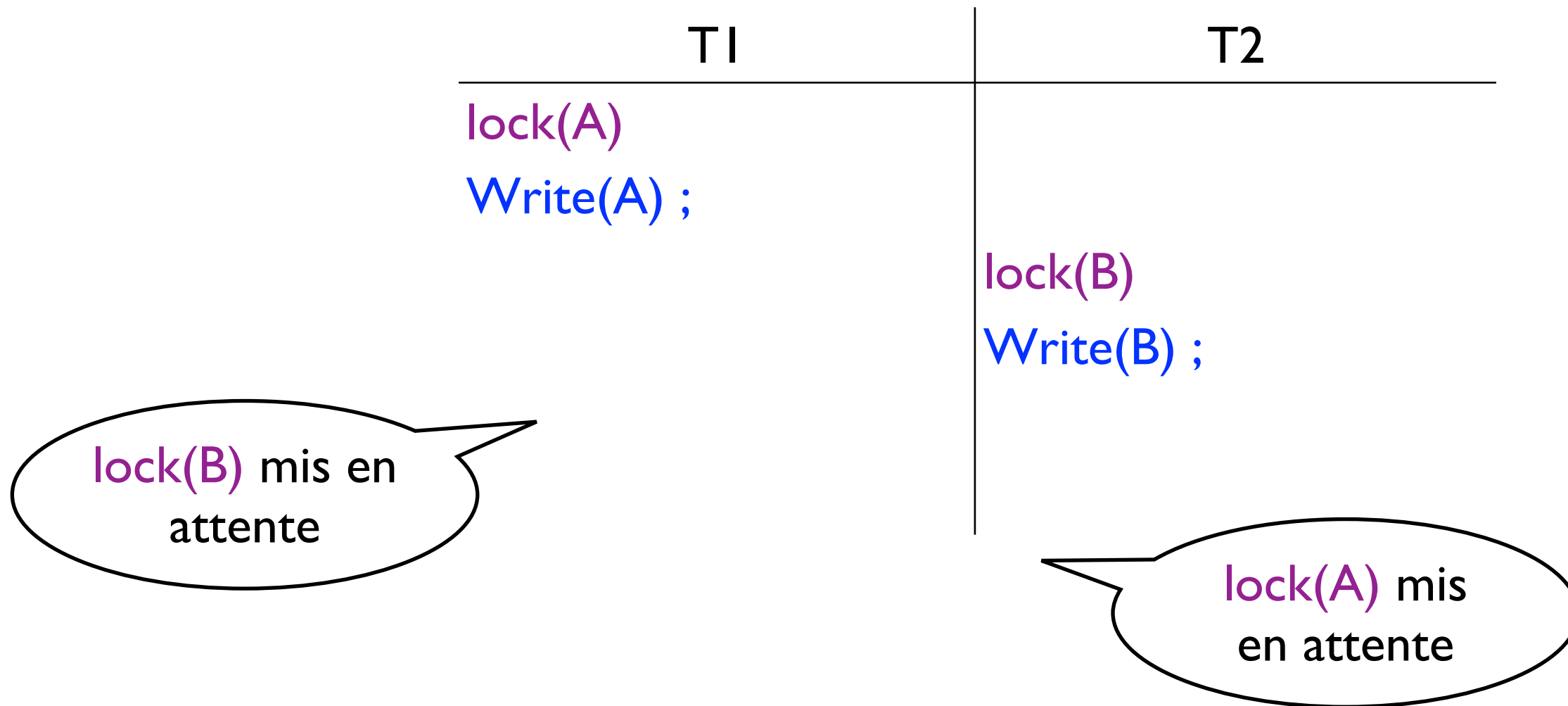
unlock(B)

Read(A)

unlock(A)

Inconvénient de 2PL

- 2PL peut être cause de *Deadlock*
 - ▶ i.e chaque transaction attend que l'autre libère un verrou



- ▶ Solution typique : le *deadlock* est détecté, les transactions sont interrompues et leur effets annulés (*rollback*)

2PL ne capture pas la sérailisabilité par conflit

- 2PL garantit ordonnancement sérialisable par conflit,
- mais tous les ordonnancements sérialisables par conflit ne peuvent pas être obtenus par 2PL

S: W1(x) W3(x) W2(y) W1(y)

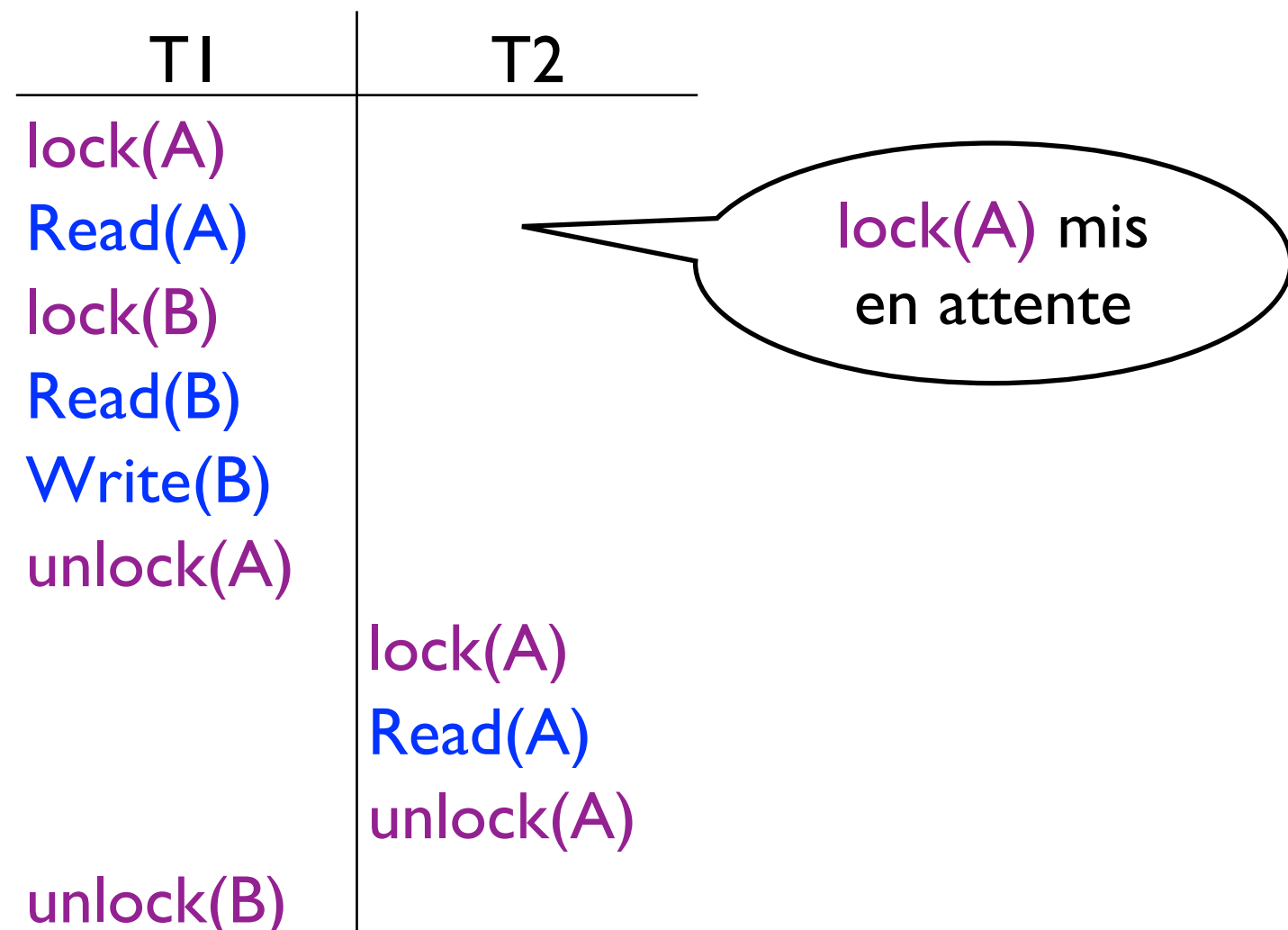
- S ne peut pas être obtenu via 2PL, parce que sous 2PL:
 - ▶ L1(y) doit avoir lieu après W2(y),
 - ▶ donc U1(x) doit avoir lieu après W2(y)
 - ▶ L1(x) doit avoir lieu avant W1(x)
 - ▶ donc, W3(x) ne peut pas avoir lieu comme dans S (T1 détient le lock sur x à ce moment-là)
- Cependant, S est sérialisable par conflit (équivalent à T2, T1, T3).

Verrouillage : améliorer les performances

- Performances du verrouillage : deux paramètres (souvent conflictuels)
 - ▶ niveaux de concurrence (temps d'attente des transactions)
 - ▶ surcharge pour la gestion de verrous
- Améliorations de la performance
 - ▶ verrous partagés
 - ▶ verrouillage hiérarchique

Verrous partagés

- Deux lectures de la même donnée ne sont jamais conflictuelles
- Mais le système de verrous vu jusqu'à maintenant empêche deux transactions de prendre un verrou en même temps pour lire la même donnée
- Cela limite la concurrence surtout quand on implémente 2PL :



Verrous partagés

Solution : plusieurs types de verrous

S-lock(A): demande un verrou partagé sur A (*lock in share mode*)

X-lock(A): demande un verrou exclusif sur A (*lock in exclusive mode*)

unlock(A) : libère tous type de verrou sur A

| Verrou demandé \ Verrou détenu | Verrou détenu | |
|--------------------------------|-----------------|-----------------|
| | S | X |
| S | Accordé | Mise en attente |
| X | Mise en attente | Mise en attente |

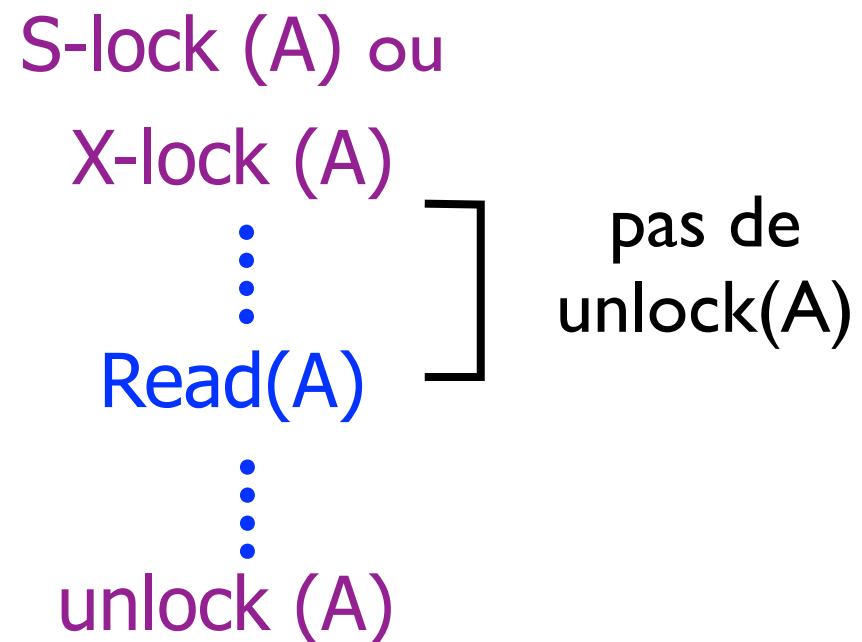
Compatibilité des verrous

Verrous partagés

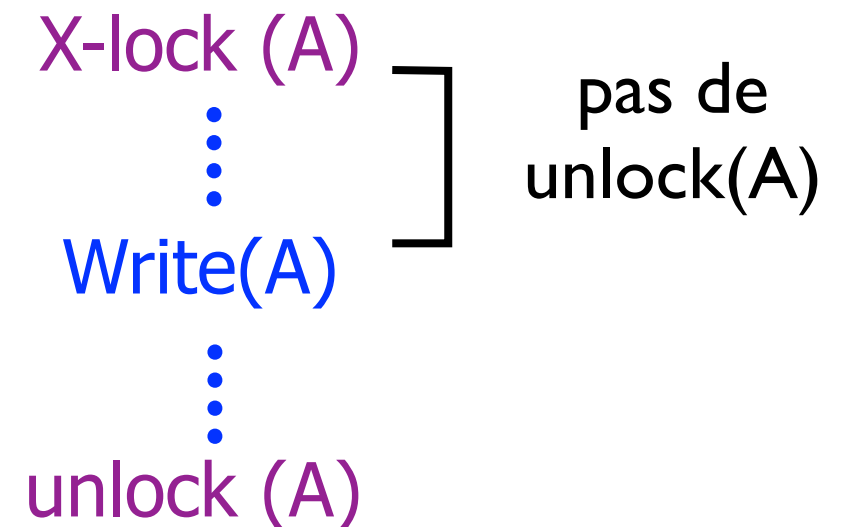
On suppose toujours que les transactions soient disciplinées

Transaction bien formée avec verrous partagés:

pour tout Read(A) :



pour tout Write(A) :



Ordonnancements possibles en présence de verrous partagés

- Seulement les ordonnancements qui respectent les compatibilités des verrous sont possibles (cf. table de compatibilité)

| T1 | T2 |
|------------------------|-----------------------------------|
| S- lock(A) Read(A) | |
| | S-lock(A) Read(A) |
| X-lock(B) Write(B) | |
| | unlock(A) |
| unlock(A) unlock(B) | |
| | S-lock(B) Read(B) unlock(B) |

possible

| T1 | T2 |
|-----------------------|-----------------------------------|
| S- lock(A) Read(A) | |
| | S-lock(A) Read(A) |
| X-lock(B) Write(B) | |
| | unlock(A) |
| unlock(A) | |
| | S-lock(B) Read(B) unlock(B) |
| unlock(B) | |

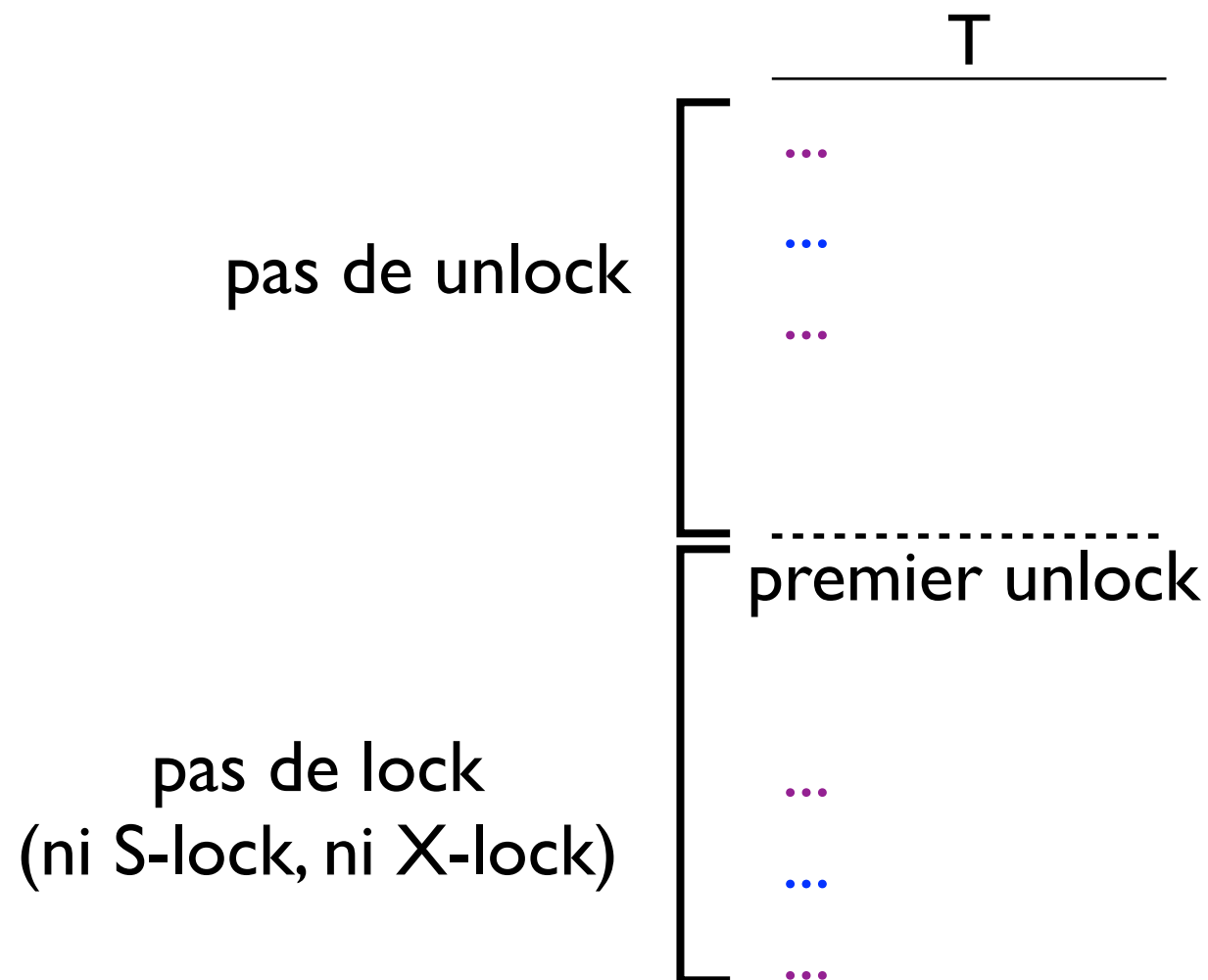
pas possible

2PL avec verrous partagés

- Les règles sont les mêmes mais concernent les deux types de lock.

I.e, pour chaque transaction,

aucun lock (ni S-lock, ni X-lock) n'est précédé par un unlock



Correction de 2PL avec verrous partagés

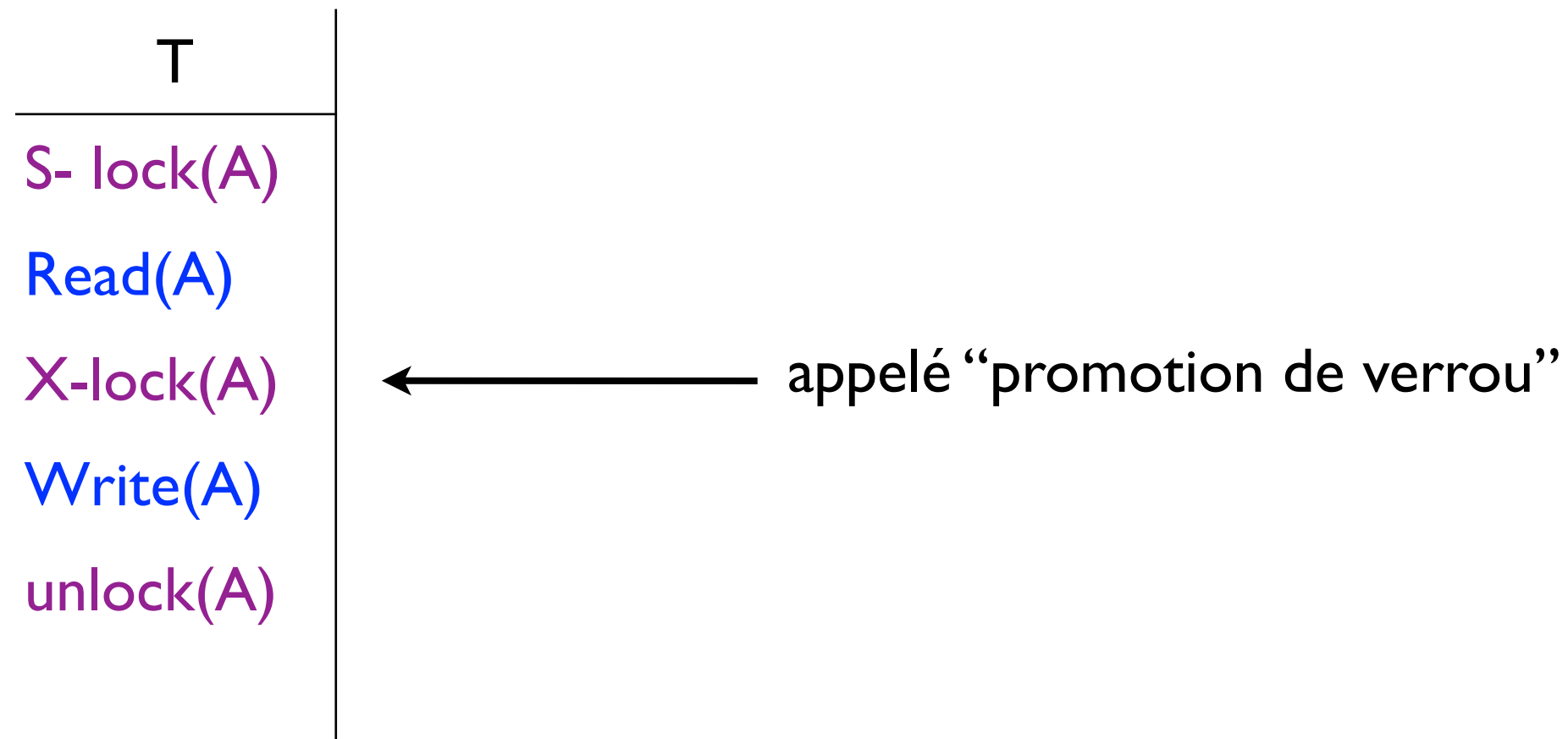
Théorème

Tout ordonnancement possible de transactions bien formées avec verrous S/X conformes à 2PL est sérialisable par conflit

Preuve: similaire au cas de verrous X

Promotion de verrou

- Remarque : une transaction peut détenir plusieurs types de verrous sur la même donnée
- T est bien formée :



Promotion de verrou

- Utile pour des opération de type “mise à jour”
sans verrouiller complètement la ressource dès le début

| |
|-----------------|
| T |
| <hr/> |
| S- lock(A) |
| Read(A, a); |
| vérifie $a > 0$ |
| X-lock(A) |
| $a := a - 1$ |
| Write(A, a) |
| unlock(A) |

Promotion de verrou

- L'alternative sans promotion de verrou n'est pas 2PL
⇒ pourrait permettre des ordonnancements non-sérialisables

| |
|-----------------|
| T |
| <hr/> |
| S- lock(A) |
| Read(A, a); |
| vérifie $a > 0$ |
| unlock(A) |
| X-lock(A) |
| $a := a - 1$ |
| Write(A, a) |
| unlock(A) |

Promotion de verrou

- L'alternative sans promotion de verrou n'est pas 2PL
⇒ pourrait permettre des ordonnancements non-sérialisables

| T | T' |
|-----------------|-------------|
| S- lock(A) | |
| Read(A, a); | |
| vérifie $a > 0$ | |
| unlock(A) | |
| | X-lock(A) |
| | Write(A, 0) |
| | unlock(A) |
| X-lock(A) | |
| $a := a - 1$ | |
| Write(A, a) | |
| unlock(A) | |

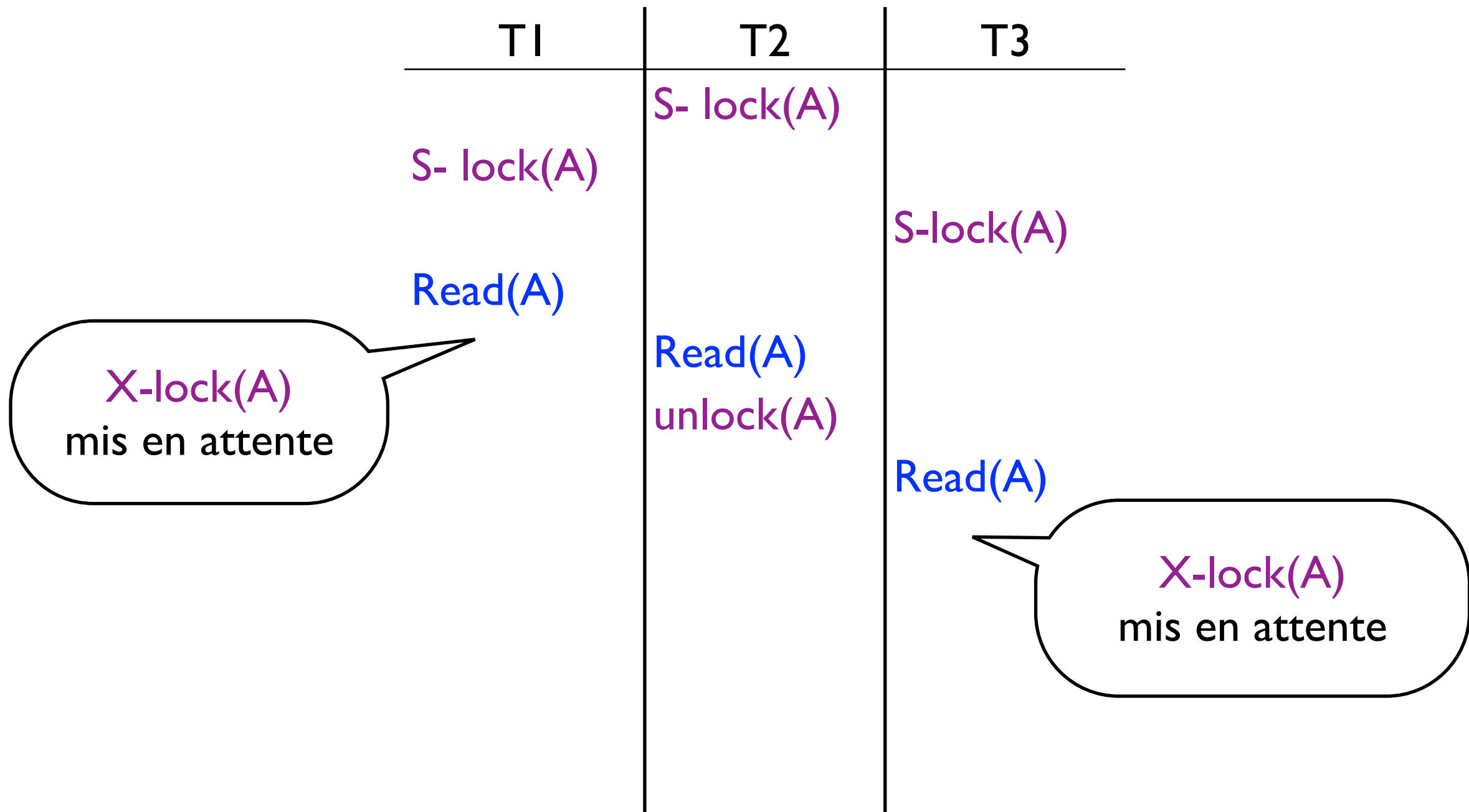
Promotion de verrou

- Inconvénient de la promotion de verrou : favorise le *deadlock*

| T1 | T2 | T3 |
|------------|------------|------------|
| S- lock(A) | S- lock(A) | S- lock(A) |
| Read(A) | Read(A) | Read(A) |
| X-lock(A) | unlock(A) | X-lock(A) |
| Write(A) | | Write(A) |
| unlock(A) | | unlock(A) |

Promotion de verrou

- Inconvénient de la promotion de verrou : favorise le *deadlock*



- Remarquer également l'avantage : le parallélisme potentiel de T1 et T2

Verrous de mise à jour

- Pour résoudre ce problème certains systèmes empêchent la promotion de verrou de type S
- Mais ils introduisent un nouveau type de verrou : le **verrou de mise à jour** (*update lock*, verrou de type U)
 - ▶ le verrou de mise à jour est le seul qui peut être promu à X
- Si une transaction veut lire une donnée pour la modifier ensuite, elle est censée prendre un verrou de type U plutôt que type S.
- Un verrou U a une table de compatibilité asymétrique

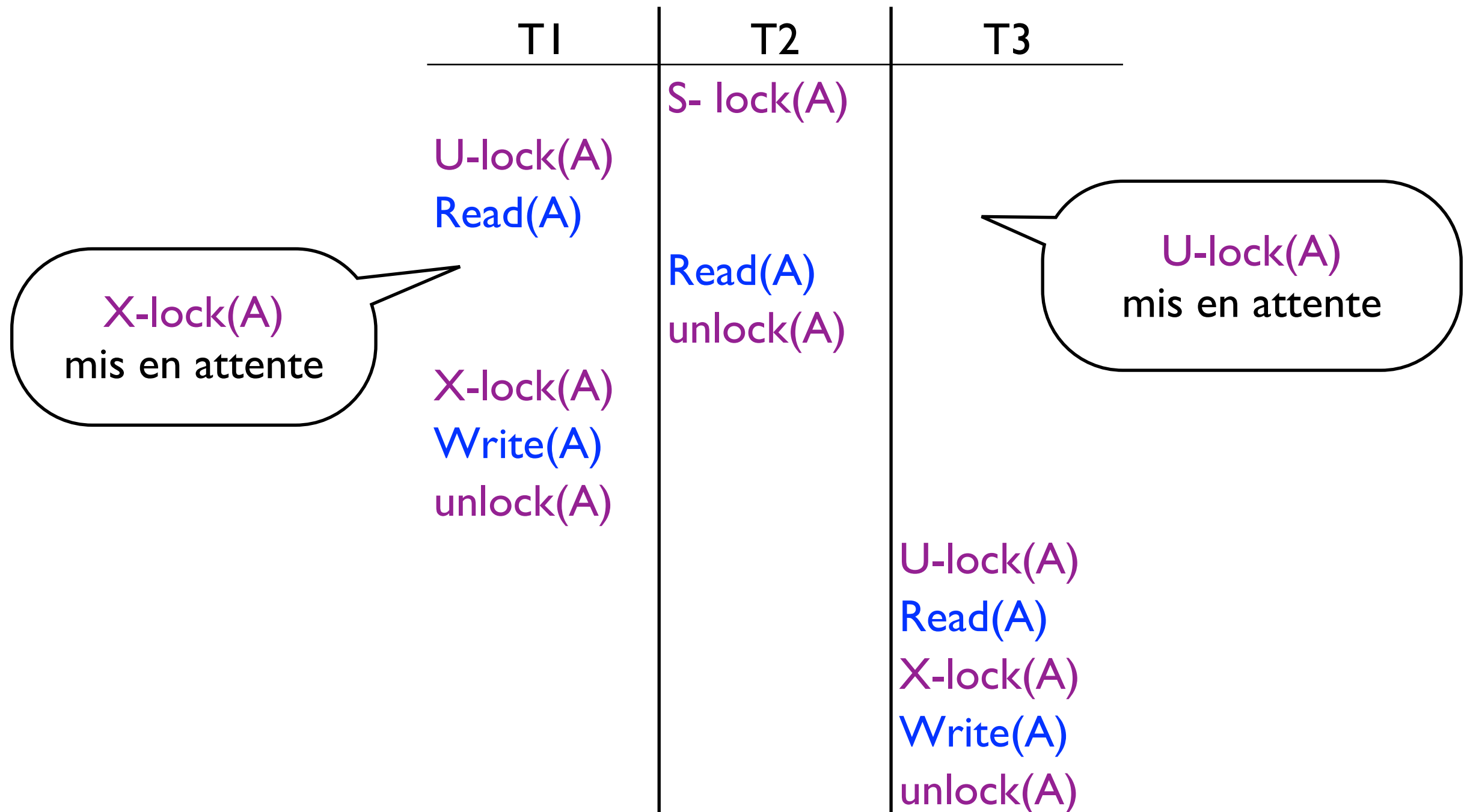
Verrous de mise à jour

- Pour résoudre ce problème certains systèmes empêchent la promotion de verrou de type S
- Mais ils introduisent un nouveau type de verrou : le **verrou de mise à jour** (*update lock*, verrou de type U)
 - ▶ le verrou de mise à jour est le seul qui peut être promu à X
- Si une transaction veut lire une donnée pour la modifier ensuite, elle est censée prendre un verrou de type U plutôt que type S.
- Un verrou U a une table de compatibilité asymétrique

| Verrou demandé \ Verrou détenu | Verrou détenu | | |
|--------------------------------|-----------------|-----------------|-----------------|
| | S | X | U |
| S | Accordé | Mise en attente | Mise en attente |
| X | Mise en attente | Mise en attente | Mise en attente |
| U | Accordé | Mise en attente | Mise en attente |

Verrous de mise à jour

- Evite le *deadlock* entre T1 et T3



- Les mêmes avantages de parallélisme entre T1 et T2

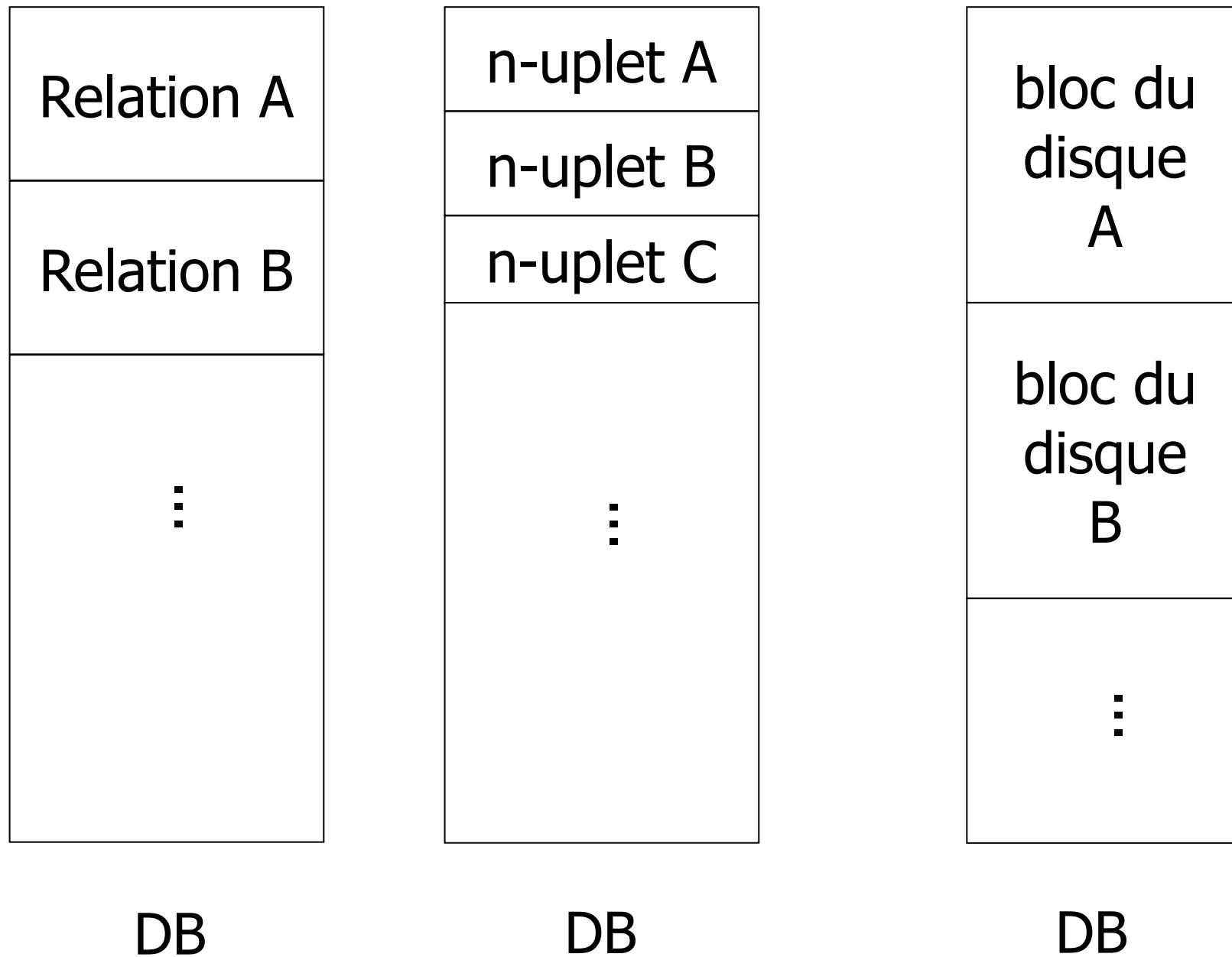
Verrouillage : améliorer les performances

- Performances du verrouillage : deux paramètres (souvent conflictuels)
 - ▶ niveaux de concurrence (temps d'attente des transactions)
 - ▶ surcharge pour la gestion de verrous
- Améliorations de la performance
 - ▶ verrous partagés
 - ▶ **verrouillage hiérarchique**
 - Permet de trouver un compromis entre le coût du verrouillage et le niveau de concurrence



Verrouillage hiérarchique (idée)

- Quels sont les éléments de données que l'on verrouille?
 - ▶ I.e quels sont les A, B, C, ... sur lesquels on effectue lock / unlock ?

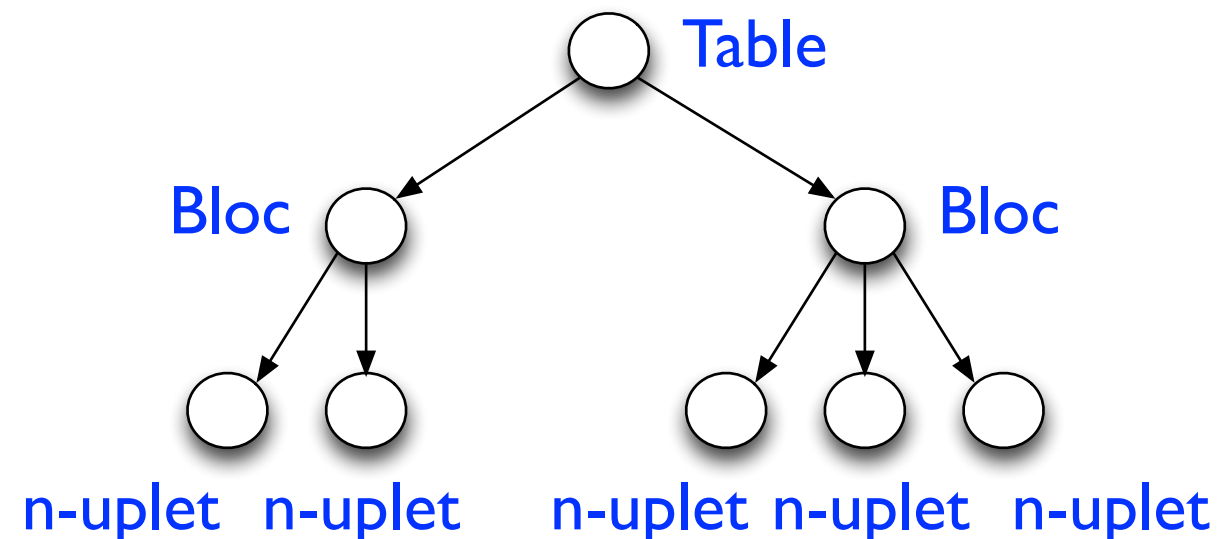


Verrouillage hiérarchique (idée)

- Le protocole de verrouillage fonctionne dans tous les cas, mais faudrait-il choisir des éléments petits ou grands?
- Verrouillage des grands éléments (e.g., Relations)
 - besoin de moins de verrous (verrouillage moins coûteux)
 - moins de concurrence possible (temps d'attente plus importants)
- Verrouillage de petits éléments (e.g., n-uplets, champs)
 - besoin de plus de verrous (verrouillage plus coûteux)
 - plus de concurrence possible (temps d'attente moins importants)
- Dans la plupart des SGBD on peut avoir les deux : verrouillage hiérarchique

Verrouillage hiérarchique (idée)

- Permet de verrouiller des objets de différente granularité
 - Adapter la taille du verrou au nombre de transactions
 - Granule gros si peu de transactions en cours
 - Granule fin sinon
- Les niveaux de verrous définissent un arbre (une hiérarchie):
 - une table contient plusieurs blocs du disque,
 - chaque bloc plusieurs n-uplets



Verrouillage hiérarchique (idée)

- Sur chaque noeud de la hiérarchie une transaction peut demander un verrou normal (S ou X) ou un verrou d’ “intention” (IS ou IX)
 - ▶ Un verrou normal verrouille le noeud entier (Ex. la table)
 - ▶ Un verrou d’intention déclare que la transaction a l’intention de demander un verrou (du même type) pour un sous-élément
 - Ex. demander un verrou IX sur une table pour ensuite demander un verrou X sur un n-uplet de la table
- Plusieurs verrous d’intention peuvent être détenus sur le même élément
- Un verrou d’intention sur un noeud est refusé uniquement quand le noeud est déjà verrouillé avec un verrou normal (S ou X) incompatible
- Un verrou S ou X est refusé quand il y a un verrou normal ou d’intention de type incompatible sur le même element
- Pour demander un verrou S ou X sur un élément, la transaction commence à la racine de la hiérarchie et demande un verrou d’ “intention” du même type sur tous les ancêtres de l'élément

Verrouillage : bilan

- Approche pessimiste de la concurrence
 - ▶ Prévient les conflits
- Assez coûteuse et assez complexe
- Avantages du verrouillage
 - ▶ Pas d'abandon
 - ▶ Les performances sont bonnes : verrous partagés, taille variable du granule

Verrouillage en pratique

Chaque SGBD implémente le verrouillage différemment, mais en général :

- **Granularité par défaut** : n-uplet (verrous demandés sur les lignes - *row level locks*)
- **Verrouillage automatique** : pour garantir sérialisabilité, pas de confiance aux transactions pour demander / libérer les verrous
 - ▶ des verrous (S / X) sur les n-uplets concernés sont automatiquement demandés quand la transaction demande une lecture / écriture (SELECT / UPDATE etc.)
 - ▶ et libérés uniquement à transaction terminée
- **Autres verrous** : La transaction peut demander / libérer des verrous additionnels d'autre type (type “*mise à jour*”, intention, etc..)

Verrouillage en pratique

Demande explicite d'autres types de verrous :

- Verrou de type “mise à jour”

SELECT ... FROM *table* ... FOR UPDATE ;

- Verrou (normal ou d'intention) sur une table :

LOCK TABLE *table* IN *type-de-verrou* MODE;

- *type-de-verrou* :

- **SHARE** (S)
- **EXCLUSIVE** (X)
- **ROW SHARE** (IS)
- **ROW EXCLUSIVE** (IX)
- ...

D'autres mécanismes de contrôle de la concurrence

- *Multi-version*
 - ▶ *Snapshot isolation*
- ...

Mécanismes multi-versions (MVCC)

- Maintiennent des vieilles versions des éléments de données pour augmenter la concurrence
 - Estampillage multi-version
 - 2PL multi-version
- Chaque écriture déclenche la création d'une nouvelle version de l'élément écrit.
- les versions sont identifiées par une estampille.
- Quand une lecture Read(A) est demandée, une version appropriée de A est retournée (qui dépend de l'estampille de la transaction)
- Les lectures ne bloquent jamais puisque une version de la donnée est retournée immédiatement.
- *Snapshot Isolation* : Un mécanisme multi-version très utilisé en pratique (variantes implémentées dans Oracle, PostgreSQL, SQL Server, ...)

Snapshot isolation

Un transaction T qui s'exécute avec Snapshot Isolation

- prends une “image” (*snapshot*) des données committées quand elle démarre
- lit/ modifie les données toujours dans son propre *snapshot*
- les mises à jour des transactions concurrentes ne sont pas visibles à T
- Les écritures de T seront appliquées à la vraie BD quand T fait commit
- Le premier qui fait commit “gagne” :
 - ▶ T fait un commit seulement si aucune autre transaction concurrente n’a encore écrit les données que T a l’intention d’écrire
 - ▶ si ce n’est pas le cas, T fait un rollback et est redémarrée.

Snapshot isolation

- Bonne performances
 - ▶ lectures jamais bloquées,
 - ▶ ni d'autres activités des transactions
- SI ne garantit pas toujours exécutions sérialisables (“trop” d'isolation!)
 - ▶ Sérialisable:
entre deux transactions concurrentes, l'une voit les effets de l'autre
 - ▶ SI: aucune ne voit les effets de l'autre
- Mais évite les anomalies habituelles
 - ▶ lecture “sale”, modification perdue, lecture non-reproductible, lecture fantôme

FIN

Merci!