# 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)

# Propriétés

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 $T_i, T_j$, pour $T_i$ tout se passe comme si $T_j$ s'est terminée avant le début de $T_i$ ou bien qu'elle commence après la fin de $T_i$ (les résultats intermédiaires de $T_j$ lui sont invisibles)

**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 Eur 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)

## Exemple (suite)

- 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

# Etats d'une transaction

- **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

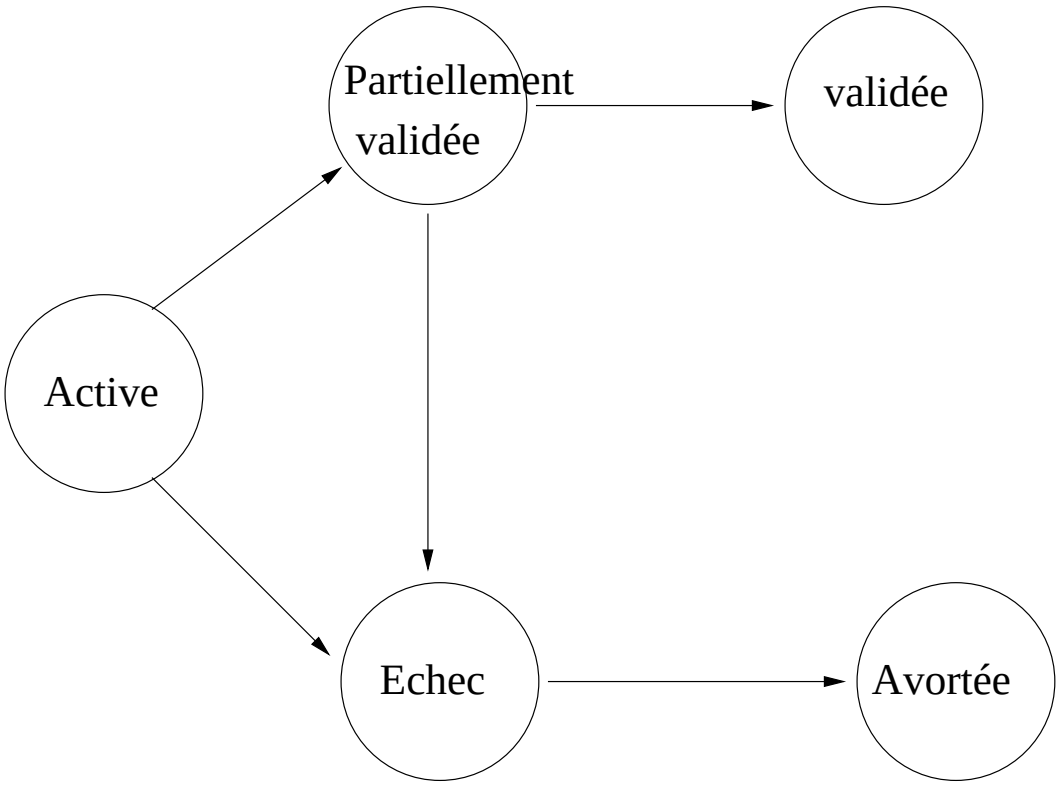
- **Avortée :** Après que toutes les modifications faites par la transaction soient annulées (Roll back). Deux options

  - Réexécuter la transaction

  - Tuer la transaction

- **Validée :** après l'exécution avec succès de la dernière opération

# Etats d'une transaction



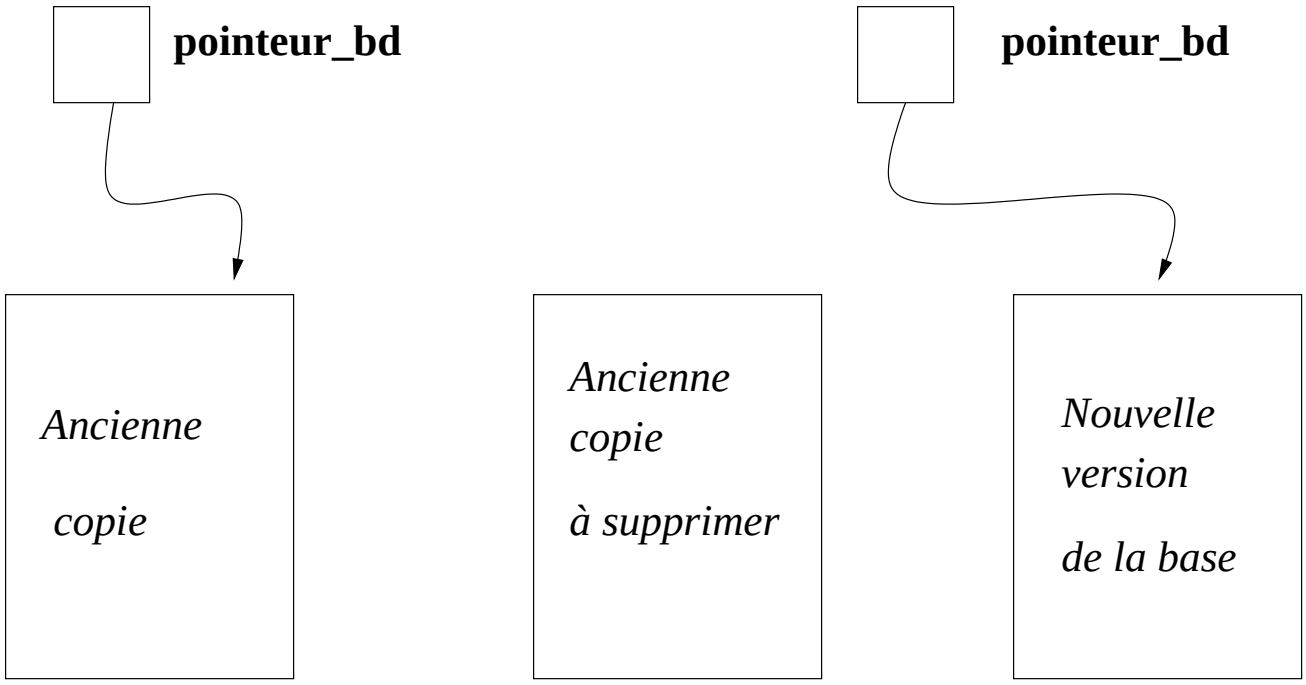
# Implémentation de l'atomicité

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

b) après la màj

inefficace si la base est volumineuse.

# Exécutions concurrentes

- Plusieurs transactions peuvent être exécutées en concurrence
  - une meilleur 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.

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

# Sérialisabilité

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.

# Remarque sur la sérialisabilité

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

$O$ est sérialisable si son exécution donne le même résultat que l'exécution de $T_1, \ldots, T_n$ 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 $n!$).

# c-sérialisabilité

- 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 $t_i = Lire(Q)$ et $t_j = Lire(Q)$ 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

| $T_3$ | $T_4$ |
|---|---|
| $Lire(Q)$ | |
| | $Ecrire(Q)$ |
| $Ecrire(Q)$ | |

# 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

| $T_1$ | $T$ |
|---|---|
| $Lire(A)$ | |
| $A := A - 1000$ | |
| $Ecrire(A)$ | |
| | $Lire(B)$ |
| | $B := B - 10$ |
| | $Ecrire(B)$ |
| $Lire(B)$ | |
| $B := B + 1000$ | |
| $Ecrire(B)$ | |
| | $Lire(A)$ |
| | $A := A + 10$ |
| | $Ecrire(A)$ |

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

# Reprise sur panne. Récupérabilité

- $O$ est récupérable si suite à l'annulation d'une transaction, on peut toujours revenir à un état cohérent sans défaire les transactions validés.

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

| $T_{10}$ | $T_{11}$ | $T_{12}$ |
|---|---|---|
| $Lire(A)$ | | |
| $Lire(B)$ | | |
| $Ecrire(A)$ | | |
| | $Lire(A)$ | |
| | $Ecrire(A)$ | |
| | | $Lire(A)$ |
| $fin$ | | |
| | $fin$ | |
| | | $fin$ |

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

# Implémentation de l'isolation

- 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

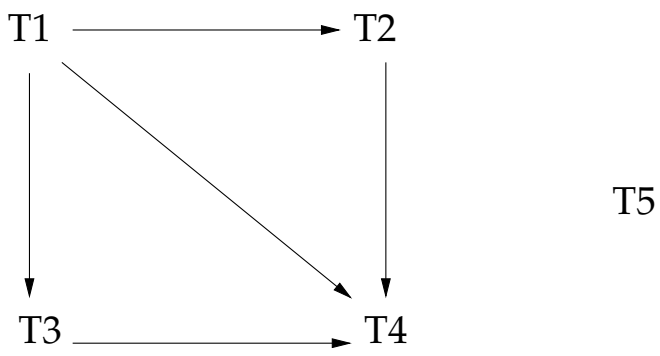
Considérer un ordonnancement $O$ des transactions $T_1, \ldots T_n$. Le graphe de précédence de $O$ est un graphe $(N, A)$ où

- $N$ 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.

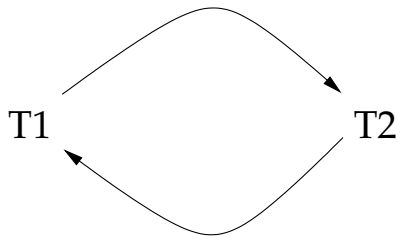
# Exemple de test

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



# Exemple de test

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



# conclusion

- 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.

# Contrôle de concurrence

- 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

| | $P$ | $X$ |
|---|---|---|
| $P$ | $oui$ | $non$ |
| $X$ | $non$ | $non$ |

  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)

$$
\begin{array}{ll}
T_1 : & Lock\_P(A) \\
& Lire(A) \\
& Unlock(A) \\
& Lock\_P(B) \\
& Lire(B) \\
& Unlock(B) \\
& Affiche r(A + B)
\end{array}
$$

- 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 à deux phases

*verrouillage 2PL (two phase locking)*

- 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"

# Extension du 2PL

- 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 $T_i$ 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 !!*

# Protocole avec estampillage

*Le but est d'avoir des ordonnancements sérialisables équivalents à l'ordre chronologique des transactions*

- A chaque transaction est associée une estampille relative au moment où elle "arrive", i.e $T_i$ avant $T_j \Leftrightarrow ST(T_i) < ST(T_j)$ (l'heure système ou bien un simple compteur)

- A chaque objet $D$ de la base sont associées 2 estampilles

  1. $E\_ST(D)$ : la plus grande des estampilles des transactions qui ont écrit $D$ avec succès

  2. $L\_ST(D)$ : la plus grande des estampilles des transactions qui ont lu $D$ avec succès

# Protocole avec estampillage

Supposons que $T_i$ veuille lire $D$

- Si $ST(T_i) \geq E\_ST(D)$, alors la lecture est autorisée et

$$L\_ST(D) \longleftarrow \max\{L\_ST(D), ST(T_i)\}$$

- Si $ST(T_i) < E\_ST(D)$, alors $T_i$ est "annulée" et relancée avec une nouvelle estampille (car cela veut dire que $T_i$ essaye de lire une valeur qui a été écrite par une transaction qui est arrivée après elle)

# Protocole avec estampillage

Supposons que $T_i$ veuille écrire $D$

- Si $TS(T_i) < L\_ST(D)$ : dans ce cas, cela veut dire qu'il y a une transaction qui est arrivée après $T_i$ et qui a lu $D$. $T_i$ est annulée

- Si $TS(T_i) < E\_ST(D)$ : si on laisse faire cette écriture, alors elle va "écraser" celle faite par une transaction arrivée après $T_i$. $T_i$ est annulée

- Sinon,
  - l'écriture est réalisée
  - $E\_ST(D) \longleftarrow ST(T_i)$

# Protocole avec estampillage

Soient les transactions $T_1, T_2, T_3, T_4$ et $T_5$ avec les estampilles resp. 1, 2, 3, 4 et 5.

L'ordonnancement suivant représente une situation où $T_2$ et $T_4$ sont annulées

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|
| | | | | $Lire(X)$ |
| | $Lire(Y)$ | | | |
| $Lire(Y)$ | | | | |
| | | $Lire(Y)$ | | |
| | | $Ecrire(Y)$ | | |
| | | $Lire(Z)$ | | |
| | | $Ecrire(Z)$ | | |
| | | | | $Lire(Z)$ |
| | $Lire(Z)$ | | | |
| | $abort$ | | | |
| $Lire(X)$ | | | | |
| | | | $Ecrire(Z)$ | |
| | | | $abort$ | |
| | | | | $Ecrire(X)$ |
| | | | | $Ecrire(Z)$ |

# Protocole avec estampillage

- L'estampillage permet d'éviter les blocages (les transactions sont exécutées ou bien annulées)

- Il garantit la c-sérialisabilité puisque tous les arcs sont de la forme $T_i \longrightarrow T_j$ avec $ST(T_i) < ST(T_j)$

- Par contre, le problème de récupérabilité persiste. Si $T_i$ est annulée alors que $T_j$ a lu une valeur écrite par $T_i$, alors $T_j$ doit aussi être annulée. Si $T_j$ a déjà validé, alors l'ordonnancement n'est pas récupérable.

# Protocole basé sur la validation

L'exécution d'une transaction $T_i$ se fait en 3 phases :

1. **Lecture :** Les écritures se font sur des "variables locales"

2. **Validation :** Tester la validité des écritures (ne violent elles pas la sérialisabilité ?)

3. **Ecriture :** Si la validation réussit, alors les écritures sont retranscrites sur la base

# Protocole basé sur la validation

Pour faire le test de validité, nous avons besoin de savoir à quels moments ont lieu les différentes phases. D'où l'utilisation d'estampilles :

1. **Début($\mathbf{T_i}$)**: Le moment où $T_i$ a débuté

2. **Validation($\mathbf{T_i}$)**: Le moment où $T_i$ a terminé sa phase précédente

3. **Fin($\mathbf{T_i}$)**: Le moment où $T_i$ termine la phase d'écriture

# Protocole basé sur la validation

La sérialisabilité est testée en se basant sur un estampillage des transactions correspondant à leur estampille de validation, i.e $ST(T_i) = Validation(T_i)$.

La validation de $T_j$ réussit si pour chaque $T_i$ telle que $ST(T_i) < ST(T_j)$

- $Fin(T_i) < Début(T_j)$, ou bien
- $Début(T_j) < Fin(T_i) < validation(T_j)$ et l'ensemble des données écrites par $T_i$ est distinct de celui des données lues par $T_j$

Pour le 2ème point:

- Les écritures de $T_j$ n'ont pas d'effet sur les lectures de $T_i$ (elles ont lieu après la fin des lectures de $T_i$)
- les écritures de $T_i$ n'ont pas d'effet sur les lectures de $T_j$ ($T_j$ ne lit aucun objet écrit par $T_i$)

# Exemple d'ordonnancement

| $T_1$ | $T_2$ |
|---|---|
| $Lire(B)$ | |
| | $Lire(B)$ |
| | $B := B - 50$ |
| | $Lire(A)$ |
| | $A := A + 50$ |
| $Lire(A)$ | |
| $Affiche(A + B)$ | |
| $< valider >$ | |
| | $Ecrire(B)$ |
| | $Ecrire(A)$ |
| | $< valider >$ |

**Note:** La méthode de validation permet d'éviter les cascades de Roll back puisque les lectures ne se font que sur des données persistantes.

# Granularité du verrouillage

Nous considérons la hiérarchie suivante :

$$Base$$
$$\downarrow$$
$$Table$$
$$\downarrow$$
$$Page$$
$$\downarrow$$
$$Tuple$$

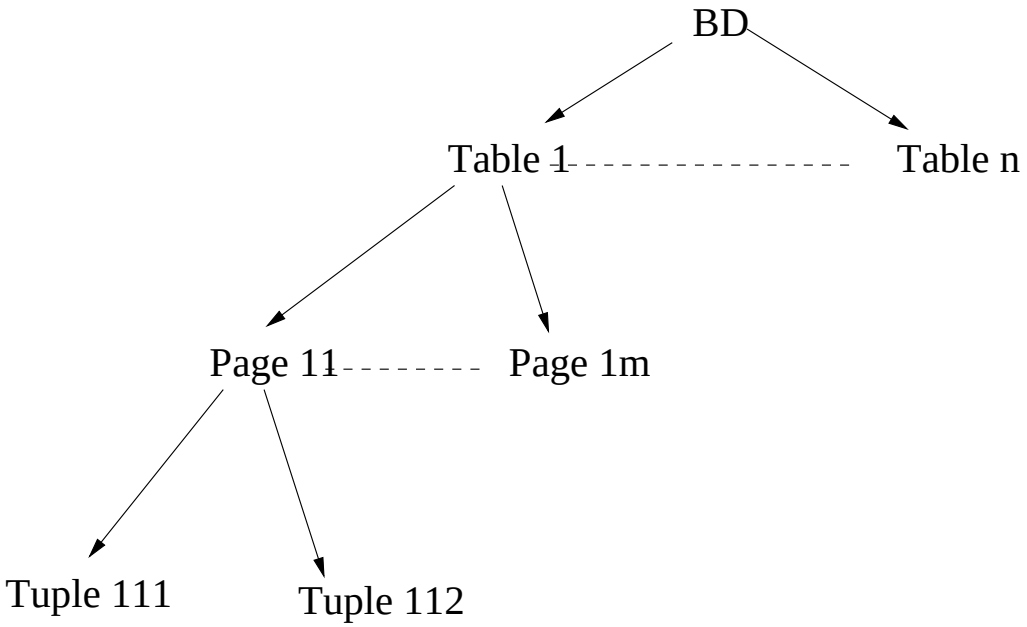
Permettre aux transactions de verrouiller n'importe quel niveau mais en respectant un nouveau protocole

Avant de verrouiller un objet $D$, $T_i$ doit avoir un <u>verrou intentionnel</u> sur tous les ancêtres de $D$

| | $IP$ | $IX$ | $P$ | $X$ | $PIX$ |
|---|---|---|---|---|---|
| $IP$ | $oui$ | $oui$ | $oui$ | | $oui$ |
| $IX$ | $oui$ | $oui$ | | | |
| $P$ | $oui$ | | $oui$ | | |
| $X$ | | | | | |
| $PIX$ | $oui$ | | | | |

# Granularité du verrouillage

- Nous sommes en présence d'une structure d'arbre de la forme




- Un noeud $D$ peut être verrouillé en $P$ ou $IP$ si le parent est déjà verrouillé en $IX$ ou $IS$

- $D$ peut être verrouillé en $X$, $PIX$ ou $X$ si le parent de $d$ est verrouillé en $IX$ ou $PIX$

- $T_i$ peut libérer un verrou sur $D$ si elle ne possède plus de verrous sur les descendants de $D$

- Là aussi, on utilise le verrouillage en deux phases

# Granularité du verrouillage

**Exemples :**

$T_1$ parcourt $R$ et met à jour quelques tuples :

- $T_1$ obtient un verrou $PIX$ sur $R$, à chaque lecture d'un tuple, elle pose d'abord un verrou $P$, et si elle a besoin de le modifier, elle transforme le $P$ en $X$

$T_2$ utilise un index pour lire une partie de $R$

- $T_2$ pose un verrou $IP$ sur $R$, ensuite elle obtient des verrous $P$ pour chaque tuple qu'elle veut lire

$T_3$ lit la table $R$ en entier

- $T_3$ demande un verrou $S$ sur $R$ ou bien

# Gestion des blocages

## 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 annulés

- Privilégier la transaction qui détient le verrou
  - $ST(T_i) < ST(T_j) \Rightarrow T_i$ est annulée
  - $ST(T_i) > ST(T_j)$ alors $T_i$ attend

# Gestion des blocages

## Prévenir vs guérir

1) Guérir

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

- $N =$ les transactions
- $T_i \longrightarrow T_j$ ssi $T_i$ attend que $T_j$ libère un verrou
- Quand $T_i$ demande un verrou détenu par $T_j$, alors l'arc $T_i \longrightarrow T_j$ est rajouté au graphe
- $T_i \longrightarrow T_j$ 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

# Gestion des blocages

**Prévenir vs guérir**

**Que faire si un blocage est détecté ?**

Il faut annuler une transaction participant au cycle.

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!!

# Les transactions et SQL

Une transaction commence automatiquement à l'exécution d'une commande SQL **SELECT, INSERT, DELETE, UPDATE, CREATE TABLE ...**

Une transaction se termine par l'exécution de **COMMIT** ou **ROLLBACK**. Toutes les commandes exécutées entre le début et la fin font partie de la transaction

Chaque transaction est caractérisée par : *un mode d'accès* et *un niveau d'isolation*

Il y a deux modes d'accès: `READ ONLY` et `READ WRITE`. C'est à l'utilisateur de le préciser au début de la transaction.

# Les transactions et SQL (1)

Le niveau d'isolation exprime le "degrés" d'exposition d'une transaction aux effets des autres transactions.

Il y a 4 niveaux

1. `READ UNCOMMITED`
2. `READ COMMITED`
3. `REPEATABLE READ`
4. `SERIALISABLE`

# Les transactions et SQL(2)

- `SERIALIZABLE`: $T_i$ obtient les verrous sur les objets qu'elle accède et les garde jusqu'à son commit (2PL rigoureux). Les autres transactions ne peuvent faire d'insertion ou de modification car $T_i$ verrouille soit (i) toute la table soit (2) utilise un verrouillage d'index

- `REPEATABLE READ`: Utilise le 2PL rigoureux mais ne verrouille que des objets individuels (pas d'index ni collection d'objets)

- `READ COMMITTED`: Les verrous X sur les objets sont gardés jusqu'à la fin. Les verrous P sont libérés immédiatement après la lecture.

- `READ UNCOMMITTED`: $T_i$ n'a pas le droit de faire des écritures et est autorisée à faire des lectures sans avoir à demander de verrous P

# Les transactions et SQL(3)

- Une *lecture impropre* (dirty read) a lieu quand $T_i$ lit un objet $A$ modifié par $T_j$ alors que $T_j$ n'a pas encore validé

- Une *lecture non répétable* (unrepeatable read) a lieu quand $T_i$ lit $A$ puis $T_j$ le modifie ensuite $T_i$ relit $A$.

| Niveau | dirty read | unrep. read | fantôme |
|---|---|---|---|
| `READ UNCOMMITTED` | possible | possible | possible |
| `READ COMMITTED` | Non | possible | possible |
| `REPEATABLE READ` | Non | Non | possible |
| `SERIALISABLE` | Non | Non | Non |

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
```

# Exemple de SGBD : Oracle 8

Oracle 8 utilise un mécanisme utilisant le multiversionnement: Chaque objet peut avoir plusieurs versions. Si $T_i$ veut lire $A$ alors le système lui fournit la version de $A$ dont le $E\_ST(A)$ est le max avec $E\_ST(A) < ST(T_i)$.

Si $T_i$ veut écrire $A$ et $ST(T_i) \geq L\_ST(A)$, alors une nouvelle version de $A$ est créée avec $L\_ST(A) = ST(T_i)$.

# 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 $\mathbf{Output(P_X)}$ 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 exécutée sur la base

- Quand une transaction $T_i$ débute, l'enregistrement $\langle T_i, start \rangle$ 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 $\langle T_i commit \rangle$ 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 $X \longleftarrow x_i$

# Màj différées

- 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 $\langle T_i Start \rangle$ **et** $\langle T_i Commit \rangle$ sont tous deux présents dans le *Log*

- Dans ce cas, l'opération $\mathbf{Redo(T_i)}$ met à jour les $X$ en utilisant le *Log*

- Exemple :

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

# 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 $C = 700$.

| $(a)$ | $(b)$ | $(c)$ |
|---|---|---|
| $\langle T_0 start \rangle$ | $\langle T_0 start \rangle$ | $\langle T_0 start \rangle$ |
| $\langle T_0, A, 950 \rangle$ | $\langle T_0, A, 950 \rangle$ | $\langle T_0, A, 950 \rangle$ |
| $\langle T_0, B, 2050 \rangle$ | $\langle T_0, B, 2050 \rangle$ | $\langle T_0, B, 2050 \rangle$ |
| | $\langle T_0 commit \rangle$ | $\langle T_0 commit \rangle$ |
| | $\langle T_1 start \rangle$ | $\langle T_1 start \rangle$ |
| | $\langle T_1, C, 600 \rangle$ | $\langle T_1, C, 600 \rangle$ |
| | | $\langle T_1 commit \rangle$ |

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

# Màj immédiates

- Ici, nous avons en plus besoin de l'opération $\mathbf{undo(T_i)}$ 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 $\mathbf{output(P)}$ 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 Màj différées

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

# Màj immédiates (suite)

- $\mathbf{undo(T_i)}$ 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 $\mathbf{undo(T_i)}$ exécutée) si le *Log* contient $\langle T_i start \rangle$ mais pas $\langle T_i commit \rangle$
  - $T_i$ doit être refaite (i.e $\mathbf{redo(T_i)}$ est exécutée) si le *Log* contient les deux enregistrements

- Les opérations $\mathbf{undo}$ sont exécutées avant les opération de $\mathbf{redo}$

# Exemple de reprise

| | | |
|---|---|---|
| $T_0 start\rangle$ | $\langle T_0 start\rangle$ | $\langle T_0 start\rangle$ |
| $T_0, A, 1000, 950\rangle$ | $\langle T_0, A, 1000, 950\rangle$ | $\langle T_0, A, 1000, 950\rangle$ |
| $T_0, B, 2000, 2050\rangle$ | $\langle T_0, B, 2000, 2050\rangle$ | $\langle T_0, B, 2000, 2050\rangle$ |
| | $\langle T_0 commit\rangle$ | $\langle T_0 commit\rangle$ |
| | $\langle T_1 start\rangle$ | $\langle T_1 start\rangle$ |
| | $\langle T_1, C, 700, 600\rangle$ | $\langle T_1, C, 700, 600\rangle$ |
| | | $\langle T_1 commit\rangle$ |
| $(a)$ | $(b)$ | $(c)$ |

(a) $\mathbf{undo(T_0)}$: $B$ reprend la valeur 2000 et $A$ la valeur 1000

(b) $\mathbf{undo(T_1)}$ et $\mathbf{redo(T_0)}$: $C$ reprend la valeur 700, ensuite $A$ et $B$ leur sont affectées resp. 950 et 2050.

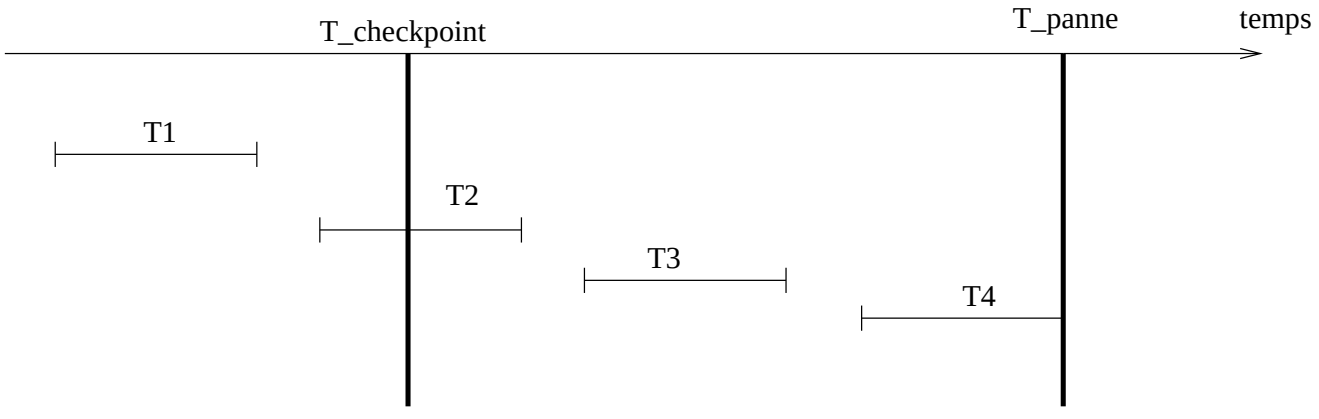
(c) $\mathbf{redo(T_0)}$ et $\mathbf{redo(T_1)}$: $A$ et $B$ sont mis à 950 et 2050 resp. Ensuite $C$ est mis à 600

# 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 $\langle checkpoint \rangle$ est inséré dans le *Log*

- Durant la reprise, nous considérons seulement seulement la transaction $T_i$ la plus récente qui a débuté avant le checkpoint, ainsi celles qui lui succèdent




- $T_1$ est ignorée, $T_2$ et $T_3$ sont refaites, $T_4$ est annulée.

# En présence de la concurrence

- Nous supposons le protocole verrouillage en deux phases strict (i.e les màj 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 $\langle checkpoint\ L \rangle$ 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 $\langle T_i commit \rangle$, alors rajouter $T_i$ à *redo-liste*
   - si c'est $\langle T_i start \rangle$ et si $T_i$ n'est pas dans la *redo-liste*, la rajouter dans *undo-liste*

3. Pour chaque $T_i$ dans $L$, si $T_i$ 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 $T_i \in$*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 $T_i \in$*redo-liste*

# Exemple

$\langle T_0 start \rangle$
$\langle T_0, A, 0, 10 \rangle$
$\langle T_0 commit \rangle$
$\langle T_1 start \rangle$
$\langle T_1, B, 0, 10 \rangle$
$\langle T_2 start \rangle$
$\langle T_2, C, 0, 10 \rangle$
$\langle T_2, C, 10, 20 \rangle$
$\langle checkpoint\{T_1, T_2\} \rangle$
$\langle T_3 start \rangle$
$\langle T_3, A, 10, 20 \rangle$
$\langle T_3, D, 0, 10 \rangle$
$\langle T_3 commit \rangle$

l'étape 4 s'arrête là