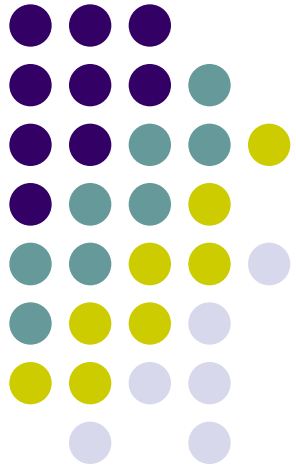
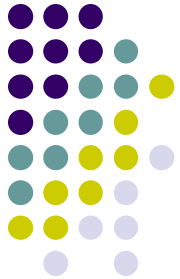
Fiabilité des données

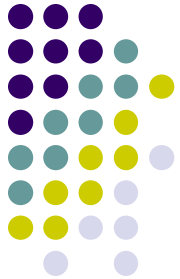


Introduction



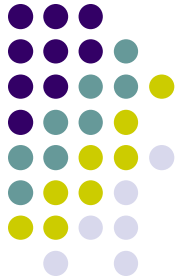
- Constatations
  - Plusieurs milliers d'« utilisateurs » utilisent les données d'une BD de manière concurrente
  - La base de donnée n'est pas à l'abri d'une panne
- En conséquence
  - Le SGBD doit assurer
    - la cohérence des données
    - Fiabilité
    - **Grâce** à la notion de **transaction**

Rappel sur les Transactions



- Suite d ’actions comprises entre
  - Begin_Transaction, qui marque le début d ’une transaction
  - et Commit_Transaction, ou Abort_Transaction, qui termine une transaction
- Exemple Typique : Begin_Transaction
  CpteA+=100;
  CpteB-=100;
  Commit_Transaction
- cette suite d ’actions répond aux propriétés ACID:
  - Atomicité : Tout ou rien
  - Cohérence : Données sont laissées dans un état cohérent
  - Isolation : Modifications pas visibles pendant la transaction
  - Durabilité : Les actions validées ne peuvent pas être perdues

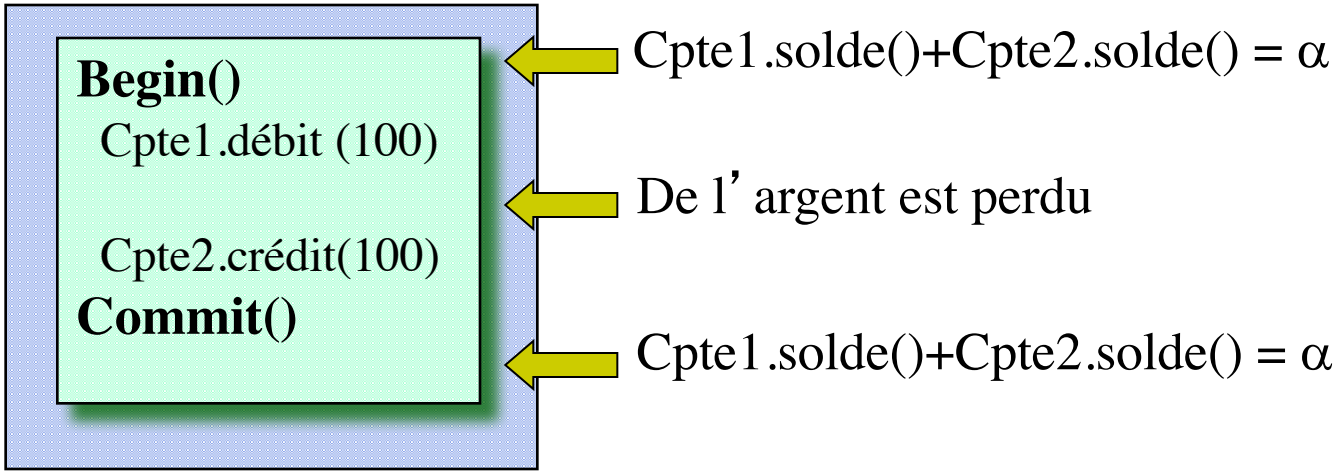
Atomicité



- Notion :
  - Les actions d ’une transaction sont indivisibles
- Besoins :
  - Pouvoir défaire des actions si elles ne sont pas validées
  - Pouvoir refaire des actions perdues après une Validation

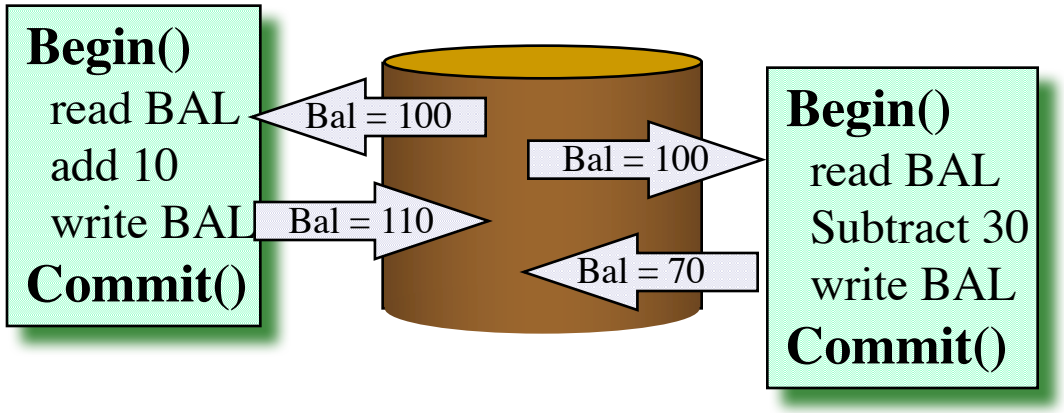
Cohérence

- Notion
  - Les données sont prises dans un état cohérent au début de la transaction, et rendues dans un état cohérent en fin de transaction
  - La cohérence peut être violée au cours de la transaction
- Exemple:



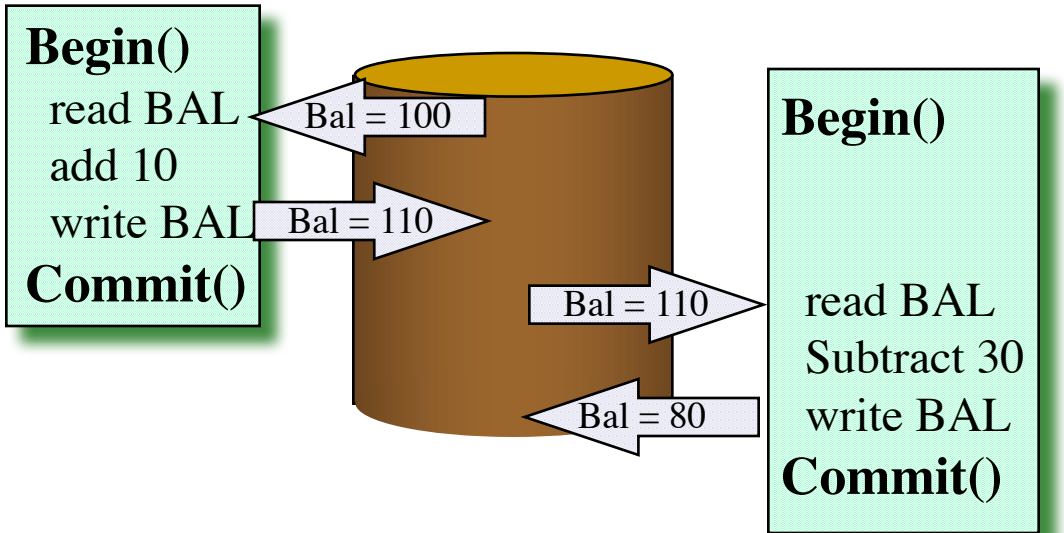
Isolation (1/2)

- Problème
  - Exécuter des programmes de manière concurrente peut amener à lire des données incohérentes
- Exemple

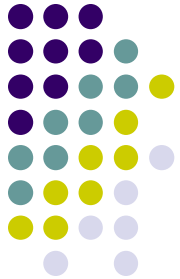


Isolation (2/2)

- Besoins
  - Les modifications d ’une transaction ne doivent pas être visibles avant sa validation
  - Un système transactionnel doit automatiquement protéger les applications (Verrouillage, Versionning, etc…)

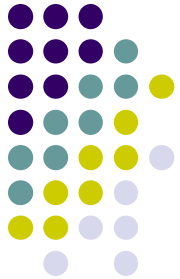


Durabilité

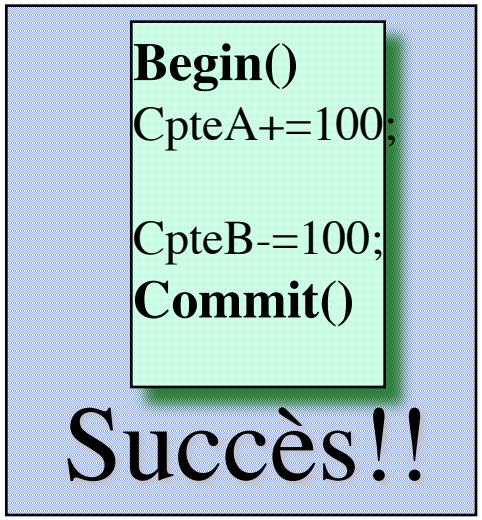


- Notion
  - Les modifications effectuées par une transaction ne doivent pas être perdues
- Besoin
  - Système tolérant aux pannes (notion de sauvegarde)
  - Pouvoir refaire des transactions validées en cas de panne
- Plusieurs niveaux de durabilité
  - Le système tolérant à 100 % n ’existe pas
  - Exemple : impossible de faire des copies des données de carte à microprocesseur

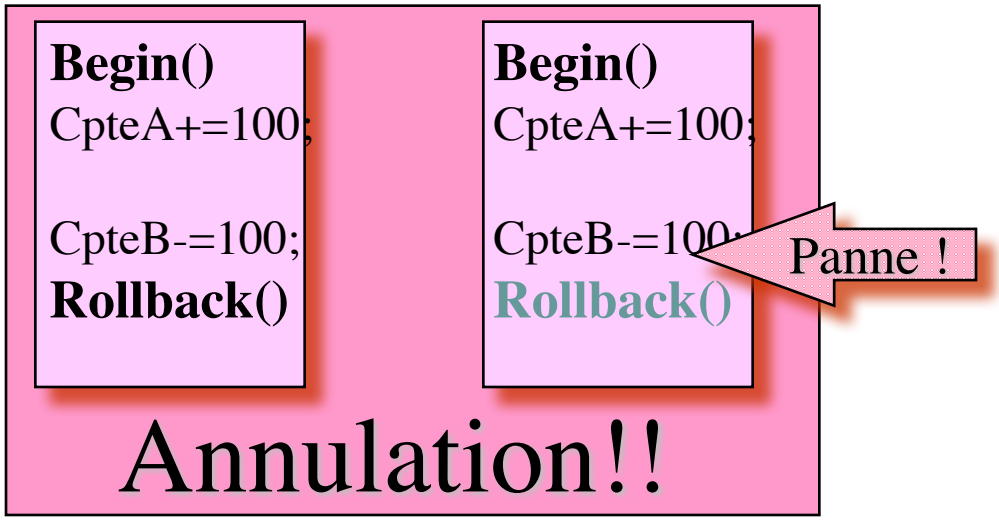
Terminaison d'une Transaction



- Seulement 2 possibilités :

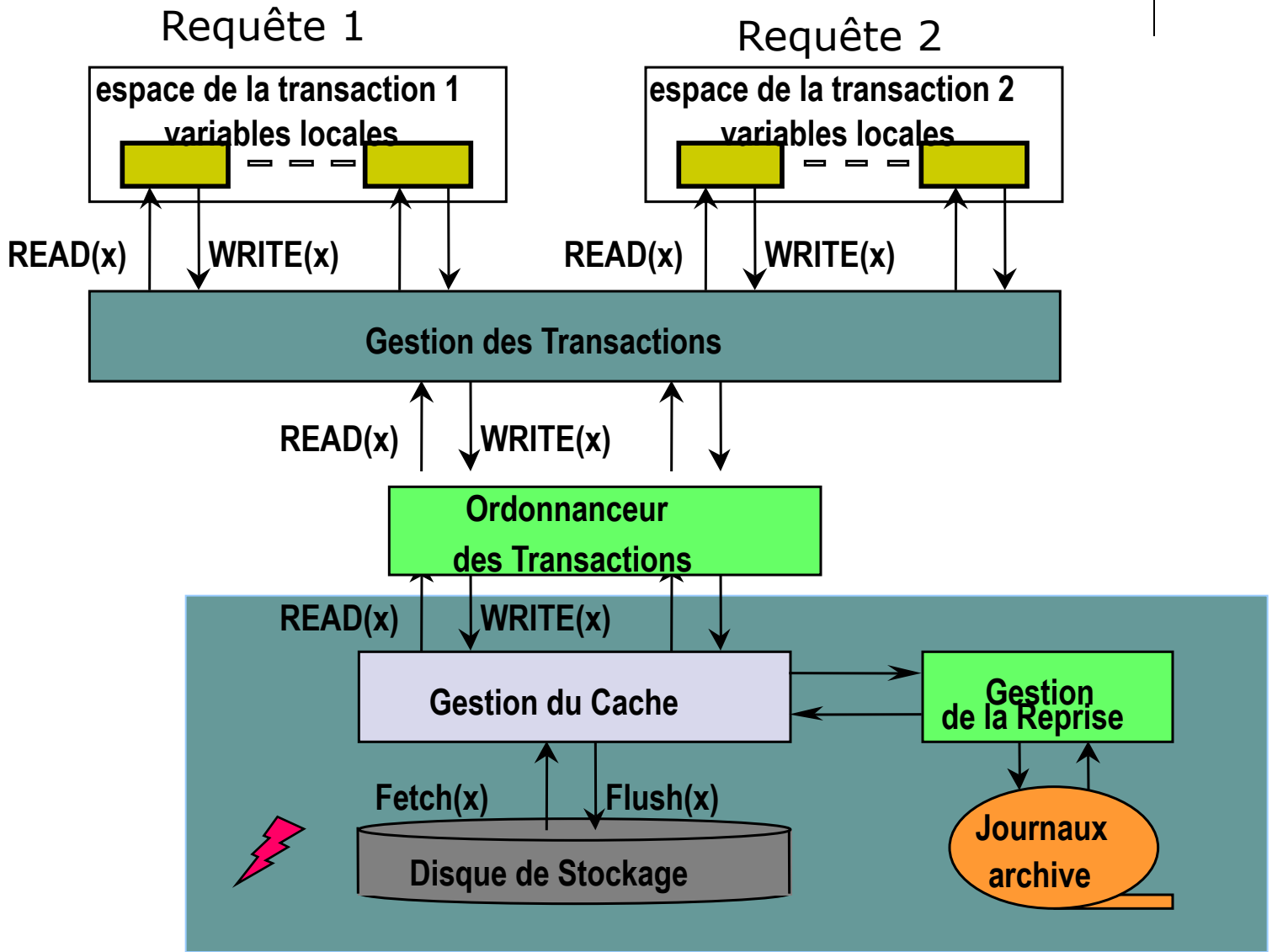


Les modifications sont Validées

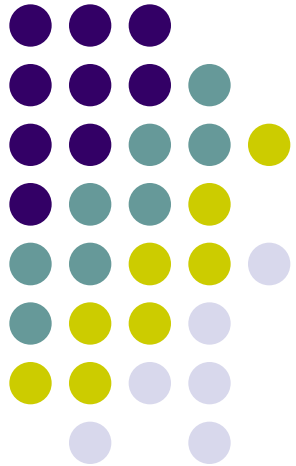


Les modifications sont Abandonnées

Architecture du gestionnaire de données



I. Le contrôle de concurrence

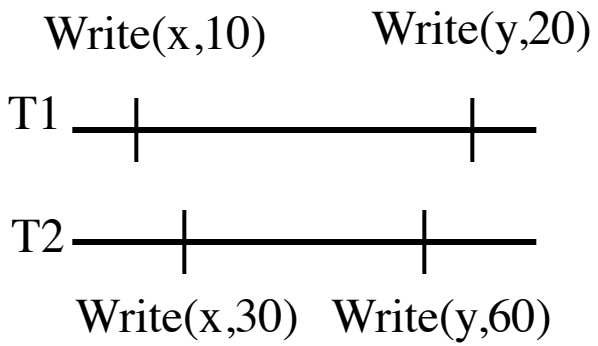


Au sommaire

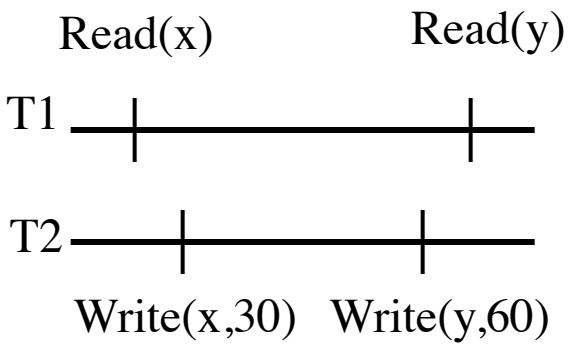
- Gestion de la concurrence d'accès
  - Les situations d'incohérence
  - Principes à respecter
    - Serialisabilité
  - Notion de transaction
    - Définition
    - Propriétés
  - Techniques utilisées
    - Verrouillage à deux phases
    - Estampillage

Notion de cohérence

- Une exécution concurrente non contrôlée de plusieurs transactions crée des incohérences :



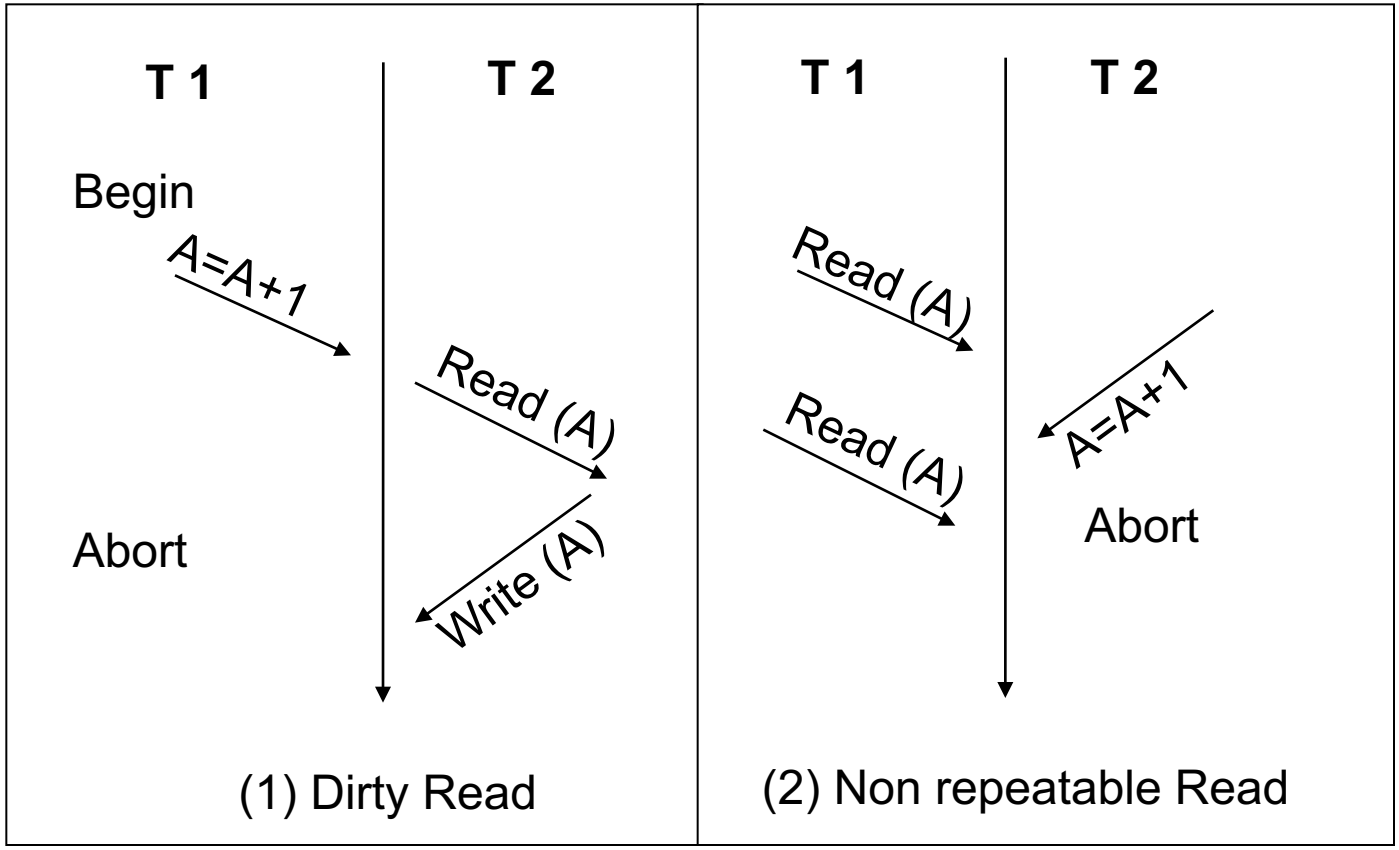
Ecritures incohérentes



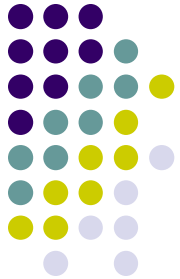
Lectures incohérentes

Autres cas

A=0



Principe de Sérialisabilité

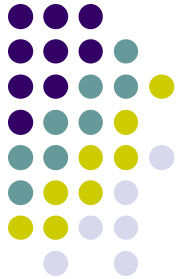


- Sérialisabilité d ’une exécution

  = équivalence à une exécution en série des transactions

  - Exécution en Série :
    - « une exécution est dite en série, si pour tout couple de transactions, tous les événements de l ’une précèdent tous les événements de l ’autre »
  - Equivalence d ’exécution :
    - « 2 exécutions du même ensemble de transactions sont équivalentes ssi
      - elles sont constituées des mêmes événements
      - elles produisent le même état final des objets et les mêmes résultats pour les transactions »
  - Exécution sérialisable :
    - « Une exécution concurrente de transactions validées est sérialisable ssi il existe une exécution en série équivalente »

Principe de Sérialisabilité (2)

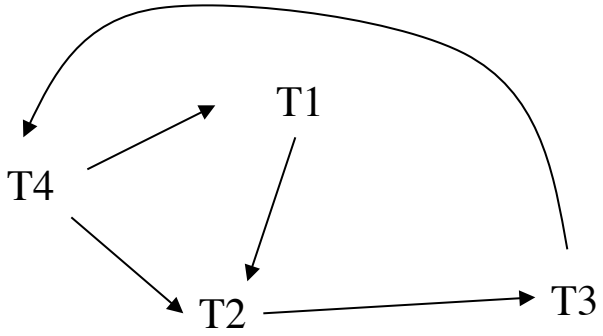


- Exécution sérialisable commutable
  - Exécution commutable = cas particulier de sérialisable
  - *op1* et *op2* commutent si ∀ (Ti, Tk) $op1_i(x)op2_k(x)$ et $op2_k(x)op1_j(x)$
    - ont les mêmes effets sur x (produisent le même résultat)
    - ont les mêmes effets sur Ti et Tk (chaque transaction produit le même résultat)

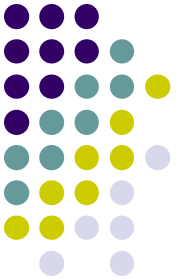
- Exécution sérialisable stricte
  - Exécution stricte = cas particulier de sérialisable
  - Dès qu'une transaction a lu un objet, une autre peut lire le même objet, mais aucune autre ne peut écrire sur cet objet
  - Dès qu'une transaction a écrit un objet, aucune autre ne peut lire ou écrire sur cet objet

Conflit et graphe de dépendance

- Conflit entre Transactions
  - Soient 2 transactions Ti et Tk, Un conflit existe si il existe un objet X accédé par Ti et Tk au moyen des opérations $op1_i$ et $op2_k$ telles que ¬ commute(op1, op2)
- Dépendance entre Transactions
  - Soient 2 transactions Ti et Tk, on a une dépendance de Ti vers Tk si il existe un conflit, et $op1_i < op2_k$
- Graphe de dépendances
  - Permet de déterminer la sérialisabilité d ' une exécution concurrente



Méthodes Pessimistes : le 2PL (1/2)

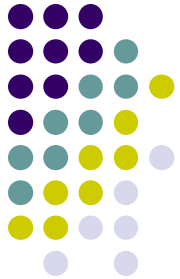


- Principe du Verrouillage à 2 Phases
  - N lecteurs XOR 1 seul écrivain
- A chaque accès à une donnée, un verrou est posé

Compatibilité	Lecture	Ecriture
Lecture	OUI	NON
Ecriture	NON	NON

- Les verrous sont tous relâchés à la fin de la transaction
- Inconvénient : risque de Dead Lock

Méthodes Pessimistes : le 2PL (2/2)

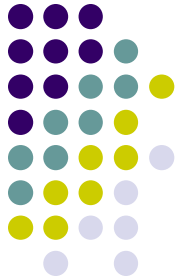


- 2 phases :
  - Une phase d ’acquisition des verrous
  - Une phase de libération
- Exemple :

T1	T2
1.Verrouiller (X, écriture)	2.Verrouiller (X, écriture)
3.Ecrire(X)	Ecrire(X)
4.Verrouiller(Y, écriture)	Verrouiller(Z, écriture)
5.Ecrire(Y)	Ecrire(Z)
6.Déverrouiller(X) Déverrouiller(Y)	Déverrouiller(X) Déverrouiller(Z)

T2 bloquée Jusque 6.

Méthodes Pessimistes : l ’estampillage (1/2)



- L ’Estampillage
  - Principes : utilisation de 2 estampilles L et M

Accès en Lecture à un Objet A par T

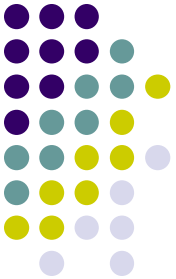
Si V(T) >= M(A) alors
** L(A) = max(V(T),L(A))**
sinon
** Abandon de T**
fin si

Accès en Modification à un Objet A par T

Si V(T) >= max (M(A), L(A)) alors
** M(A) = V(T)**
sinon
** Abandon de T**
fin si

- Avantage :
  - Facile à mettre en œuvre, pas d ’interblocage possible
- Inconvénients :
  - Abandons inutiles
  - Possibilité pour une transaction de redémarrer indéfiniment

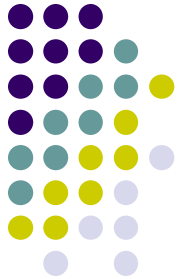
Méthodes Pessimistes : l ’estampillage (2/2)



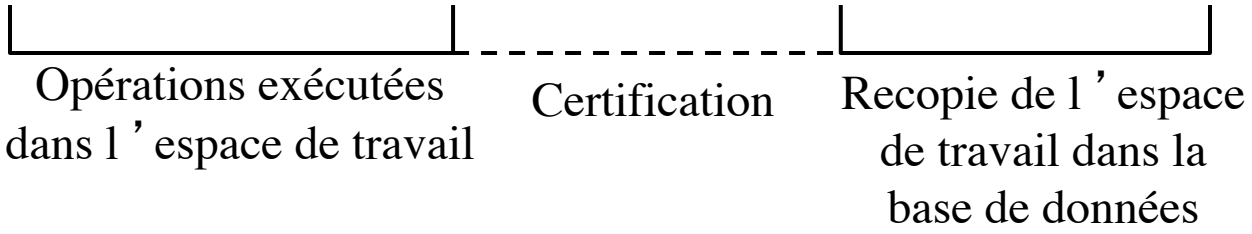
- Exemple :

T1 Estampille = 1	T2 Estampille = 2	Estampille Obj	Observation
Lire(A)		Estampille =1	OK
	Lire(A)	Estampille = 2	OK
	Ecrire(A)	Estampille =2	OK
Ecrire(A)		Estampille =2	Impossible

Méthode Optimiste : La certification (1/2)

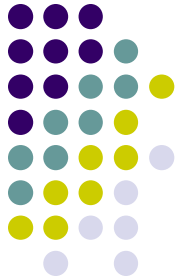


- Une transaction = 3 étapes



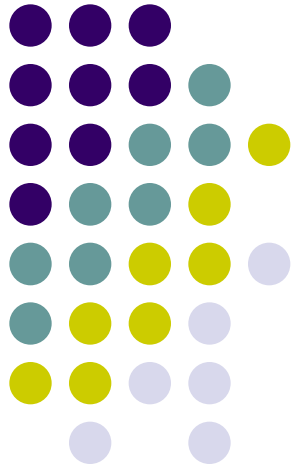
- Les transactions reçoivent une estampille en arrivant à la phase de certification
  - Si conflit, la transaction est relancée
  - Sinon, elle est validée : ces modifications sont recopiées dans la base

Méthode Optimiste : La certification (2/2)

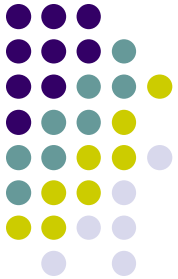


- Efficace si Tx faible de conflits :
  - si les conflits sont importants, il y a fort risque d'abandon
  - Cela consomme inutilement des ressources
- Les transactions ne doivent pas mettre à jour immédiatement la base (c'est à dire « salir » le support avant la validation) car :
  - Une transaction peut utiliser librement les effets d'une autre transaction pas encore validée
  - Cela peut conduire à des abandons en cascade
- Il faut utiliser un mode de mise à jour « différé »
  - espace de travail de la transaction <> espace de travail dans la BD

II. La reprise sur panne

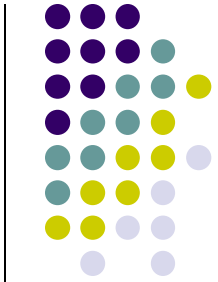


Au sommaire



- Principe
- Notion de Checkpoint
  - Rôle
- Méthodes
  - UNDO
  - REDO
  - UNDO/REDO
- Archivage

Contexte & définition

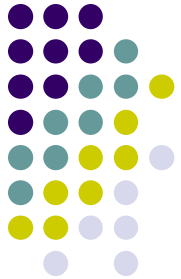


- Définition:
  - Fiabilité d'une BD est l'aptitude du SGBD de préserver la cohérence en cas de la survenue d'une panne ou l'introduction d'erreurs
- Exemples d'erreur
  - Violation des contraintes d'integrité
    - Trigger
- Exemples de pannes
  - Crash disque
  - Défaillance du système (bugs, panne de courant)
  - Catastrophes naturelles

Outils

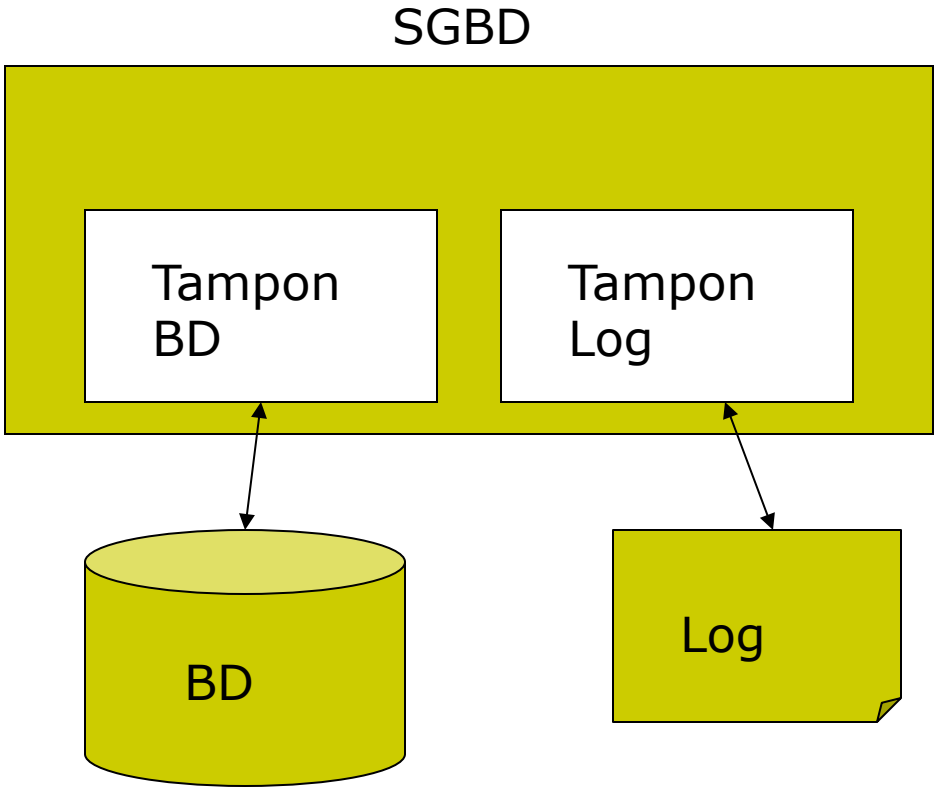
  - Journaux
  - Copie ombre
  - RAID
  - Sauvegarde multi-site

Techniques de reprise sur panne

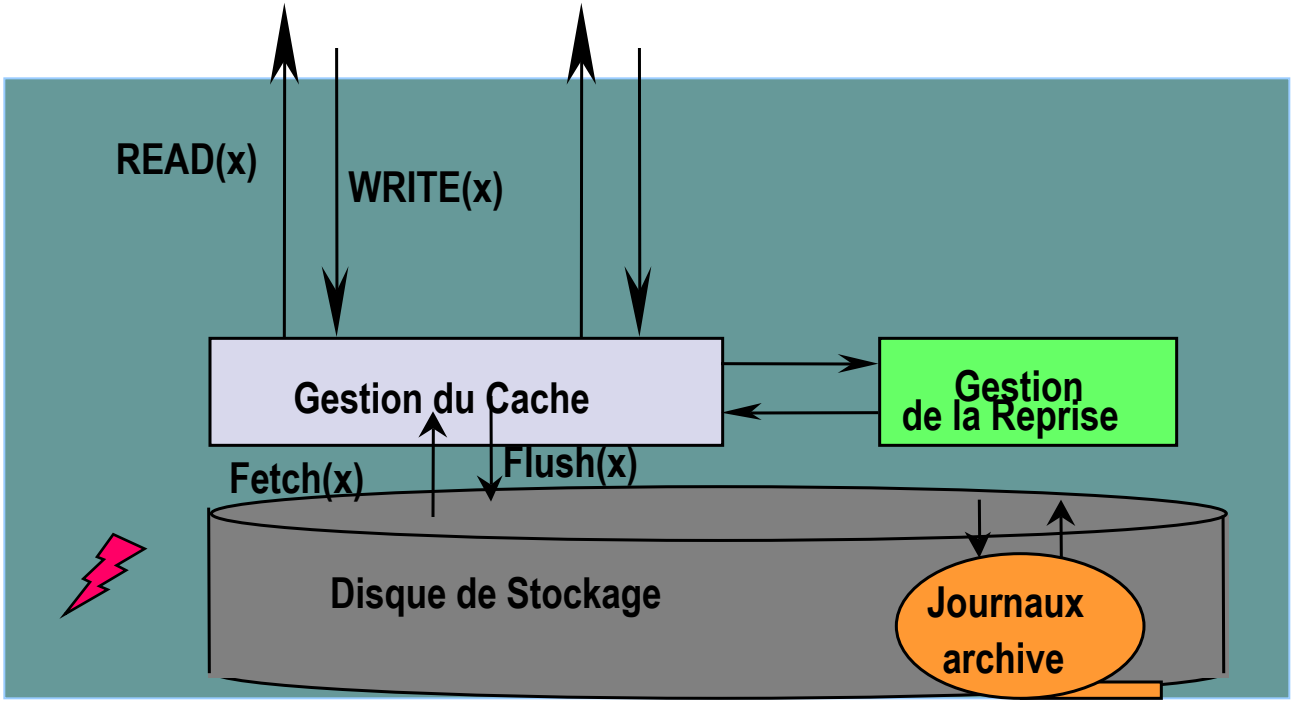


- Recouvrement:
  - à partir d'un journal (Log), reconstituer l'état cohérent de la base.
- Points de contrôle
  - Optimise la technique du recouvement
  - Contrôle régulièrement le contenu du fichier Log
- Archivage
  - utilisation de BD ombre

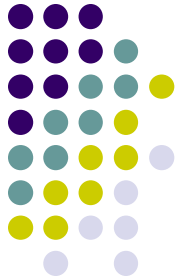
Gestion des Logs



La reprise sur panne & gestion cache



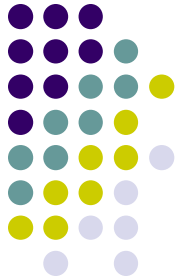
Reprise sur panne & gestion cache



- Le SGBD utilise un cache pour minimiser les accès disque.
- Les données modifiées sont d'abord modifiées dans le cache
- puis stockés sur disque à la validation d'une transaction

- Quelles sont les données valides?
  -

Les opérations élémentaires & utilisation de cache



- Read (x, buf) :
  - si x n'est pas dans le cache faire Input(x)
  - Buf:= x

- Write(x, Buf):
  - Si x n'est pas dans le cache faire Input(x)
  - X := Buf (en mémoire)
- Input (x) :
  - transfert de x du disque en mémoire
- Output(x):
  - Transfert de x de la mémoire vers le disque (mise à jour persistante)

Exemple de transaction

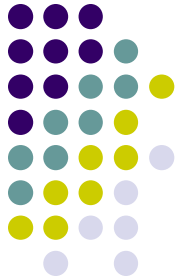
- Exemple de transaction
  - T1:
    - A=A+1;
    - B=B+2;

cache

Op/ mem	Buf	A	B	@A	@B
				10	20
Read(A,Buf)	10	10			
Buf+=1	11	10		10	20
Write(A,Buf)	11	11		10	20
Read(B,Buf)	20	11	20	10	20
Buf+=2	22	11	20	10	20
Write(B,Buf)	22	11	22	10	20
Output A	22	11	22	11	20
Output B	22	11	22	11	22

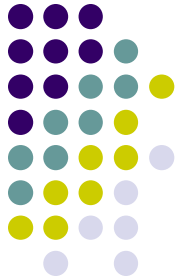
Validation
De T1

Le principe du recouvrement



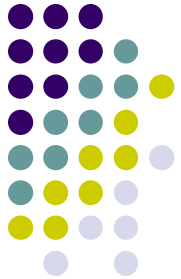
- Le gestionnaire du recouvrement inscrit les événements dans un journal.
- Le journal est inscrit sur une mémoire stable : le disque
- En cas de panne, le journal servira à reconstituer la base de données dans un état cohérent.
- Le buffer du Log: évite le coût d'écriture sur disque à chaque mise à jour
- Respect des transactions validées (durabilité)

Types d'événements journalisés



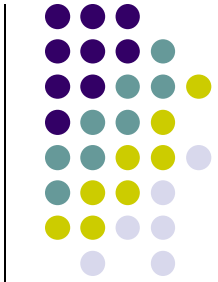
- <START T>:
  - au démarrage d'une nouvelle transaction
- <Commit T>:
  - A la validation de T
- <Abort T>:
  - En cas de rollback
- <T, A, v> :
  - Dans le cadre de la transaction T, l'ancienne (et/ou la nouvelle) valeur de l'attribut A est v

Les points de contrôle



- Problématique
  - Le parcours de l'ensemble des événements depuis la dernière panne est lent et coûteux!!
- Technique
  - Vérifier à intervalle régulier la validité des transactions.
  - Laisser une trace dans le journal à chaque point de contrôle
  - Seuls les événements entre deux points de contrôle doivent être pris en compte
- Conséquence
  - Améliore considérablement le temps de recouvrement.

La méthode de journalisation UNDO



- Principe:
  - Permet d'annuler les effets des transactions non abouties
- fonctionnement:
  - <T,A,v>: A avait la valeur v sur disque avant une mise à jour effectuée par T
  - L'événement <commit T> ne peut être inscrit dans le journal qu'une fois les données modifiées sur disque
  - En conséquence, au commit de T:
    1. Flush() : permet de rendre persistant les écritures dans le journal pour une transaction
    2. Transfert des valeurs modifiées par T sur disque
    3. Ecrire <commit T> dans le journal
    4. Flush()

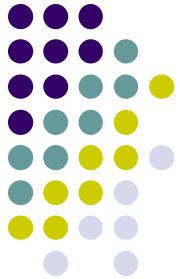
Exemple

Op/mem	Buf	A	B	@A	@B
				10	20
Read(A,Buf)	10	10			
Buf+=1	11	10		10	20
Write(A,Buf)	11	11		10	20
Read(B,Buf)	20	11	20	10	20
Buf+=2	22	11	20	10	20
Write(B,Buf)	22	11	22	10	20
Flush()					
Output A	22	11	22	11	20
Output B	22	11	22	11	22
Flush()					

Journal

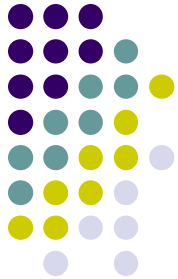
cache	Sur disque
<Start T>	<START T>
<T,A,10>	<T,A,10>
<T,B,20>	<T,B,20>
<Commit T>	<Commit T>

Procédure de reprise



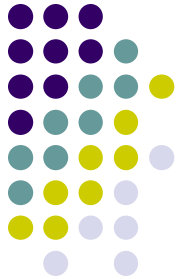
- Pour chaque transaction T démarrée avant panne:
  - Si commit T sur disque
    - Rien
  - Sinon
    - Pour chaque inscription <T,A,v>, restaurer la valeur v dans A sur disque
    - Inscrire <Abort T> sur journal
    - Flush()

Technique des points de contrôle pour la methode UNDO



- Contrôle effectué à intervalles réguliers
- But :
  - Simplifier le recouvrement
- Une nouvelle inscription dans le Log:
  - <CKPT>
- Effet:
  - Recenser les transactions actives.
- Mise en œuvre
  - Soit T_A cet ensemble
  - Jusqu'à la fin du point de contrôle, différer toute nouvelle transaction.
  - Attendre que toute transaction de T_A soit fini et l'evenement abort ou commit inscrit sur journal
  - Effets sur le Log
    - flush()
    - Ecrire un <CKPT>
    - Flush()

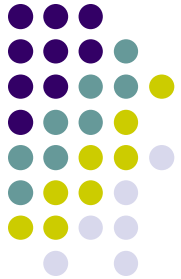
Procédure de reprise améliorée



- La reprise commence à partir du dernier point de contrôle
  - Lire le journal à partir de la fin jusqu'à <CKPT>.
  - Seules les transactions ayant démarré apres le CKPT sont à défaire

- Inconvénient du ckpt:
  - Blocage des transactions nouvelles
- Inconvénient de la journalisation UNDO
  - Oblige à réécrire systématiquement sur disque
  - N'optimise pas les E/S

La méthode de journalisation REDO



- Principes:
  - La reprise consiste à refaire les transactions validées à partir du journal et ignorer les autres
  - <T,A,v>: signifie que la nouvelle valeur de A modifiée dans le cadre de T est v
  - L'événement <commit T> sera inscrit dans le journal avant toute modification sur disque
  - En conséquence:
    - Un flush() sur disque du journal avant toute écriture sur disque

Exemple

Journal

Op/mem	Buf	A	B	@A	@B
				10	20
Read(A,Buf)	10	10			
Buf+=1	11	10		10	20
Write(A,Buf)	11	11		10	20
Read(B,Buf)	20	11	20	10	20
Buf+=2	22	11	20	10	20
Write(B,Buf)	22	11	22	10	20
Flush()					
Output A	22	11	22	11	20
Output B	22	11	22	11	22
Flush()					

cache Sur disque

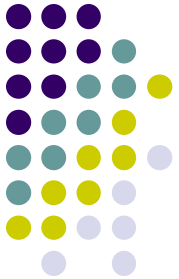
<Start T><Start T>

<T,A,11> <T,A,11>

<T,B,22> <T,B,22>

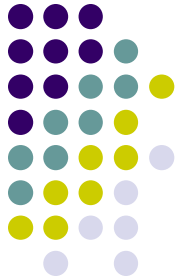
<Commit T><Commit T

Procédure de reprise



- Lecture du journal par le début
- Toute transaction ayant validé est refaite à partir du journal
- Toute transaction non validé:
  - Inscrire un <Abort T>
- Flush() : mise à jour du journal

Procédure de reprise avec Check Point

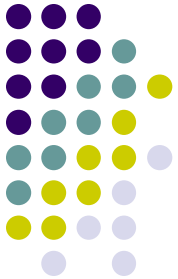


- L'idée:
  - Le CKPT doit permettre de forcer l'écriture sur disque des transactions validées
- Caractéristique:
  - Le START CKPT n'est pas bloquant pour les nouvelles transactions
- Fonctionnement:

  Début checkpoint
  - Rechercher l'ensemble des transactions actives à l'instant t. Soit TA1,TA2,…,TAn cet ensemble
  - Inscrire **<START CKPT (TA1,TA2,…,TAn)>**
  - Flush buffer Log
  - Rechercher les transactions ayant validé depuis le dernier CKPT. Soit TC=TC1,TC2,…TCm
  - Pour chaque TCi
    - Rechercher les modifications effectuées (remonter jusqu'au premier CKPT où TCi non encore active)
    - Mise à jour des variables modifiées à partir du cache
  - Inscrire **<END CKPT>** dans le fichier Log

  End checkpoint

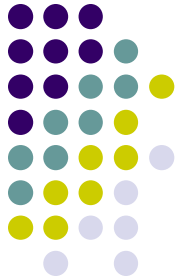
Check Point pour la méthode REDO: Traitement de la panne



- Deux cas:
  - La panne après END CKPTi:
    - Transactions à refaire:
      - Toute transaction T active au <START>
        - Si commit alors REDO
        - Sinon abandonner les mises à jour
      - Les transactions ayant validé avant le début du checkpoint sont déjà prises en compte
  - La panne après START $CKPT_i$:
    - La mise à jour sur disque pour les transactions validées avant le START $CKPT_i$ est incertaine.

    Début
    - Retrouver le START-END $CKPT_{i-1}$
    - Soit $TA_{i-1,1}$,$TA_{i-1,2}$,…, $TA_{i-1,m}$ l'ensemble des transactions ayant démarré après le START CKPTi-1 ou actives (non validées) au START CKPTi-1
      - Refaire toute transaction t ayant validé (commit dans le Log)
      - Abandonner les transactions n'ayant pas validé

Exemple de reprise REDO avec Check point



- START T1
- START T2
- COMMIT T2
- START T3
- **START CKPT (T1,T3)**
- COMMIT T3
- **END CKPT**
- START T4
- COMMIT T4
- **START CKPT(T1)**
- START T5
- COMMIT T1

Cas 1

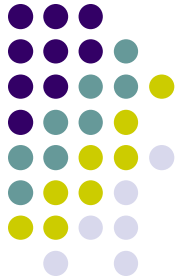
- START T6
- COMMIT T6
- **END CKPT**

Cas 2

- Cas 1: Panne après commit T1
  - (T2 est validé)
  - Refaire T3,T4,T1,T6
  - Abandonner T5

- Cas 2: panne après END CKPT
  - (T4 et T3 validé)
  - Refaire T6,T1
  - Abandonner T5

Jounalisation UNDO/REDO



- Pallie les défauts des méthodes précédentes:
  - UNDO: I/O inutiles
  - REDO: Monopolisation des buffers

Un nouveau type d'enreg: <T,A,Va, Vn>: mise à jour de la variable A par la transaction T, Ancienne valeur Va, nouvelle valeur Vn

Permet une indépendance entre l'inscription du commit dans le Log et entre les transferts disque

Exemple

Journal

Op/mem	Buf	A	B	@A	@B
				10	20
Read(A,Buf)	10	10			
Buf+=1	11	10		10	20
Write(A,Buf)	11	11		10	20
Read(B,Buf)	20	11	20	10	20
Buf+=2	22	11	20	10	20
Write(,Buf)	22	11	22	10	20
Output A	22	11	22	11	20
Output B	22	11	22	11	22

cache Sur disque

<Start T>

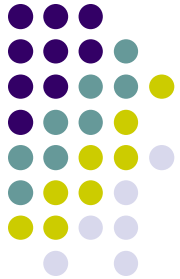
<T,A,10,11>

<T,B,20, 22>

<Commit T>

<Commit T>

Procédure de reprise UNDO/REDO



- Commencer la reprise à partir du début du fichier Log
- Pour chaque commit rencontré,
  - Refaire les mises à jour à partir du log (même redondantes)

Exemple de reprise

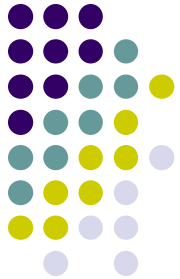
- START T1
- START T2
- **COMMIT T2**
- START T3
- **…**
- **COMMIT T3**
- **…**
- START T4
- **COMMIT T4**
- **…**
- START T5
- **COMMIT T1**

- START T6
- **COMMIT T6**
- **…**

Sens de la reprise

- Refaire:
  - T2,T3,T4,T1 et T6

Reprise UNDO/REDO avec points de contrôle

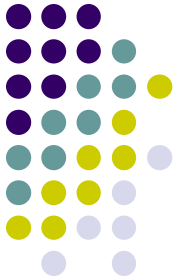


- L'idée: le point de contrôle permet d'anticiper sur les écritures pour libérer les buffers
- Description

  Début checkpoint
  - Rechercher l'ensemble des transactions actives à l'instant t. Soit TA1,TA2,…,TAn cet ensemble
  - Inscrire **<START CKPT (TA1,TA2,…,TAn)>**
  - Flush buffer Log
  - Vider les tampons
    - Ecrire sur disque toutes les mises à jour effectuées par
      - Les transactions actives
      - Les transactions validés entre deux checkpoints
  - Inscrire **<END CKPT>** dans le fichier Log
  - Flush Buffer Log

  End checkpoint

Fonctionnement des points de contrôle UNDO/REDO



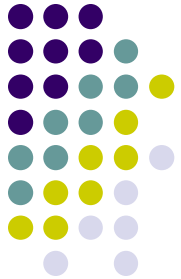
- START T1
- <T1, A, 10, 20>
- START T2
- <T2, B, 1, 2>
- COMMIT T2
- START T3
- <T3, B, 2, 5>
- **START CKPT (T1,T3)**
- <T3, C, 1, 5>
- COMMIT T3
- **END CKPT**
- START T4
- <T4, B, 5, 6>
- COMMIT T4
- **START CKPT(T1)**
- START T5
- <T5,B,6,10>
- COMMIT T1

- START T6
- <T6,A, 20, 30>
- **END CKPT**

- COMMIT T6

- Pour le premier CKPT
  - A = 20
  - B = 5
- CKPT 2
  - C = 5
  - B = 6

Checkpoint UNDO/REDO Cas de panne



- Description
  - Rechercher le dernier END CKPT
  - Toutes les transactions
    - ayant validé avant le START CKPT : rien
    - actives au CKPT,
      - si validé avant la panne :
        - completer la mise à jour sinon défaire les premières mises à jours
    - ayant démarré pendant ou apres le CKPT
      - si validé avant la panne :
        - Refaire
      - Sinon défaire
  -

Exemple

- START T1
- <T1, A, 10, 20>
- START T2
- <T2, B, 1, 2>
- COMMIT T2
- START T3
- <T3, B, 2, 5>
- **START CKPT (T1,T3)**
- <T3, C, 1, 5>
- COMMIT T3
- **END CKPT**
- START T4
- <T4, B, 5, 6>
- COMMIT T4
- **START CKPT(T1)**
- START T5
- <T5,B,6,10>

Cas 2

- COMMIT T1

Cas 1

- START T6
- <T6,A, 20, 30>
- **END CKPT**

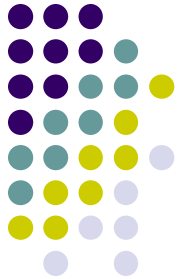
Cas 3

- COMMIT T6

Cas 4

- Cas 1: Le dernier CKPT est le CKPT1
  - C =5;B = 6;A = 20
- Cas 2: Le dernier CKPT est le CKPT1
  - T1 est annulé donc A = 10
- Cas 3: dernier CKPT = CKPT2
  - Rien (Les maj de T1 sont déjà sur disque)
- Cas 4: dernier CKPT = CKPT2
  - A = 30

En conclusion

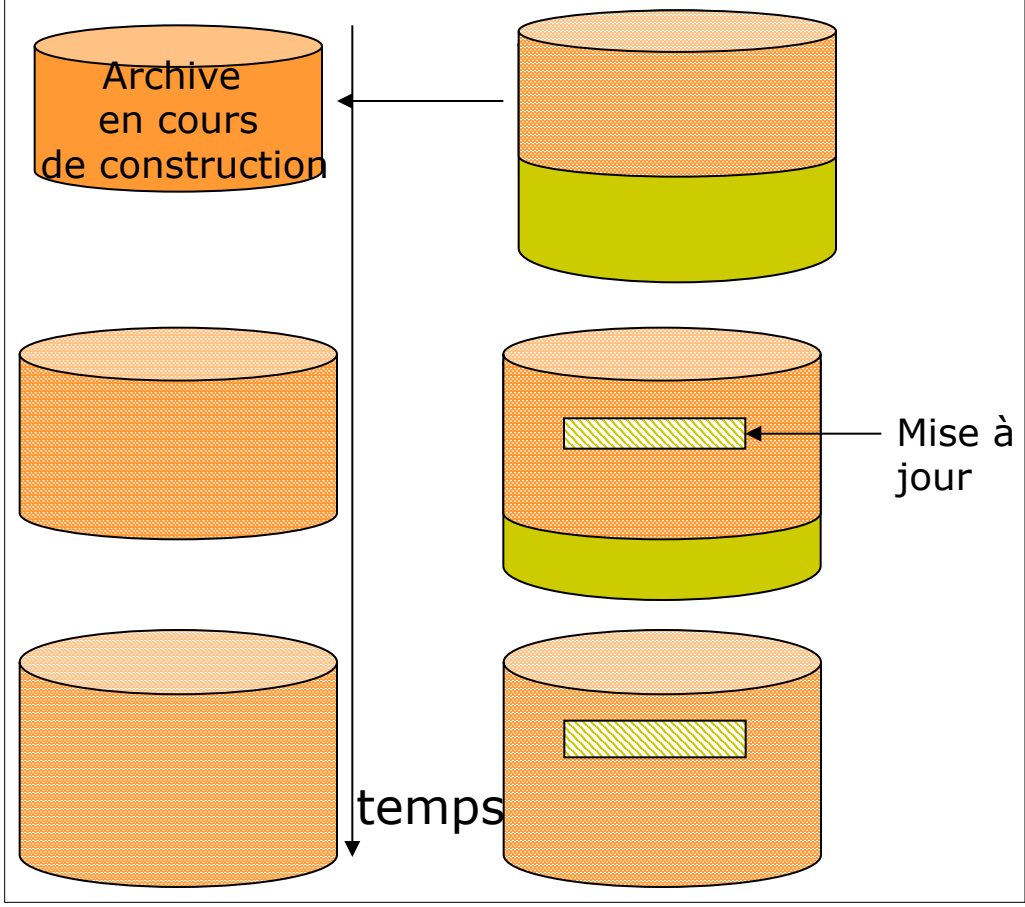


- La technique des Logs permet de retrouver un état cohérent en cas de panne mémoire
- Pas efficace en cas de crash disque
- Plusieurs techniques:
  - Archivage
  - Utilisation des disques tolérants aux fautes
  - Répartition

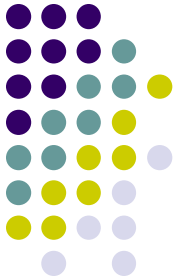
Archivage des bases de données

- 2 types d'archives
  - archive intégrale
  - archive incrémentale
- Problème:
  - La base de donnée doit rester accessible pendant un archivage
- Restauration
  - Utilisation des logs

- Illustration



RAID Redundant Array of Inexpensive Disks [Patterson 88]



RAID : Redundant Array of Inexpensive Disks

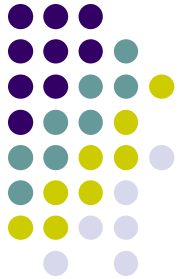
- tableau de disques peu coûteux pour améliorer les débits I/O
- cependant il faut introduire de la redondance

Niveaux de fonctionnement

- Redondance
  - Niveau 1 : Les Disques Miroirs
    - TANDEM Mirrored Disks
    - Conséquence: amélioration des lectures, espace de stockage :50%
    - Variante: mirroring assuré par le SGBD/SGF
  - Niveau 2 : une amélioration de RAID1 : seul un disque fonctionne en cas de lecture
  - Niveau 3 : Un Seul Disque de Contrôle par Groupe de Disque
  - Niveau 4 : Lectures et Ecritures indépendantes
  - Niveau 5 : Contrôle réparti sur les disques du Groupe

- Sans redondance
  - Niveau 0 : Stripping
    - répartition des blocs contigus d'un fichier entre les disques mais pas de redondance (ie AID)

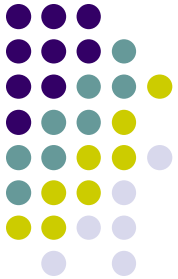
Niveaux de fonctionnement des RAID



- Redondance
  - Niveau 1 : Les Disques Miroirs
    - TANDEM Mirrored Disks
    - Conséquence: amélioration des lectures, espace de stockage :50%
    - Variante: mirroring assuré par le SGBD/SGF
  - Niveau 2 : une amélioration de RAID1 : seul un disque fonctionne en cas de lecture
  - Niveau 3 : Un Seul Disque de Contrôle par Groupe de Disque
  - Niveau 4 : Lectures et Ecritures indépendantes
  - Niveau 5 : Contrôle réparti sur les disques du Groupe

- Sans redondance
  - Niveau 0 : Stripping
    - répartition des blocs contigus d'un fichier entre les disques mais pas de redondance (ie AID)

Redondance multisites



- Préserve les données de pannes dues à des catastrophes naturelles
- Consiste à gérer plusieurs copies sur des sites différents.
- Variante:
  - Gestion de RAID sur plusieurs sites
  - Avec ou sans redondance