Module BDR
Master d'Informatique

Cours 7- Reprise sur pannes

Stéphane Gançarski
Stephane.Gancarski@lip6.fr

Gestion de transactions

Définition

Exemples

Propriétés des transactions

Fiabilité et tolérance aux pannes

Journaux

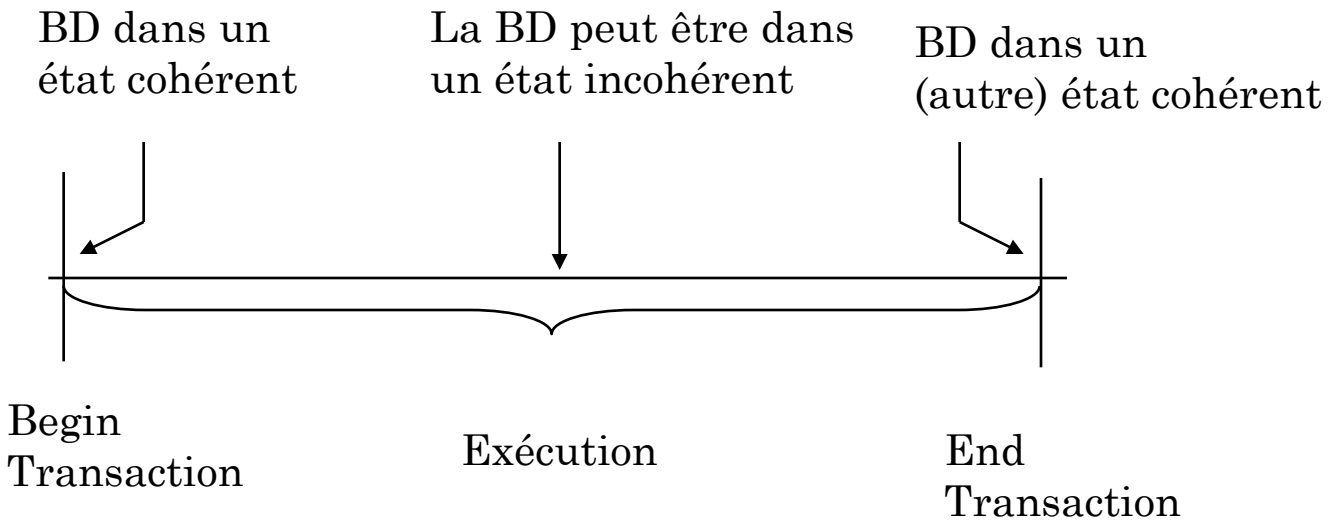
Protocoles de journalisation

Points de reprise

Transaction

Ensemble d'actions qui réalisent des transformations cohérentes de la BD

- opérations de lecture ou d'écriture de données, appelées *granules* (tuples, pages, etc.)



Syntaxe

Une transaction est délimitée par Begin_transaction et End_transaction et comporte :

- des opérations de lecture ou d'écriture de la BD
- des opérations de manipulation (calculs, tests, etc.)
- des opérations transactionnelles: commit, abort, etc.

On ne s'intéresse pas aux opérations de manipulation (logique interne de la transaction) car leur analyse serait trop coûteuse et pas toujours possible (ex. client jdbc, communique uniquement les opérations de lecture/écriture et transactionnelles)

Exemple de transaction simple

```
Begin_transaction Budget-update
begin
  EXEC SQL UPDATE Project
      SET Budget = Budget * 1.1
      WHERE Pname = `CAD/CAM’;
end
```

BD exemple

Considérons un système de réservation d'une compagnie aérienne avec les relations:

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)

CUST(CNAME, ADDR, BAL)

FC(FNO, DATE, CNAME)

Exemple de transaction de réservation

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL  UPDATE     FLIGHT
              SET        STSOLD = STSOLD + 1
              WHERE      FNO = flight_no AND DATE = date; /*
         1 place vendue
    EXEC SQL  INSERT
              INTO       FC(FNO, DATE, CNAME);
              VALUES     (flight_no, date, customer_name,);
         /* 1 résa en plus
    output("reservation completed")
end . {Reservation}
```

Problème : s'il n'y a plus de place dans l'avion ?

- **Surbooking ?**
- **Contrainte d'intégrité (STSOLD <= CAP) ? Message d'erreur…**

Terminaison de transaction

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL    SELECT      STSOLD,CAP
                INTO        temp1,temp2
                FROM        FLIGHT
                WHERE       FNO = flight_no AND DATE =  date;
    if temp1 = temp2 then
        output("no free seats");
        Abort
    else
        EXEC SQL   UPDATE  FLIGHT
                   SET     STSOLD = STSOLD + 1
                   WHERE   FNO = flight_no AND DATE = date;
        EXEC SQL   INSERT
                   INTO    FC(FNO, DATE, CNAME, SPECIAL);
                   VALUES  (flight_no, date, customer_name,
          null);
      Commit
      output("reservation completed")
  endif
end . {Reservation}
```

Propriétés des transactions

ATOMICITE

- tout (commit) ou rien (abort)

COHERENCE

- pas de violation de contrainte d'intégrité
- Cohérence mutuelle des répliques (réplication synchrone)

ISOLATION

- les mises-à-jour concurrentes sont invisibles

DURABILITE

- les mises-à-jour validées persistent

Les transactions délimitent

Les responsabilités respectives du programmeur et du système.

Programmeur :

- Écrire des transactions implantant la logique de l'appli

Système doit garantir :

- l'exécution *atomique* et *fiable* en présence de pannes
- l'exécution *correcte* en présence d'utilisateurs concurrents

Fiabilité

Problème:

Comment maintenir

atomicité

durabilité

des transactions

Types de pannes

Panne de transaction

- abandon (normal –if- ou dû à un interblocage)
- en moyenne 3% des transactions abandonnent anormalement

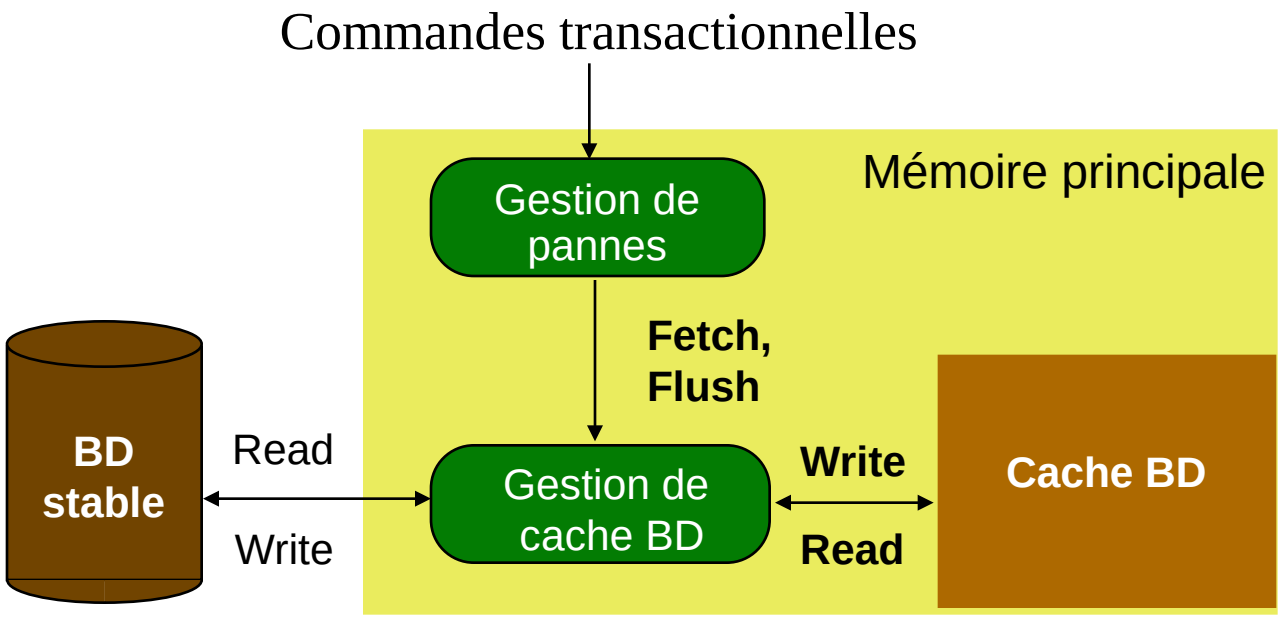
Panne système

- panne de processeur, mémoire, alimentation, …
- le contenu de la mémoire principale est perdu mais disk ok

Panne disque

- panne de tête de lecture ou du contrôleur disque
- les données de la BD sur disque sont perdues

Architecture pour la gestion de pannes



Stratégies de mise-à-jour

Mise-à-jour en place

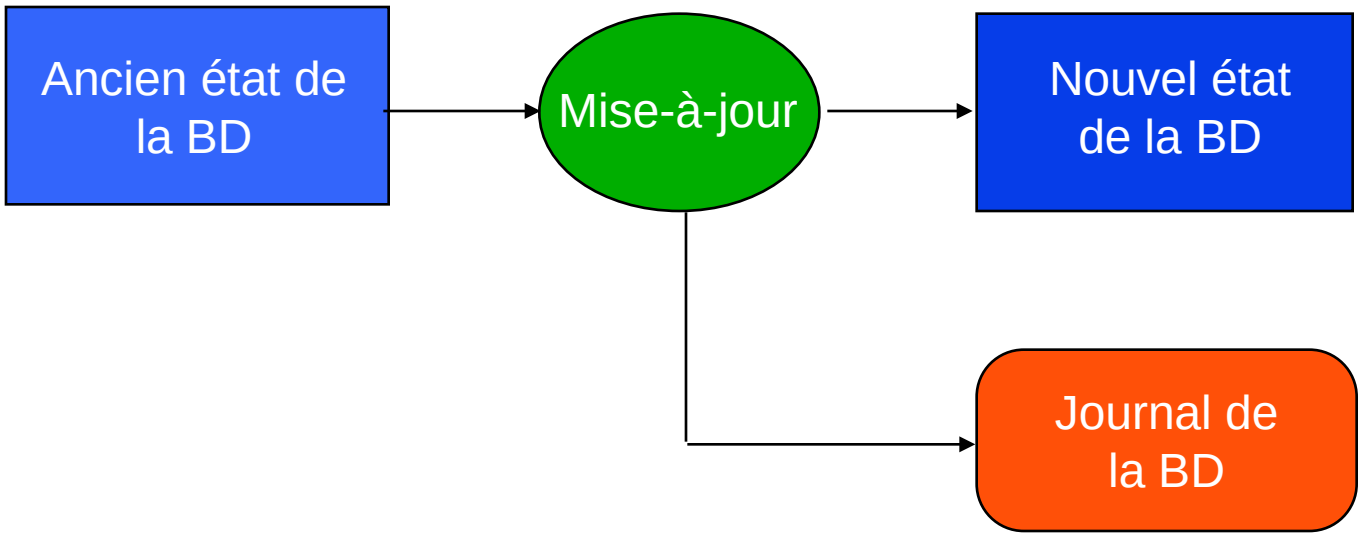
- chaque mise-à-jour cause la modification de données dans des pages dans le cache BD
- l'ancienne valeur est écrasée par la nouvelle

Mise-à-jour hors-place

- les nouvelles valeurs de données sont écrites séparément des anciennes dans des pages ombres
- peu utilisé en pratique car très cher
- mises-à-jour des index compliquée

Journal de la BD

Chaque action d'une transaction doit réaliser l'action, ainsi qu'écrire un enregistrement dans le journal (fichier en ajout seulement avec purge de temps à autre)



Journalisation

Le journal contient les informations nécessaires à la restauration d'un état cohérent de la BD

- identifiant de transaction
- type d'opération (action)
- granules accédés par la transaction pour réaliser l'action
- ancienne valeur de granule (image avant)
- nouvelle valeur de granule (image après)
- …

Structure du journal

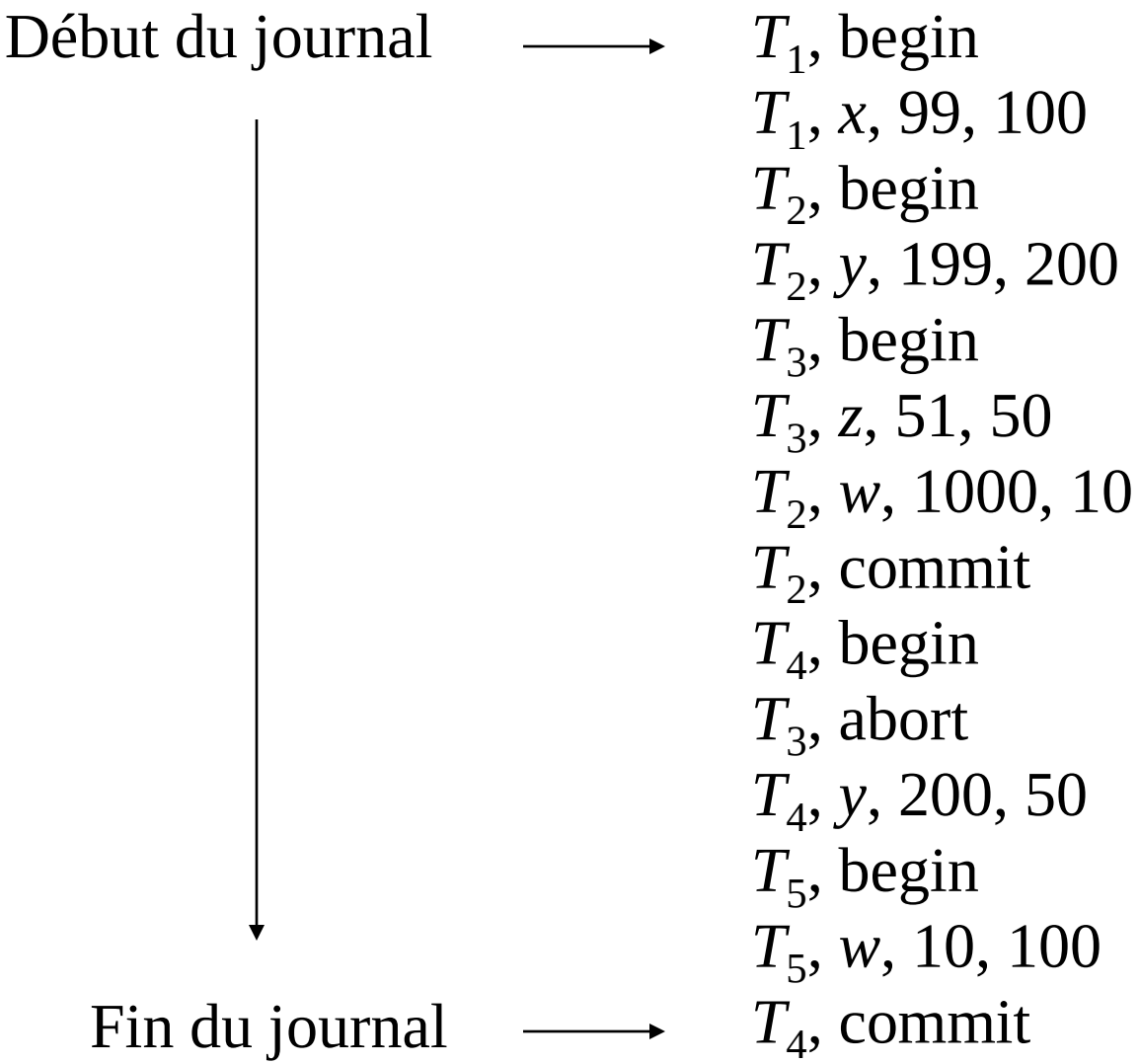
Structure d'un enregistrement :

- N° transaction (Trid)
- Type enregistrement {début, update, insert, commit, abort}
- TupleId (rowid sous Oracle)
- [Attribut modifié, Ancienne valeur, Nouvelle valeur] ...

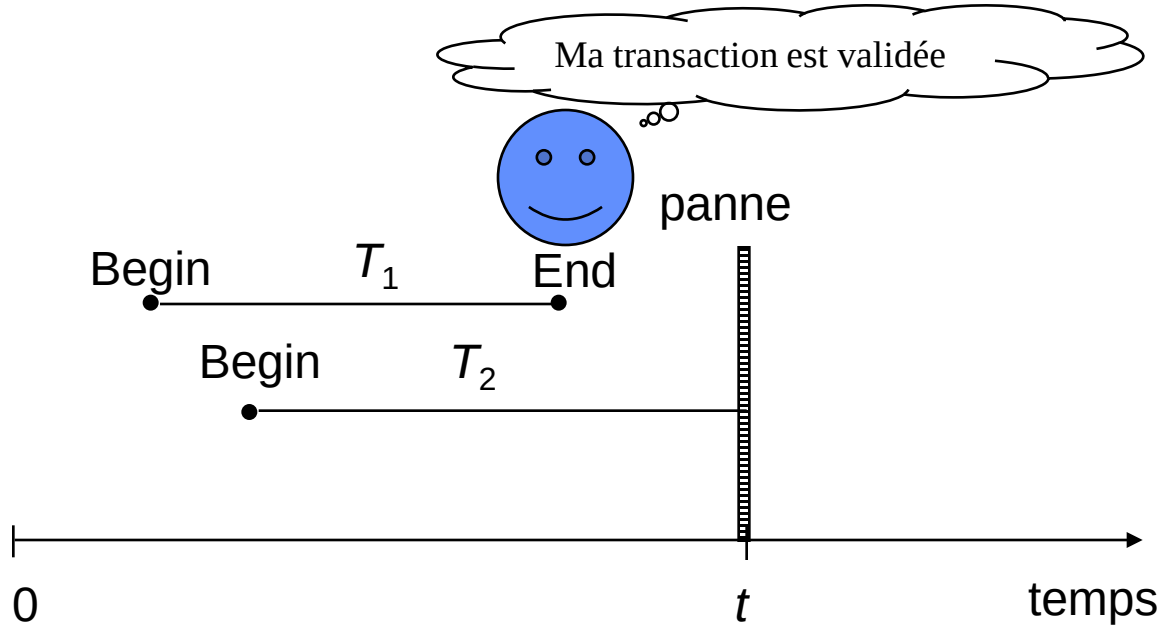
Problème de taille

- on tourne sur N fichiers de taille fixe
- possibilité d'utiliser un fichier haché sur Trid/Tid

Exemple de journal



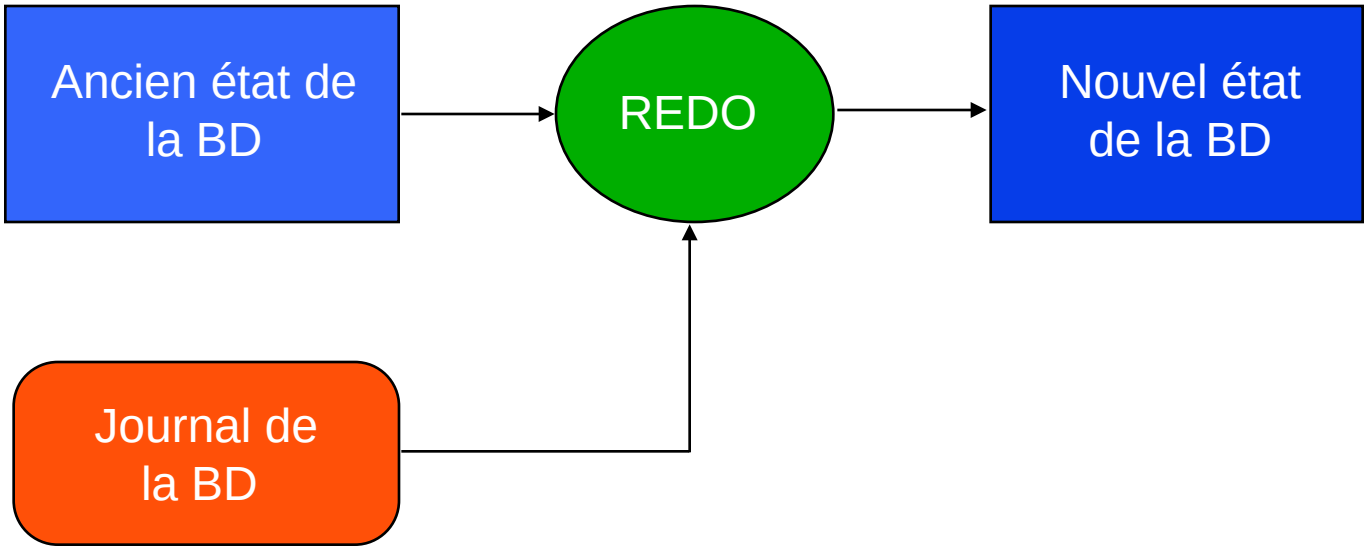
Pourquoi journaliser?



Lors de la reprise

- toutes les mises-à-jour de T_1 doivent être faites dans la BD (REDO)
- aucune mise-à-jour de T_2 ne doit être faite dans la BD (UNDO)

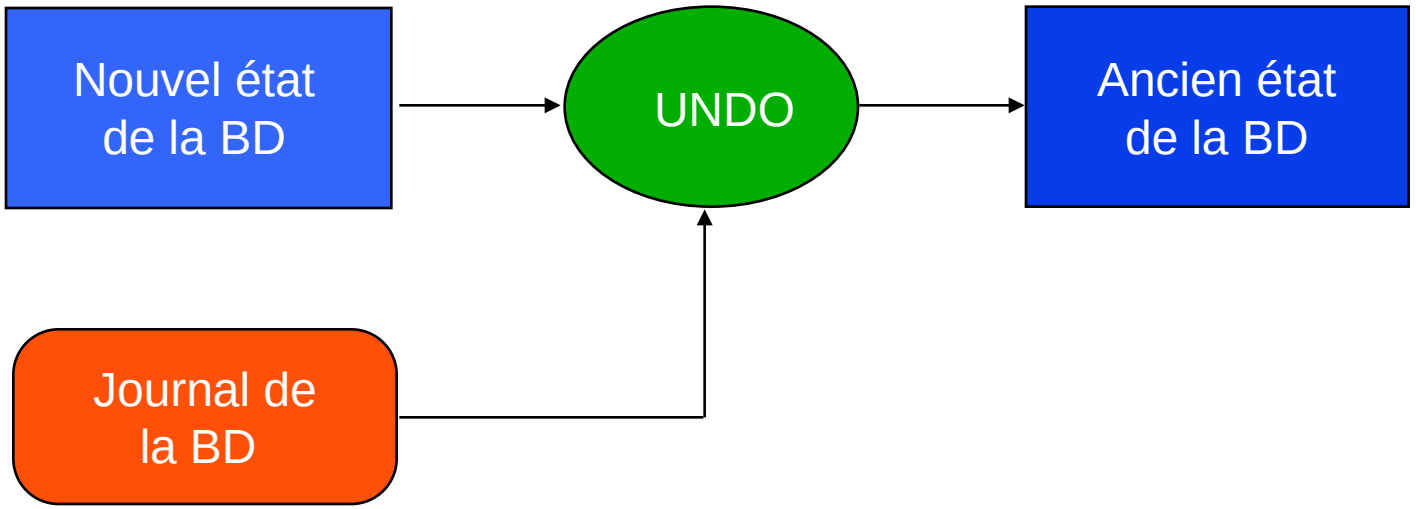
Protocole REDO



L'opération REDO utilise l'information du journal pour refaire les actions qui ont été exécutées ou interrompues

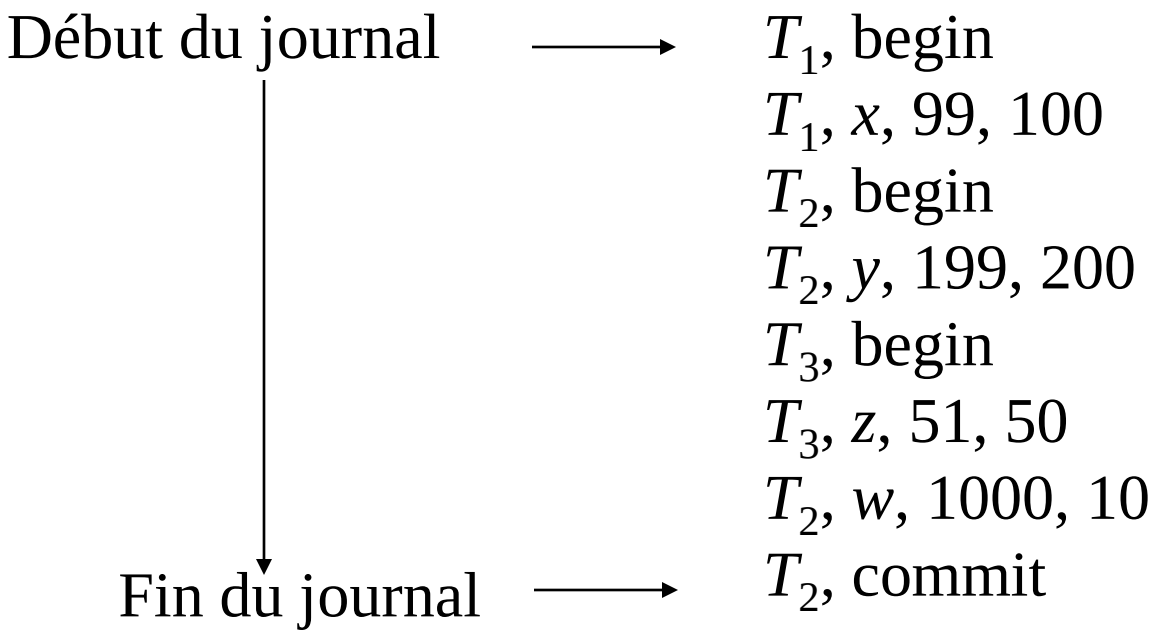
Elle génère la nouvelle image

Protocole UNDO



L'opération UNDO utilise l'information du journal pour restaurer l'image avant du granule. Faite en principe avant le REDO

UNDO: parcours vers l'arrière, REDO : parcours vers l'avant



UNDO : T_2 rien (marquée pour Redo), z:=51, x:=99
REDO : y:=200, w:=10

Abandons en cascade, recouvrabilité (1/2)

Soient deux transactions, T0 et T1, exécutant l'une après l'autre les instructions suivantes :

1. *variable1 := Lire (A);*
2. *variable1 := variable1 – 2 ;*
3. *Ecrire (A, variable1);*
4. *variable2 := Lire (B);*
5. *variable2 := variable2 / variable1 ;*
6. *Ecrire (B, variable2);*

Le système, sur lequel elles s'exécutent, tient à jour un journal susceptible de contenir les enregistrements suivants:

<*No de Transaction*, **start** | **commit** | **abort** >

<*No de Transaction*, *identification de granule*, *ancienne valeur*, *nouvelle valeur*>

Les valeurs initiales de A et B étant respectivement 4 et 14, quel est le contenu du journal lorsque la seconde transaction (T1) se termine ?

Abandons en cascade, recouvrabilité (2/2)

Soient deux transactions, T0 et T1, exécutant l'une après l'autre les instructions suivantes :

1. *variable1 := Lire (A);*
2. *variable1 := variable1 – 2 ;*
3. *Ecrire (A, variable1);*
4. *variable2 := Lire (B);*
5. *variable2 := variable2 / variable1 ;*
6. *Ecrire (B, variable2);*

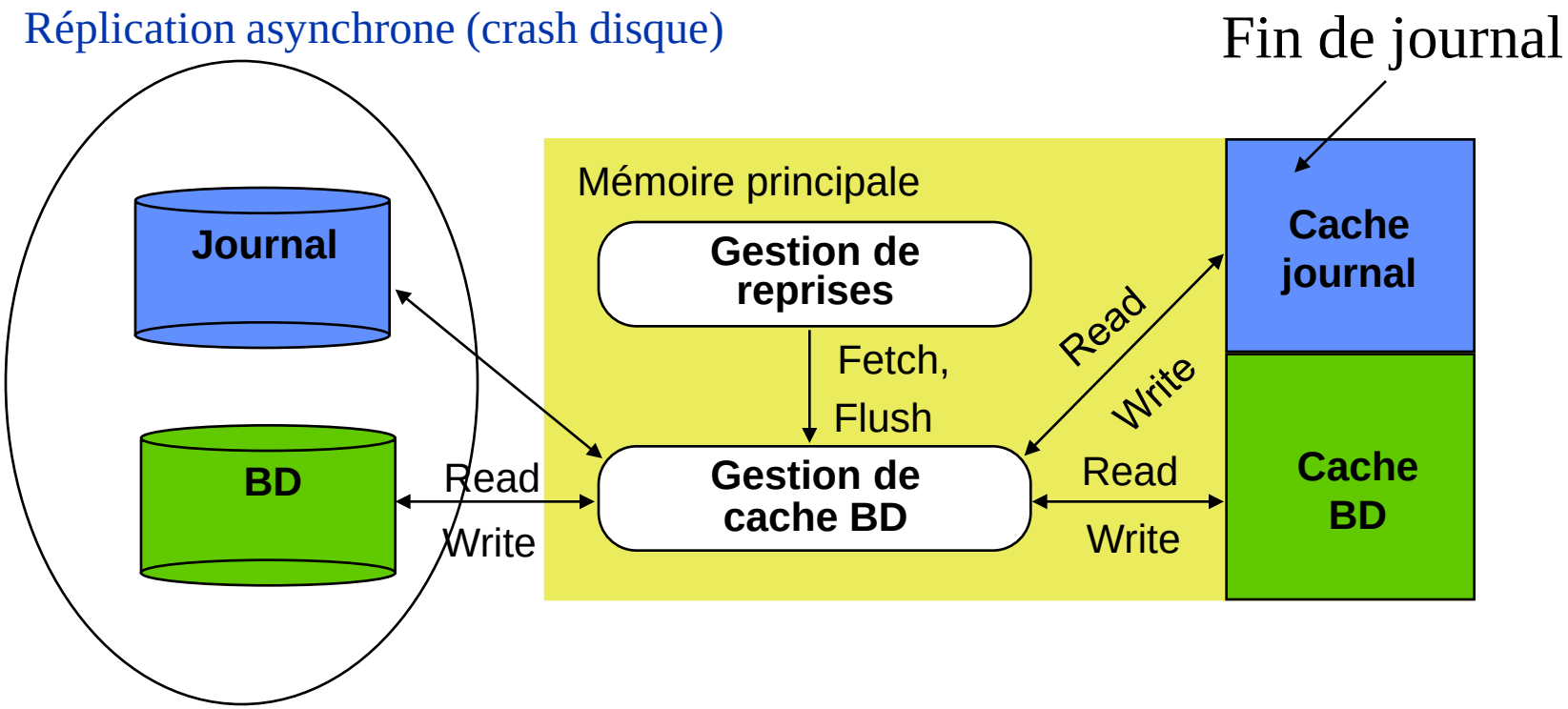
On suppose maintenant qu'une transaction T2 effectue le morceau de code suivant :

variable := Lire(A);
Ecrire(A, variable + 2);

entre l'exécution des instructions (3) et (4) de T1, sur un système qui fait les *écritures en mode immédiat*.

Comment pourra-t-on restaurer une base cohérente à la terminaison de T1 sur erreur dans chacun des cas suivant : (a) T2 a encore d'autres instructions à exécuter, et (b) T2 ayant terminé son code avec l'exécution de ses 2 instructions, l'enregistrement < *T2*, ***commit*** > figure dans le journal ?

Interface du journal



Ecriture du journal sur disque

Synchrone (forcée): à chaque ajout d'un enregistrement

- ralentit la transaction
- facilite le recouvrement

Asynchrone : périodique ou quand le buffer est plein ou…

- Au plus tard quand la transaction valide

Gestion du cache BD

Le cache améliore les performances du système, mais a des répercussions sur la reprise (dépend de la politique de migration sur le disque).

Pour simplifier le travail de reconstruction, on peut

- empêcher des migrations cache->disque
  - Fix : ne peut migrer *pendant* la transaction
- forcer la migration en fin de transaction
  - Flush : doit migrer à chaque commit

Fix et flush facilite le recouvrement mais contraignent la gestion du cache

Gestion du cache BD

Impact sur la reprise :

- No-fix/no-flush : UNDO/REDO

Undo nécessaire car les écritures de transactions non validées ont peut être été écrites sur disque et donc rechargées à la reprise.

Redo nécessaire car les écritures de transactions validées n'ont peut être pas été écrites sur disque

- Fix/no-flush : REDO
- No-fix/flush : UNDO

Quand écrire le journal sur disque?

Supposons une transaction *T* qui modifie la page *P*

Cas chanceux

- le système écrit *P* dans la BD sur disque
- le système écrit le journal sur disque pour cette opération
- PANNE!... (avant la validation de *T*)

Nous pouvons reprendre (undo) en restaurant *P* à son ancien état grâce au journal

Cas malchanceux

- le système écrit *P* dans la BD sur disque
- PANNE!... (avant l'écriture du journal)

Nous ne pouvons pas récupérer car il n'y a pas d'enregistrement avec l'ancienne valeur dans le journal

Solution: le protocole Write-Ahead Log (WAL)

Protocole WAL

Observation:

- si la panne précède la validation de transaction, alors toutes ses opérations doivent être défaites, en restaurant les images avant (*partie undo* du journal)
- dès qu'une transaction a été validée, certaines de ses actions doivent pouvoir être refaites, en utilisant les images après (*partie redo* du journal)

Protocole WAL:

- avant d'écrire dans la BD sur disque, la partie *undo* du journal doit être écrite sur disque
- lors de la validation de transaction, la partie *redo* du journal doit être écrite sur disque avant la mise-à-jour de la BD sur disque

Points de reprise

Réduit la quantité de travail à refaire ou défaire lors d'une panne

Un point de reprise enregistre une liste de transactions actives

Pose d'un point de reprise:

- écrire un enregistrement begin_checkpoint dans le journal
- écrire les buffers du journal et de la BD sur disque
- écrire un enregistrement end_checkpoint dans le journal

Remarque :

- Procédure similaire pour rafraichissement des sauvegardes

Procédures de reprise

Reprise à chaud

- perte de données en mémoire, mais pas sur disque
- à partir du dernier point de reprise, déterminer les transactions
  - validées : REDO
  - non validées : UNDO

Reprise à froid

- perte de données sur disque
- à partir de la dernière sauvegarde et du dernier point de reprise, faire REDO des transactions validées
- UNDO inutile

Il peut y avoir des pannes pendant la procédure de reprise….

Modèles étendus

Applications longues composées de plusieurs transactions coopérantes

Seules les mises-à-jour sont journalisées

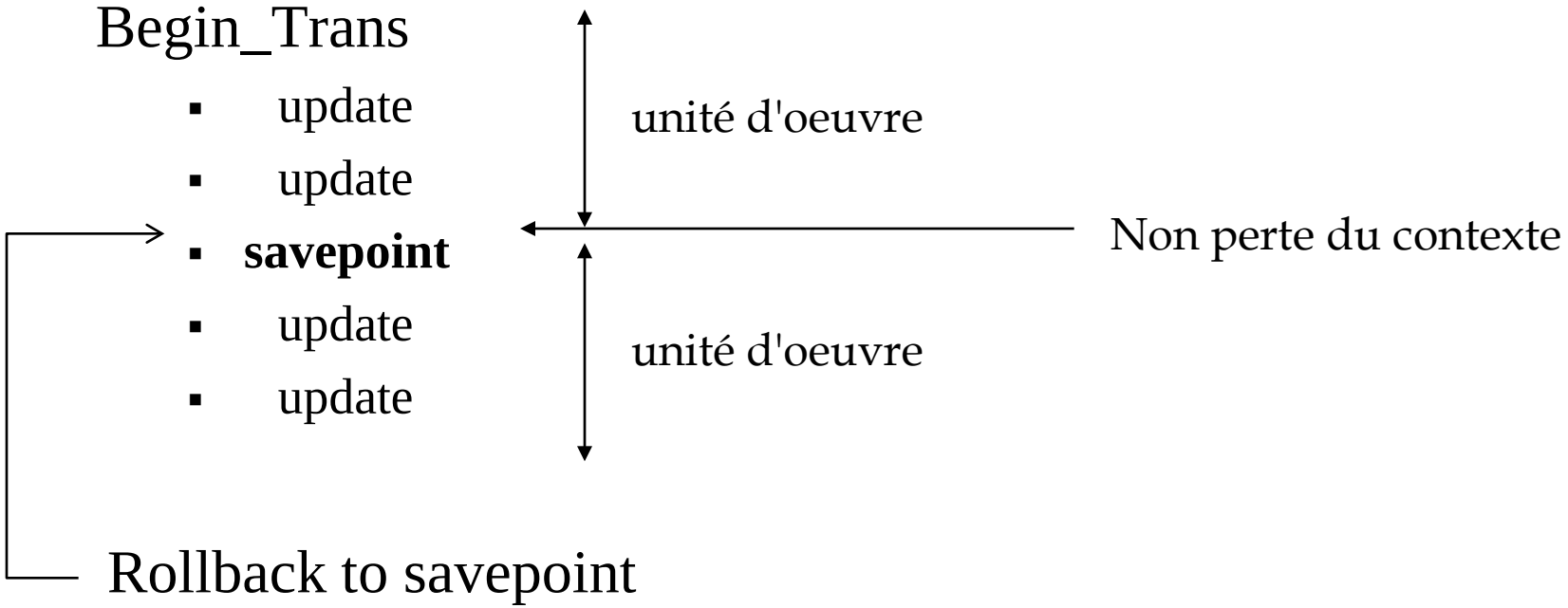
Si nécessité de défaire une suite de transactions:

- contexte ad-hoc dans une table temporaire
- nécessité d'exécuter des compensations

Points de Sauvegardes

Introduction de points de sauvegarde intermédiaires

- (savepoint, commitpoint)



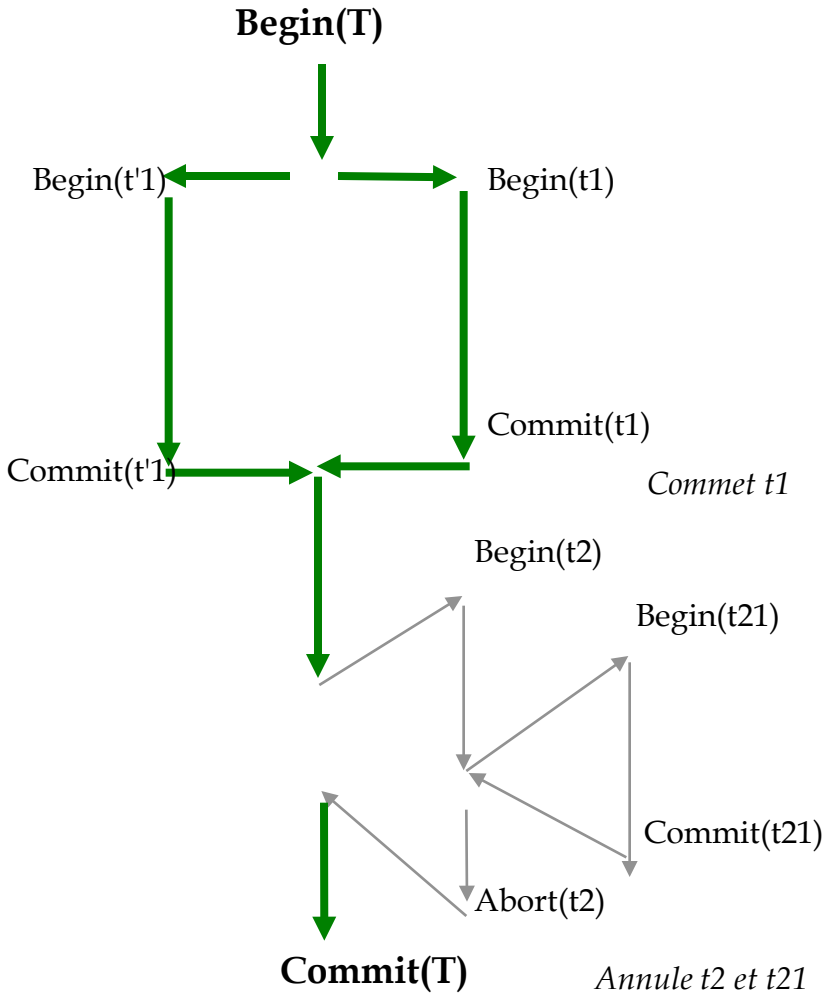
Transactions Imbriquées

OBJECTIFS

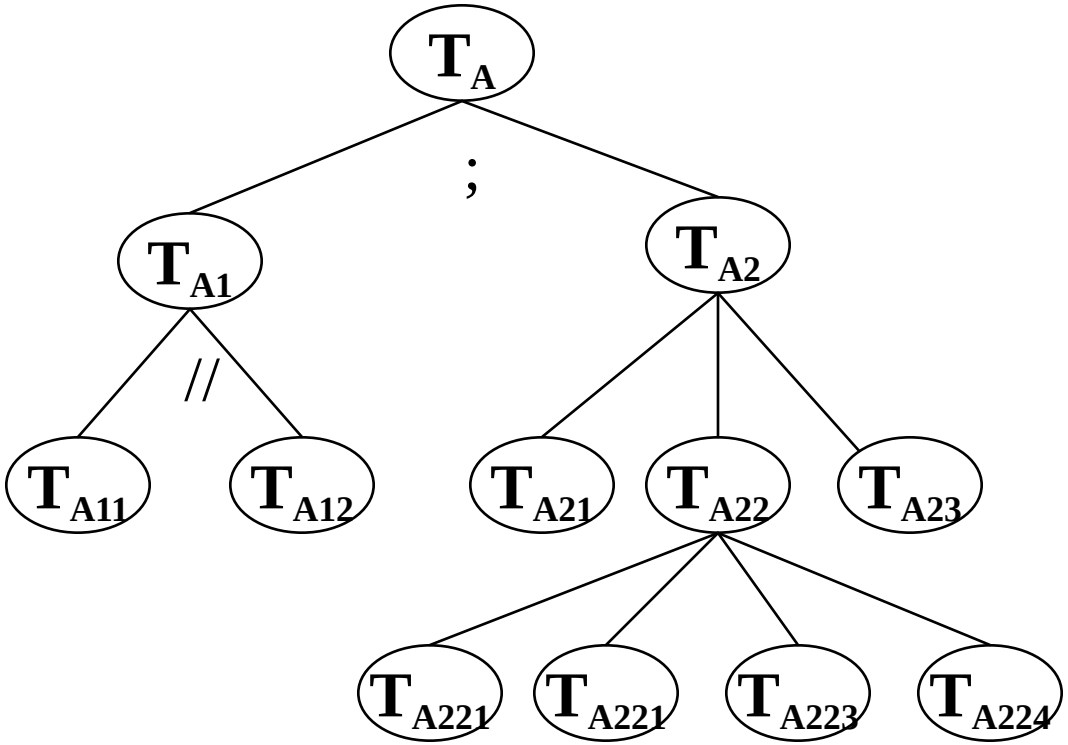
- Obtenir un mécanisme de reprise multi-niveaux
- Permettre de reprendre des parties logiques de transactions
- Faciliter l'exécution parallèle de sous-transactions

SCHEMA

- Reprises et abandons partiels
- Possibilité d'ordonner ou non les sous-transactions



Transactions Imbriquées T.I. (1)
[J. E. Moss 85]



TI = Ensemble de transactions qui peuvent être elles mêmes imbriquées (on dit aussi « emboitées »)

Transactions Imbriquées (2)

Sous-transaction : *unité d'exécution*

- Une sous-transaction démarre après et finit avant sa mère. Des sous-transactions au même niveau peuvent être exécutées en concurrence (sur différents sites)
- Chaque sous-transaction est exécutée de manière indépendante; elle peut décider soit de valider soit d'abandonner

Sous transaction : *unité de reprise*

- Si une sous-transaction valide, la mise à jour de la BD a lieu seulement lorsque la transaction racine valide.
- Si une sous-transaction abandonne, ses descendants abandonnent.

Toutes les sous-transactions (inclus la racine) doivent respecter cette nouvelle définition d'atomicité et la propriété d'isolation

Seule la racine doit préserver les propriétés de cohérence et de (nouvelle) durabilité.

Transactions Imbriquées (3)

Une sous-transaction peut-être:

- **Obligatoire**: si elle abandonne, son père doit abandonner
- **Optionnelle**: si elle abandonne, son père peut continuer (abandon partiel)
- **Contingente** : si elle abandonne, une autre peut être exécutée à sa place

Si d'abandon d'une sous-transaction, le père soit :

- Abandonne aussi (et donc son sous-arbre abandonne)
- Continue sans les résultats de la sous-transaction abandonnée
- Relance la sous-transaction ou une alternative

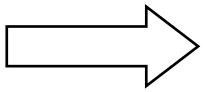
Transactions Imbriquées (4)
Contrôle de la concurrence

(R1) Seules les transactions feuilles accèdent aux objets

(R2) Quand une sous-transaction valide, ses verrous sont hérités par sa mère

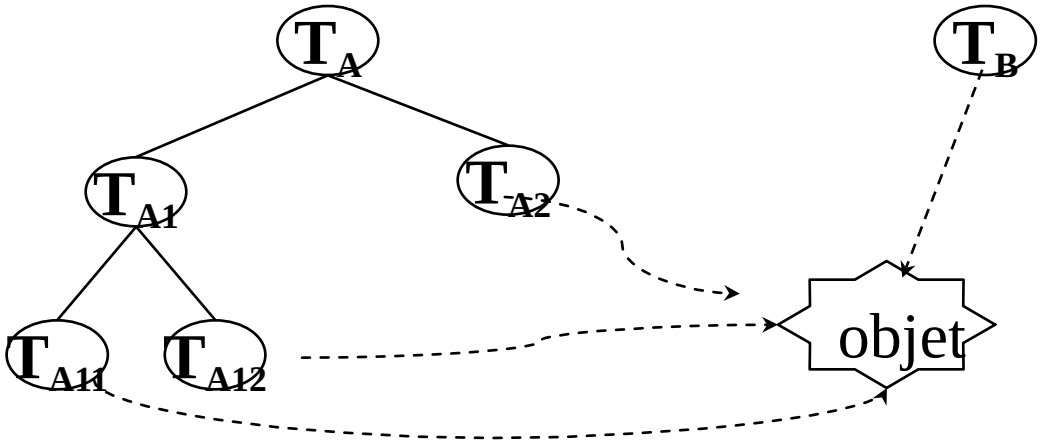
(R3) Quand une sous-transaction abandonne, ses verrous sont relâchés

(R4) Une sous-transaction ne peut accéder à un verrou que si il est libre ou détenu par un ancêtre

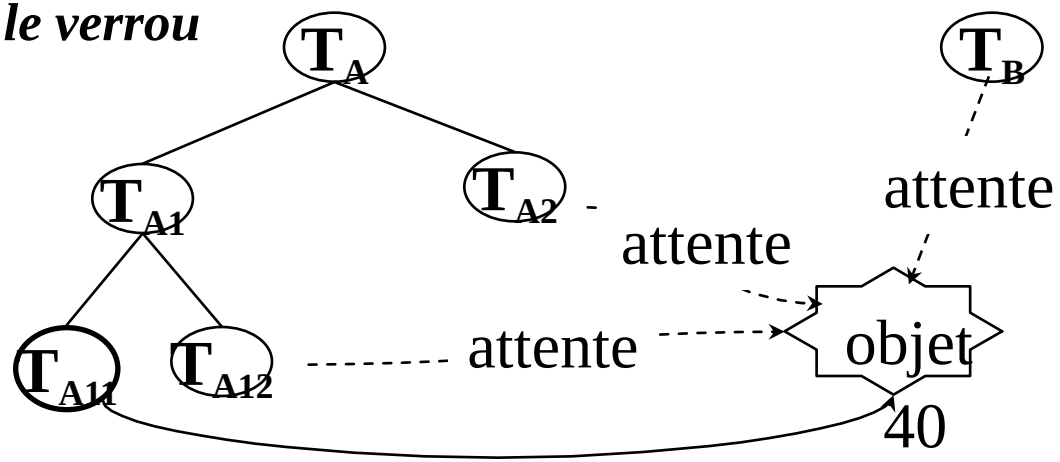


R1 à R4 garantissent l'isolation

Transactions Imbriquées (5)
Contrôle de la concurrence

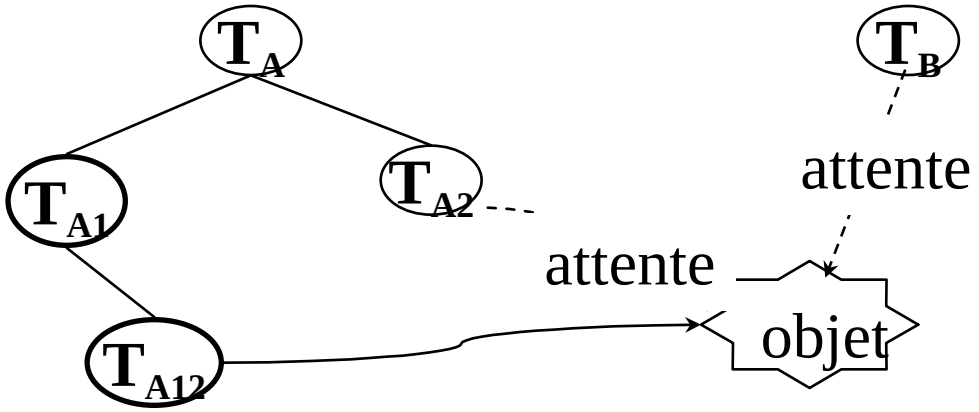


T_{A11} obtient le verrou

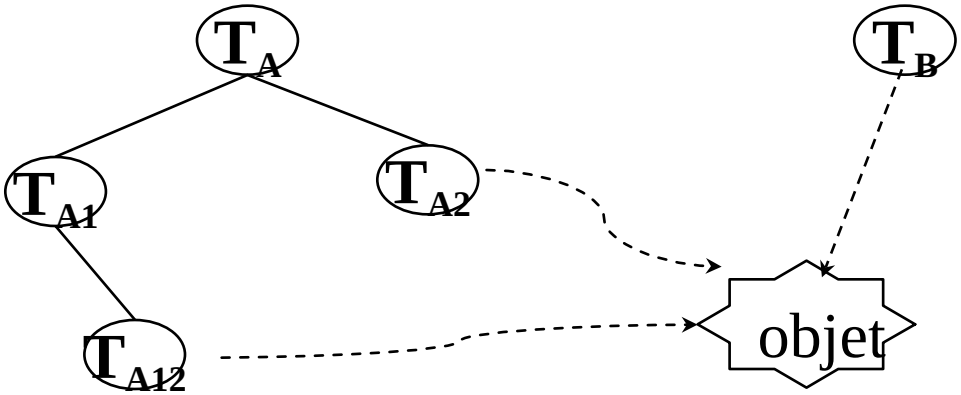


Transactions Imbriquées (6)
Contrôle de la concurrence

Si T_{A11} valide



Si T_{A11} abandonne



41

Transactions Imbriquées (7)

(R’1) Quand une sous-transaction valide, ses effets sont hérités par sa mère

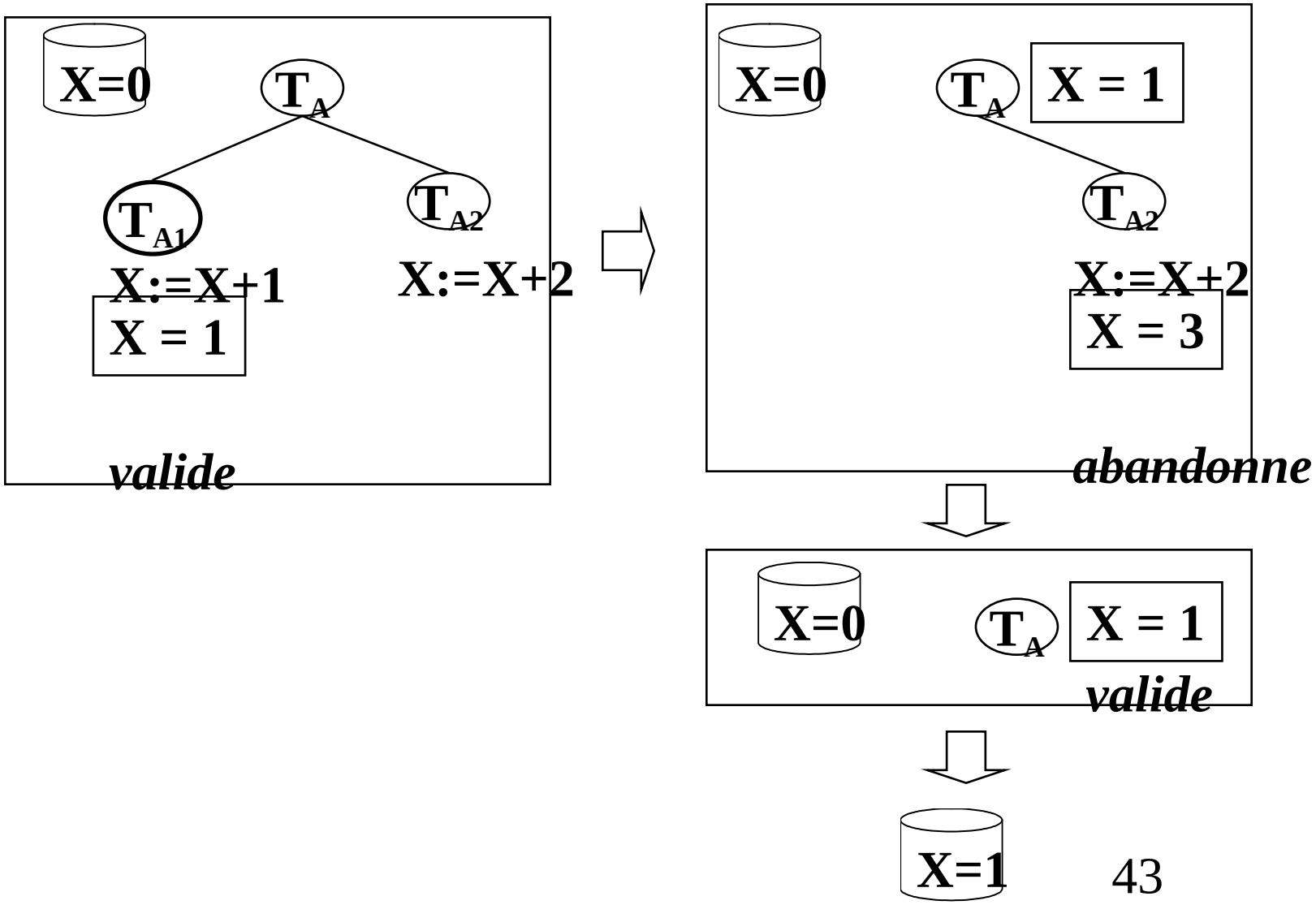
(R’2) Quand une sous-transaction abandonne, ses effets sont abandonnés.

(R’3) Quand la transaction racine valide, ses effets sont écrits dans la base

R’1 à R’3 garantissent atomicité et durabilité

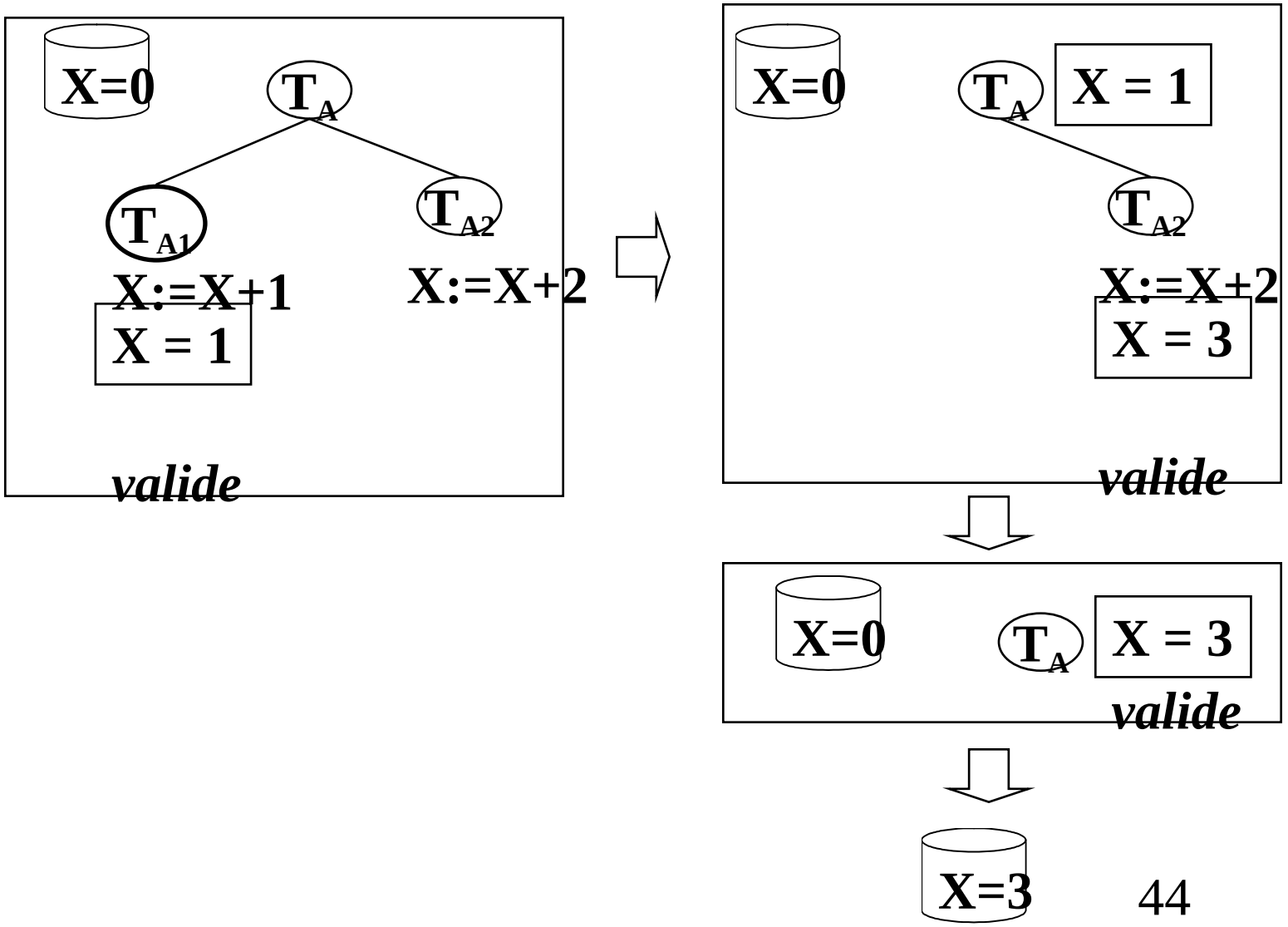
42

Transactions Imbriquées (8)



43

Transactions Imbriquées (8b)



44

Sagas

Groupe de transactions avec transactions compensatrices

En cas de panne du groupe, on exécute les compensations

Seules les sous-transactions sont isolées…. Ce n'est plus vraiment un modèle transactionnel