

Gestion des transactions

Dr N. BAME

Plan

Définition

Exemples de programmes

Propriétés des transactions

Fiabilité et tolérance aux pannes

- Journaux
- Protocoles de journalisation
- Points de reprise

Introduction

L'exécution **concurrente** des programmes des utilisateurs est essentielle dans un SGBD.

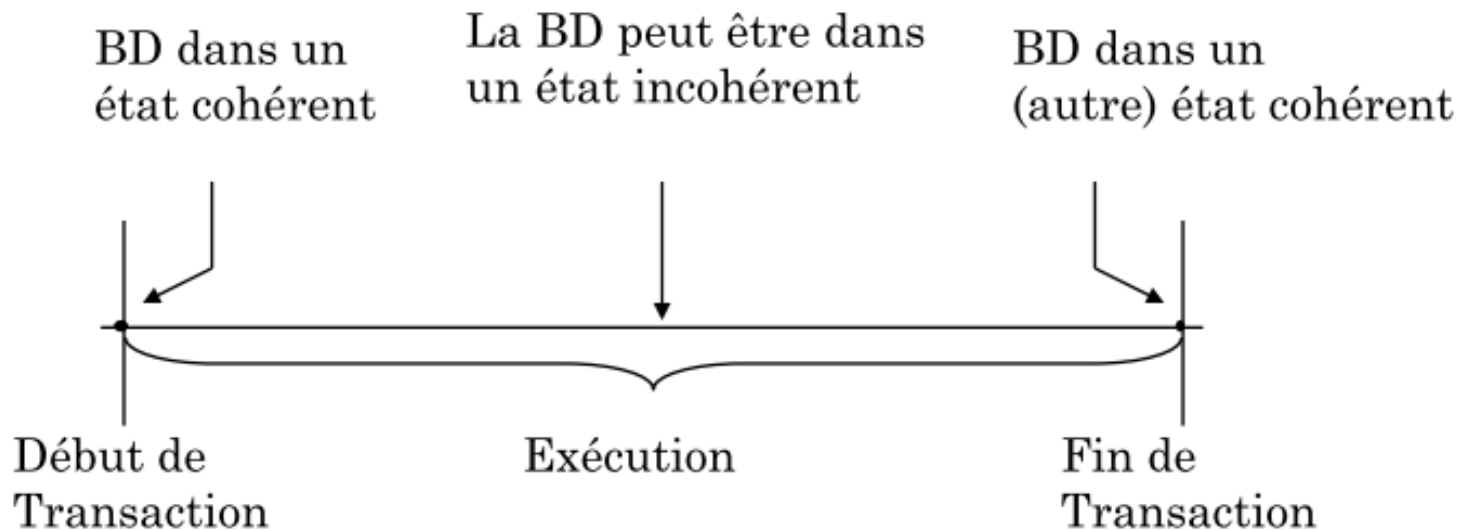
Les programmes des utilisateurs peuvent contenir **plusieurs opérations** sur les données de la BD, mais le SGBD n'est concerné **que par** les opérations de **écriture/lecture** vers/de la base.

Une **transaction** correspond à une vision d'**un programme** d'utilisateur du côté du SGBD :

- une **séquence** de lectures/écritures.

Transaction

- **Transaction** = séquence d'actions qui transforment une BD d'un **état cohérent** vers un **autre état cohérent**
- opérations de lecture et d'écriture de données de différentes granularités
- **granules** = tuples, tables, pages disque, etc...



Transaction

Begin_transaction

read(*account1*, *v1*) -- opération 1

read(*account2*, *v2*) -- opération 2

$v1 \leftarrow v1 - 100$

$v2 \leftarrow v2 + 100$

write(*account1*, *v1*) -- opération 3

write(*account2*, *v2*) -- opération 4

End_transaction

Programmation

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*** qui terminent la transaction :

- **Commit :**

- validation des modifications (explicite ou implicite à la fin)

- **Abort** (ou **Rollback**):

- annulation de la transaction : on revient à l'état cohérent initial avant le début de la transaction

Exemple de transaction simple

Begin_transaction Budget-update

begin

EXEC SQL UPDATE Project

SET Budget = Budget * 1.1

WHERE Pname = `CAD/CAM`;

end . {Budget-update}

/* validation (commit) implicite à la fin de la transaction */

BD exemple

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

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

CUST(CNAME, ADDR, TEL)

FC(#FNO, DATE, #CNAME, SPECIAL)

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 réservation 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}

Transactions : Implantation

BD : $a = 2$; $b = 3$;

T: $x = \text{read}(a)$; $y = \text{read}(b)$; $\text{write}(a, y)$; $\text{write}(b, x)$;

Les transactions fournissent des exécutions :

- **fiables** : $a = 3$ et $b = 2$ après la validation d'une exécution de T (même en **présence de pannes** : coupure de courant, panne disque, ...)
- **correctes** : une exécution de T échange a et b (**commit**) ou ne fait rien (**abort**)
- **cohérentes** : $a = 2$ et $b = 3$ après deux **exécutions concurrentes et validées** de T

T1: $x = \text{read}(a)$; $y = \text{read}(b)$; $\text{write}(a, y)$; $\text{write}(b, x)$;

T2: $x = \text{read}(a)$; $y = \text{read}(b)$; $\text{write}(a, y)$; $\text{write}(b, x)$;

- **Problème**: Comment implanter/garantir ces propriétés ?

Gestion de pannes

Gestion de concurrence

Propriétés ACID des transactions

A**TOMICITE** : Les opérations entre le début et la fin d'une transaction forment une unité d'exécution. soit toutes les opérations de la transaction sont exécutées, soit aucune.

C**OHERENCE** : Chaque transaction accède et retourne une base de données dans un état cohérent (pas de violation de contrainte d'intégrité).

I**SOLATION** : Le résultat d'un ensemble de transactions concurrentes et validées correspond au résultat d'une exécution successive des mêmes transactions.

D**URABILITE**: Les mises-à-jour des transactions validées persistent.

Propriétés des transactions

ATOMICITE : Les opérations entre le début et la fin d'une transaction forment une *unité d'exécution*

DURABILITE: Les mises-à-jour des transactions validées *persistent*.

COHERENCE : Chaque transaction accède et retourne une base de données dans un état cohérent (pas de violation de contrainte d'intégrité).

ISOLATION : Le résultat d'un ensemble de *transactions concurrentes* et validées correspond au résultat d'une exécution *successive* des mêmes transactions.

1. Gestion de pannes

- Cache
- Journalisation

2. Gestion de cohérence

- Sérialisabilité
- Algorithmes de contrôle de concurrence

Gestion des pannes

Types de pannes

Panne de transaction

- abandon normal (prévu dans le programme/incohérence *logique*) ou dû à un *deadlock* (conflit entre transactions)
- *pas de perte* « physique » de contenu

Panne processeur/mémoire

- panne de processeur, mémoire, alimentation, ...
- *le contenu de la mémoire principale* (programme et *buffer*) **est perdu**


Panne disque

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

Violation de l'atomicité

Transaction T1:

```
1: begin_transaction
2: read(account1, v1)
3: read(account2, v2)
4: v1 ← v1 - 100
5: v2 ← v2 + 100
6: write(account1, v1)
7: write(account2, v2)
8: end_transaction
```



Exemple:

- Défaillance **après ligne 6** et **avant ligne 7**.
- Pour éviter l'incohérence, l'effet de la ligne 6 doit être défait (*undo*) avant que **account1** soit visible à d'autres transactions.

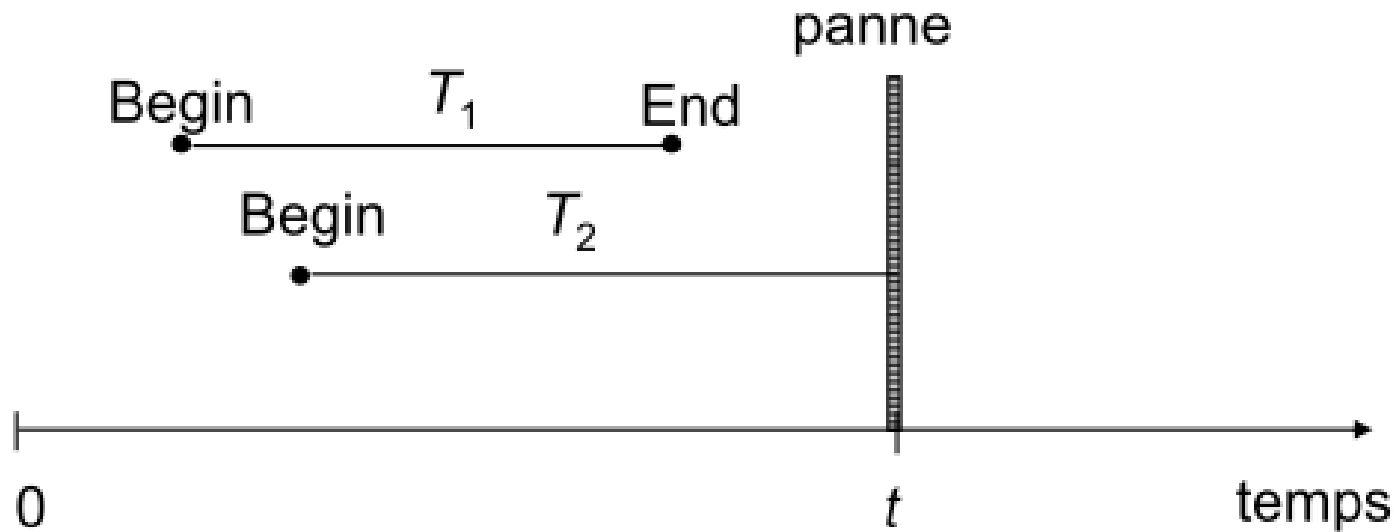
Violation de la durabilité

- Transaction T1 ci-dessus
- Initialement: **account1 = account2 = 0**
- Après exécution de T1: **account1 = -100** et **account2 = 100**

Si **account1** et **account2** sont en mémoire volatile, un **crash** conduit à **perdre le nouvel état** de **account1** et **account2**.

L'effet de T1 est perdu.

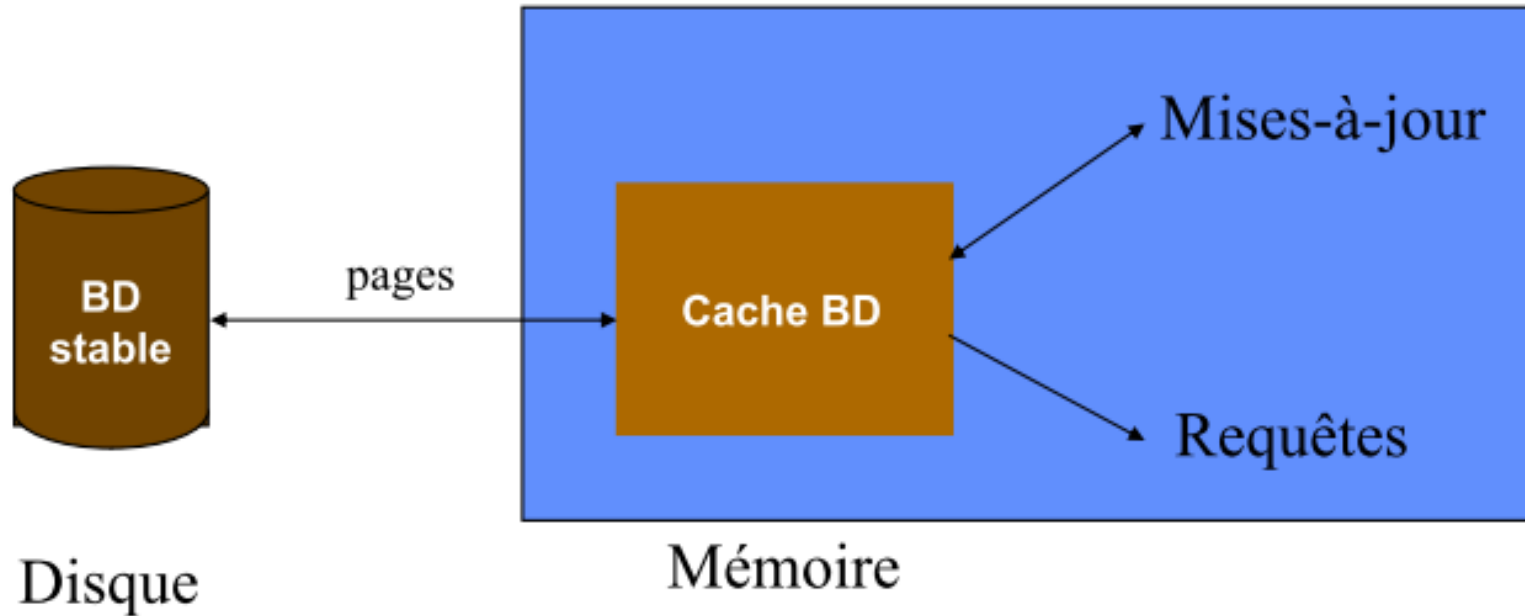
Gestion de pannes



Lors de la panne

- toutes les mises-à-jour de T_1 doivent perdurer (**durabilité**)
- aucune mise-à-jour de T_2 ne doit être faite dans la BD (**atomicité**)

Lecture/écriture BD



Cache BD : sert à augmenter la performance du système.

Mises-à-jour d'une base de données

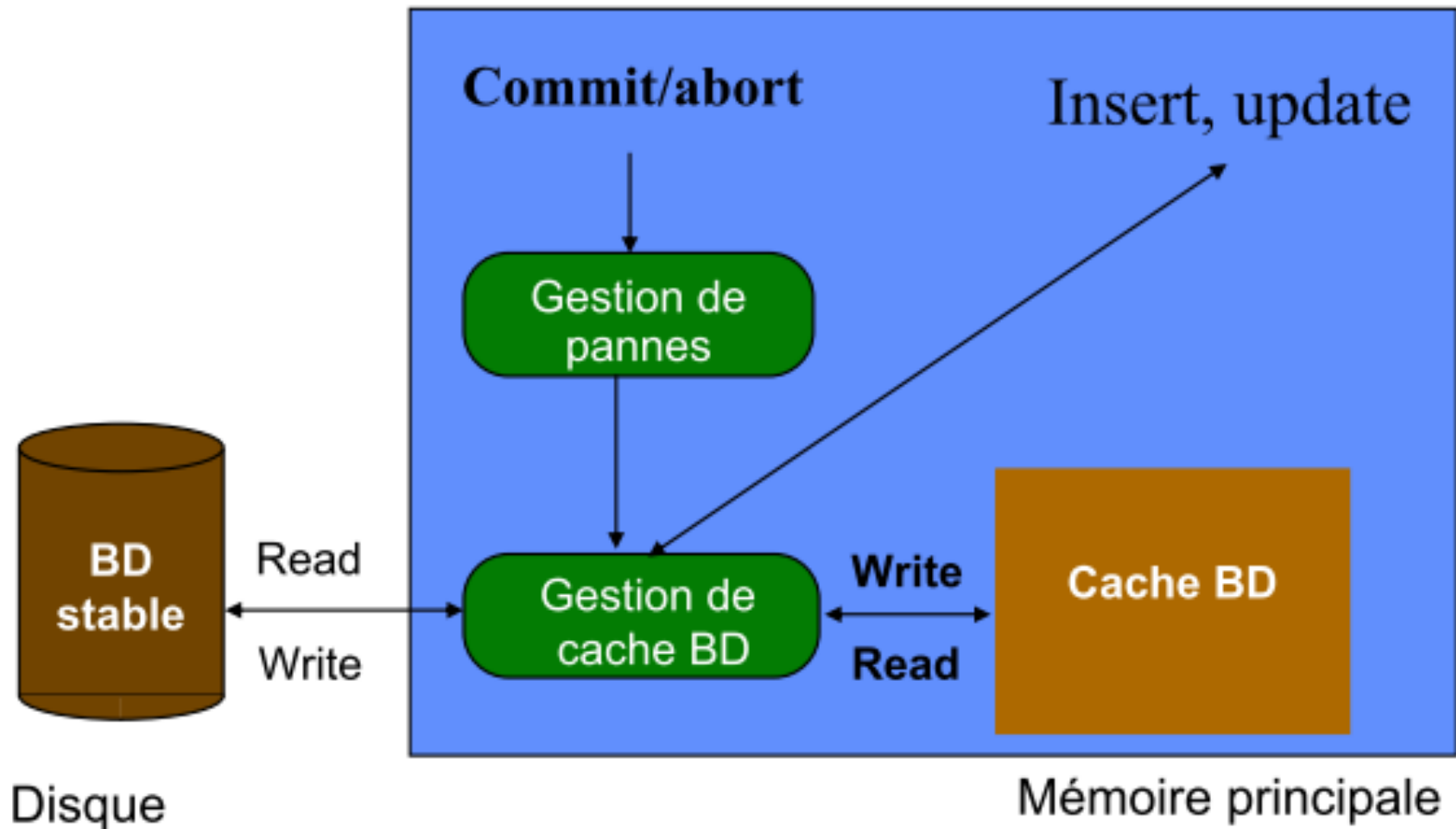
Mise-à-jour *en place*

- chaque mise-à-jour cause la modification de données dans des **pages du 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** qui remplacent les pages d'origine au moment du commit.
- peu utilisée en pratique car *très coûteuse* :
 - *fragmentation des données sur disque*
 - nécessite la mise-à-jour d'index (même quand la clé ne change pas)

Architecture pour la gestion de pannes



Problème du cache BD

Cache BD :

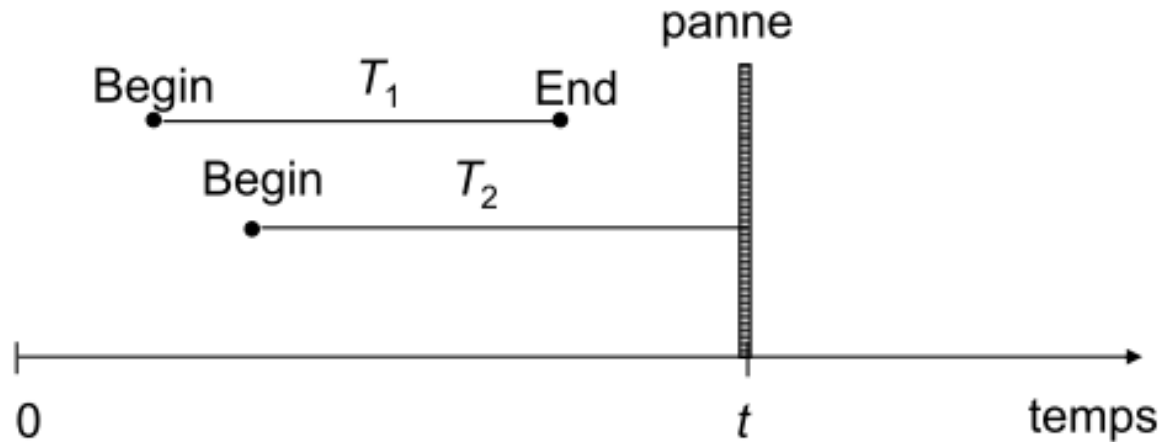
- sert à augmenter la performance du SGBD en évitant les lectures/écritures disque
- une page du cache peut contenir des données validées et non-validées

Problème : Comment garantir la ***durabilité*** et ***l'atomicité***

- *sans forcer l'écriture* des données validées (**commit**) sur disque (non-force)
 - **REDO** : il faut garantir que les modifications de *transaction validées* seront prises en compte après une panne
- *sans empêcher d'écrire* des données non-validées sur disque (steal) :
 - **UNDO** : il faut être capable d'annuler des modifications de *transactions annulées*

Solution : maintenir un **journal** des mises-à-jour

Pourquoi journaliser?

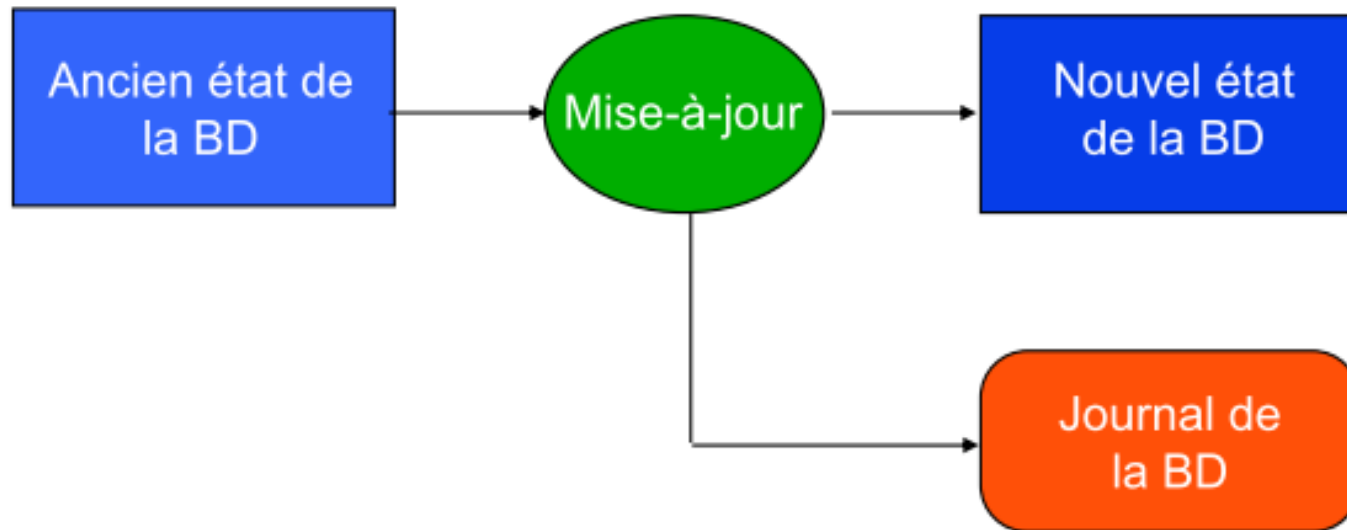


Après la reprise :

- 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**)

Atomicité : Journal de la BD

- Chaque action d'une transaction est enregistrée dans le journal qui est un fichier séquentiel répliqué sur des disques différents de la BD (un crash disque ne doit pas détruire le journal et les données!) :



Journalisation

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

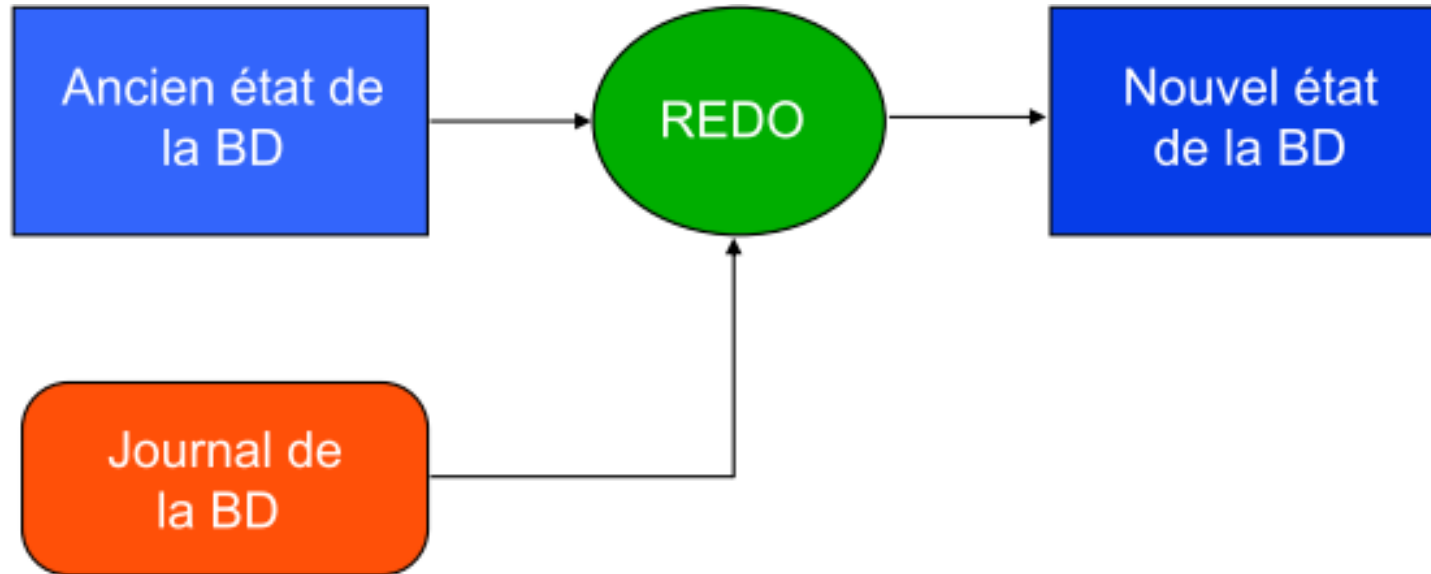
- identifiant de transaction
- type d'opération (action)
- granules accédés par la transaction pour réaliser l'action
- ancienne valeur de granule (image avant : UNDO)
- nouvelle valeur de granule (image après : REDO)
- ...

Exemple de journal

Début du journal \longrightarrow

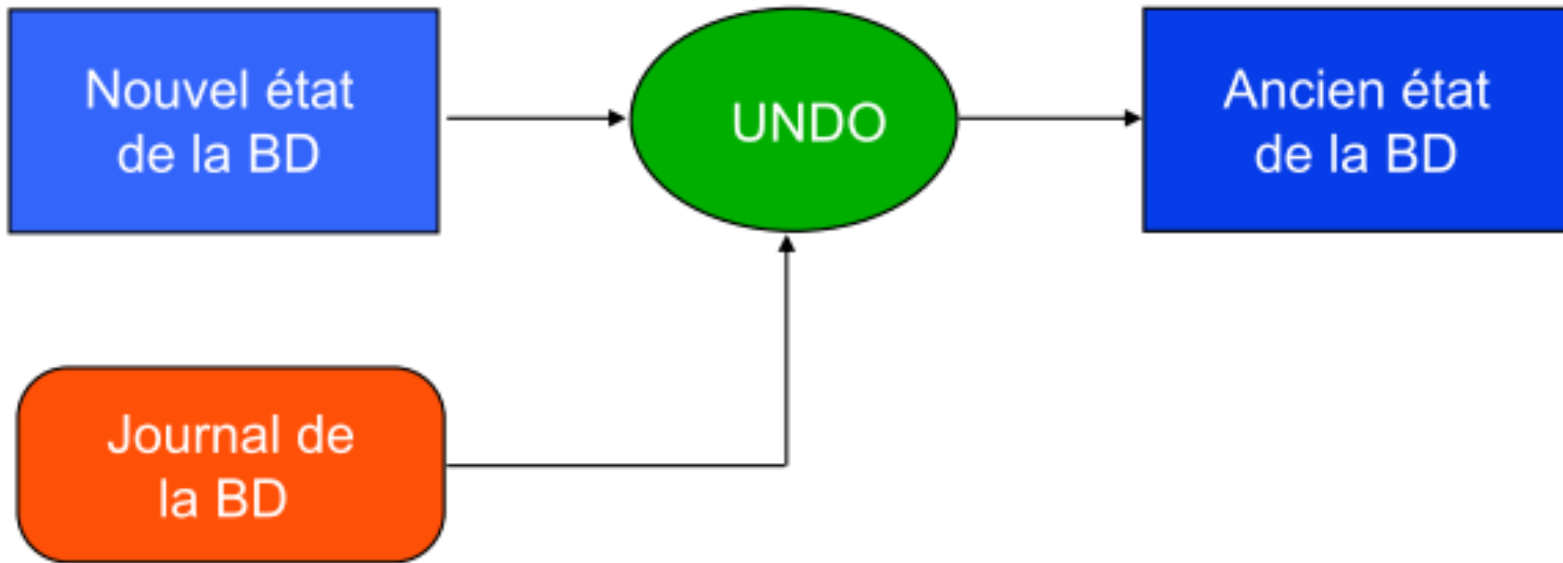
- T_1 , begin
- T_1 , x, 99, 100
- T_2 , begin
- T_2 , y, 199, 200
- T_3 , begin
- T_3 , z, 51, 50
- T_2 , w, 1000, 10
- T_2 , commit
- T_4 , begin
- T_3 , abort
- T_4 , y, 200, 50
- T_5 , begin
- T_5 , w, 10, 100
- T_4 , commit

Protocole REDO



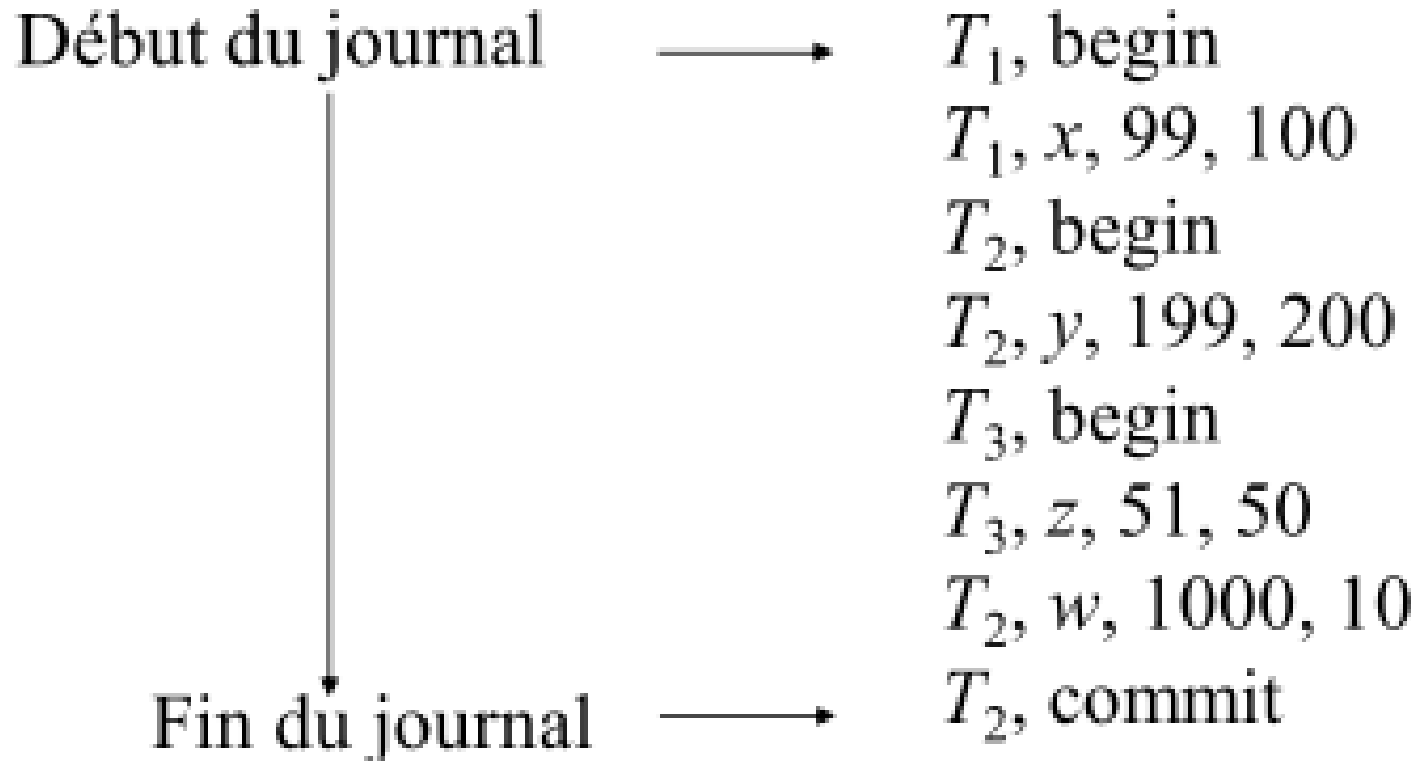
L'opération REDO utilise l'information du journal (image après) pour **refaire les actions qui ont été exécutées ou interrompues.**

Protocole UNDO



L'opération UNDO utilise l'information du journal (image avant) pour **restaurer l'image avant** du granule.

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



UNDO : T_2 rien (marquée pour Redo), $z:=51$, $x:=99$

REDO : $y:=200$, $w:=10$

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

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 ?

Comment restaurer la base en mode nofix ? En mode fix ?

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 (no-Fix)*.

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 ?

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

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

Observations :

- si la panne précède la validation de la 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

Point de reprise : enregistrement de toutes les modifications d'une liste de transactions actives pour réduire la quantité de travail à refaire ou défaire lors d'une panne

Pose d'un point de reprise:

- écrire *begin_checkpoint* dans le journal
- écrire les *buffers du journal et de la BD sur disque*
- écrire *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 validée : UNDO
- Variante *ARIES* (*IBM DB2, MS SQL Server*) : refaire *toutes* les transactions et défaire les transactions non terminées au moment du crash

Reprise à froid :

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

Conclusion

- La gestion de pannes garantit la *durabilité* et *l'atomicité* des transactions.
- Elle ne doit pas trop pénaliser la performance du système (trop de lectures et écritures de disque).
- La **gestion de concurrence entre transactions** est *indépendante* de la gestion des pannes.