

TRANSACTIONS ET CONCURRENCE

- Le concept de transaction
- Les états d'une transaction
- Exécutions concurrentes
- Sérialisabilité
- Définition des transactions dans SQL
- Test de sérialisabilité

Le concept de transaction

Une transaction est une *unité de programme* qui accède aux données de la base en lecture et/ou écriture

Une transaction accède à un état *cohérent* de la base

Durant l'exécution d'une transaction, l'état de la base peut ne pas être cohérent

Quand une transaction est *validée* (commit), l'état de la base **doit** être cohérent

Deux types de problèmes

- problèmes systèmes (récupérabilité)
- exécutions concurrentes de plusieurs transactions (sérialisabilité)

Pour préserver l'intégrité des données, le système doit garantir certaines propriétés :

- **Atomicité** : Soit toutes les opérations de la transaction sont validées, ou bien aucune opération ne l'est.
- **Cohérence** : L'exécution d'une transaction préserve la cohérence de la base.
- **Isolation** : Même si plusieurs transactions peuvent être exécutées en concurrence, aucune n'est censée prendre en compte les autres transactions ; i.e pour chaque paire de transactions T_i, T_j , pour T_i tout se passe comme si T_j s'est terminée avant le début de T_i ou bien qu'elle a commencé après la fin de T_i (les résultats intermédiaires de T_j ne lui sont pas apparents)
- **Durabilité** : Si une transaction est validée, alors tous les changements qu'elle a faits sont persistants (même s'il y a un crash)

Ce sont les propriétés **ACID**

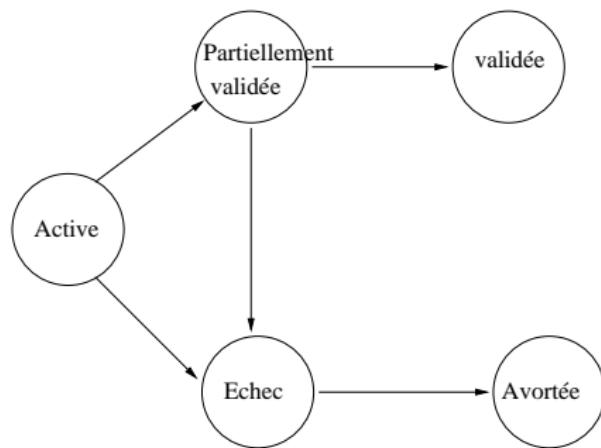
Exemple

- Une transaction qui transfert 1000 Euros du compte A vers le compte B
 - 1 $Lire(A)$
 - 2 $A := A - 1000$
 - 3 $Ecrire(A)$
 - 4 $Lire(B)$
 - 5 $B := B + 1000$
 - 6 $Ecrire(B)$
- La base est cohérente si la somme $A + B$ ne change pas suite à l'exécution de la transaction (cohérence)
- Si la transaction “échoue” après l'étape 3, alors le système doit s'assurer que les modifications de A ne soient pas persistantes (atomicité)

- Une fois l'utilisateur est informé que la transaction est validée, il n'a plus à s'inquiéter du sort de son transfert (durabilité)
- Si entre les étapes 3 et 6, une autre transaction est autorisée à accéder à la base, alors elle "verra" un état incohérent ($A + B$ est inférieur à ce qu'elle doit être). L'isolation n'est pas assurée. La solution triviale consiste à exécuter les transactions en séquence

- **Active** : la transaction reste dans cet état durant son exécution
- **Partiellement validée** : juste après l'exécution de la dernière opération
- **Echec** : après avoir découvert qu'une exécution "normale" ne peut pas avoir lieu
- **Avortée** : Après que toutes les modifications faites par la transaction soient annulées (Roll back). Deux options
 - Réexécuter la transaction
 - Tuer la transaction
- **Validée** : après l'exécution avec succès de la dernière opération

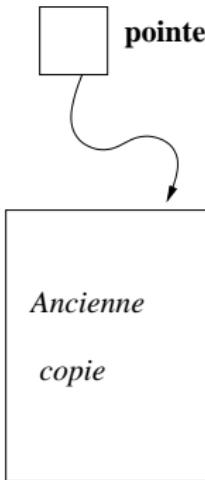
Etats d'une transaction



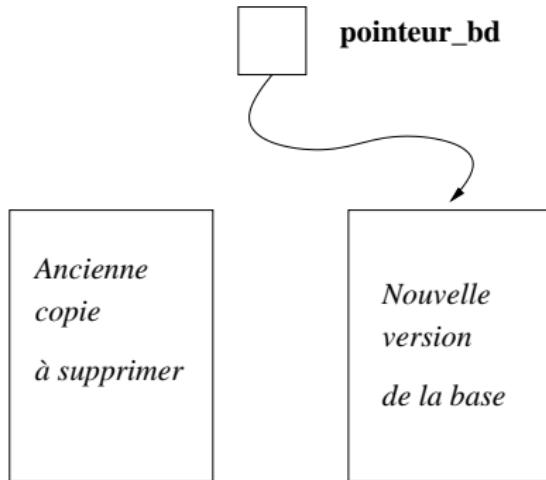
Approche naïve

- C'est le *mécanisme de reprise sur panne*
- La notion de *copie* (shadow database)
 - On suppose qu'une seule transaction peut être exécutée à la fois
 - Un pointeur **pointeur_bd** pointe vers la version cohérente courante de la base.
 - Toutes les mises à jour sont exécutées sur une copie. **pointeur_bd** ne pointera sur la copie que si la transaction est validée.
 - Si la transaction échoue, alors la copie est supprimée.

Implémentation de l'atomicité



a) avant la mise à jour
inefficace si la base est volumineuse.



b) après la māj

- Plusieurs transactions peuvent être exécutées en concurrence
 - une meilleure utilisation du processeur (une transaction peut utiliser le processeur pendant qu'une autre accède au disque)
 - réduction du temps de réponse aux transactions (une transaction courte n'a pas à attendre la fin d'une transaction longue)
- **Contrôle de la concurrence** : mécanisme permettant l'interaction entre transactions tout en assurant l'intégrité de la base.

Ordonnancement

En anglais : Schedule.

- *Une séquence chronologique spécifiant l'ordre d'exécution d'opérations de plusieurs transactions*
- **Exemple :**

| T_1 | T_2 |
|--|--|
| $Lire(A)$ $A := A - 1000$ $Ecrire(A)$ $Lire(B)$ $B := B + 1000$ $Ecrire(B)$ | $Lire(A)$ $Temp := A * 0, 1$ $A := A - Temp$ $Ecrire(A)$ $Lire(B)$ $B := B + Temp$ $Ecrire(B)$ |

C'est un ordonnancement “en série” de T_1 et T_2 . On l'appellera O_1

Ordonnancement (suite)

| T_1 | T_2 |
|-----------------|--------------------|
| $Lire(A)$ | |
| $A := A - 1000$ | |
| $Ecrire(A)$ | |
| | $Lire(A)$ |
| | $Temp := A * 0, 1$ |
| | $A := A - Temp$ |
| | $Ecrire(A)$ |
| $Lire(B)$ | |
| $B := B + 1000$ | |
| $Ecrire(B)$ | |
| | $Lire(B)$ |
| | $B := B + Temp$ |
| | $Ecrire(B)$ |

L'ordonnancement O_3 représente une exécution “entrelacée” de T_1 et T_2 . Il est équivalent à $\langle T_1, T_2 \rangle$.

Ordonnancement (suite)

L'ordonnancement suivant (O_4) ne préserve pas la valeur de $A + B$. Pourtant T_1 et T_2 prises à part préservent cette valeur. C'est un ordonnancement qu'on ne doit pas accepter.

| T_1 | T_2 |
|--|--|
| $\frac{\begin{array}{l} Lire(A) \\ A := A - 1000 \end{array}}{Ecrire(A) \\ Lire(B) \\ B := B + 1000 \\ Ecrire(B)}$ | $\begin{array}{l} Lire(A) \\ Temp := A * 0, 1 \\ A := A - Temp \\ Ecrire(A) \\ Lire(B) \\ \\ B := B + Temp \\ Ecrire(B) \end{array}$ |

Dans la suite, on ne va considérer que les opérations de lecture et d'écriture.

- Hypothèse : Chaque transaction prise à part préserve la cohérence de la base.
- Ainsi, l'exécution en série préserve la cohérence.
- Un ordonnancement entrelacé est sérialisable s'il est équivalent à un ordonnancement en série.

Soit O un ordonnancement composé des transactions $\{T_1, \dots, T_n\}$.

O est sérialisable si son exécution donne le même résultat que l'exécution de T_1, \dots, T_n dans un ordre quelconque.

Il faut donc tester toutes les permutations possibles qui sont en nombre de $n!$

Ceci est la définition générale mais qui est difficile à implémenter. C'est pour ça qu'on a proposé d'autres définitions qui peuvent être testées sans avoir à exécuter tous les ordonnancements en série possibles (i.e les $n!$).

- Les instructions t_i et t_j des transactions T_i et T_j sont en conflit s'il existe un objet Q accédé par t_i et t_j et l'une d'elles écrit Q . Si $t_i = \text{Lire}(Q)$ et $t_j = \text{Lire}(Q)$ alors il n'y a pas de conflit.
- Si un ordonnancement O peut être transformé en O' par une série de remplacements (*swaps*) d'instructions non conflictuelles, alors O et O' sont *c-équivalents*.
- O est *c-sérialisable* s'il est c-équivalent à un ordonnancement en série.
- L'ordonnancement ci-dessous n'est pas c-sérialisable

| | |
|--------------------|--------------------|
| T_3 | T_4 |
| $\text{Lire}(Q)$ | |
| | $\text{Ecrire}(Q)$ |
| $\text{Ecrire}(Q)$ | |

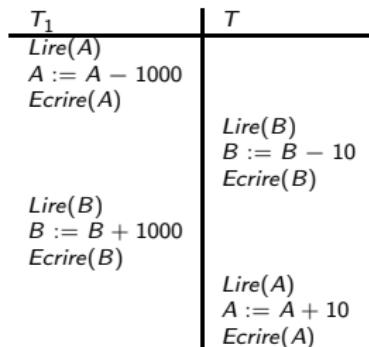
c-sérialisabilité

L'ordonnancement O_3 ci-dessous peut être transformé en O_1 . Il est donc c-sérialisable.

| T_1 | T_2 |
|-------------|-------------|
| $Lire(A)$ | |
| $Ecrire(A)$ | |
| | $Lire(A)$ |
| | $Ecrire(A)$ |
| $lire(B)$ | |
| $Ecrire(B)$ | |
| | $lire(B)$ |
| | $Ecrire(B)$ |

sérialisabilité

- L'ordonnancement ci-dessous est équivalent à $\langle T_1, T \rangle$ pourtant il n'est pas c-sérialisable



- Pour déterminer ce type d'équivalence, il faut analyser des opérations autres que *Lire* et *Ecrire*.

Reprise sur panne. Récupérabilité

- Ordonnancement *récupérable* : Si T_i lit un objet précédemment écrit par T_j alors la validation de T_j a lieu avant la validation de T_i ;
- L'ordonnancement suivant n'est pas récupérable si T_9 valide tout de suite après la lecture

| T_8 | T_9 |
|-------------|-------------|
| $Lire(A)$ | |
| $Ecrire(A)$ | |
| | $Lire(A)$ |
| | $Ecrire(B)$ |

Si T_8 doit être avortée, alors T_9 aura lu une valeur qui peut être “incohérente”

Le SGBD doit s'assurer que les ordonnancements soient récupérables.

Récupérabilité (suite)

- L'échec d'une transaction peut conduire à l'avortement de plusieurs transactions

| T_{10} | T_{11} | T_{12} |
|------------------|----------------|------------------|
| <i>Lire(A)</i> | | |
| <i>Lire(B)</i> | | |
| <i>Ecrire(A)</i> | <i>Lire(A)</i> | <i>Ecrire(A)</i> |
| | | <i>Lire(A)</i> |

Si T_{10} échoue, alors T_{11} et T_{12} doivent être avortées.

Récupérabilité (suite)

- Ordonnancement sans cascade : Si pour chaque paire T_i, T_j t.q T_j lit une donnée précédemment écrite par T_i , alors la validation de T_i a lieu avant celle de T_j .
- Un ordonnancement sans cascade de roll back est récupérable.
- Il est souhaitable de restreindre les ordonnancements à ceux qui sont sans cascade

- Les ordonnancements doivent être c-sérialisables, récupérables pour garder la cohérence de la base et de préférence sans cascade.
- Si l'on n'autorise que les ordonnancements en série, alors toutes les propriétés sont garanties.
- Faire la balance entre le taux de concurrence et le traitement en plus qui s'y greffe.

Tester la c-sérialisabilité

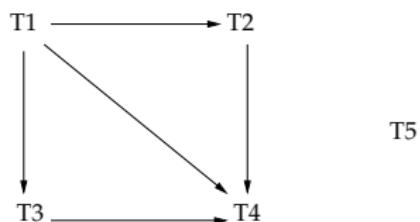
Considérer un ordonnancement O des transactions T_1, \dots, T_n . Le graphe de précédence de O est un graphe (N, A) où

- N est l'ensemble des transactions
- Il y a un arc (T_i, T_j) s'il y a un conflit entre T_i et T_j sur un objet Q et T_i accède à Q avant T_j

O est c-sérialisable ssi son graphe de précédence est acyclique.

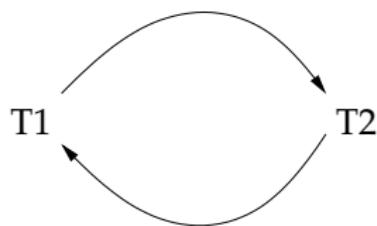
Exemple de test

| T_1 | T_2 | T_3 | T_4 | T_5 |
|--------------------------|--------------------------|-------|--|---------------------------------------|
| $Lire(Y)$ $Lire(Z)$ | $Lire(X)$ | | | |
| | $Lire(Y)$ $Ecrire(Y)$ | | $Ecrire(Z)$ | |
| $Lire(U)$ | | | | $Lire(V)$ $Lire(W)$ $Ecrire(W)$ |
| $Lire(U)$ $Ecrire(U)$ | | | $Lire(Y)$ $Ecrire(Y)$ $Lire(Z)$ $Ecrire(Z)$ | |



Exemple de test

| T_1 | T_2 |
|-------------|--------------------------|
| $Lire(A)$ | |
| $Ecrire(A)$ | $Lire(A)$ $Ecrire(A)$ |



- Tester si un ordonnancement est sérialisable *après* son exécution (ou bien sur son graphe de précédence) est inefficace.
- But : Développer des stratégies de contrôle de concurrence qui puissent garantir la c-sérialisabilité. Ces protocoles n'auront pas besoin de faire le test sur le graphe de précédence (utilisation de techniques de verrouillage, cf prochain cours)
- Pourquoi ce cours ? Pouvoir décider si un protocole est correct par rapport à une notion de sérialisabilité.

- Protocoles basés sur les verrous
- Protocoles basés sur les estampilles
- Protocoles basés sur la validation
- Différentes granularités
- Gestion des deadlock (verrous mortels)

Notion de verrouillage

Les transactions *posent des verrous* sur les données auxquelles elles veulent accéder.

Les données peuvent être verrouillées de deux manières

- **Verrou exclusif** : Dans ce cas, la donnée peut être lue et écrite. Le verrou X est attribué lors de l'exécution de l'opération $Lock_X$ (donnée)
- **Verrou partagé** : Dans ce cas, la donnée ne peut être que lue. Le verrou P est attribué suite à l'exécution de $Lock_P$ (donnée)

Les demandes de verrous sont adressées au gestionnaire de la concurrence.

Une transaction ne peut avancer tant que le verrou qu'elle demande ne lui est pas attribué.

Notion de verrouillage (suite)

- Matrice de compatibilité de verrous

| | <i>P</i> | <i>X</i> |
|----------|------------|------------|
| <i>P</i> | <i>oui</i> | <i>non</i> |
| <i>X</i> | <i>non</i> | <i>non</i> |

Une transaction ne peut poser un verrou sur une donnée que si ce verrou est compatible avec les verrous qui y sont déjà posés.

- Lors de son exécution, une transaction peut libérer certains verrous.

Utilisation de la commande *Unlock*(donnée)

Notion de verrouillage (suite)

$T_1 :$

- $Lock_P(A)$
- $Lire(A)$
- $Unlock(A)$
- $Lock_P(B)$
- $Lire(B)$
- $Unlock(B)$
- $Afficher(A + B)$

- Un protocole de verrouillage est une discipline qui dicte aux transactions comment elles demandent et comment elles libèrent les verrous

Notion de verrouillage (suite)

- Considérer l'ordonnancement suivant

| T_1 | T_2 |
|-----------------|--------------|
| $Lock_X(B)$ | |
| $Lire(B)$ | |
| $B := B - 1000$ | |
| $Ecrire(B)$ | |
| | $Lock_P(A)$ |
| | $Lire(A)$ |
| | $Lock_P(B)$ |
| $Lock_X(A)$ | |
| $Ecrire(A)$ | |
| | $Lire(B)$ |

T_1 et T_2 seront bloquées. Nous sommes en situation de *deadlock*. Pour résoudre ce problème, une des deux transactions doit être avortée et ses verrous libérés

- La possibilité de deadlock existe dans presque tous les protocoles

verrouillage 2PL

- Ce protocole garantit la c-sérialisabilité
- Phase 1 :
 - La transaction peut poser des verrous
 - Elle ne peut pas en libérer
- Phase 2 :
 - La transaction peut libérer des verrous
 - Elle ne peut plus en poser
- On associe à chaque transaction un point de verrouillage qui correspond au moment où elle pose son dernier verrou (la fin de la première phase). On montre alors que les ordonnancements sont équivalents à l'exécution en série selon l'ordre des points de verrouillage des transactions.

Verrouillage à deux phases

- Le 2PL ne garantit pas l'absence de deadlocks
- Le 2PL ne garantit pas l'absence de cascades

Extension : “2PL strict” et “2PL rigoureux”

- Dans le 2PL stricte, les transactions gardent leurs verrous exclusifs jusqu'au commit.
- Dans le 2PL rigoureux, les transactions gardent **tous** leurs verrous jusqu'au commit.

La différence : T_i , qui vient après T_j , peut écrire un objet A après que T_j l'ait lu et avant que T_j ne soit validée.

Alors que dans le 2ème cas, T_i ne peut pas modifier un objet accédé par T_j tant que celle-ci n'a pas validé.

Acquisition des verrous

Ce n'est généralement pas à l'utilisateur d'inclure dans son code les demandes de verrous.

Si une transaction T_i veut lire/écrire un objet D sans demander explicitement un verrou, alors l'algo suivant est utilisé

Pour l'opération de lecture :

Si T_i a un verrou sur D **Alors**

Lire(D)

Sinon

Tant que il y a un transaction avec un verrou

X sur D **Faire**

Attendre

FinTantQue

Fournir un verrou P à T_i sur D

Lire(D)

FinSi

Acquisition des verrous

Pour l'opération d'écriture :

Si T_i a un verrou X sur D **Alors**

Ecrire(D)

Sinon

Tant que il y a une transaction avec

un verrou sur D **Faire**

Attendre

FinTantQue

Si T_i a déjà un verrou P sur D **Alors**

Transformer P en X

Sinon

Fournir un verrou X sur D à T_i

Ecrire(D)

On ne dit pas quand est-ce que les verrous sont libérés !!

Prévenir vs guérir

1) Prévenir

Soit T_i qui demande un verrou en conflit avec celui déjà détenu par T_j

- Privilégier les plus anciennes transactions (estampillage) :
 - $ST(T_i) < ST(T_j) \Rightarrow T_i$ peut attendre T_j
 - $ST(T_i) > ST(T_j) \Rightarrow T_i$ est annulée
 - Si T_i est relancée avec une nouvelle estampille, alors elle risque d'attendre longtemps !! La relancer avec la même estampille.
 - Noter qu'ici, seules les transactions demandeuses sont annulées
- Privilégier la transaction qui détient le verrou
 - $ST(T_i) < ST(T_j) \Rightarrow T_j$ est annulée
 - $ST(T_i) > ST(T_j)$ alors T_i attend

Prévenir vs guérir

1) Guérir

Ici, il faut détecter le blocage. Le système maintient un graphe $G = (N, A)$ avec

- N = les transactions
- $T_i \rightarrow T_j$ ssi T_i attend que T_j libère un verrou
- Quand T_i demande un verrou détenu par T_j , alors l'arc $T_i \rightarrow T_j$ est rajouté au graphe
- $T_i \rightarrow T_j$ est supprimé quand T_j ne détient plus le verrou demandé par T_i
- Le système est bloqué ssi G contient un cycle
- Un algorithme est lancé périodiquement pour tester l'acyclicité

Prévenir vs guérir

Que faire si un blocage est détecté ?

Il faut annuler une transaction participant au cycle.

Choisir celle qui permet de réduire au maximum le nombre de cycles

Choisir celle qui est la moins proche de son état de validation

Choisir une transaction qui n'a pas été annulée plusieurs fois

Le problème

- **La suppression** d'un tuple ne peut se faire que si T_i détient un verrou X sur ce tuple
 - **L'insertion** d'un tuple par T_j implique la détention d'un verrou X par T_j sur ce tuple
- 1 T_i veut supprimer tous les comptes dont le solde est > 300 . Elle verrouille donc tous ces tuples.
 - 2 T_j insère un compte avec un solde > 400 . Noter qu'il n'y a pas conflit.
 - 3 Ensuite, T_i affiche les comptes dont le solde est > 200 . Le nouveau tuple est affiché.

Noter que cette exécution n'est pas équivalente à une exécution en série pourtant l'ordonnancement est c-sérialisable !!

Reprise sur Panne

Accès aux données

- Deux types de pages : *pages disque* des *pages mémoire*
- Les mouvements des pages entre disque et MC se font par l'appel des procédures : **Input(P)** et **Output(P)**
- Chaque transaction T_i possède un espace de travail privé. Elle manipule des “copies” des objets de la base. La copie de l'objet X manipulée par T_i sera noté x_i
- Les affectations entre X et x_i se font par les appels à **Lire(X)** et **Ecrire(X)**
- L'opération **Output(P_X)** n'est pas obligatoirement exécutée juste après l'appel à **Ecrire(X)**

Le problème

Lors d'une panne, certaines transactions doivent être *refaites* (celles qui ont fait le commit) et d'autres doivent être *annulées*. Supposons que le tampon puisse contenir 3 pages, les tuples t_1, t_2, t_3 et t_4 se trouvent dans 4 pages différentes

- 1 Soit T_0 qui modifie les 4 tuples. A un certain moment, une page doit être remplacée. Si T_i n'arrive pas à faire le commit, alors il faut annuler l'effet de ses modifications.
- 2 Soit T_1 qui modifie t_1 et t_2 puis fait son commit. Avant que les modif's de T_1 ne soient sur le disque, T_2 est lancée. T_2 modifie t_2, t_3 et t_4 puis elle échoue. Il faut dans ce cas, annuler T_2 et relancer T_1 car ses modif's ne sont pas reportées sur le disque

Reprise avec “Log”

- Nous supposons d'abord que les transactions sont exécutées en série (i.e pas de concurrence)
- Un fichier *Log* (ou fichier *Trace*) est maintenu pour garder la trace des différentes opérations exécutées sur la base
- Quand une transaction T_i débute, l'enregistrement $\langle T_i, \text{start} \rangle$ est inséré dans le *Log*
- Avant que T_i n'exécute **Ecrire(X)**, l'enregistrement $\langle T_i, X, V_1, V_2 \rangle$ est rajouté dans le *Log* avec V_1 et V_2 les valeurs avant et après l'écriture
- Quand T_i termine, l'enregistrement $\langle T_i, \text{commit} \rangle$ est inséré
- On suppose que les écritures dans le *Log* ne passent pas par un tampon mais se font directement sur le disque

Modes de réalisation des mises à jour

Nous allons considérer deux modes de réalisation des mises à jours

- ① **Différé** : Ce n'est qu'après l'exécution du commit, que les nouvelles valeurs sont effectivement réalisées.
- ② **Immédiat** : Les écritures sont traduites en des affectations
$$X \leftarrow x_i$$

- Toutes les modifications sont retranscrites dans le *Log*
- T_i commence par écrire $\langle Start \ T_i \rangle$
- L'opération **Ecrire(X)** résulte en une insertion de $\langle T_i, X, V \rangle$, où V est la nouvelle valeur. Jusque là, X n'est pas modifié.
- Quand T_i termine, $\langle T_i, Commit \rangle$ est inséré
- A la fin, le *Log* est utilisé pour réaliser les màj différées
- Remarque : Noter la forme des enregistrements $\langle T_i, X, V \rangle$. On n'a pas besoin de l'ancienne valeur car elle est dans X qui n'est pas modifié.

Màj différées (Suite)

- Durant la reprise, suite à un crash, une transaction doit être *refaite* ssi $\langle T; Start \rangle$ et $\langle T; Commit \rangle$ sont tous deux présents dans le *Log*
- Dans ce cas, l'opération **Redo(T_i)** met à jour les X en utilisant le *Log*
- Exemple :

| T_0 | T_1 |
|------------------|------------------|
| Lire(A) | Lire(C) |
| $A := A - 50$ | $C := C - 100$ |
| Ecrire(A) | Ecrire(C) |
| Lire(B) | |
| $B := B + 50$ | |
| Ecrire(B) | |

Màj différées (Suite)

- Ci-dessous, la configuration du *Log* est représentée à trois instants différents. On suppose qu'au départ $A = 1000$, $B = 2000$ et $C = 700$.

$\langle T_0 start \rangle$

$\langle T_0, A, 950 \rangle$

$\langle T_0, B, 2050 \rangle$

$\langle T_0 start \rangle$

$\langle T_0, A, 950 \rangle$

$\langle T_0, B, 2050 \rangle$

$\langle T_0 commit \rangle$

$\langle T_1 start \rangle$

$\langle T_1, C, 600 \rangle$

$\langle T_0 start \rangle$

$\langle T_0, A, 950 \rangle$

$\langle T_0, B, 2050 \rangle$

$\langle T_0 commit \rangle$

$\langle T_1 start \rangle$

$\langle T_1, C, 600 \rangle$

$\langle T_1 commit \rangle$

(a)

(b)

(c)

- Si une panne a lieu à l'un de ses instants alors
 - (a) Pas de *redo* à faire
 - (b) **redo(T_0)** doit être exécuté
 - (c) **redo(T_0)** et **redo(T_1)** doivent être exécutés dans cet ordre

- Ici, nous avons en plus besoin de l'opération **undo(T_i)** qui permet d'annuler l'effet d'une écriture non validée. Les enregistrements du *Log* doivent donc contenir l'ancienne valeur
- Se rappeler que l'opération **output(P)** peut avoir lieu après **ou** avant le commit
- Noter aussi que l'ordre de l'output des pages peut être différent de celui dans lequel ils sont modifiés (cela dépend de la technique de gestion des tampons)

Exemple de M^àj différées

| Log | Ecrire | Output |
|--------------------------------------|------------|------------|
| $\langle T_0 start \rangle$ | | |
| $\langle T_0, A, 1000, 950 \rangle$ | | |
| $\langle T_0, B, 2000, 2050 \rangle$ | | |
| | $A = 950$ | |
| | $B = 2050$ | |
| $\langle T_0 commit \rangle$ | | |
| $\langle T_1 start \rangle$ | | |
| $\langle T_1, C, 700, 600 \rangle$ | | |
| | $C = 600$ | |
| $\langle T_1 commit \rangle$ | | P_B, P_C |
| | | P_A |

- **undo(T_i)** restaure la valeur des objets modifiés par T_i . Ceci est réalisé en parcourant le *Log* en commençant par la fin.
- Durant la reprise suite à une panne,
 - T_i doit être annulée (i.e **undo(T_i)** exécutée) si le *Log* contient $\langle T_i;start \rangle$ mais pas $\langle T_i;commit \rangle$
 - T_i doit être refaite (i.e **redo(T_i)** est exécutée) si le *Log* contient les deux enregistrements
- Les opérations **undo** sont exécutées avant les opérations de **redo**

Exemple de reprise

| | | |
|---------------------------------|--------------------------------------|--------------------------------------|
| $\langle T_0start \rangle$ | $\langle T_0start \rangle$ | $\langle T_0start \rangle$ |
| $\langle A, 1000, 950 \rangle$ | $\langle T_0, A, 1000, 950 \rangle$ | $\langle T_0, A, 1000, 950 \rangle$ |
| $\langle B, 2000, 2050 \rangle$ | $\langle T_0, B, 2000, 2050 \rangle$ | $\langle T_0, B, 2000, 2050 \rangle$ |
| | $\langle T_0commit \rangle$ | $\langle T_0commit \rangle$ |
| | $\langle T_1start \rangle$ | $\langle T_1start \rangle$ |
| | $\langle T_1, C, 700, 600 \rangle$ | $\langle T_1, C, 700, 600 \rangle$ |
| | | $\langle T_1commit \rangle$ |

(a)

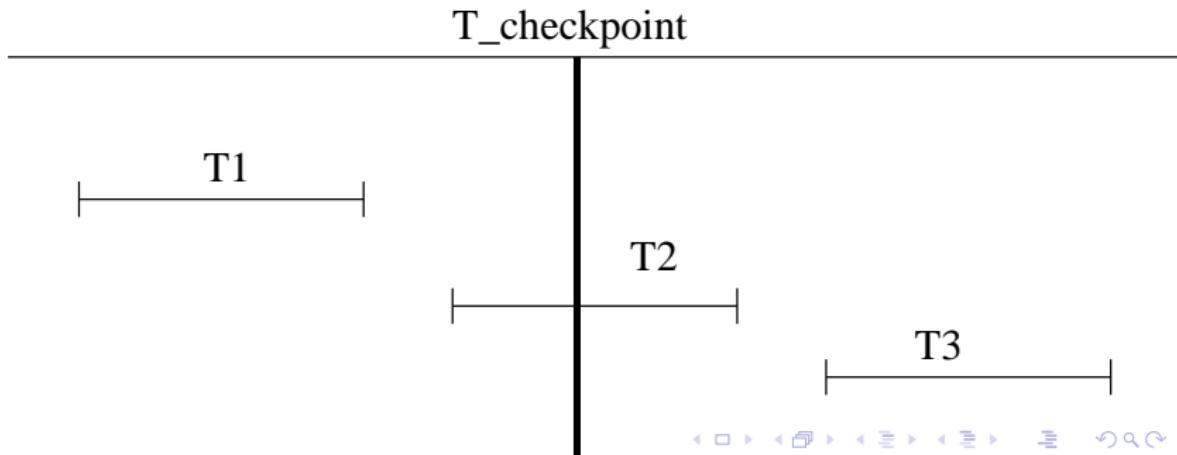
(b)

(c)

- (a) **undo(T_0)** : B reprend la valeur 2000 et A la valeur 1000
- (b) **undo(T_1)** et **redo(T_0)** : C reprend la valeur 700, ensuite A et B leur sont affectées resp. 950 et 2050.
- (c) **redo(T_0)** et **redo(T_1)** : A et B sont mis à 950 et 2050 resp.
Ensuite C est mis à 600

Points de reprise (Checkpoint)

- Il n'est pas pratique de parcourir tout le *Log* durant la reprise.
- Périodiquement, on "output" toutes les pages sur le disque, et suite à ça, un enregistrement *(checkpoint)* est inséré dans le *Log*
- Durant la reprise, nous considérons seulement la transaction T_i la plus récente qui a débuté avant le checkpoint, ainsi celles qui lui succèdent
-



- Nous supposons le protocole verrouillage en deux phases strict (i.e les m^àj des T_i non encore validées sont invisibles aux autres T_j)
- Nous supposons aussi qu'une page peut être modifiée par plusieurs transactions
- Les insertions dans le *Log* restent inchangées
- Le checkpoint est modifié :
L'enregistrement $\langle \text{checkpoint } L \rangle$ est inséré où L représente la liste des T_i actives lors du checkpoint

En présence de la concurrence (suite)

La reprise se fait comme suit :

- 1 Initialiser les listes *undo-liste* et *redo-liste* à vide
- 2 Parcourir le *Log* en commençant par la fin jusqu'à la rencontre de $\langle checkpointL \rangle$. Pour chaque enregistrement rencontré durant le parcours
 - si c'est $\langle T; commit \rangle$, alors rajouter T ; à *redo-liste*
 - si c'est $\langle T; start \rangle$ et si T ; n'est pas dans la *redo-liste*, la rajouter dans *undo-liste*
- 3 Pour chaque T ; dans L , si T ; n'est pas dans *redo-liste*, alors la rajouter dans *undo-liste*

En présence de la concurrence (suite)

A ce niveau, *undo-liste* contient les T_i incomplètes à annuler, et *redo-liste* contient les T_i validées donc à refaire.

- 4 Parcourir le *Log* de la fin et s'arrêter quand tous les $\langle startT_i \rangle$ des T_i dans *undo-liste* ont été rencontrés.
Durant ce parcours, annuler toute opération de $T_i \in \text{undo-liste}$
- 5 Se positionner au niveau du plus récent $\langle checkpointL \rangle$
- 6 Parcourir le *Log* jusqu'à la fin, et refaire chaque opération de $T_i \in \text{redo-liste}$

Exemple

$\langle T_0 start \rangle$

$\langle T_0, A, 0, 10 \rangle$

$\langle T_0 commit \rangle$

$\langle T_1 start \rangle$

l'étape 4 s'arrête là

$\langle T_1, B, 0, 10 \rangle$

$\langle T_2 start \rangle$

$\langle T_2, C, 0, 10 \rangle$

$\langle T_2, C, 10, 20 \rangle$

$\langle checkpoint \{ T_1, T_2 \} \rangle$

$\langle T_3 start \rangle$

$\langle T_3, A, 10, 20 \rangle$

$\langle T_3, D, 0, 10 \rangle$

$\langle T_3 commit \rangle$