

# Module 6 – Transactions ACID

Cours IGE487 – Modélisation de bases de données

Chargé de cours Tudor Antohi

# Objectifs

- Mise en contexte
- Problématique
- Modèle transactionnel
- Propriétés recherchées (ACID)
- Récupérabilité (reprise)
- Sérialisabilité
- Gestion transactionnelle sous SQL
- SGF, SGBD, SQL et ACID

# Introduction

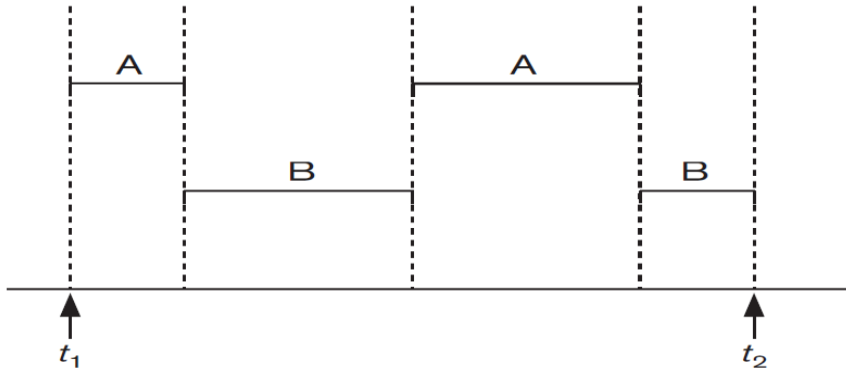
On a discuté l'algebre relationnelle , la normalisation et l'optimisation  
On va revenir sur l'optimisation dans chaque cours.

- Les SGBDs modernes viennent avec beaucoup des utilitaires modernes, en majorité analytiques
- Une SGBD sur laquelle on peut mettre nos vies doit garder nos données propres, integres et disponibles
- Un élément essentiel dans le choix d'une BD est de comprendre si le mecanisme transactionnel offert par un vendeur répond a nos besoins d'affaires

*Learn before you choose.*

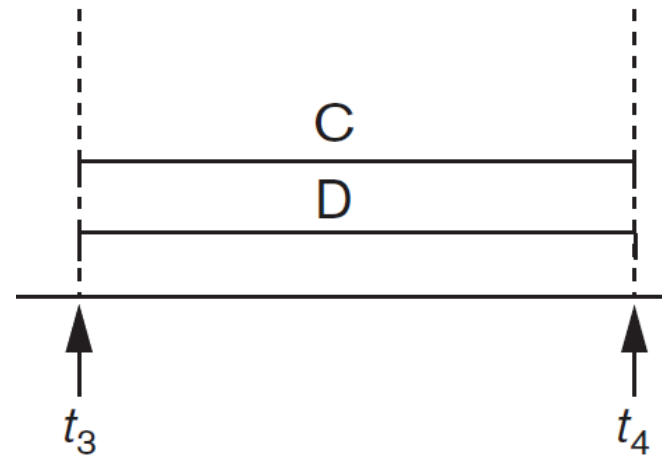
# Parallélisme – la cause des conflits

## Interlacées – une ressource



Dans l'ordonnancement préemptif, la ressource est allouée aux processus pour une durée limitée, tandis que dans l'ordonnancement non préemptif, la ressource est allouée au processus jusqu'à ce qu'il se termine ou passe à l'état d'attente.

## Parallélisme – multiple ressources



Le problème est quand plusieurs processus au même temps essayent d'actualiser une seule ressource.

# Problématique

- Une base de données doit :
  - Desservir simultanément plusieurs clients et sessions
  - Garder l'intégrité de données
  - Fournir un mécanisme de récupération en cas de defaillance
  - Fournir un mécanisme de restauration en cas de perte de données

Les environnements sont :

- Multi-taches
- Multi-CPUs
- Multiples zones de mémoire
- Multiples entrees et sorties disque

Les données peuvent être partagées et doivent garder leur intégrité, quand ils sont accédes en mémoire ou sur le disque.

# Solutions pour gérer le probleme

- Systèmes qui ne partages rien , les données D1 sont allouées au système S1 et les exécutions sont sérialisées en S1, les données D2 etc.
- Une meilleure approche (potentiellement) consiste à permettre l'exécution simultanée des opérations indépendantes.

## **Pourquoi accepter les opérations concurrentes et indépendantes ?**

- Une meilleure utilisation/un meilleur débit.
- Augmentation des temps de réponse aux utilisateurs.

Mais nous voudrions aussi : la correctitude, l'équité

# Concurrence

Ressources uniques accèdes par plusieurs processus parallèles

## Dangers

- Collisions sur les
  - demandes de CPUs qui dépassent le nombre de CPUs disponibles – gestion de CPU (*CPU scheduler*)
  - zones communes de mémoire – gestion de la mémoire
  - **zones communes de données – gestion de concurrence de transactions de données nécessaires**
- États transitoire de la base de données incorrecte par rapport a l'intégrité de son schéma logique qui peuvent entrainer des corruptions logiques définitives

***Important : un mecanisme de gestion de concurrence de transactions est essentiel***

# Recouverte

Un mecanisme de recouverte de transactions est important pour :

- Erreurs des systèmes physiques (failles disques, ordinateurs)
- Gestion des certaines erreurs transactionnelles (ex. : les deadlocks)

Sujet a reprendre en Module 8.



# Problématique

9

**LUC LAVOIE SLIDES 8-14**

# Modèle transactionnel

# Transactions - notions

- **Une transaction** est une unité logique de traitement exécutée par le SGBD.
- **Une unité logique de traitement** déplace une BD d'un état initial logiquement consistant et intègre à un autre état final consistant et intègre final, par rapport au schéma logique existant au moment du traitement.
- **Consistance BD** – tous les utilisateurs voient les mêmes données
- **Intégrité BD** – les contraintes d'intégrité de schéma de données sont respectées.

# Transactions – opérations de base

- **Lire(X)**, qui transfère l'élément de données X de la base de données vers une variable, également appelée X, dans une mémoire tampon

Ex.: SELECT

- **Écrire (X)**, qui transfère la valeur de la variable X dans l'élément de données X dans la base de données (via tampon intermédiaire dans certains cas).

Ex.: UPDATE, INSERT, DELETE. MERGE

# Transactions – opérations de base

- La transaction est une séquence ordonnées de lecture et d'écriture d'opérations ( Lire(X), Ecrire(Y), ...)
  - soit toutes les opérations sont exécutées dans l'ordre,
  - soit aucune n'est exécutée

Ex.: Déplacez 100 \$ du compte bancaire de Tudor vers le payment de sa carte Visa Dejsardins

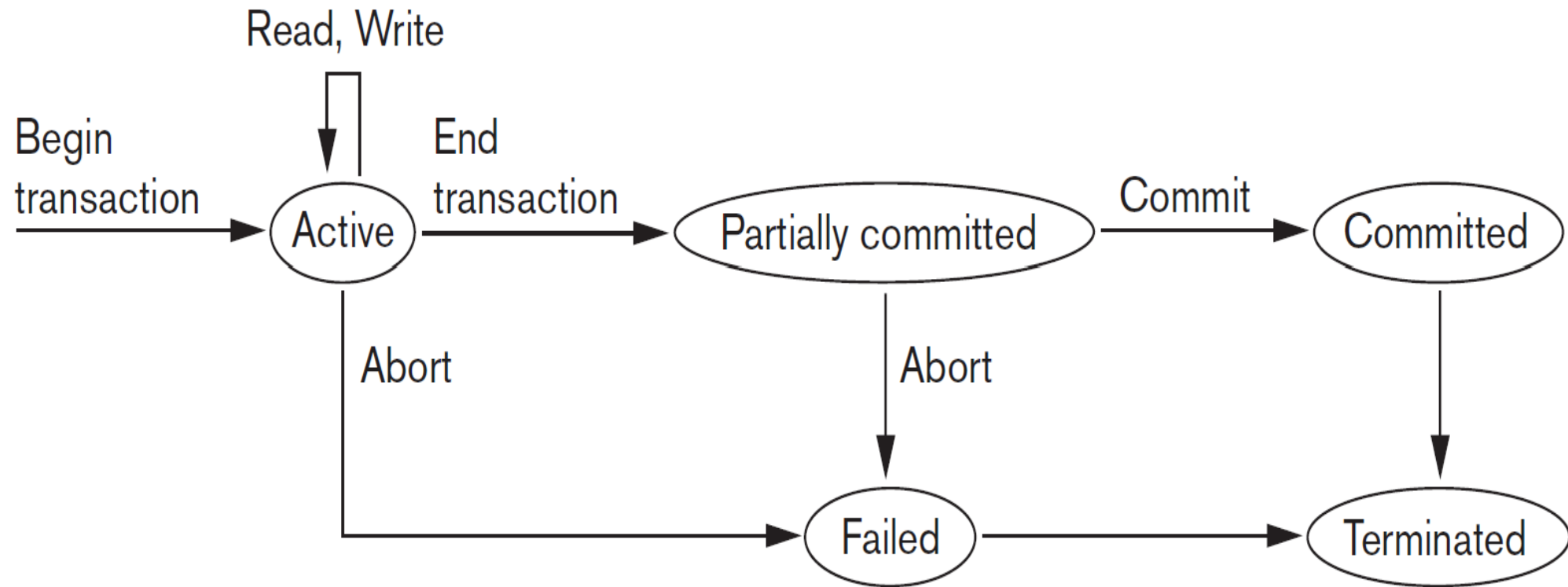
Transaction:

- Vérifiez si Tudor a 100 \$.
- Déduire 100 \$ de son compte. – *je n'aime pas s'arrêter ici !*
- Ajoutez 100 \$ à son compte carte crédit – *OK fin*

# Transaction – cycle de vie

- **Début transaction** (BEGIN TRANSACTION)
- **Opérations** lecture, écriture
- **Fin transaction** (END\_TRANSACTION) – fin de transaction, vérifications si les changements peuvent être appliquées dans la base de données (COMMIT partial)
- **Confirmer** - COMMIT\_TRANSACTION – les transactions sont écrites dans la base de données d'une façon permanente.
- **Annuler** - ROLLBACK – les vérifications BD ou applicatives rejettent la transaction et la BD fait un marche-arrière de données actualisées
- **Términation** – les vues de systèmes sont mises a jour avec le cycle de vie de la transaction.

# Transaction – états de transition



# Transaction – recouvrement

En arrière , un mécanisme de recouvrement en cas d'échec

Opérations propres à la récupération, avant chaque COMMIT

- défaire la transaction (*undo*) écriture de blocks UNDO dans le journal
- refaire la transaction (*redo*) écriture des blocks REDO dans le journal

Le journal contient un enregistrement de chaque opération WRITE qui modifie la valeur d'un élément de la base de données

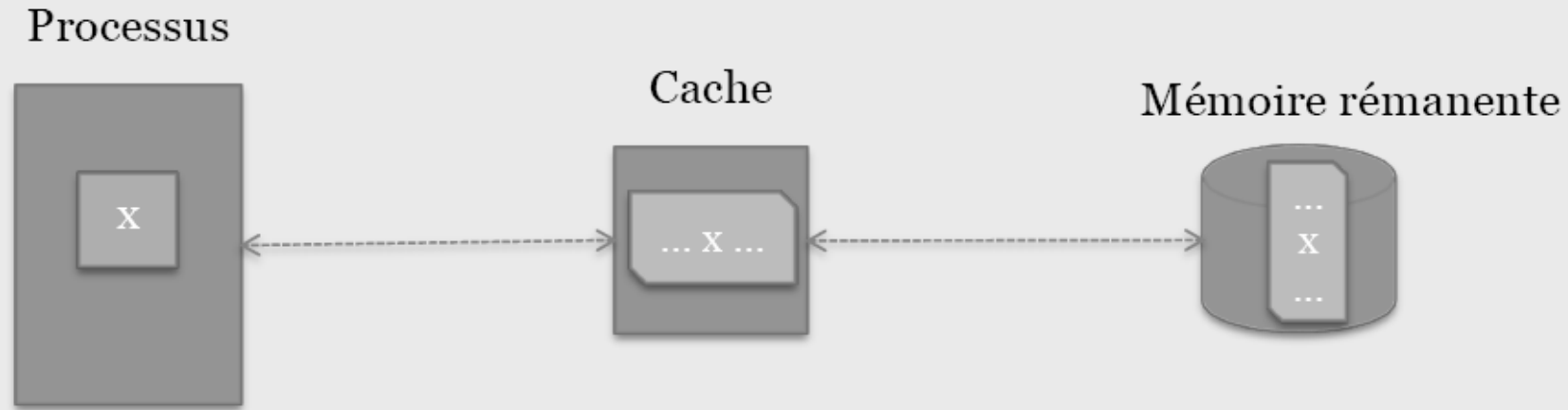
Il est possible pour annuler l'effet de ces opérations WRITE d'une transaction T en retraçant le journal et en réinitialisant tous les éléments modifiés par une opération WRITE de T à leurs anciennes valeurs.



# Transaction – recouvrement

- Slides 31 - 44 Luc Lavoie

# Gestion de la mémoire tampon (cache)



- X est un élément dénotable.
- Il est stocké dans un ou plusieurs blocs (du cache comme de la mémoire).
- Il pourrait y avoir plusieurs niveaux de cache.
- En général, les frontières de blocs et d'éléments ne coïncident pas.

# Log Buffer (journal cache, tampon)

- Le SGBD conservent certains blocs de fichier journal dans le tampon de journal pour des raisons de performance. Cela évite la surcharge de plusieurs écritures sur disque du même tampon de fichier journal.
- Au moment d'une panne du système, seules les entrées de journal qui ont été écrites sur le disque sont prises en compte dans le processus de récupération si le contenu de mémoire principale sont perdues.
- Par conséquent, avant qu'une transaction n'atteigne son point de COMMIT, toute partie du journal qui n'a pas encore été écrite sur le disque est écrite sur le disque.
- Ce processus est appelé FORCE-WRITING.

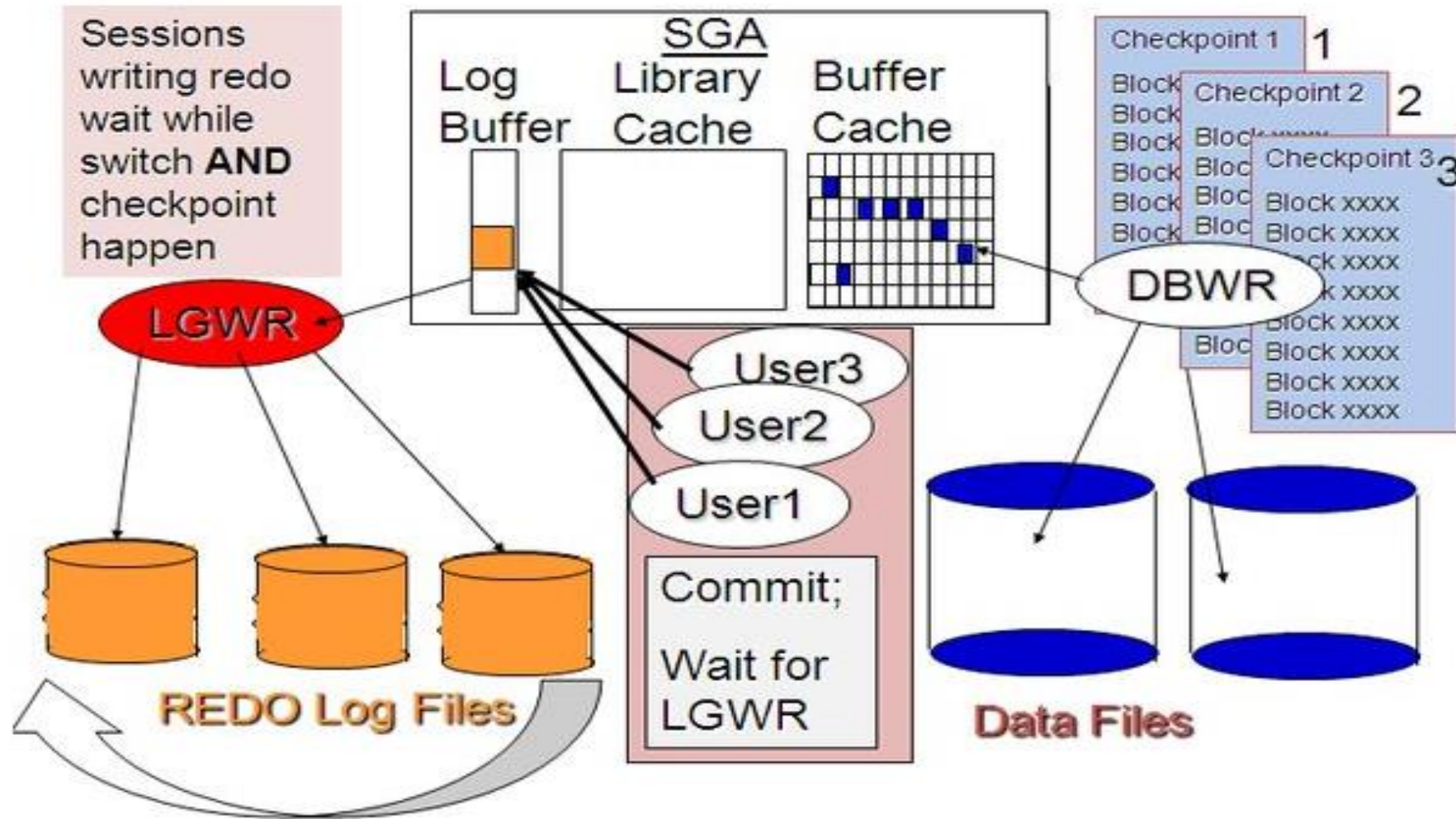
# DBMS cache (cache BD, tampon BD)

- Le cache du SGBD contient les pages du disque contenant les données en cours de traitement.
- Si tous les tampons du cache du SGBD sont occupés, une politique de remplacement de page est nécessaire pour sélectionner les tampons particuliers à remplacer.

Plusieurs méthodes, les plus connues :

- **DS** - Dans un SGBD, il existe différents types de pages disque : pages d'index, pages de fichiers de données, pages de fichiers journaux, etc. Dans cette méthode, le cache du SGBD est divisé en domaines distincts (ensembles de tampons). LRU – le moindre utilisé , GRU – priorités par domaines
- **Hot Set** - détermine pour chaque algorithme de traitement BD l'ensemble des pages disque qui seront accédées de manière répétée, et elle ne les remplace pas tant que leur traitement n'est pas terminé.

# Gestion de la mémoire



# Sérialisabilité

# Cédule , ordonnancement, planification (*schedule*)

L'ordre d'exécution des opérations de toutes les différentes transactions est connu sous le nom de calendrier (ou historique).

Un **ordonnancement (ou historique) S** de  $n$  transactions  $T_1, T_2, \dots, T_n$  est un ordre des opérations dans les transactions. Les opérations de différentes transactions peuvent être **entrelacées** en  $S$ .

L'ordre des opérations dans  $S$  est considéré comme un **ordre total**, ce qui signifie que pour deux opérations quelconques dans le programme, l'une doit se produire avant l'autre.

# Conflit

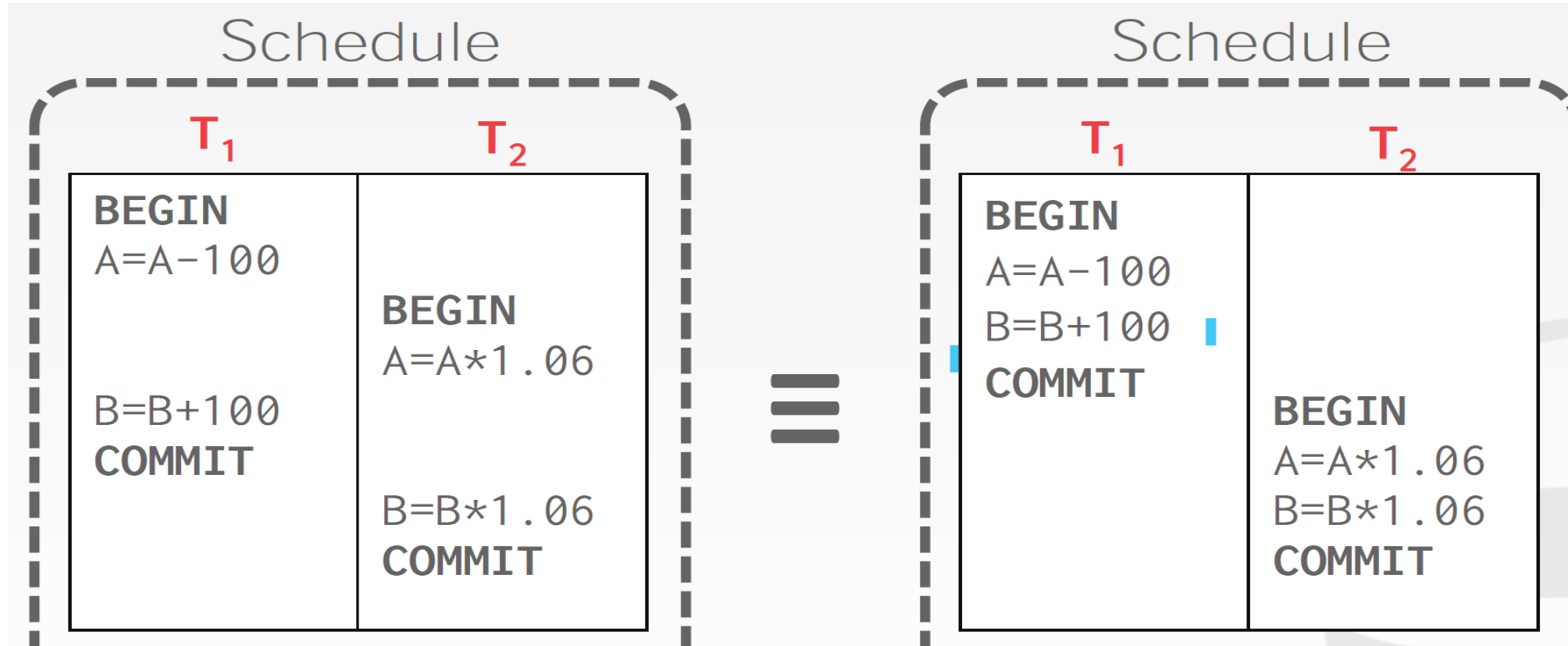
Deux opérations dans un échéancier de transactions sont dites **en conflit** si :

- (1) elles appartiennent à des transactions différentes ;
- (2) ils accèdent au même élément X ; et
- (3) au moins une des opérations est un **écrire(X)**.

Intuitivement, deux opérations sont en conflit si la **modification de leur ordre peut entraîner un résultat différent**.



# Conflicts - BON

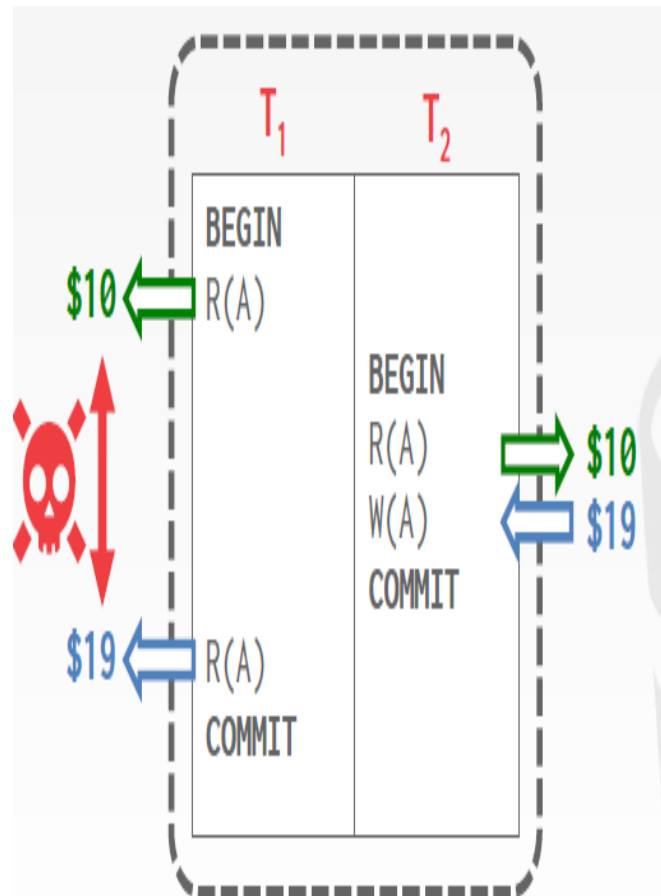


# Conflits - MAUVAIS

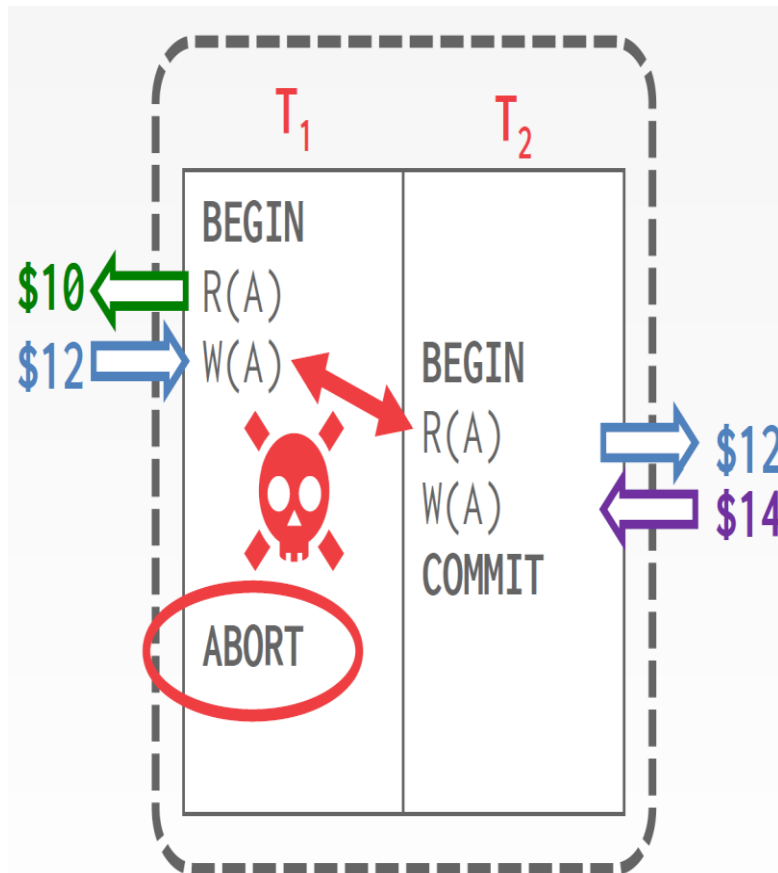
$T_1$	$T_2$
<b>BEGIN</b> A=A-100       B=B+100 <b>COMMIT</b>	<b>BEGIN</b> A=A*1.06 B=B*1.06 <b>COMMIT</b>

# Conflits d'entrelacement

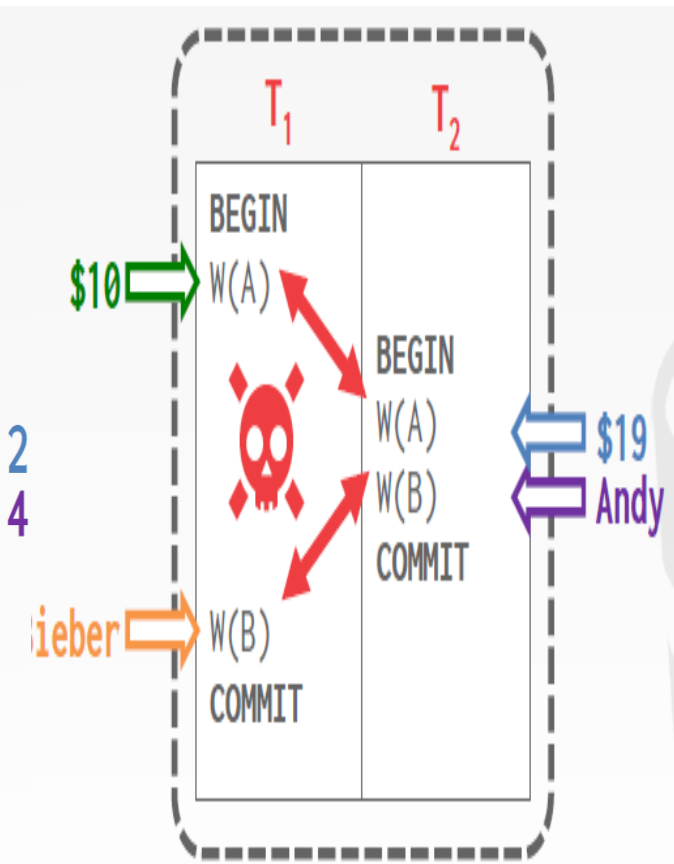
Conflits Lecture – Écriture



Conflits Écriture - Lecture



Conflits Écriture - Écriture



# Cédoule complète

1. L'ordre des opérations dans  $S$  est exactement l'ordre des opérations dans  $T_1, T_2, \dots, T_n$  y compris une opération de validation ou d'abandon comme dernière opération pour chaque transaction dans le programme.
2. Pour tout couple d'opérations d'une même transaction  $T_i$ , leur **ordre relatif** d'apparition dans  $S$  **est le même** que leur ordre d'apparition dans  $T_i$ .
3. Pour deux opérations en conflit, l'une des deux doit se produire avant l'autre dans le programme.

# Sérialisabilité et entrelacement

Nous entrelaçons les txns pour maximiser le temps

→ E/S disque/réseau lentes.

→ Processeurs multicœurs.

Lorsqu'un txn se bloque à cause d'une ressource , (en attente d'une entrée de données), une autre txn peut continuer à s'exécuter et progresser.

**Est-il possible d'entrelacer toutes les transactions ?**

- Recouvrir les transactions en conflit dans les cédules ?
- Avoir des résultats correctes comme dans le cas où toutes les transactions sont sérialisées ?

# Recouvrement de cédules

- Une planification dans laquelle une transaction **validée** doit être **annulée** pendant la récupération est appelée **non récupérable** et ne doit donc pas être autorisée par le SGBD.
- Un ordonnancement S est **récupérable** si aucune transaction T dans S n'est validée (COMMIT) tant que toutes les transactions T' qui ont écrit un élément X que T lit n'ont pas été validées (COMMIT).

# Ex. Recouvrement de cédules

## *Recouvrement possible*

*Sa'*:  $r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;$

## *Recouvrement impossible*

*Sc*:  $r1(X); w1(X); r2(X); r1(Y); w2(X); c2; a1;$

**Sc n'est pas récupérable car T2 lit l'élément X à partir de T1, mais T2 commit avant T1.**

**Si T1 annule (*aborts*) , la valeur sera incorrecte.**

**Pour que l'horaire soit récupérable, l'opération c2 dans Sc doit attendre après le COMMIT de T1.**

# Recouvrement de cédules

- **Restauration en cascade** (ou abandon en cascade) se produise quand une transaction non validée doit être annulée car elle a lu un élément d'une transaction qui a échoué. Ceci est illustré dans le programme *Se*, où la transaction *T2* doit être annulée car elle a lu l'élément *X* de *T1*, et *T1* a ensuite été abandonnée.

*Se*: *r1(X)*; *w1(X)*; *r2(X)*; *r1(Y)*; *w2(X)*; *w1(Y)*; *a1*; *a2*;

- Planification **sans cascade**, si chaque transaction de la planification ne lit que les éléments qui ont été écrits par des transactions validées. En vas, *r2(X)* va attendre que *T1* fait commit.

*Se*: *r1(X)*; *w1(X)*; ***r2(X)* va attendre**; *r1(Y)*; *w2(X)*; *w1(Y)*; *a1*; *a2*;

- **Planification stricte**, dans lequel les transactions ne peuvent ni lire ni écrire un élément *X* tant que la dernière transaction qui a écrit *X* n'a pas été validée (ou abandonnée).



# Caractériser les planifications en fonction de la sérialisabilité

Un échéancier S est **sériel** si, pour chaque transaction T participant à l'échéancier, toutes les **opérations** de T sont **exécutées consécutivement** dans l'échéancier ; sinon, l'horaire est dit non sériel. Le problème avec les horaires en série est qu'ils limitent la simultanéité

Un échéancier S de n transactions est **sérialisable** s'il est équivalent à un **échéancier sériel des n transactions**.

# Équivalences de sérialisabilité

Deux ordonnancements sont dits **équivalents de résultat** s'ils produisent le même état final de la base de données. Cependant, deux programmes différents peuvent produire accidentellement le même état final.

Deux ordonnancements sont dits **équivalents de conflit** si l'ordre relatif de deux opérations en conflit est le même dans les deux ordonnancements.

En utilisant la notion d'équivalence de conflit, nous définissons un **ordonnement S comme étant sérialisable s'il est (en conflit) équivalent à un ordonnement sériel S'**.

# Test de sérialisabilité

1. Pour chaque transaction  $T_i$  participant à l'horaire  $S$ , créez un nœud étiqueté  $T_i$  dans le graphe de priorité.
2. Pour chaque cas dans  $S$  où  $T_j$  exécute un **read\_item(X)** après que  $T_i$  exécute un **write\_item(X)**, créez une arête  $(T_i \rightarrow T_j)$  dans le graphe de priorité.
3. Pour chaque cas dans  $S$  où  $T_j$  exécute un **write\_item(X)** après que  $T_i$  exécute un **read\_item(X)**, créez une arête  $(T_i \rightarrow T_j)$  dans le graphe de priorité.
4. Pour chaque cas dans  $S$  où  $T_j$  exécute un **write\_item(X)** après que  $T_i$  exécute un **write\_item(X)**, créez une arête  $(T_i \rightarrow T_j)$  dans le graphe de priorité.
5. L'ordonnancement  $S$  est sérialisable si et seulement si le graphe de précedence n'a pas de cycles.

# Merci Andy Pavlo - CMSU

## DEPENDENCY GRAPHS

One node per txn.

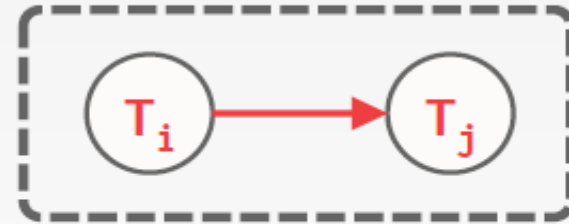
Edge from  $T_i$  to  $T_j$  if:

- An operation  $O_i$  of  $T_i$  conflicts with an operation  $O_j$  of  $T_j$  and
- $O_i$  appears earlier in the schedule than  $O_j$ .

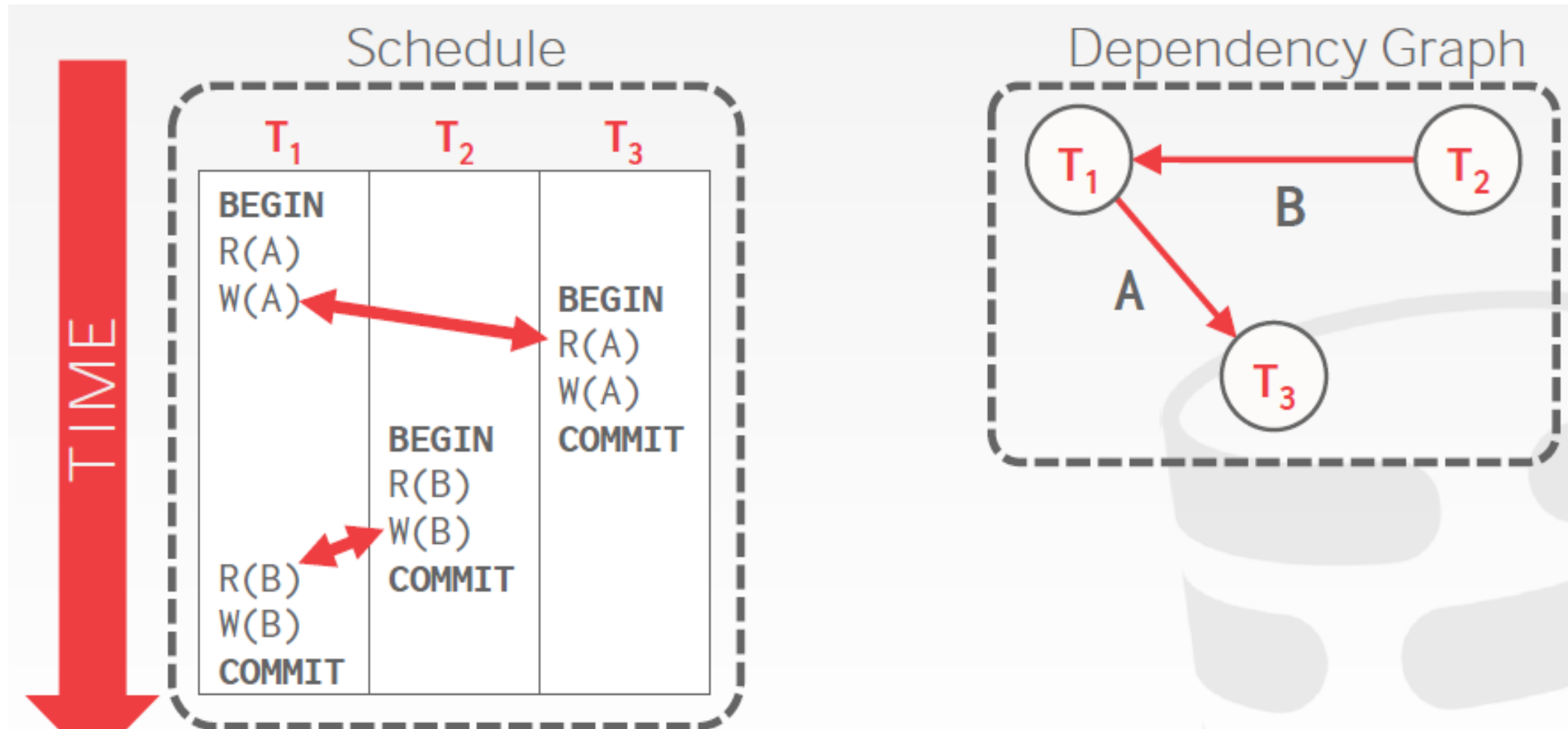
Also known as a **precedence graph**.

A schedule is conflict serializable iff its dependency graph is acyclic.

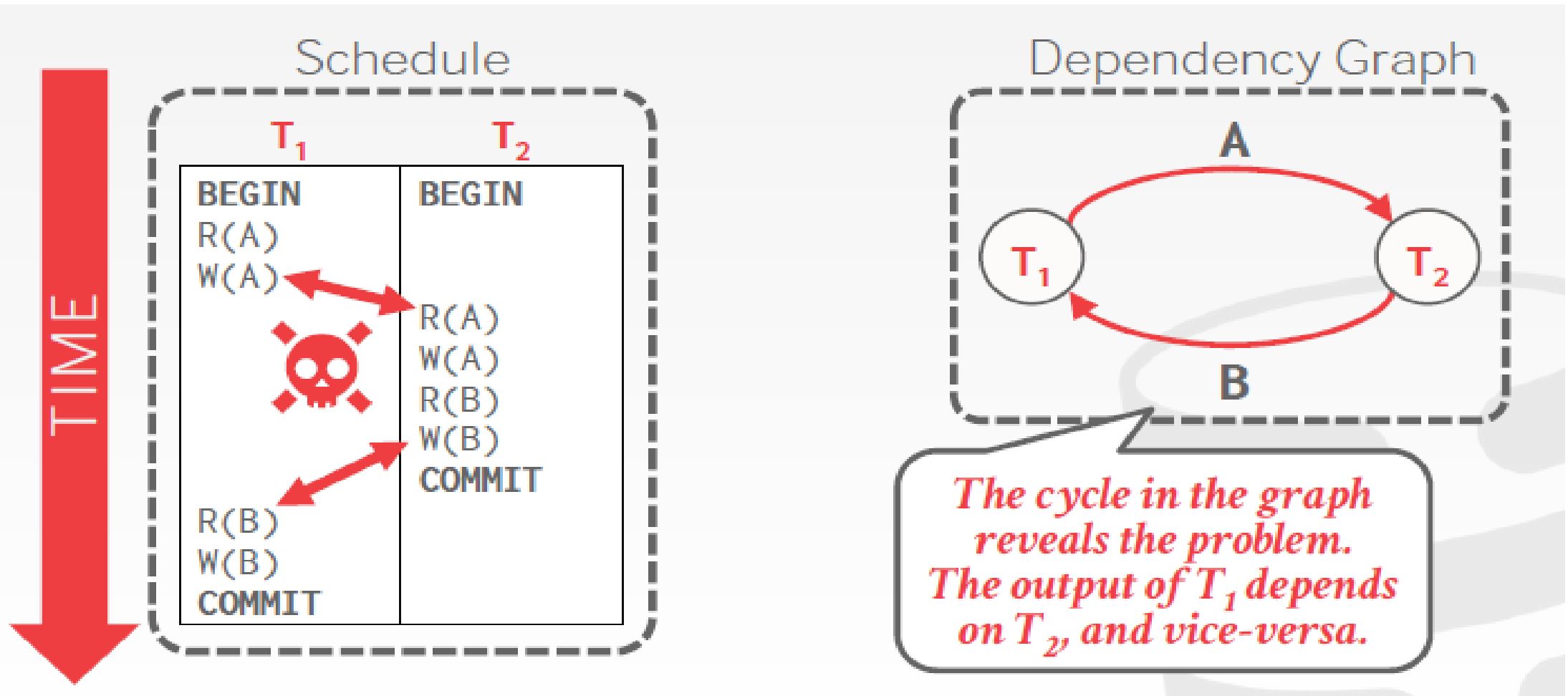
Dependency Graph



# Ex:1: C-sérialisable



## Ex:2: Non C-sérialisable



# Sérialisabilité

- Slides 45 – 51

# On continue dans le prochain cours

- Équivalence de cédules
- Graphes de dépendances
- C-Sérialisabilité , la solution la plus utilisée



ACID

# Propriétés désirables (1)

**Atomicité** : la transaction soit être exécuté dans son intégralité, soit ne pas être exécuté du tout.

**Cohérence** : La transaction est exécutée complètement du début à la fin sans sans interférence d'autres transactions, elle doit faire passer la base de données d'un état cohérent à un autre.

**Isolation** : Une transaction doit apparaître comme si elle était exécutée indépendamment des autres transactions, même si de nombreuses transactions s'exécutent simultanément. C'est-à-dire que l'exécution d'une transaction ne doit pas être perturbée par d'autres transactions exécutées simultanément.

**Durabilité** : Les modifications appliquées à la base de données par une transaction validée doivent persister dans la base de données.

# Propriétés désirables (2)

**Atomicité** : «tout ou rien"

**Cohérence** : «tout est correct"

**Isolement** : «comme si on est seul »

**Durabilité** : «survivre aux pannes»

# Atomicité

## **Approche 1 – la grande majorité de BD**

Journalisation avec de mecanismes UNDO et REDO

Les rangées UNDO sont gardées sur le disque ou dans la mémoire  
*(analogie : la boite noire d'un avion)*

## **Approche 2 – quelques systèmes comme CouchDB**

Pages copiées dans la mémoire (*shadow paging*)

SGBD fait de changements sur les copies

Le COMMIT fait l'écriture finale

# Isolation

L'isolation implique de mécanismes et algorithmes de verouillages qu'on va regarder dans le chapitre suivant.

- Un niveau d'isolement inférieur augmente la capacité de nombreux utilisateurs à accéder aux données en même temps. Mais cela augmente le nombre d'effets de concurrence, tels que les lectures erronées ou les mises à jour perdues.
- Un niveau d'isolement plus élevé réduit les effets négatifs. Mais cela nécessite plus de ressources système et augmente les chances qu'une transaction en bloque une autre.
- Le niveau d'isolement le plus élevé, sérialisable, garantit qu'une transaction récupérera exactement les mêmes données à chaque fois qu'elle répétera une opération de lecture. Mais il utilise un niveau de verrouillage susceptible d'avoir un impact sur les autres utilisateurs dans les systèmes multi-utilisateurs.

# Isolation - Trois types d'erreur possibles

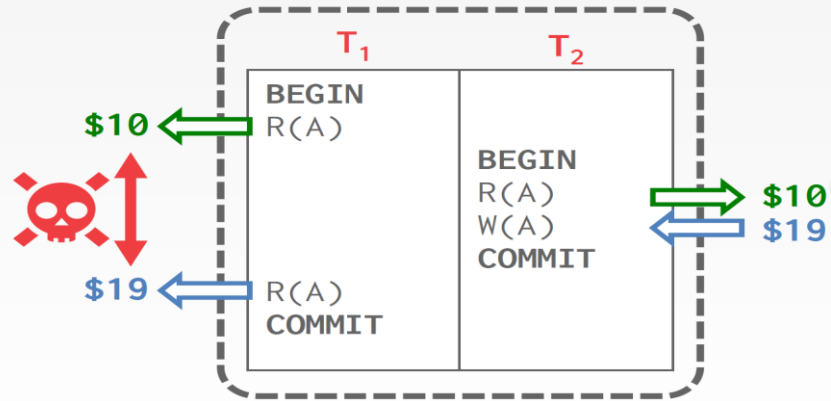
**Lectures sales (dirty reads)** - se produit lorsqu'une transaction est autorisée à lire les données d'une ligne qui a été modifiée par une autre transaction en cours d'exécution et qui n'a pas encore été validée en commit.

**Lectures non-repetables** - une lecture non répétable se produit lorsque, au cours d'une transaction, une ligne est extraite deux fois et les valeurs de la ligne diffèrent d'une lecture à l'autre..

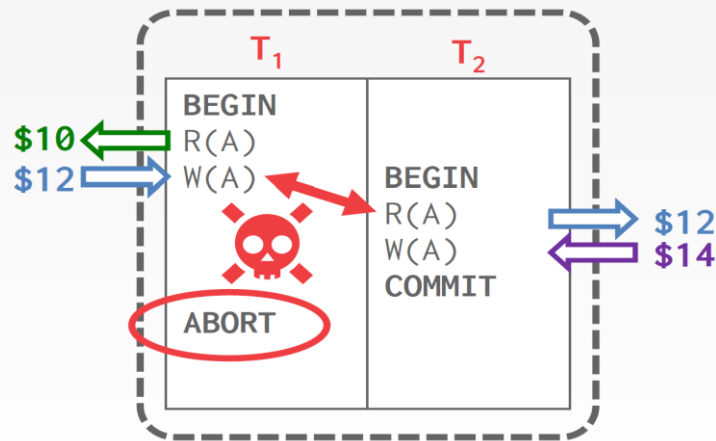
**Lectures phantomes** - une lecture fantôme se produit lorsque, au cours d'une transaction, de nouvelles lignes sont ajoutées ou supprimées par une autre transaction dans les enregistrements en cours de lecture.

# Isolation – les 3 types de conflits

## Unrepeatable Reads



## Reading Uncommitted Data ("Dirty Reads")



2. Tx1:

```
begin;
```

```
select * from ab; // empty set
```

3. Tx2:

```
begin;
```

```
insert into ab values(1,1);
```

```
commit;
```

4. Tx1:

```
select * from ab; // empty set, expected phantom read missing.
```

```
update ab set b = 2 where a = 1; // 1 row affected.
```

```
select * from ab; // 1 row. phantom read here!!!!
```

```
commit;
```

# Isolation – niveaux configurables par session

Niveau 0

Niveau 1

Niveau 2

Niveau 3

Niveau 4

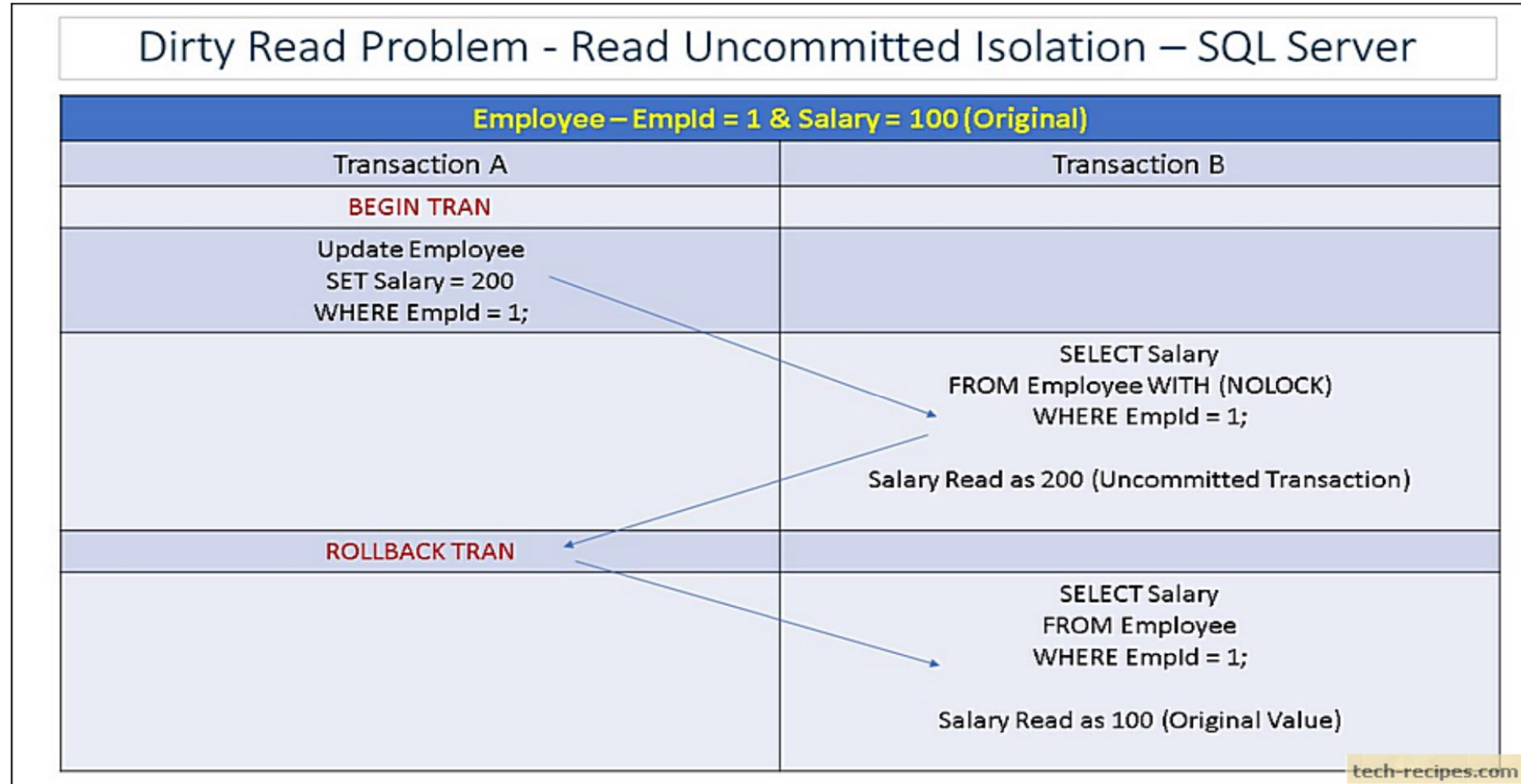
Isolation Level	Dirty Read	Non-Repeatable Read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Snapshot	No	No	No
Serializable	No	No	No

**Niveau 3 snapshot** accepte que autres sessions font des transactions sur les tuples utilisées dans ses transactions.

**Niveau 4 serializable** verrouille et ne permet pas d'autres transactions sur les tuples lues ou écrites.



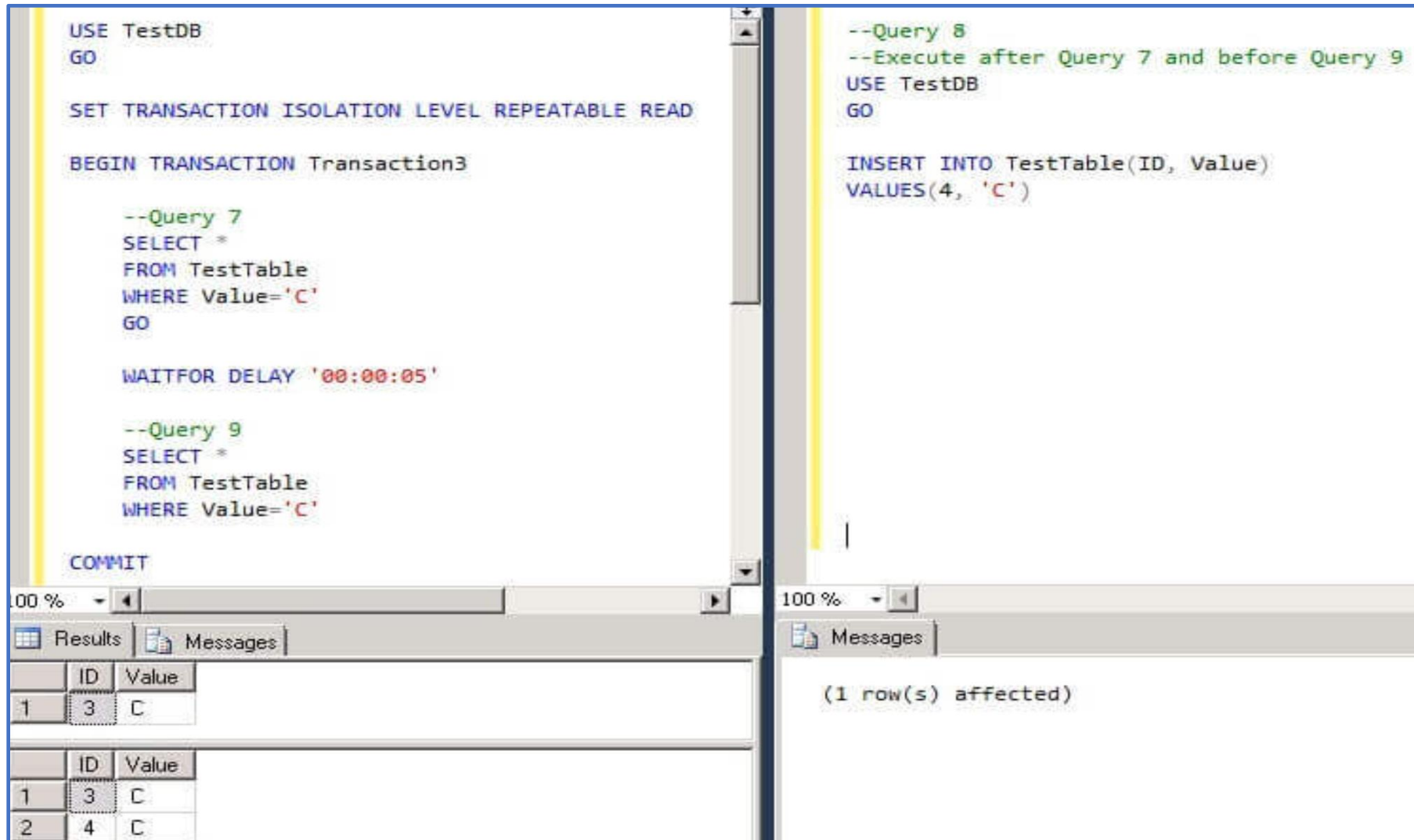
# Isolation – Read Uncommitted



# Isolation – Read Committed

Employee – EmpId = 1 & Salary = 100 (Original)	
Transaction A	Transaction B
<b>BEGIN TRAN</b>	
Update Employee SET Salary = 200 WHERE EmpId = 1;	
	SELECT Salary FROM Employee WITH (NOLOCK) WHERE EmpId = 1;  Salary Read as 100 (Original Value)
<b>ROLLBACK TRAN</b>	
	SELECT Salary FROM Employee WHERE EmpId = 1;  Salary Read as 100 (Original Value)

# Isolation – Repeatable Read



```
USE TestDB
GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRANSACTION Transaction3

--Query 7
SELECT *
FROM TestTable
WHERE Value='C'
GO

WAITFOR DELAY '00:00:05'

--Query 9
SELECT *
FROM TestTable
WHERE Value='C'

COMMIT
```

```
--Query 8
--Execute after Query 7 and before Query 9
USE TestDB
GO

INSERT INTO TestTable(ID, Value)
VALUES(4, 'C')
```

	ID	Value
1	3	C
2	4	C

(1 row(s) affected)

Si c'était juste un UPDATE on voyait juste une fois le C.

Un INSERT est visible (Phantom reads)

# Isolation – Snapshot

Les changements sur les tables sont permises dans chaque session.

Cependant, même après COMMIT la deuxième session ne voit le changement de la première session.

The screenshot displays two SQL Server sessions in a Snapshot isolation context. Session 1 (left) performs an update and then waits for 7 seconds. Session 2 (right) performs a select, an update, and another select. The results pane for Session 2 shows that the update in Session 1 is not visible to Session 2.

```
USE TestDB
GO

BEGIN TRANSACTION

UPDATE TestTable
SET Val='X'
WHERE Val='A'

WAITFOR DELAY '00:00:07'

COMMIT
```

Messages

(1 row affected)

```
USE TestDB
GO

BEGIN TRANSACTION

SELECT * FROM TestTable

UPDATE TestTable
SET Val='Y'
WHERE Val='X'

SELECT * FROM TestTable

COMMIT
```

Results

	ID	Val
1	1	A
2	2	B
3	3	C

	ID	Val
1	1	Y
2	2	B
3	3	C

# Isolation – Serializable

The screenshot displays two SQL query windows. The left window contains the following T-SQL code:

```
USE TestDB
GO

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION Transaction4

--Query 10
SELECT *
FROM TestTable
WHERE Value='C'
GO

WAITFOR DELAY '00:00:05'

--Query 12
SELECT *
FROM TestTable
WHERE Value='C'
GO

COMMIT

WAITFOR DELAY '00:00:05'

SELECT *
FROM TestTable
WHERE Value='C'
```

The right window contains the following T-SQL code:

```
--Query 11
--Execute after Query 10 and before Query 12
USE TestDB
GO

INSERT INTO TestTable(ID, Value)
VALUES(5, 'C')
```

At the bottom, the 'Results' pane shows the output of the first SELECT query (Query 10) as a table with 2 rows:

	ID	Value
1	3	C
2	4	C

Below this, the 'Messages' pane shows the output of the second SELECT query (Query 12) as a table with 3 rows:

	ID	Value
1	3	C
2	4	C
3	5	C

The 'Messages' pane on the right shows the output of the INSERT query (Query 11) as a message:

```
(1 row(s) affected)
```

La table TestDB est verrouillée.

Quand le COMMIT de la première session est exécuté, le INSERT de la première session est exécuté (il était bloqué).

Finalement la première session peut voir l'INSERT.

# Cohérence

Une fois le COMMIT effectué dans ma session, si les autres sessions interrogent mes modifications, sont-elles capables de la voir immédiatement ?

Les BDs distribuées qui reposent sur la réplication, une faible latence, ou les deux, doivent faire un compromis entre la cohérence de lecture, la disponibilité, la latence et le débit tels que définis par les théorèmes de CAP et PACELC.



# Cohérence forte – toujours en ACID

La cohérence forte offre des lectures garanties à la version validée la plus récente de l'élément. Le client n'obtiendra pas d'écriture non validée (non COMMIT) ou d'écriture partielle d'un réplica.

La cohérence forte convient aux applications qui ne tolèrent aucune perte de données due aux temps d'arrêt.

1. Cohérence la plus élevée
2. Performance la plus basse
3. Disponibilité la plus faible – on va regarder plus tard

# Cohérence forte en ACID – toujours





# Cohérence éventuelle - jamais en ACID

il n'y a aucune garantie d'ordre pour les lectures. En l'absence de toute autre écriture, les répliques finissent par converger. La cohérence éventuelle est la forme de cohérence la plus faible, un client peut lire des valeurs plus anciennes que celles qu'il avait lues auparavant. **Ex:** sommaire de tweets



# SQL – exemple Elmasri

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;

EXEC SQL SET TRANSACTION READ WRITE DIAGNOSTIC SIZE 5
        ISOLATION LEVEL SERIALIZABLE;

EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
VALUES ('Robert', 'Smith', '991004321', 2, 35000);

EXEC SQL UPDATE EMPLOYEE
SET Salary = Salary * 1.1 WHERE Dno = 2;

EXEC SQL COMMIT;

GOTO THE_END;

UNDO: EXEC SQL ROLLBACK;

THE_END: ... ;
```

# SQL PostgreSQL – rollback plus difficile en SQL

```
BEGIN;  
UPDATE accounts SET balance = balance - 100.00  
    WHERE name = 'Alice';  
SAVEPOINT my_savepoint;  
UPDATE accounts SET balance = balance + 100.00  
    WHERE name = 'Bob';  
-- oops ... forget that and use Wally's account  
ROLLBACK TO my_savepoint;  
UPDATE accounts SET balance = balance + 100.00  
    WHERE name = 'Wally';  
COMMIT;
```