Cours 2

La Gestion des Transactions

BDA
Master GL/RT - 2019- 2020

Plan du cours

- Le concept de transaction
- Les propriétés ACID
- Le Contrôle de concurrence
  - Two-phase Locking (2PL) & Two-phase commit (2PC)
- La Reprise après une panne

Traitement de fichiers (Pb)

Redondance et inconsistance des données

– certaines info se trouvent sur plusieurs fichiers

Difficulté d'accès aux informations non prévues

– nécessité d'écrire de nouveaux prog. d'accès

Dépendance : rep. interne / Applications

– changement de structure -> re-programmation des App

Atomicité et pb de concurrence

– erreur, pannes, accès concurrents -> introduisent des inconsistances

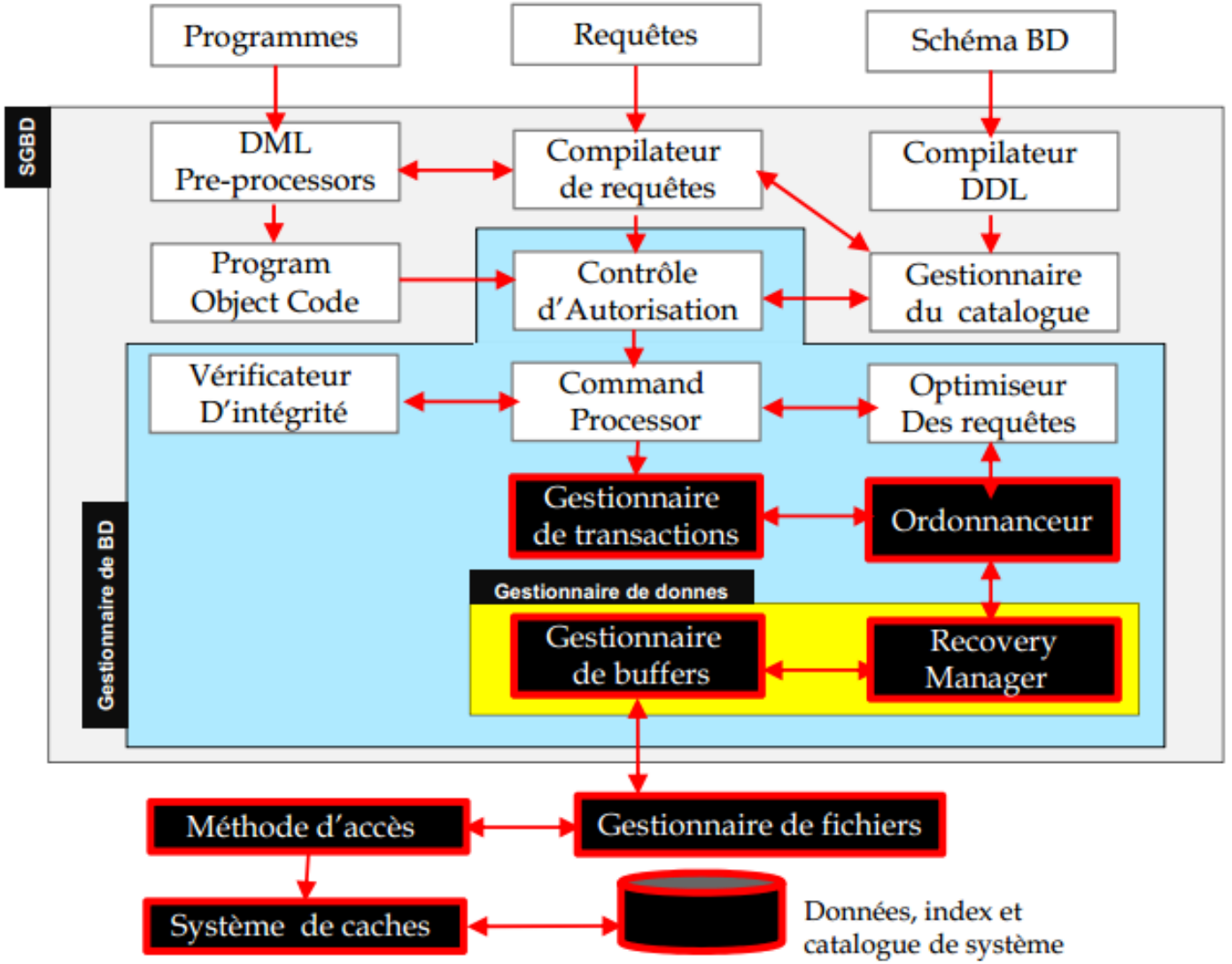
1960: les SGFs

➲Problème de surcharge ☹
➲Comment libérer le programmeur ?

⇨ SGBD ☺

- **1970**: Codd paper; les fondements des BDR
- **1980**: Les SGBD-R sur le marché

Composantes d'un SGBD



Le concept de transaction

- **Définition:**
  - Une **transaction** est une séquence d'opérations (lecture/ écriture) qui forment une seule unité de travail.
  - **Exemple:** Virements en banque, achats en ligne, inscription aux cours
  - Une transaction est souvent déclenchée par un programme d'application

    - commencer une transaction **START TRANSACTION**
    - Accès à la base (lire/écrire)
    - Calculs en MC
    - fin transaction : Une instruction COMMIT ou ROLLBACK est exécutée.

```
BEGIN TRANSACTION
   [SQL statements]
COMMIT     or
ROLLBACK (=ABORT)
```

Challenges

- Voulez-vous exécuter plusieurs applications simultanément
  - Toutes ces applications lisent et écrivent des données dans le même DB
- **Solution simple**: ne servir qu'une application à la fois
  - Quel est le problème?

- Voulez-vous que plusieurs opérations soient exécutées de manière atomique sur le même SGBD?

C'est quoi le problème ?

Account 1 = $100
Account 2 = $100
Total = $200

- **App 1:**
  - Set Account 1 = $200
  - Set Account 2 = $0
- **App 2:**
  - Set Account 2 = $200
  - Set Account 1 = $0
- At the end:
  - Total = $200

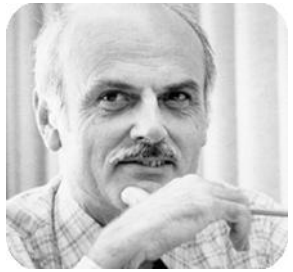
- **App 1**: Set Account 1 = $200
- **App 2**: Set Account 2 = $200
- **App 1**: Set Account 2 = $0
- **App 2**: Set Account 1 = $0
- At the end:
  - Total = $0

C'est ce qu'on appelle la mise à jour perdue aka conflit WRITE-WRITE

Turing Awards in Data Management

Charles Bachman, 1973

IDS and CODASYL



Ted Codd, 1981

Relational model

Jim Gray, 1998

Transaction processing

Michael Stonebraker, 2014

INGRES and Postgres

Responsabilité du SGBD

- SGBD doit vérifier les propriétés règles **ACID** (*atomicity*, *consistency*, *isolation* et *durability*)
- Garantir l'exécution correcte des transactions
- Gérer l'exécution concurrente des transactions

Propriétés des Transaction ACID

Atomicité: Le fait qu'une transaction est indivisible, elle doit s'exécuter entièrement ou pas du tout

Cohérence: Une transaction doit préserver la cohérence de la base de données.

- contraintes d'intégrités
- règles métier définies
- règles complexes
- responsabilité du développeur/DBA

Propriétés des Transaction ACID

Isolation: Vu qu'une transaction est une unité indivisible, ces actions ne doivent pas être séparées par des opérations de base de données ne faisant pas partie de la transaction.

- pas **d'interférence** entre les transactions qui s'exécutent en même temps dans le système

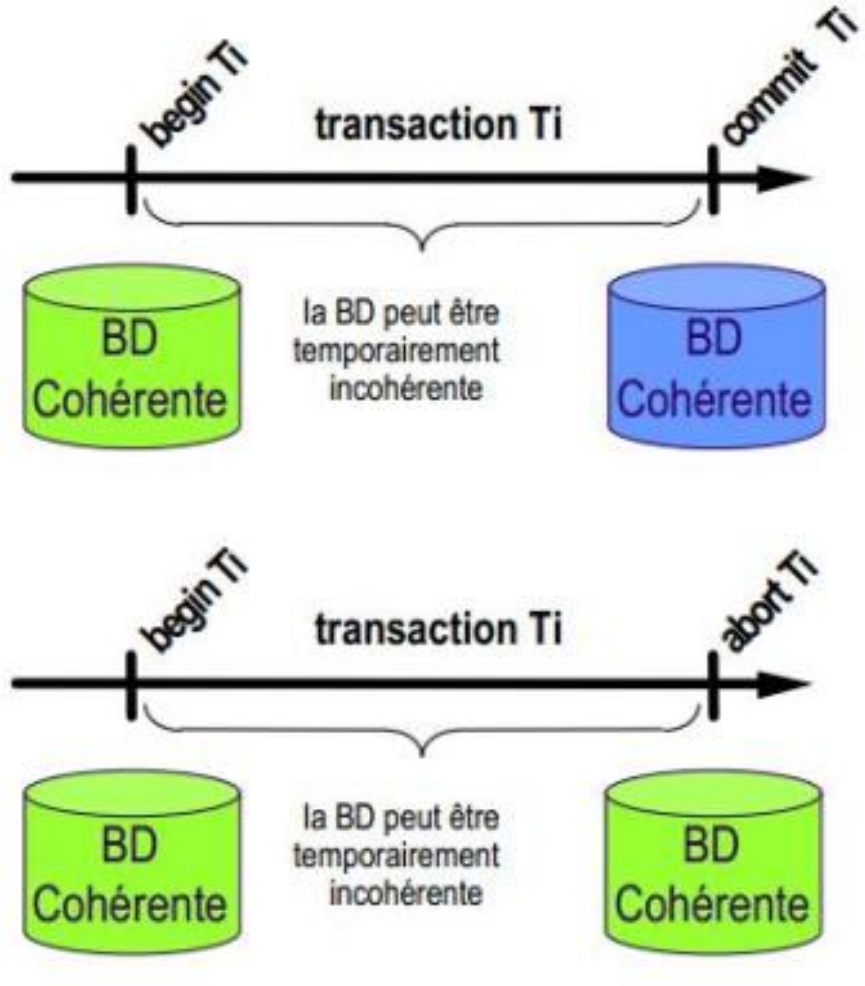
Durabilité: Même si le système exécute correctement les transactions, ceci ne servira pas à grand chose si une situation d'erreur fait que le système perd la trace des transactions terminées avec succès. Pour cela, **les actions des transactions doivent persister** même en cas des situations d'échec les plus graves.

- Gestionnaire de pannes : journalisation des opérations, algorithme REDO
- **Point fort des SGBD**, qui peuvent résister aux pannes, sans perdre de données et en restituant la base dans un état cohérent.

Propriétés des Transaction ACID

A retenir :

- **A**tomicité
  - tout ou rien
- **C**onsistance
  - cohérence sémantique
- **I**solation
  - pas de propagation de résultats non validés
- **D**urabilité
  - persistance des effets validés

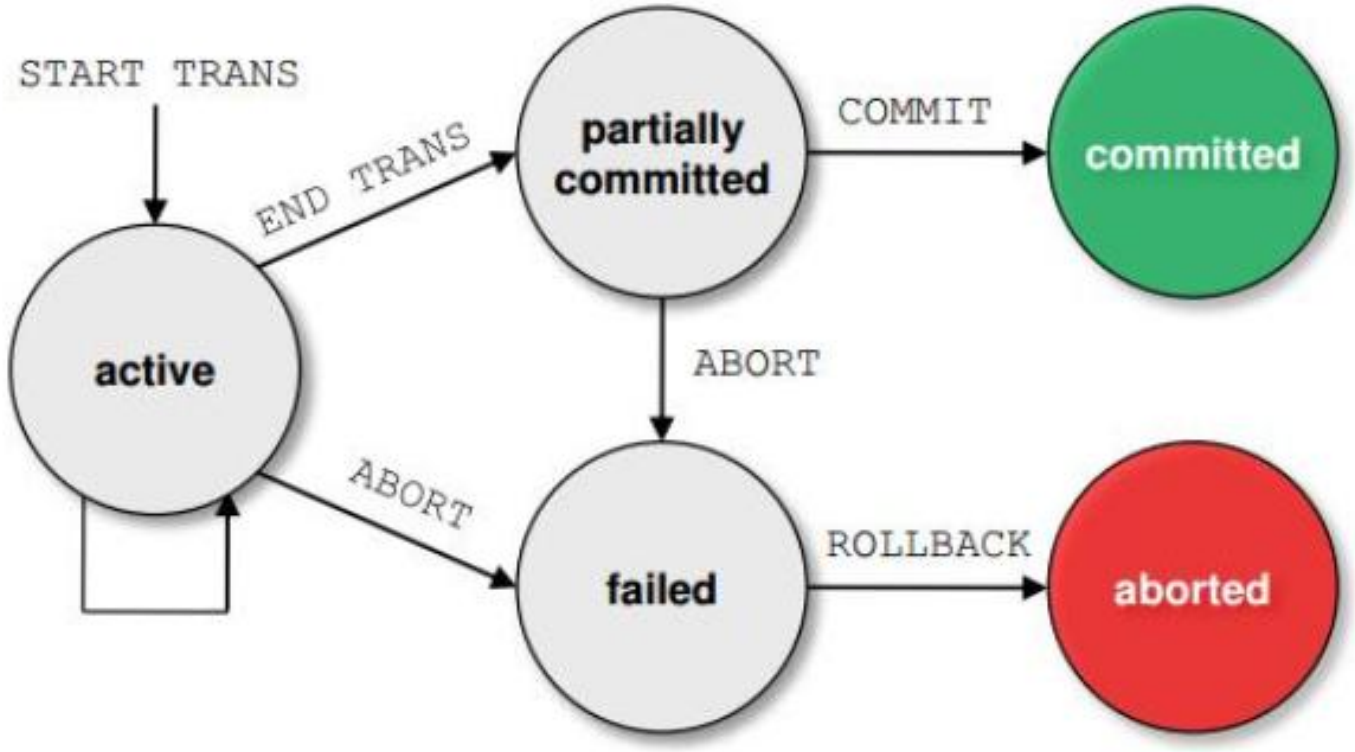


Types de transaction

Type	Description
Langage de manipulation de données (LMD)	Un ensemble d'instructions Insert, Update, Delete
Langage de définition de données (LDD)	Une instruction Create, Alter, Drop. L'instruction Truncate fait aussi partie du LDD
Langage de contrôle de données (LCD)	Une instruction Grant ou Revoke

Etat d'une transaction

- **Actif**
- **Partiellement validée**
- **Validée**
- **Echec**
- **Avortée (Abondonnée)**



Deux principaux problèmes de transaction ..

Il y a deux principaux problèmes de transaction

– exécution simultanée de plusieurs opérations (concurrence)

– récupération après des pannes matérielles et des plantages du système (panne)

• Pour préserver **l'intégrité des données,** le SGBD doit se assurer les propriétés **ACID** pour toute transaction

Problèmes rencontrés par l'exécution concurrente

perte des modifications

• 2 transactions accèdent au même élément d'une BD et l'exécution des opérations est décalée

• X=4, Y=8, N=2, M=3

T1: (joe)	T2: (fred)	X	Y
read_item(X);			
X:= X - N;			
	read_item(X);		
	X:= X + M;		
write_item(X);			
read_item(Y);			
	write_item(X);		
Y:= Y + N;			
write_item(Y);			

Résultat incorrect: : X=7 et Y=10
Résultat correct : X= 5 et Y=10

➲ Solution ??

Problèmes du Verrouillage

- T_1: R(A), W(B)
- T_2: R(B), W(A)

- T_1 holds the lock on A, waits for B
- T_2 holds the lock on B, waits for A

C'est un interblocage (deadlock!)

Question ?

Quelles sont les propriétés ACID à assurer pour garantir :

- Le contrôle de concurrence
- La reprise après une panne

Les pannes

Erreur de Transaction

Erreurs logiques (opération incorrecte, condition non satisfaite, ...)

Interblocage, …

Crash système

Pb d'alimentation, Reboot, …

Erreur de disque

En partie ou en totalité

Contrôle de concurrence …

Ordonnancement séquentiel

- Un Ordonnanceur en série est un programme dans lequel les transactions sont exécutées les unes après les autres, dans un ordre séquentiel.

- Rien ne peut aller mal si le système exécute les transactions en série

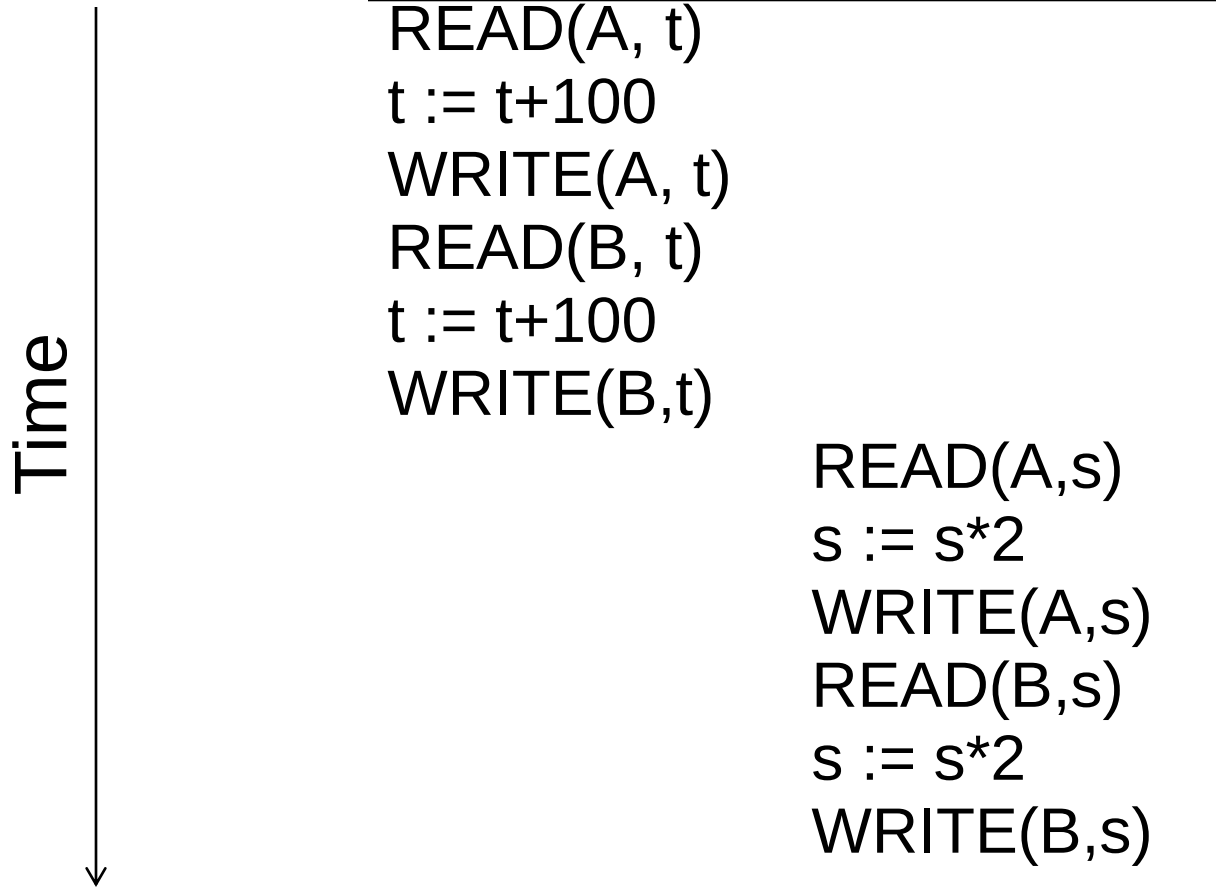
- Les SGBDs ne le font pas parce que nous voulons de meilleures performances globales du système

Exemple

A et **B** sont des éléments
dans la base de données
t et **s** sont des variables
dans le code source de la transaction

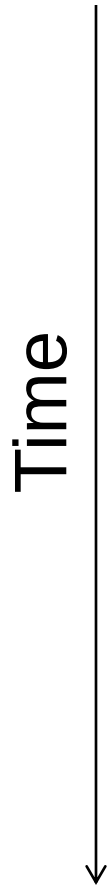
T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

Exemple d'ordonnancement séquentiel



T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

Exemple d'ordonnancement séquentiel



T1	T2
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

Ordonnancement sérialisable

Un Ordonnancement est sérialisable si elle est équivalent à un Ordonnancement en série (séquentiel)

Ordonnancement sérialisable

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

This is a serializable schedule.
This is NOT a serial schedule

Ordonnancement non-sérializable

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Comment détecter si un Ordonnancement est sérialisable?

Notation:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Idée clé: se concentrer sur les opérations en conflit

conflicting operations

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW
- Read-Read?

Conflit-sérialisable

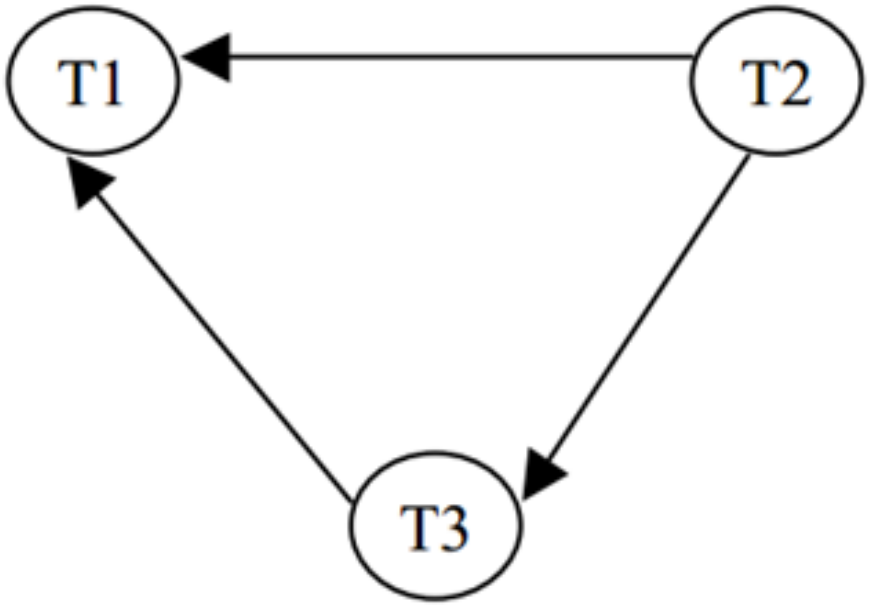
- Un Ordonnancement est Conflit-sérialisable si elle peut être transformée en un ordonnancement en série par une série d'échanges d'actions adjacentes non conflictuelles (WR, RW, WW).

- Chaque Ordonnancement Conflit-sérialisable est sérialisable

Exemple 1

c). *r3(X); r2(X); w3(X); r1(X); w1(X);*

The serialization graph is:

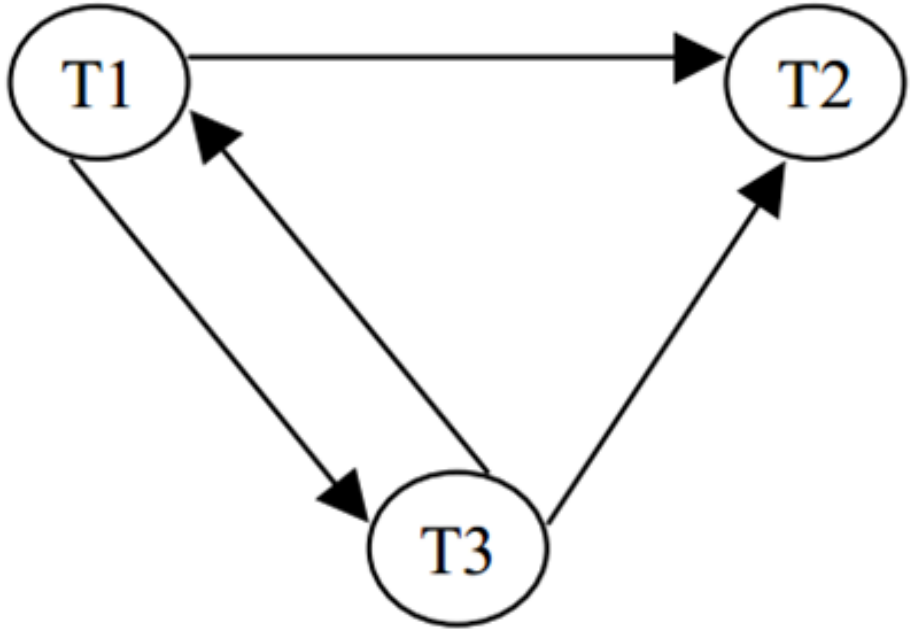


This schedule is conflict-serializable (Serializable)

Exemple 2

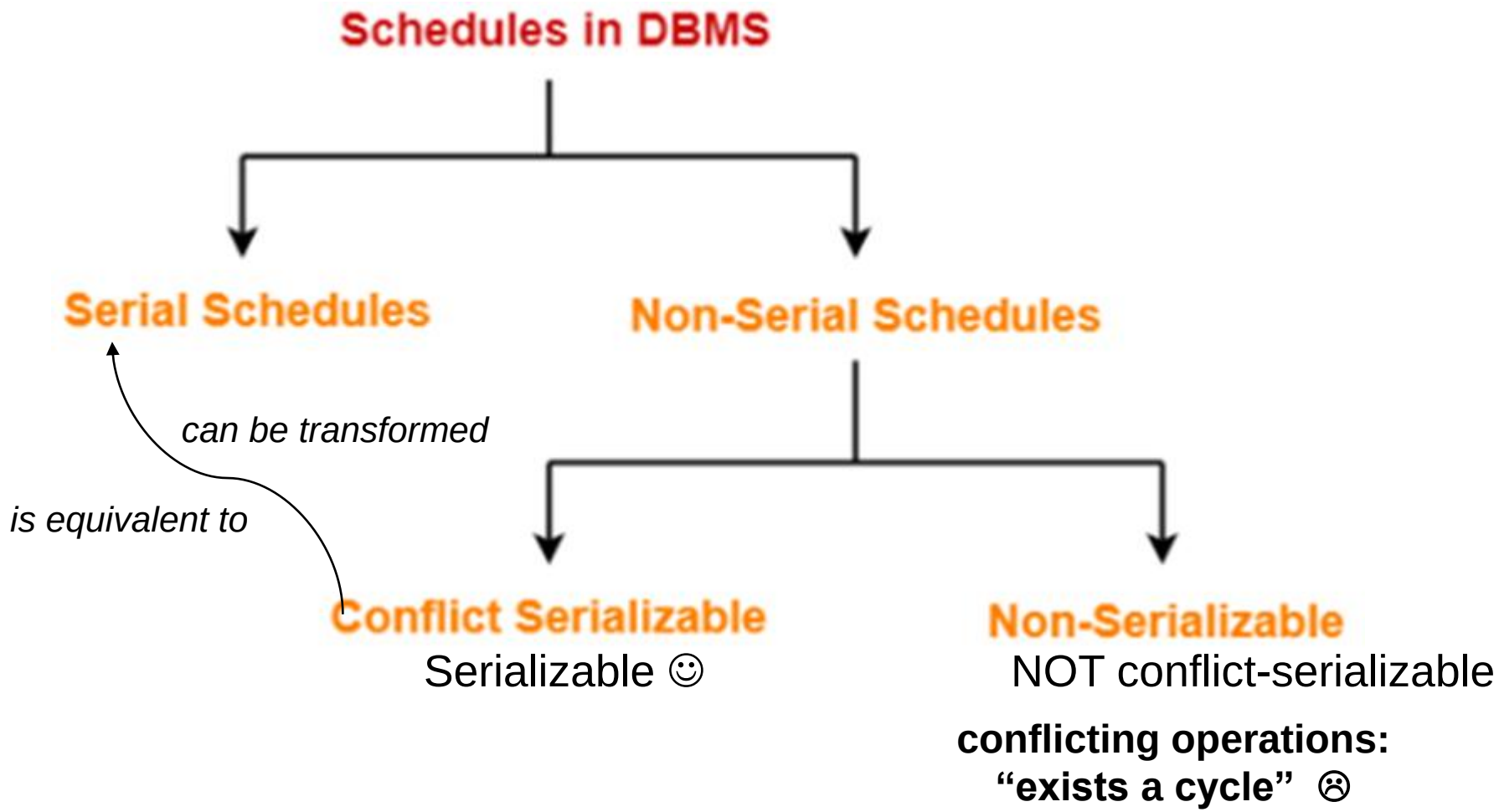
b). *r1(X); r3(X); w3(X); w1(X); r2(X);*

The serialization graph is:



This schedule is NOT conflict-serializable

Schedules in DBMS



Implémentation d'un ordonnanceur

Ordonnanceur

- Ordonnanceur= le module qui planifie les actions de la transaction, assurant la possibilité de **la sérialisation**
- Aussi appelé: Gestionnaire de contrôle de concurrence
- Nous discutons ensuite comment un planificateur peut être implémenté

Un ordonnanceur avec verrouillage

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)

By using locks scheduler ensures conflict-serializability

Verrouiller la granularité: quels éléments de données sont verrouillés?

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
  - E.g., SQL Server
- Coarse grain locking (e.g., tables, entire database)
  - Many false conflicts
  - Less overhead in managing locks
  - E.g., SQL Lite
- Solution: lock escalation changes granularity as needed

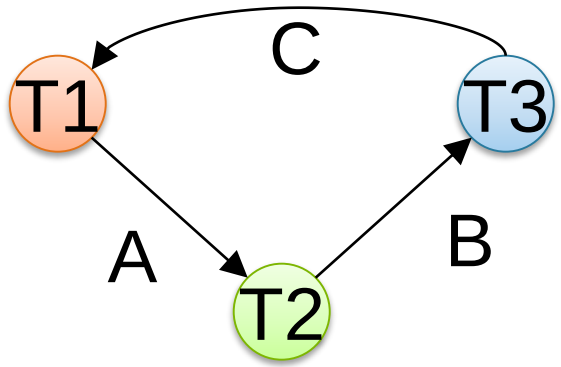
Verrouillage en deux phases (2PL)

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Ordonnancement non-sérializable

T1	T2
READ(A)	
A := A+100	
WRITE(A)	
	READ(A)
	A := A*2
	WRITE(A)
	READ(B)
	B := B*2
	WRITE(B)
READ(B)	
B := B+100	
WRITE(B)	

Exemple

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Scheduler has ensured a conflict-serializable schedule

Un nouveau problème: Non Ordonnancement récupérable

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

Un nouveau problème: Non Ordonnancement récupérable

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
Rollback	Commit

Elements A, B written by T1 are restored to their original value.

Un nouveau problème: Non Ordonnancement récupérable

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
Rollback	

Dirty reads of A, B lead to incorrect writes.

Elements A, B written by T1 are restored to their original value.

Un nouveau problème: Non Ordonnancement récupérable

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

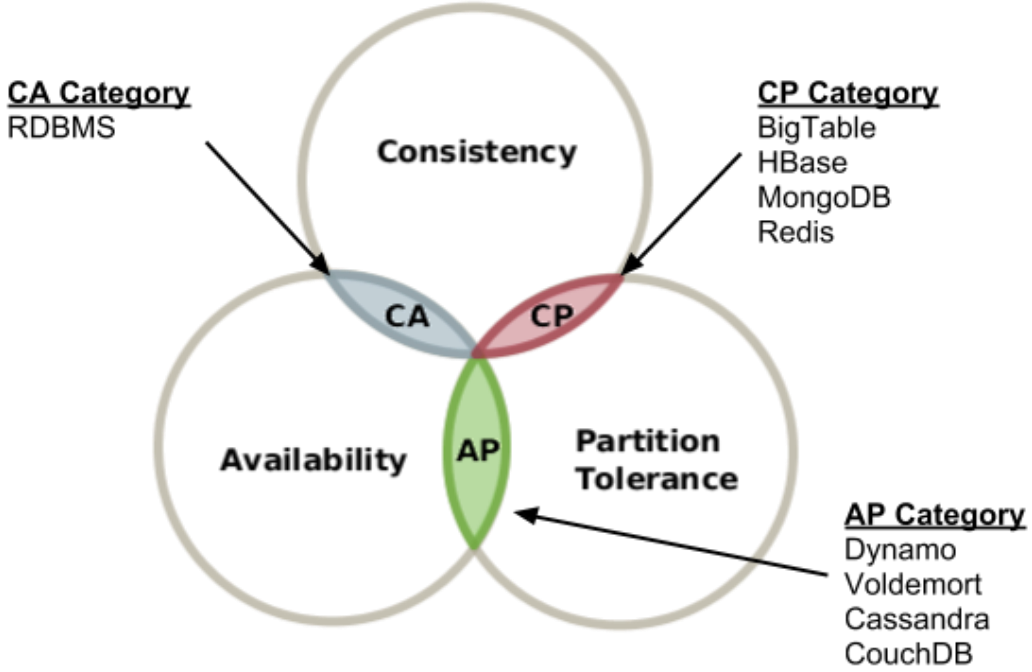
Dirty reads of A, B lead to incorrect writes.

Elements A, B written by T1 are restored to their original value.

Can no longer undo!

NoSQL (No Only SQL)

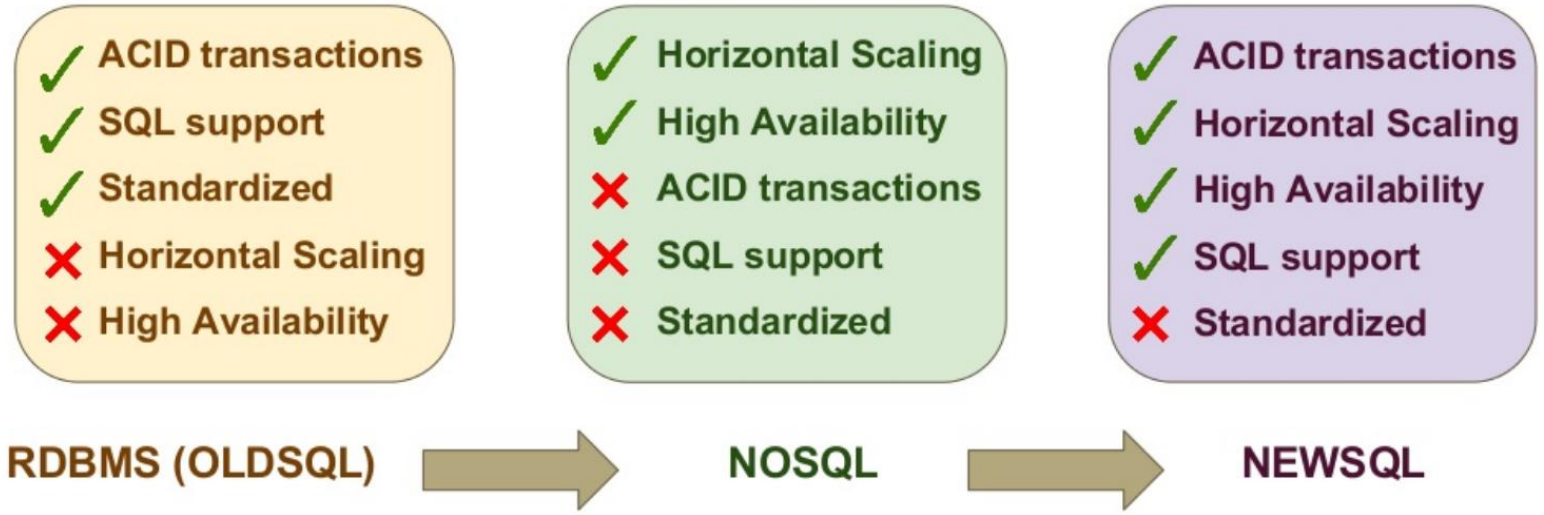
- Focus on high-availability & high-scalability (cap theory):
- No ACID transactions
- CAP Theorem



We can not achieve all the three items
In distributed database systems (center)

NewSQL

- Provide same performance for OLTP workloads as NoSQL DBMSs without giving up ACID:
  → Relational / SQL
  → Distributed
  → Usually closed-source (e.g. VoltDB, NuoDB)



Le prochain cours

La reprise après une panne