

UFR 919 Ingénierie – module 3I009 cours 7 : Transactions et concurrence d'accès

- Définition, exemples et propriétés des transactions
- Fiabilité et tolérance aux pannes
- Contrôle de concurrence

UPMC - UFR 919 Ingénierie - Bases de données 3I009

Transactions, concurrence -1

Contrôle de concurrence

- Les problèmes de concurrence
- Degrés d'isolation dans SQL
- Exécutions et sérialisabilité
- Contrôle de concurrence
- Améliorations

UPMC - UFR 919 Ingénierie - Bases de données 3I009

Transactions, concurrence -2

Contrôle de concurrence

Objectif : *synchroniser les transactions concurrentes* afin de **maintenir la cohérence** de la BD, tout en **maximisant le degré de concurrence**

Principes:

- Exécution simultanée des transactions pour des raisons de performance
par ex., exécuter les opérations d'une autre transaction quand la première commence à faire des accès disques
- Les résultats doivent être équivalents à des exécutions non simultanées (isolation)
besoin de *raisonner sur l'ordre d'exécution* des transactions

UPMC - UFR 919 Ingénierie - Bases de données 3I009

Transactions, concurrence -3

Problèmes de concurrence

Perte d'écritures : on perd *une partie* des écritures de T1 et *une partie* des écritures de T2 sur des données partagées

[T1: Write a1 → A; T2: Write b2 → B; T1: Write b1 → B; T2: Write a2 → A;]

A=a2, B=b1 : on perd une écriture de T1 et une écriture de T2

Non reproductibilité des lectures : une transaction écrit une donnée entre deux lectures d'une autre transaction

[T1: Read A; T2: Write b2 → A; T1: Read A;]

Si b2 est différente de la valeur initiale de A, alors T1 lit deux valeurs différentes.

Conséquence : *introduction d'incohérence*

Exemple avec une contrainte (A = B) et deux transactions T1 et T2

Avant : A=B

[T1: A*2 → A; T2: A+1 → A; T2: B+1 → B; T1: B*2 → B;]

Après : A = 2*A+1, B=2*B+2 → A <> B

UPMC - UFR 919 Ingénierie - Bases de données 3I009

Transactions, concurrence -4

Degrés d'isolation SQL-92

Lecture sale (lecture d'une maj. non validées):
 T1: Write(A); T2: Read(A); T1: abort (abandon en cascade)

- T1 modifie A qui est lu ensuite par T2 avant la fin (validation, annulation) de T1
- Si T2 annule, T1 a lu des données qui n'existent pas dans la base de données

Lecture non-répétable (maj. intercalée) :
 T1: Read(A); T2: Write(A); T2: commit; T1: Read(A);

- T1 lit A; T2 modifie ou détruit A et valide
- Si T1 lit A à nouveau et obtient *résultat différent*

Fantômes (requête + insertion) :
 T1: Select where R.A=...; T2: Insert Into R(A) Values (...);

- T1 exécute une requête Q avec un prédicat tandis que T2 insère de nouveaux n-uplets (fantômes) qui satisfont le prédicat.
- Les *insertions* ne sont pas détectées comme concurrentes pendant l'évaluation de la requête (résultat incohérent possible).

UPMC - UFR 919 Ingénierie - Bases de données 31009 Transactions, concurrence -5

Degrés d'isolation SQL-92

	Degré	Lecture sale	Lectures non répétable	Fantômes
↑ degré de concurrence	READ_UNCOMMITTED	possible	possible	possible
	READ_COMMITTED	impossible	possible	possible
	REPEATABLE_READ	impossible	impossible	possible
↓ degré d'isolation	SERIALIZABLE	impossible	impossible	impossible

Ne devrait pas s'appeler comme ça

UPMC - UFR 919 Ingénierie - Bases de données 31009 Transactions, concurrence -6

Exécution (où Histoire)

- Exécution** : ordonnancement des opérations d'un ensemble de transactions
- Ordre** : total (séquence = transactions plates) ou *partiel* (arbre, modèles avancés))

T ₁ : Read(x) Write(x) Commit	T ₂ : Write(x) Write(y) Read(z) Commit	T ₃ : Read(x) Read(y) Read(z) Commit
--	--	--

H₁ = W₂(x) R₁(x) R₃(x) W₁(x) C₁ W₂(y) R₃(y) R₂(z) C₂ R₃(z) C₃

UPMC - UFR 919 Ingénierie - Bases de données 31009 Transactions, concurrence -7

Exécution en série

- Exécution en série** : histoire où il n'y a pas d'entrelacement des opérations de transactions
- Hypothèse** : chaque transaction est *localement* cohérente
- Si la BD est cohérente avant l'exécution des transactions, alors elle sera également cohérente après leur exécution en série.

T ₁ : Read(x) Write(x) Commit	T ₂ : Write(x) Write(y) Read(z) Commit	T ₃ : Read(x) Read(y) Read(z) Commit
--	--	--

H_s = W₂(x) W₂(y) R₂(z) C₂ R₁(x) W₁(x) C₁ R₃(x) R₃(y) R₃(z) C₃

T₂ → T₁ → T₃

UPMC - UFR 919 Ingénierie - Bases de données 31009 Transactions, concurrence -8

Exécution sérialisable

Opérations conflictuelles : deux opérations de deux transactions différentes sont en *conflict* si elles accèdent le même granule et *une des deux opérations est une écriture*.

Exécutions équivalentes : deux exécutions H1 et H2 d'un ensemble de transactions sont *équivalentes (de conflict)* si

- l'ordre des opérations de chaque transaction et
- l'ordre des opérations conflictuelles (validées) sont identiques dans H1 et H2.

La bonne définition

Exécution sérialisable : exécution où il existe *au moins une* exécution en série (ou *sérielle*) équivalente (de conflict).

Exécutions équivalentes et sérialisables

T_1 : Read(x)	T_2 : Write(x)	T_3 : Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

Les exécutions suivantes ne sont pas équivalentes :

$H_1 = W_2(x) R_1(x) R_3(x) W_1(x) C_1 R_3(y) W_2(y) R_2(z) C_2 R_3(z) C_3$

$H_2 = W_2(x) R_1(x) W_1(x) C_1 R_3(x) W_2(y) R_3(y) R_2(z) C_2 R_3(z) C_3$

H_2 est équivalente à H_s , qui est sérielle $\Rightarrow H_2$ est *sérialisable* :

$H_s = W_2(x) W_2(y) R_2(z) C_2 R_1(x) W_1(x) C_1 R_3(x) R_3(y) R_3(z) C_3$

Est ce que H1 est sérialisable ?

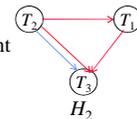
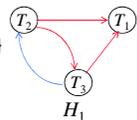
Graphe de précedence (GP)

Graphe de précedence $GP_H = \{V, P\}$ pour l'exécution H :

- $V = \{T_i \mid T_i \text{ est une transaction validée dans } H\}$
- $P = \{T_i \rightarrow T_k \mid o_{ij} \in T_i \text{ et } o_{kl} \in T_k \text{ sont en conflit et } o_{ij} <_H o_{kl}\}$

$H_1 = W_2(x) R_1(x) R_3(x) W_1(x) C_1 R_3(y) R_2(z) W_2(y) C_2 R_3(z) C_3$

$H_2 = W_2(x) R_1(x) W_1(x) C_1 R_3(x) W_2(y) R_3(y) R_2(z) C_2 R_3(z) C_3$



Théorème: l'exécution H est sérialisable ssi GP_H ne contient pas de cycle (facile à prouver, ordre partiel).

H_2 correspond à l'exécution en série : $T_2-T_1-T_3$

Méthodes de contrôle de concurrence

- Méthode optimiste
- Verrouillage à deux-phases (2PL)
- Multiversion (snapshot)
- Estampillage

Algorithmes de verrouillage

Les transactions font des demandes de verrous à un *gérant de verrous* :

- verrous en lecture (vl), appelés aussi **verrous partagés**
- verrous en écriture (ve), appelés aussi **verrous exclusifs**

Compatibilité (de verrous sur le même granule et deux transactions différentes) :

	vl	ve
vl	Oui	Non
ve	Non	Non

Algorithme Lock

```

Bool Function Lock (Transaction t, Granule G, Verrou V) {
  /* retourne vrai si t peut poser le verrou V sur le granule G et faux sinon
  (t doit attendre) */
  Cverrous := {};
  Pour chaque transaction t' ≠ t ayant verrouillé le granule G faire {
    Cverrous = Cverrous ∪ t'.verrous(G) ; // cumuler les verrous sur G
  }
  si Compatible(V, Cverrous) alors {
    t.verrous(G) = t.verrous(G) ∪ { V } ; // marquer l'objet verrouillé
    return true ;
  } sinon {
    /* insérer le couple (t, V) dans la liste d'attente de G */
    G.attente = G.attente ∪ { t } ;
    bloquer la transaction t ;
    return false ;
  }
}
    
```

Algorithme Unlock

```

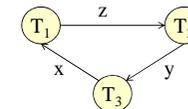
Procédure Unlock(Transaction t, Granule G) {
  /* t libère tous les verrous sur G et redémarre les transactions en
  attente (si possible) */
  t.verrou(G) := {};
  Pour chaque couple (t', V) dans G.attente faire {
    si Lock(t', G, V) alors {
      G.attente = G.attente - {(t', V)} ;
      débloquer la transaction t' ;
    }
  }
}
    
```

Grappe d'Attente (GA)

vl₁(x), ve₂(z), ve₃(y), vl₁(z), ve₃(x), vl₂(y)...

Granule	ve	vl	Attente ve	Attente vl
x		T ₁	T ₃	
y	T ₃			T ₂
z	T ₂			T ₁

Grappe d'attente :



Cycle → **Interblocage**

Ne pas confondre avec le Grappe de Précédence (GP, sérialisabilité)

Résolution des interblocages

Prévention

- Définir des critères de priorité de sorte à ce que le problème ne se pose pas
- Ne pas accepter toutes les attentes (abandon)

Détection

- Gérer le Graphe d'Attente (GA)
- Lancer un algorithme de détection de circuits dès qu'une transaction attend trop longtemps
- Choisir une victime qui brise le circuit

Prévention des interblocages

Algorithme : Les transactions sont numérotées par ordre d'arrivée et on suppose que T_i désire un verrou détenu par T_j :

- Choix préemptif (« seuls plus jeunes attendent »):
 - $j > i$: T_i prend le verrou et T_j est *abandonnée*.
 - $j < i$: T_i attend.
- Choix non-préemptif (« seuls plus vieux attendent »):
 - $j > i$: T_i attend.
 - $j < i$: T_i est *abandonnée*.

Théorème : Il ne peut pas avoir d'interblocages, si une transaction abandonnée est toujours relancée avec *le même numéro*.

Verrouillage et sérialisabilité

Est-ce que le verrouillage permet de garantir la sérialisabilité ???

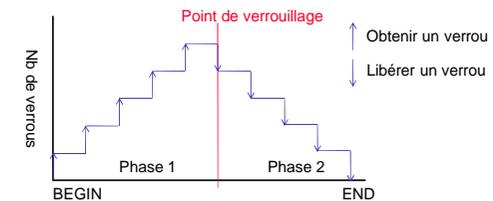
$$H_1 = W_2(x) R_1(x) R_3(x) W_1(x) C_1 R_3(y) R_2(z) W_2(y) C_2 R_3(z) C_3$$

On a vu que H1 n'est pas sérialisable, pourtant...

=> Il ne faut pas libérer les verrous *trop tôt*

Verrouillage à deux phases

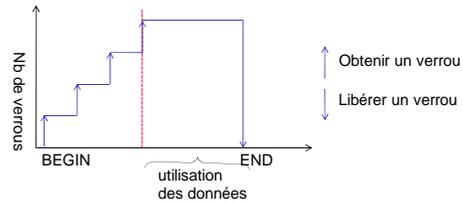
- Chaque transaction verrouille l'objet avant de l'utiliser.
- Quand une demande de verrou est en conflit avec un verrou posé par une autre transaction *en cours*, la transaction qui demande doit attendre.
- **Quand une transaction libère son premier verrou, elle ne peut plus demander d'autres verrous.**



Verrouillage à deux phases strict

On tient les verrous jusqu'à la fin (commit, abort).

Evite les abandons en cascade



UPMC - UFR 919 Ingénierie - Bases de données 31009

Transactions, concurrence -21

Verrouillage à deux phases (2PL): Conclusion

Théorème : Le protocole de verrouillage à deux phases génère des *historiques sérialisables en conflit* (mais n'évite pas les fantômes).

Autres versions de 2PL (Oracle, snapshot isolation):

- Relâchement des verrous en lecture après l'opération :
 - non garantie de la reproductibilité des lectures (READ_COMMITTED)
 - + verrous conservés moins longtemps : plus de parallélisme
- Accès à la version précédente lors d'une lecture bloquante :
 - nécessité de conserver une version (journaux)
 - + une lecture n'est jamais bloquante

UPMC - UFR 919 Ingénierie - Bases de données 31009

Transactions, concurrence -22

Transactions dans Oracle

Une transaction **démarre** lorsqu'on exécute une instruction SQL qui modifie la base ou le catalogue (DML et DDL).

- Ex : **UPDATE, INSERT, CREATE TABLE...**

Une transaction **se termine** dans les cas suivants :

- L'utilisateur valide la transaction (**COMMIT**)
- L'utilisateur annule la transaction (**ROLLBACK sans SAVEPOINT**)
- L'utilisateur se déconnecte (la transaction est validée)
- Le processus se termine anormalement (la transaction est défaite)

Amélioration des performances dans Oracle :

- Niveaux d'isolation
- Contrôle de concurrence multiversion

Verrouillage

UPMC - UFR 919 Ingénierie - Bases de données 31009

Transactions, concurrence -23

Commandes transactionnelles

COMMIT

- Termine la transaction courante et écrit les modifications dans la base.
- Efface les points de sauvegarde (SAVEPOINT) de la transaction et relâche les verrous.

ROLLBACK

- Défait les opérations déjà effectuées d'une transaction

SAVEPOINT

- Identifie un point dans la transaction indiquant jusqu'où la transaction doit être défaite en cas de rollback.
- Les points de sauvegarde sont indiqués par une étiquette (les différents points de sauvegarde d'une même transaction doivent avoir des étiquettes différentes).

UPMC - UFR 919 Ingénierie - Bases de données 31009

Transactions, concurrence -24

SET TRANSACTION

SET TRANSACTION

- Spécifie le comportement de la transaction :
 - Lectures seules ou écritures (**READ ONLY** ou **READ WRITE**)
 - Établit son niveau d'isolation (**ISOLATION LEVEL**)
 - Permet de nommer une transaction (**NAME**)

Cette instruction est facultative. Si elle est utilisée, elle doit être la première instruction de la transaction, et n'affecte que la transaction courante.

Niveaux d'isolation

Oracle propose deux niveaux d'isolation, pour spécifier comment gérer les mises à jour dans les transactions

- **SERIALIZABLE**
- **READ COMMITTED**

Définition

- Pour une transaction :
 - **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE**
 - **SET TRANSACTION ISOLATION LEVEL READ COMMITTED**
- Pour toutes les transactions à venir (dans une session)
 - **ALTER SESSION SET ISOLATION LEVEL = SERIALIZABLE;**
 - **ALTER SESSION SET ISOLATION LEVEL = READ COMMITTED;**

Différence read-committed serializable pour lectures

en mode RC, on lit la dernière valeur validée depuis le début de la commande SQL Select

en mode SR, on lit la dernière valeur validée depuis le début de la transaction (multiversion).

En d'autre terme, RC permet des lectures non reproductibles alors que SR ne le permet pas

Différence read committed / serializable pour écriture

- Dans les deux modes, une transaction attend que le verrou X se libère pour prendre le verrou et faire la mise à jour
- En mode SR, une mise-à-jour de T sera refusée s'il y a eu une modif. faite et validée par une autre transaction depuis le démarrage de T.
- La valeur que T « écrase » doit avoir été validée *avant le début* de T

Mais : E1(a), E2(b), L1(b), L2(a), V1, V2 sera accepté en mode SR n'est ni équivalent à T1, T2, ni équivalent à T2, T1 car les deux lectures vont lire l'état initial (avant le début de l'exécution)

Sérialisation par estampillage : Idée principale

- Hypothèse :
 - Une « vieille » transaction ne génère pas de conflit avec une plus « jeune ».
- Une transaction t peut
 - lire une granule g si la dernière transaction qui a écrit sur g est plus âgée que t
 - écrire sur une granule g si les dernières transactions qui on lu ou écrit sur g sont plus âgées que t
- Sinon on annule t en entier et on la redémarre (elle renaît)

Sérialisation par estampillage (1/3)

On affecte

à chaque transaction t une estampille unique $TS(t)$ dans un domaine ordonné.

à chaque granule g

une étiquette de lecture $EL(g)$ et

une étiquette d'écriture $EE(g)$

qui contient l'estampille de la dernière transaction qui a lu, respectivement, écrit g .

Sérialisation par estampillage (2/3)

La transaction t veut lire g :

• Si $TS(t) \geq EE(g)$, la lecture est acceptée et
 $EL(g) := \max(EL(g), TS(t))$.

• Sinon la lecture est refusée et t est relancée (abort) avec une nouvelle estampille (*plus grande que toutes les autres*).

La transaction t veut écrire g :

• Si $TS(t) \geq \max(EE(g), EL(g))$, l'écriture est acceptée et
 $EE(g) := TS(t)$.

• Sinon l'écriture est refusée et t est relancée avec une nouvelle estampille (*plus grande que toutes les autres*).

Sérialisation par estampillage (3/3)

$L_1(b), L_2(b), E_2(b), L_1(a), L_2(a), E_2(a), E_1(b)?$

• $TS(t_1) < TS(t_2)$: T_1 sera relancée avec une nouvelle estampille
 $TS'(t_1) > TS(t_2)$:

$L_2(b), E_2(b), L_2(a), E_2(a), L_3(b), L_3(a), E_3(b)$

Remarque :

• Il existe des historiques qui ne peuvent pas être produits par 2PL mais sont admis par estampillage et vice-versa.

Règle de Thomas

$L_1(b), L_1(a), E_2(b,v), L_2(a), E_2(a), E_1(b,v)?$
 $L_1(b), L_1(a), E_1(b,v'), E_2(b,v), L_2(a), E_2(a)$

Observation : Aucune transaction avec $TS(t) > TS(t_1)$ a lu b :
 L'abandon de T_1 n'est pas nécessaire car v' n'aurait jamais été lue, si l'écriture s'était passée plus tôt.

Nouvelle règle pour l'écriture:

- Si $TS(t) \geq \max(EE(g), EL(g))$, l'écriture est acceptée et $EE(g) := TS(t)$.
- Si $TS(t) < EE(g)$ et $TS(t) \geq EL(g)$, l'écriture est *ignorée*.
- Sinon l'écriture est refusée et T est relancée avec une nouvelle estampille.

Le problème des annulations

$H = W_2(x) R_1(x) W_1(x) C_1 R_3(x) W_2(y) R_3(y) R_2(z) R_3(z) A_2$

Quand une transaction est abandonnée (A_2) il faut annuler

- les écritures de T_2
- et les transactions T_1 et T_3 qui utilisent ces écritures

Problèmes :

- annulation de transactions *en cascade* : possible mais coûteux
- annulation de transactions déjà validées (T_1) : impossible !
 Exécution non recouvrable

Recouvrabilité

• Exécution *recouvrable* : pas d'annulation de transactions validées

• Solution : **retardement du commit**

• Si T « lit de » T' , alors T doit valider après T'

$H = W_2(x) R_1(x) W_1(x) C_1 R_3(x) W_2(y) R_3(y) R_2(z) R_3(z) A_2$

Éviter annulation en cascade

$H = W_2(x) R_1(x) W_1(x) R_3(x) W_2(y) R_3(y) R_2(z) R_3(z) A_2$

• T_2 annule $\Rightarrow T_1$ et T_3 doivent annuler

• Solution : **retardement des lectures**

• Une transaction ne doit lire que des transactions validées

$H = W_2(x) R_1(x) W_1(x) R_3(x) W_2(y) R_3(y) R_2(z) R_3(z) A_2$

Exécutions strictes

$$H = W_2(x) W_1(x) \dots A_2 \dots A_1$$

T2 annule \Rightarrow comment restaurer la valeur de x ?

A_2 restaure x initial, A_1 restaure x écrit par T_2

(il est possible de changer le comportement, mais complexe)

Solution : **retardement des lectures et écritures**

Une transaction ne doit écrire que sur des données validées

$$H = W_2(x) W_1(x) \dots A_2$$

2PL strict garantit des exécutions strictes

Modèles étendus

Applications longues composées de plusieurs transactions coopérantes

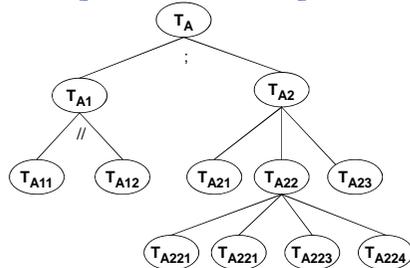
Seules les mises-à-jour sont journalisées

Si nécessité de défaire une suite de transactions:

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

Transactions Imbriquées T.I. (1)

[J. E. Moss 85]



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

Transactions Imbriquées (2)

Sous-transaction : *unité d'exécution*

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

Sous transaction : *unité de reprise*

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

Transactions Imbriquées (3)

Une sous-transaction peut-être:

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

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

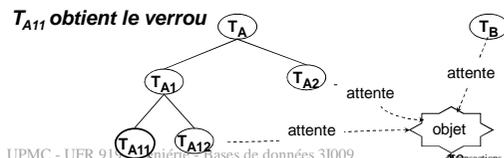
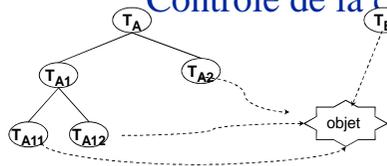
- Abandonne aussi (et donc son sous-arbre abandonne)
- Continue sans les résultats de la sous-transaction abandonnée
- Relance la sous-transaction ou une alternative

Transactions Imbriquées (4) Contrôle de la concurrence

- (R1) Seules les transactions feuilles accèdent aux objets
- (R2) Quand une sous-transaction valide, ses verrous sont hérités par sa mère
- (R3) Quand une sous-transaction abandonne, ses verrous sont relâchés
- (R4) Une sous-transaction ne peut accéder à un verrou que si il est libre ou détenu par un ancêtre

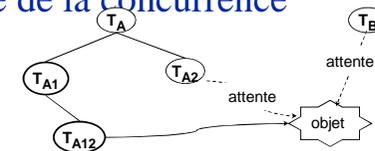
⇒ R1 à R4 garantissent l'isolation

Transactions Imbriquées (5) Contrôle de la concurrence

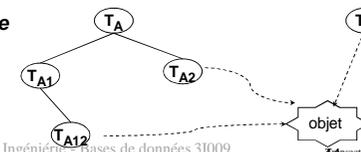


Transactions Imbriquées (6) Contrôle de la concurrence

Si T_{A11} valide



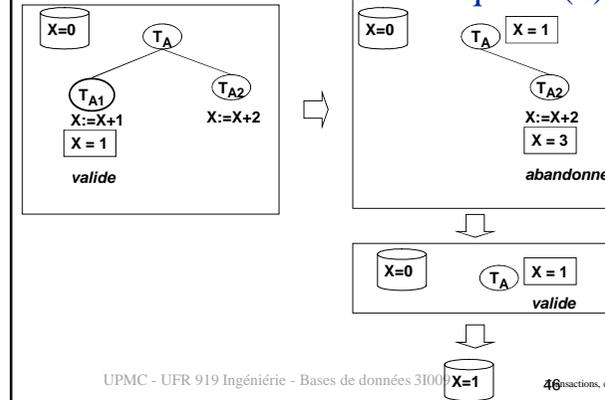
Si T_{A11} abandonne



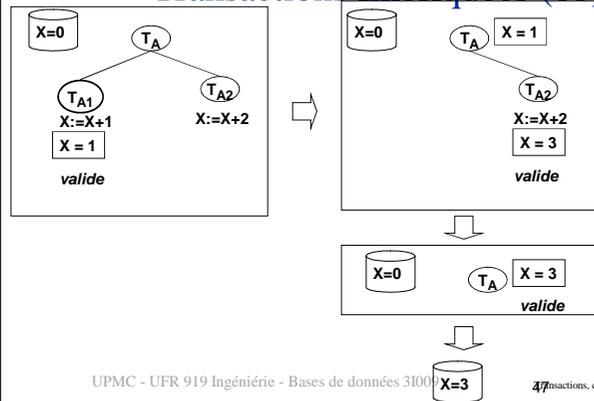
Transactions Imbriquées (7)

- (R'1) Quand une sous-transaction valide, ses effets sont hérités par sa mère
 - (R'2) Quand une sous-transaction abandonne, ses effets sont abandonnés.
 - (R'3) Quand la transaction racine valide, ses effets sont écrits dans la base
- R'1 à R'3 garantissent atomicité et durabilité*

Transactions Imbriquées (8)

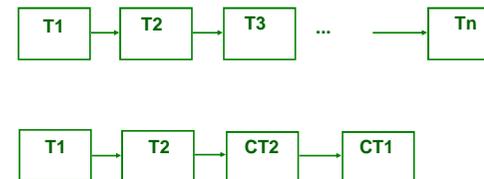


Transactions Imbriquées (8b)



Sagas

Groupe de transactions avec transactions compensatrices
 En cas de panne du groupe, on exécute les compensations
 Seules les sous-transactions sont isolées.... Ce n'est plus vraiment un modèle transactionnel



Conclusion

La gestion des transactions est une tâche importante dans un SGBD :

- Gestion de pannes
- Contrôle de concurrence
- ACID
- garantir la cohérence sans trop bloquer les ressources