Bases de Données

Reprise sur panne et contrôle de concurrence

Fiabilité (reliability) d'un système sa capacité à surmonter les erreurs et anomalies qui y surviennent.

- Principales erreurs ou dysfonctionnements :
  - erreurs de programmation des transactions : locales
  - erreurs systèmes (défaillances) : globales
  - erreurs de médias
- => induisent le fonctionnement incorrect de la base ou sa destruction.

Objectifs des mécanismes qui assurent la fiabilité

- Eviter les incohérences de la base
- Rétablir la cohérence de la base suite à un problème
- Erreurs de médias (disques) : destruction de parties de la base => solution : avoir des copies d ’archives sur supports distincts et/ou BD dupliquées
- Erreurs locales (transactions) : résolues à l ’aide de contraintes d ’intégrité et/ou triggers, …
- **Erreurs systèmes : suite à une panne (principalement coupure de courant) ==>** perte du contenu des buffers de la MC (volatile), donc perte des infos des transactions actives.

Solution aux défaillances système

- Avoir des duplicatas des données (invisibles à l ’utilisateur)

  ==> le système utilise un journal (fichiers log)

- Pour restaurer la base dans un état cohérent

  ==> déroulement d *’une procédure de reprise*

- Un programme = une séquence de transactions

- Une transaction = une séquence d ’actions (lire, écrire, calculs en MC),

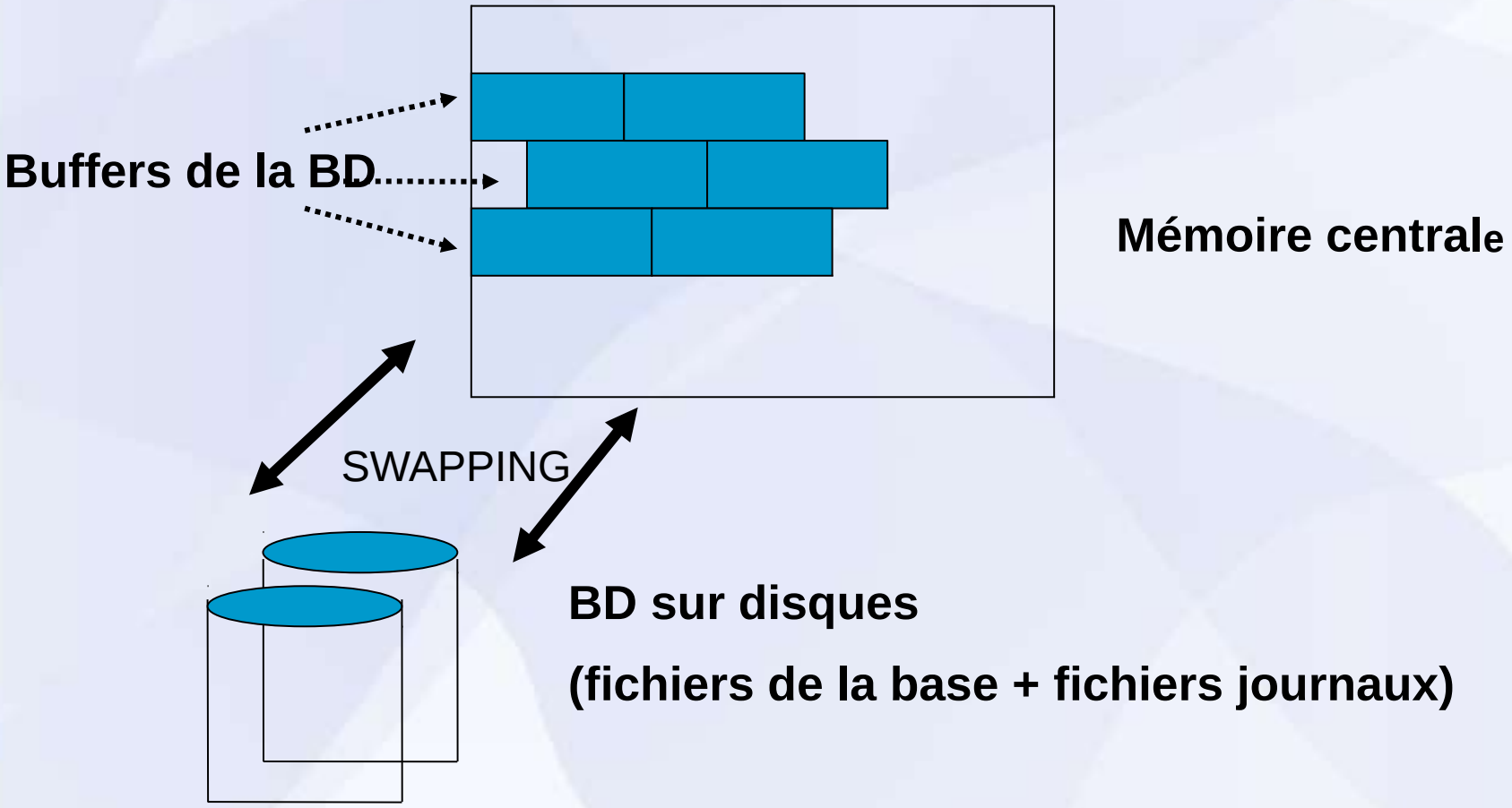
  mais pour la BD, les opérations importantes sont lire/écrire (on néglige le temps d'exécution)

- Une transaction :
  - début transaction (ou équivalent)
    - accès à la base (lire/écrire)
    - calculs en MC
  - fin transaction (COMMIT ou ROLLBACK)

- COMMIT : fin de la transaction avec succès (les modifications deviennent permanentes dans la base)

- ROLLBACK : fin de la transaction avec échec (la base revient à l'état où elle était avant le début de la transaction)

- Une BD = ens. de fichiers sur supports stables (disques)
- Le SGBD exploite la base en travaillant sur une partie de cette base en MC (dans des buffers)



Journalisation

- Écriture sur support stable du déroulement de l'activité des transactions => pour que le SGBD puisse respecter l'atomicité des transactions

- Il y a 3 types de journalisation :
  - UNDO (défaire)
  - REDO (refaire)
  - UNDO/REDO (défaire et refaire)

- **Mécanisme de reprise après panne (recovery) :** permet de remettre la BD dans un état cohérent suite à une panne (défaillance système => perte du contenu de la MC)

- Les journaux principaux sont localisés sur des supports stables (disques).

- En fonction du type de SGBD, suite à une panne, le processus de recouvrement peut consister à :

- annuler les actions effectuées par les transactions non terminées avec succès (UNDO), ou
- refaire les actions effectués par les transactions terminées avec succès (REDO) ou
- refaire certaines actions et en annuler d ’autres (UNDO/REDO)

Enregistrements dans le journal (log)

<start T> : début transaction

<Commit T> : validation de la transaction

<Abort T> : annulation de la transaction

<T, X, old_v, new_v> : la transaction T a mis à jour la donnée X avec la nouvelle valeur *new_v* (l 'ancienne valeur étant *old_v*)

La transaction = notion fondamentale dans un SGBD

= unité logique de travail.

Exemple de transaction :

```
Insert into SP (s : ’S5 ’, P : ’P1 ’, QTY: 1000);
If <erreur> Go To Then undo;
   Update SP set QTY := QTY + 500 Where P := ‘ P1 ’;
   If <erreur> GO TO Then undo;
         -- si erreur, on défait la tansation
         Commit transaction; - - si pas d'erreur, on valide
         Go To fin;
undo :    rollback transaction
fin :      return;
```

L ’idéal : toutes les actions d'une transaction s ’exécutent sans problèmes. *

Mais : défaillances systèmes, débordements, ….

=> le gestionnaire de transactions doit garantir que si certaines mises à jour ne sont pas exécutées, alors il est capable de restaurer un état cohérent de la base (ex. annuler les màj effectuées).

Donc soit la transaction s ’exécute entièrement, soit pas du tout

=> une séquence d ’opérations non atomique apparaît comme atomique au niveau externe.

Atomicité : fournie par le gestionnaire de transaction :

- avec COMMIT : il signale la fin d ’une transaction avec succès. Les màj effectuées deviennet permanentes.

- avec ROLLBACK : il signale la fin anormale de la transaction. La BD pourrait être dans un état incohérent et les màj effectuées doivent être défaites (annulées).

Comment annuler une màj ?

Grâce au fichier *journal* (log) localisé sur support stable, et géré par le système. Contient toutes les màj (valeurs avant et valeurs après modif, notamment).

Pour annuler une màj particulière, le système utilise l'entrée correspondante du journal pour retrouver la valeur avant màj.

Reprise d ’une transaction après panne

Une transaction commence par BEGIN TR (ou équivalent) et se termine :

- par COMMIT (si succès) ou
- ROLLBACK (si échec).

Le COMMIT établit un point de commit (syncpoint). Il correcspond à la terminaison d'une unité logique de travail, à un point où la BD est (ou devrait être) cohérente.

Le ROLLBACK remet la BD dans l'état où elle était avant le BEGIN TR (retour au point de commit de la transaction précédente).

Point de commit (syncpoint)

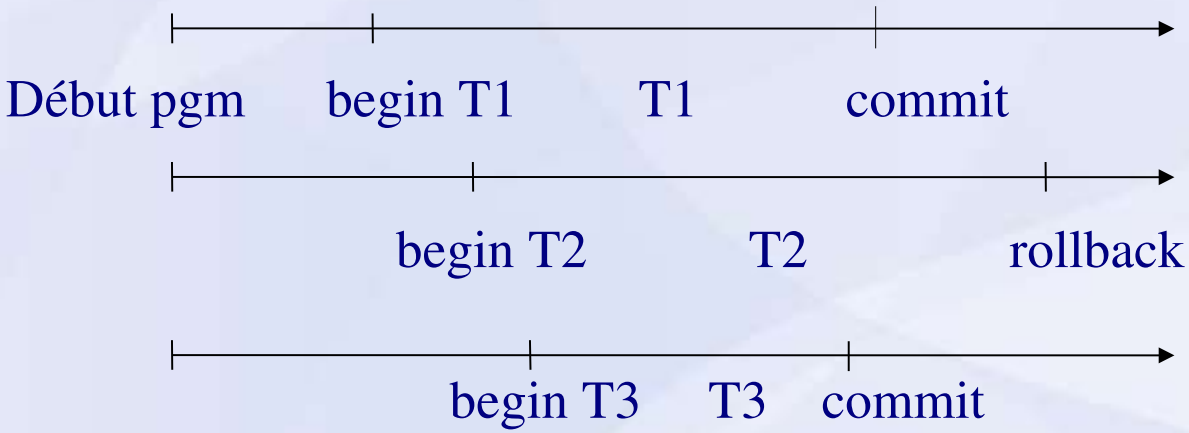
Lorsqu ’un **point de commit** est établi, on a :

- Toutes les màj terminées par COMMIT effectuées depuis le dernier point de commit sont validées (deviennent permanentes). Avant ce point, les màj sont assimilées à des tentatives, car elles pourraient être annulées.

- Toutes les variables de positionnement (varrous et références à des curseurs, par exemple) sont effacés.

Exécution d ’un programme

En général, un programme est une séquence de plusieurs transactions qui s ’exécutent les unes après les autres.



Remarque : Dans l ’exemple, des tests explicites sont inclus pour tenir compte des erreurs. **Mais** d ’autres erreurs non prévues peuvent survenir (panne de courant, …)

=> le système exécute des **ROLLBACK implicites** pour ces programmes.

La transaction est également une unité de reprise

Si la transaction se termine par COMMIT (succès) :

alors le système garantit que les màj deviennent permanentes, même si une défaillance survient après le COMMIT (Au COMMIT, les màj pourraient avoir été effectuées sur un tampon en MC,

et si une défaillance se produit juste à ce moment là, la procédure de redémarrage du système est capable de positionner ces màj en examinant les entrées du journal).

=> Il s ’ensuit que le contenu du journal doit être écrit physiquement sur support stable avant de terminer l ’opération de COMMIT.

Cette règle est appelée **règle WAL** (Write-Ahead Log rule) ou écriture en avant dans le journal.

Propriétés des transactions

Une transaction doit posséder les 4 propriétés suivantes :
Atomicité, **C**ohérence, **I**solation et **D**urabilité.

Atomicité: toutes les opérations d ’une transaction sont exécutées ou aucune ne l’est (principe du tout ou rien).

Cohérence : une transaction préserve la cohérence de la BD. Elle transforme un état cohérent de la BD en un autre état cohérent (si transaction “bien” programmée).

Isolation : une transaction s'exécute isolément des autres. Même si plusieurs transactions s'exécutent en concurrence, les màj de chacune ne deviennent visibles aux autres transactions qu'après son COMMIT.

Durabilité : une fois qu'une transaction est validée (COMMIT), ses màj deviennent permanentes dans la base, même si ensuite une panne survient.

Reprise du système

- Si défaillance globale, dite douce : pannes de courant, …
  => effacement de la MC.

Défaillances *système*

Effacement du contenu de la **MC.**

=> l'état précis des transactions en cours d 'exécution au moment de la défaillance est inconnu.

=> doivent être **annulées** (rollback).

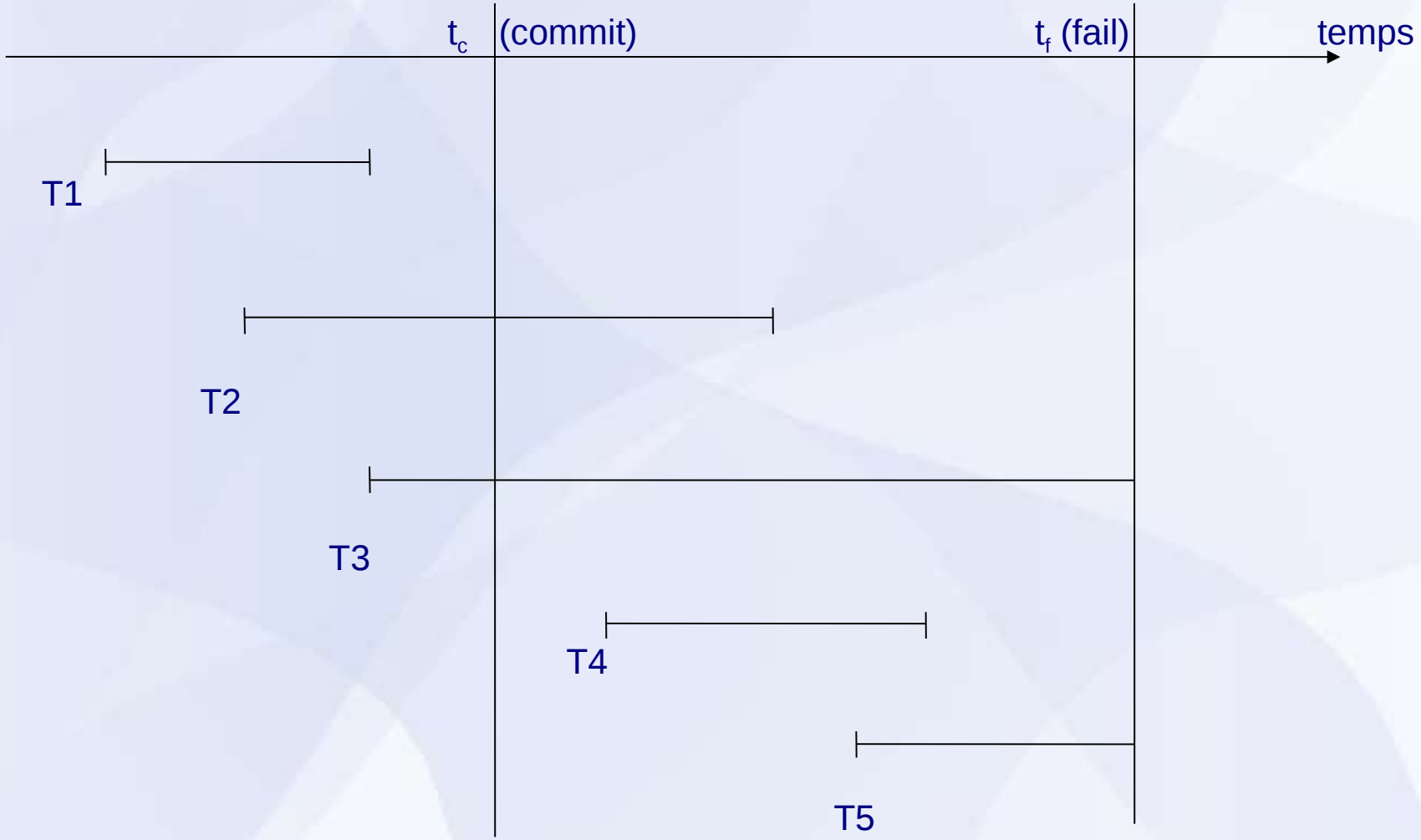
et nécessaire de **« rejouer »** certaines transactions lors du redémarrage du système. Ce sont celles qui ont effectué leur COMMIT avant la défaillance et dont les mises à jour n 'ont pas eu le temps d 'être inscrites dans la base.

Pour effectuer ce travail, le système utilise le contenu du **journal (log)**. A certains intervalles prédéfinis, le système positionne un **point de contrôle (checkpoint)** qui consiste :

- à écrire physiquement (écriture forcée) le contenu des tampons en MC sur les fichiers disque.

- à écrire physiquement le compte-rendu du point de contrôle dans le journal physique sur disque (log). Ce compte-rendu donne la liste de toutes les transactions qui étaient en cours d'exécution lors du positionnement du point de contrôle.

Illustration : tous les cas de transactions :



A l ’instant t_c, il y a pose d ’un point de contrôle.

A l ’instant t_f, une défaillance survient (fail).

Les transactions T1, …, T5 sont en cours d ’exécution.

On a :

- Les transactions de type T1 sont terminées avec succès (commit) avant t_c.
- Les transactions de type T2 ont débuté avant t_c, et se sont terminées avec succès (commit) après t_c et avant $t_{f.}$

- Les transactions de type T3 ont également débuté avant t_c, mais ne se sont pas terminées à la date t_f.
- Les transactions de type T4 ont débuté après t_c, et se sont terminées avec succès (commit) avant t_f.
- Les transactions de type T5 ont aussi débuté après t_c, mais ne se sont pas terminées à la date t_f.

Lorsque le système est redémarré,

- les transactions de types T3 et T5 doivent être annulées,
- les transactions de types T2 et T4 doivent être rejouées.
- lestransactions de type T1 n'interviennent pas car leurs màj ont déjà été transférées sur disque lors du chekpoint précédent (écriture forcée à la date t_c).

Par conséquent, lors du redémarrage du système, la procédure suivante est exécutée pour identifier les types de transactions T2, T3, T4 et T5 :

1- Commencer par tenir 2 listes UNDO et REDO.

- Initialiser la liste UNDO à toutes les transactions enregistrées dans le compte-rendu du point de contrôle le plus récent avant la panne.
- Initialiser la liste REDO à vide.

2- Faire une recherche en avant dans le journal en partant du point de contrôle :

2.1. Si un COMMIT T est rencontré, alors transférer la transaction T de la liste UNDO vers REDO.

2.2. A la fin du journal :

- la liste UNDO contiendra les transactions de types T3 et T5.
- La liste REDO contiendra les transactions de types T2 et T4.

Puis le système effectue un parcours en arrière du journal, annulant les transactions de la liste UNDO.

Ensuite, il effectue à nouveau un parcours en avant, rejouant les transactions de la liste REDO.

Enfin, lorsque l ’activité de reprise est terminée, le système devient prêt à accepter de nouveau travaux sur la BD.

Gestion des transactions

Contrôle de concurrence

- Plusieurs utilisateurs se connectent simultanément pour manipuler la BD (écriture, màj, suppression, ...)

=> risque d ’incohérence des données

+ chute des performances.

3 principaux problèmes lors de l ’exécution concurrente de transactions

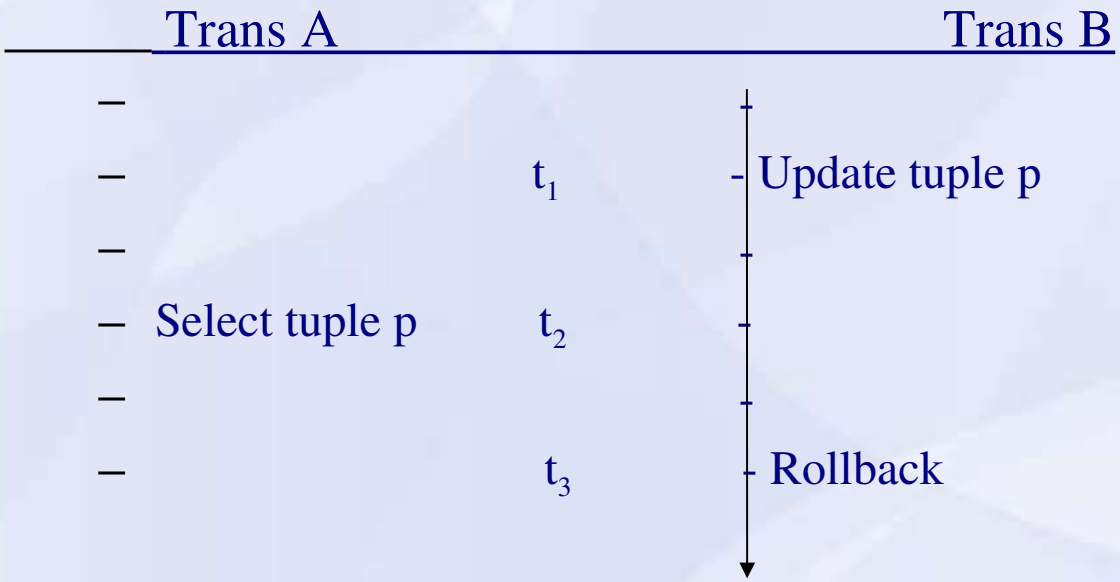
Problème de perte de mise à jour

- une transaction A récupère un tuple p à t_1; ensuite, la transaction B récupère le même tuple p à t_2; puis la transaction A met à jour le tuple p à t_3 (sur la base des valeurs de p lues à t_1); enfin, la transaction B met à jour le tuple p à t_4 (sur la base des valeurs de p lues à t_2, c-à-d identiques à celles lues à t_1).
- **=> la màj effectuée par A est perdue (écrasée par celle de B)**

Trans A		Trans B
–		-
– Select tuple p	t_1	-
–	t_2	- Select tuple p
– Update tuple p	t_3	-
–	t_4	- Update tuple p

La mise à jour effectuée par A à l 'instant t_3 est perdue à l 'instant t_4.

Problème des dépendances non validées (1)

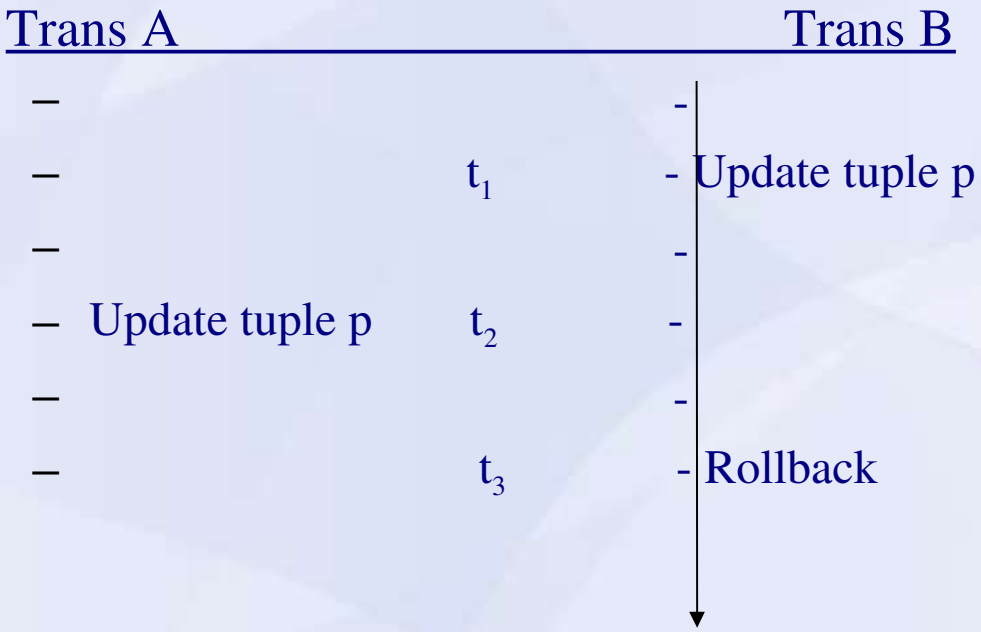


La transaction A devient dépendante d'une modification non validée de la transaction B, i.e., la transaction A a lu des valeurs de p à t_2 qui ne sont plus valides à l'instant t3 (car B a fait un Rollback)

=>lecture salissante - dirty read)

Problème des dépendances non validées (2)

Cas encore pire (car A fait une màj du tuple p à t2, qui n 'est plus valide à t_3)



La transaction A devient dépendante d'une modification non validée de la transaction B, i.e., la transaction A a mis à jour des valeurs de p à t_2 qui ne sont plus valides à l 'instant t3 (car B a fait un Rollback)

Problème de l ’analyse incohérente

Soient 3 comptes bancaires ACC1, ACC2 et ACC3, initialisés à :

ACC1 ← 40, ACC2 ← 50 et ACC3 ← 30

Nous avons 2 transactions A et B :

- A fait le cumul des comptes dans la variable sum, et
- B transfère la somme de 20 de ACC3 vers ACC1.

Voici une exécution possible de ces transactions en concurrence :

		Trans A	Trans B
ACC1:40 ACC2:50 ACC3:30			
	t_1	Select ACC1: sum = 40	
	t_2	Select ACC2: sum:=40+50=90	
	t_3		Select ACC3
	t_4		Update ACC3: ACC3 := ACC3-10
	t_5		Select ACC1
	t_6		Update ACC1: ACC1:=ACC1+10
	t_7		COMMIT
	t_8	Select ACC3: sum=90+20 = 110 au lieu de 120 !!	

Le résultat correct est : **ACC1:50 ACC2:50 ACC3:20**

Solution par Verrouillage

- Les pbs vus précédemment peuvent être résolue grâce à une technique de CC simple : **le verrouillage.**

- **Idée de base** : quand une transaction veut s 'assurer qu 'un objet (ex. un tuple) dont elle a besoin ne sera pas modifié de façon imprévisible, alors elle pose un verrou sur l 'objet.
- Le verrou a pour effet de verrouiller l 'accès à l 'objet par d 'autres transactions. Il les empêche en particulier de modifier l 'objet.
- La première transaction est ainsi sûre que l 'objet est stable aussi longtemps qu 'elle le souhaite.

Fonctionnement détaillé du verrouillage

- **Hypothèse : le** système gère 2 types de verrous. Les verrous exclusif (X locks), et les verrous partagés (S, Shared locks), parfois appelés respectivement verrous d ’écriture et verrous de lecture.
- On suppose que les seuls verrous disponibles sont X et S, et que le seul objet ‘ verrouillable ’ est le tuple.
-

Fonctionnement

- **1.** Si la transaction A détient un verrou exclusif X sur le tuple p, alors la demande d ’un verrou de n ’importe quel type (X ou S) sur p faite par une autre transaction sera refusée.

- **2.** Si la transaction A détient un verrou partagé S sur le tuple p, alors :
  - une demande d ’un verrou X sur p faite par une autre transaction sera refusée.
  - une demande d ’un verrou S sur p faite par une autre transaction B pourra être accordée (et la transaction B aura également posé un verrou S sur le tuple p)

Matrice de compatibilité des verrous X et S

B demande le verrou \ A détient le verrou de type	X	S	Pas de verrou
X	NON	NON	OUI
S	NON	OUI	OUI

Protocole d ’accès aux données (avec les verrous X et S)

- **1.** Une transaction qui souhaite lire un tuple doit d ’abord obtenir un verrou S sur ce tuple.

- **2.** Une transaction qui souhaite mettre à jour un tuple doit d ’abord obtenir un verrou X sur ce tuple.

  Si elle détient déjà un verrou S sur le tuple et qu ’elle désire le mettre à jour (séquence select/update), elle doit alors *promouvoir* le verrou S en X.

  Rmq : les demandes de verrous sont souvent implicites :

  Select (verrou S), update, delete (verrou X)

- **3.** Si une demande de verrou émise par B est refusée (car conflit avec un verrou détenu par A),

  alors B est mise en attente (jusqu ’à libération du verrou par A). Le système doit garantir que B n ’attende pas indéfiniment (éviter le verrouillage à vie, deadlock).

- **4.** Les verrous exclusifs, X, sont conservés jusqu’au COMMIT de la transaction (ou son ROLLBACK).

  En général, les verrous S également, mais pas toujours.

Eviter ou Résoudre le verrouillage à vie

- Ce qui est souhaitable : détection puis suppression du blocage.

- => Construire un graphe d'attente des transactions (qui attend la fin de qui). Puis :

  - **Détecter** : voir si le **graphe d 'attente** des transactions comporte un cycle.
  - **Supprimer** le blocage : choisir une transaction *victime* pour l 'annuler, et ensuite la redémarrer. On espère que l 'indéterminisme du système fera que la situation de blocage ne se renouvelle pas.

Remarque *certains système annulent et relancent une transaction qui reste inactive pendant un certain temps fixé : le time-out*

Résolution des 3 problèmes de concurrence (avec les verrous)

- **Pb de perte de màj :**

Trans A		Trans B
–		-
– Select tuple p	t_1	-
(obtention d'1 verrou S sur p)		
–	t_2	- Select tuple p
		(obtention d'1 verrou S sur p)
Update tuple p	t_3	-
(demande d'1 verrou X sur p)		
attente (car p est verrouillé en S par B)		
attente	t_4	- Update tuple p
attente		(demande d'1 verrou X sur p)
attente		*attente (car p est verrouillé en S par A)*
attente		*attente*

Pas de perte de màj, mais un interblocage survient !

Pb des dépendances non validées (1)

Trans A		Trans B
–		-
–	t_1	- Update tuple p (obtention d 'une verrou X sur p)
–		-
– Select tuple p (demande d 'un verrou S sur p) attente car p est verrouillé en X par B)	t_2	-
–		-
–	t_3	- Rollback (libération du verrou X sur p)
– resume : Select p (obtention d 'un verrou S sur p)	t_4	
–		

La transaction A ne "voit" pas, à l 'instant t_2 ,la modif non validée de B (car elle doit se mettre en attente)

Pb des dépendances non validées (2)

Trans A		Trans B
–		-
–	t_1	- Update tuple p (obtention d 'une verrou X sur p)
–		-
– Update tuple p (demande d 'un verrou X sur p) attente car p est verrouillé en X par B)	t_2	-
–		-
–	t_3	- Rollback (libération du verrou X sur p)
– resume : Update p	t_4	
– (obtention d 'un verrou X sur p)		

La transaction A ne met pas à jour p (modif non validée) à l 'instant t_2 (car elle se met en attente)

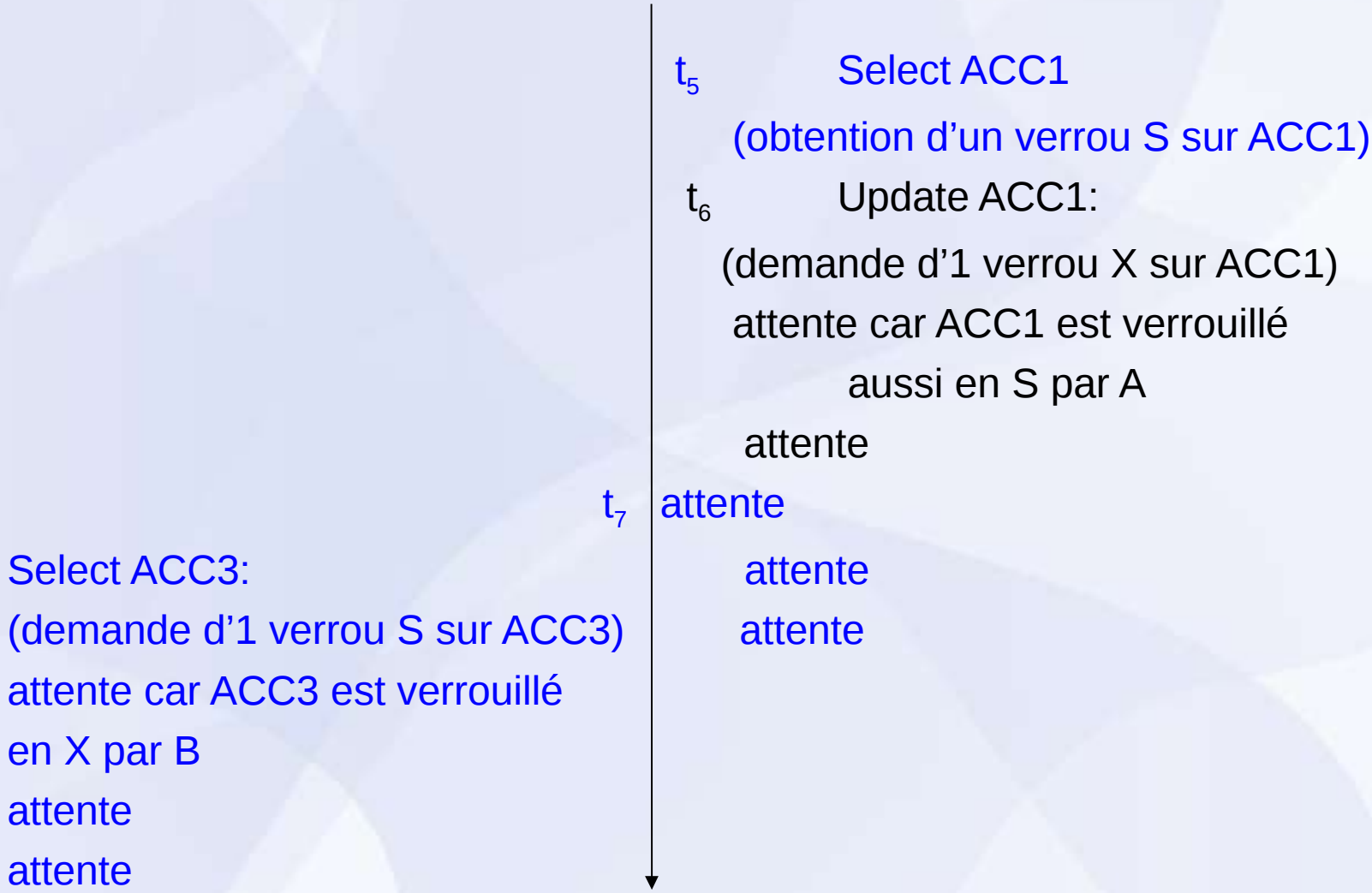
Rmq : A devient indépendant d 'une màj non validée (dans les 2 cas).

Problème de l'analyse incohérente (1)

ACC1:40 ACC2:50 ACC3:30

Trans A		Trans B
Select ACC1: sum = 40 (obtention d'1 verrou S sur ACC1)	t_1	
Select ACC2: sum:=40+50 (obtention d'1 verrou S sur ACC2)	t_2	
	t_3	Select ACC3 (obtention d'1 verrou S sur ACC3)
	t_4	Update ACC3: (obtention d'1 verrou X sur ACC3) ACC3 := ACC3-10

Problème de l ’analyse incohérente (2)



Rmq *:* L ’analyse incohérente est évitée, mais un blocage survient !

Notion de sérialisabilité

Il s'agit du **critère de correction** généralement accepté pour le contrôle de concurrence, i.e., **une exécution entrelacée de transactions est considérée comme correcte si elle est *sérialisable***.

Une exécution (entrelacée) de transactions est sérialisable si elle produit le même résultat que leur exécution en série (l'une après l'autre).

=> *justification :*

** prise séparément, chaque transaction est supposée correcte (transforme un état cohérent de la BD en un autre état cohérent)*

** l'exécution des ces transactions les unes après les autres (séquentiel) est donc aussi correcte, puisque les transactions sont supposées indépendantes les une des autres*

Notion de sérialisabilité

* Donc une exécution entrelacée est correcte, si elle est équivalente à une exécution en série (si elle est sérialisable).

Remarque : pour les exemples précédents, le problème est que, dans chaque cas, l'exécution n'était pas sérialisable.

L 'effet du verrouillage était précisément de *forcer* la sérialisabilité.

Terminologie

Soit un ensemble de transactions.

Un ordonnancement est toute exécution (entrelacée ou non) de ces transactions.

Un ordonnancement séquentiel : exécution des transactions les unes après les autres (sans entrelacements).

Soit A = {a1, a2, …, an}, B = {b1, b2, …, Bk}, on a :

Ordonnancement séquentiel : a1, …, an, b1, …, bk **ou** b1, …, bk, a1, …, an

Un exemple d ’ordonnancement entrelacé : b1, b2, a1, b3, a2, b4, …., bk, an)

Deux ordonnancements sont équivalents : s'ils produisent les mêmes résultats, quelque soit l'état initial de la base.

Théorème de verrouillage à 2 phases

Le concept de sérialisabilité fut introduit par Eswaran et al. 1976.

Le même article fournit également l'énoncé d'un théorème important :

Le théorème du verrouillage à 2 phases (2PL - Two-Phase Locking) :

Thèorème : si toutes les transactions satisfont au « protocole de verrouillage à 2 phases » alors tous les ordonnancements entrelacés sont sérialisables.

Protocole de verrouillage à 2 phases :

1. Avant d ’agir sur un objet, (un tuple de BD, par exemple), une transaction doit obtenir un verrou sur cet objet (d'un certain type).

2. Après l ’abandon d ’un verrou, une transaction ne doit plus jamais pouvoir obtenir de verrous.

Rmq : une transaction qui obéit à ce protocole a ainsi 2 phases : une phase d ’acquisition de verrous et une phase de libération.

Dans la pratique, la phase de libération est souvent condensée en une seule opération : COMMIT (ou ROLLACK), à la fin de la transaction.

On peut considérer que le protocole présenté précédemment est un protocole de verrouillage à 2 phases.

- **Le théorème est donc une condition *suffisante*.**

- **Rmq** : la gestion de la concurrence peut être assurée sans réaliser de verrouillage, mais par une technique de **gestion de versions**.

  Chaque requête est exécutée dans une copie virtuelle de la base qui l'isole des autres transactions.

  Mais d ’autres types de problèmes se posent pour gérer les différentes versions.

Niveaux d ’isolation

Sont utilisés pour désigner pour décrire le ***degré d ’interférence*** qu ’une transaction donnée est prête à tolérer avec les autres transactions concurrentes.

Pour garantir la sérialisabilité, le seul degré d ’interférence toléré est le degré zéro (pas d ’interférences) !

Mais les systèmes réels autorisent des exécutions de transactions à des niveaux d ’isolation inférieurs à la sérialisabilité.

Niveaux d ’isolation :

READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALISABLE.

Le niveau le plus stricle est SERIALISABLE,

le moins strict est READ UNCOMMITTED.

Niveaux d ’isolation

Si les transactions s ’exécutent au niveau SERIALISABLE, alors n ’importe quelle exécution entrelacée de ces transaction est garantie être sérialisable.

Si elles s ’exécutent avec un niveau d ’isolation inférieur, alors un de ces problèmes peut se poser :

1. **Lecture salissante (dirty read)** : possible avec READ UNCOMMITTED

   T1 effectue une mise à jour sur un tuple p

   T2 récupère ensuite ce tuple

   T1 se termine par un ROLLBACK

   => la transaction T2 a alors observé un tuple qui n ’existe plus ou qui n ’a jamais existé (car T1 n’a en fait jamais été exécutée)

2. Lecture non renouvelable :

possible avec READ COMMITTED et
avec READ COMMITTED

T1 récupère un tuple p

T2 effectue ensuite une mise à jour de ce tuple

T1 récupère à nouveau ce tuple

=> la transaction T1 a alors récupéré le même tuple 2 fois et a observé des valeurs différentes !

3. Fantômes : possible avec tous les niveaux d ’isolation sauf SERIALISABLE

T1 récupère un ensemble de tuples (qui satisfont une condition)

T2 insère ensuite un tuple qui satisfait la même condition

Si T1 récupère de nouveau l ’ensemble de tuples qui fatisfont la condition, elle va observer un nouveau tuple qui n ’était pas auparavant (un fantôme !)

Niveaux d ’isolation

Les différents niveaux d ’isolation sont définis selon le type de violation de la sérialisabilité qu ’ils autorisent. La figure suivante en donne un récapitulatif.

PROBLEME *Niveau d ’isolation*	LECT. SALIS.	LECT. NON RENOUV.	FANTOME
Lect. non validée **(read uncommitted)**	**oui**	**oui**	**oui**
lecture validée **(read committed)**	**non**	**oui**	**oui**
lect. renouvelable	**non**	**non**	**oui**
sérialisable	**non**	**non**	**non**

- Avec SQL : la modification du type de transaction est réalisée avec :

- SET TRANSACTION
  [ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }] [READ WRITE | READ ONLY]