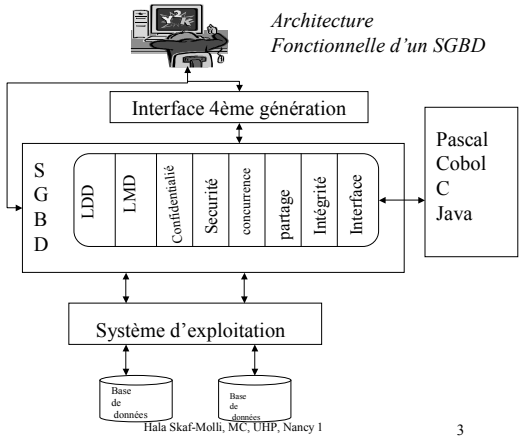
Sécurité de fonctionnement et gestion des accès concurrents

Hala Skaf-Molli

Références

- Database Transaction Models for Advanced Applications, edited by Ahmed K. Elmagarmid, 1992.
- Environnements de Développement Coopératifs, thèse en Informatique, Pascal Molli, 1997..
- Sybase SQL Server, Transact-SQL User's Guide
- + sur le WEB ...

Architecture Fonctionnelle d'un SGBD



Fonctionnalités de SGBD

- Multi-utilisateurs:
  - plusieurs utilisateurs peuvent interagir avec une DB simultanément…
  - régler les problèmes dûs au travail à plusieurs en même temps sur les mêmes objets…
  - assurer la cohérence de la base même en présence des accès concurrents (même en cas dysfonctionnement!!)

  => Transaction…

Définition: Transaction

- Une transaction : un ensemble d'actions permettant de prendre une base données dans état **cohérent** et elle la rend dans un autre état cohérent..
- Elle donne l'illusion à l'utilisateur d'être tous seul

Exemple

- Retrait 50 Fr du compte x; valeur initiale, x= 100Fr; CI: x>=0
- *Programme:*

```
k ← lire(x)
if k >= 50 then
    k = k-50
    écrire(x,k)
else
    print(« pas assez d 'argent sur le
      compte »)
endif
```

Transaction
T1:Retrait(x,50)

lire(x, 100)
écrire(x,50)

Transaction est une séquence d'actions de lire et écrire...

Terminaison de transaction

- L'exécution de transaction se termine : commit ou abort (rollback).
- **Commit**: la transaction est réussie, ses mises à jour doivent être incorporées dans la BD.
- **Abort**: la transaction a échoué, la BD doit annuler ou supprimer tous les effets de la transaction sur la BD

Transaction et cohérence

- ***Par définition**, une transaction qui modifie les objets d'une BD doit préserver les contraintes d'intégrité de la base…*
- Deux approches:
  1. l'utilisateur connaît toutes les CI, il écrit que des programmes corrects (qui respecte les CI)…
  2. pas d'hypothèse sur la correction individuelle des transactions : la BD doit vérifier statiquement et dynamiquement les CI…

Transaction et Cohérence

- La première approche est attractive, pourquoi !!!
- Deuxième approche:
  - sur coût «*overhead*» sur le SGBD..
  - Réduire la disponibilité de la base:
    - l'accès aux données est bloqué: assurer dynamiquement la cohérence de la base après chaque transaction implique que l'état de la base ne peut pas être touché à la fin de la transaction par un autre utilisateur..
  - Le SGBG utilisent approche mixte….

Transaction

- L'hypothèse standard

si une transaction s'exécute toute seule, dans une base de données cohérente, alors elle va laisser la base dans un autre état cohérent..

- Attention: pendant son exécution une transaction peut violer les CI mais lorsqu'elle termine (commit), elle met la base dans un état cohérent..

Propriétés de transactions

Chaque transaction doit être **ACID**:

- **A**tomicité: toute ou rien. Soit toutes les opérations de la transaction sont exécutées soit aucune..
- **C**ohérence: une transaction prise individuellement doit faire passer la base d'un état cohérent dans un autre état cohérent. **I**solation: une transaction ne doit observer que des états cohérents de la base (pas de résultats intermédiaires d'une autre transaction)
- **D**urabilité: lorsqu'une transaction termine, ses résultats deviennent permanents et ne peuvent plus être remis en cause, ni par une panne du système, ni par une autre transaction (remettre en cause la durabilité renvoie aux problème d'exécution partielles)...

Propriétés de transactions

Deux classes d'algorithmes ou protocoles pour assurer l'ACID de transactions:

- Correction des exécutions concurrentes (Execution Atomicity): **protocoles des accès concurrents**
  - maintenir la cohérence des transactions même si elles sont exécutées de manière concurrente : garantir que l'exécution concurrente d 'un ensemble de transactions individuellement correcte fait passer la base dans un état cohérent…
- Prennent en compte les exécutions partielles des transactions (Failure Atomicity) : **protocoles de recouvrement :**
  - maintenir la loi de toute ou rien, isolation et durabilité.

Correction des exécutions concurrentes

- Le système garantit que l'exécution concurrente d 'un ensemble de transactions individuellement correcte fait passer la base dans un état cohérent…
- Pas de violation de cohérence à cause des exécution concurrente des transactions..

Quelles sont les problèmes liés aux exécutions concurrentes ?

Recouvrement

- Trouver un état cohérent de la base à la suite d'une exécution partielle...
- 2 types d 'échecs
  - échec transaction: l'utilisateur veut annuler la transaction, division par zéro, erreur !!
  - Échec système: coupure d'électricité, crash disque !!!
- N'oubliez pas une transaction est atomique (tout ou rien..)

Exécution concurrente

- 3 problèmes :
  - mise à jour perdue
  - lectures impropres
  - lecture non reproductibles

Mise à jour perdue

Etat initial de s:
s=300 ; CI: s>= 0

t_1: programme user1 Retrait(200F, s)	t_2: programme user2 Retrait(300F, s)
temp1 ← **lire(s)**	
temp1 ← temp1 -200	
	temp2 ← **lire(s)**
	temp2 ← temp2 -300
	écrire(s,temp2)
écrire(s,temp1)	

Ordonnancement de t1 et t2

$lire_1$(s,300)
$lire_2$(s,300)
$écrire_2$(s,0)
$écrire_1$(s,100)

t1 toute seule est correcte; t2 toute seule est correcte
Après l 'exécution concurrente : s= 100 !!!

La mise à jour de Retrait(300F, s) est **perdue**!!

Lectures impropres

CI:balance= c + s = 400

t_1:programme user1 Virement(100F, s, c)	t_2:programme user2 Balance()
temp1 ← **lire(s)**	
temp2 ← **lire(c)**	
écrire(s, temp1-100)	
	temp3 ← **lire(s)**
	temp4 ← lire(c)
écrire(c,temp2+100)	
	imprimer(temp3+temp4)

Ordonnancement de t1 et t2

$lire_1$(s,200)
$lire_1$(c,200)
$écrire_1$(s,100)
$lire_2$(s,100)
$lire_2$(c,200)
$écrire_1$(c,300)

Après l exécution:
balance= 300 !!!
Balance() a fait de lectures impropres

Lectures non reproductibles

t_1:programme user1	t_2:programme user2
lire(s, 100)	
imprimer(s)	
	lire(s, 100)
	écrire(s, 300)
lire(s, 300)	
imprimer(s)	

Les 2 valeurs affichées ne sont pas identiques !!

Conclusion sur les exécutions concurrentes

- Imposer que chaque transaction soit individuellement correcte (respecte les CI) ne garantit pas que l'exécution concurrente de ces transactions soit correcte..

i.e. Une exécution concurrente de deux transactions bien écrites peut laisser la base dans un état incohérent..

- Solution possible:
  - SGBD exécute les transactions l'une après l'autre de manière séquentielle (ou en série).
  - **Une exécution séquentielle est correcte**.. Pourquoi??

Inconvénients

- De manière générale, l'entrelacement des opérations permet une meilleure utilisation de CPU..
- De plus, deux transactions n'accédant pas à des données partagées doivent aussi s'exécuter de manière séquentielle !!!

- **Existe-il des exécutions non séquentielles correctes?**

  En d'autre terme:

  Deux transactions accédant à des données partagées peuvent avoir un entrelacement correct de leur opérations !!!

Exemple

t1:programme user1 Virement(100F, s, c)	t2:programme user2 Balance()
temp1 ← **lire(s)** **écrire(s, temp1-100)**	
	temp3 ← **lire(s)**
temp2 ← **lire(c)** **écrire(c,temp2+100)**	
	temp4 ← lire(c) **imprimer(temp3+temp4)**

Ordonnancement de t1 et t2

lire1(s,200)
écrire1(s,100)
lire2(s,100)
lire1(c,200)
écrire1(c,300)
lire2(c,200)

- Exécution correcte..

Pour l'utilisateurs : c'est équivalent à l'exécution de transfert suivie par Balance ..

=> accepter les exécutions dont leur comportement n'est pas différent d'une exécution en série ...

Sérialisabilité

- Critère de correction des exécutions concurrentes..
- **Une exécution concurrente d'un ensemble de transactions est correcte s'il existe une exécution en série telle que dans les deux exécutions:**
  - **chaque transaction lit les mêmes valeurs**
  - **et les valeurs finales sur la base sont les mêmes..**
- Une exécution concurrente respectant ce critère est dite sérialisable (View sérialisable)...

Exemple: exécution view sérialisable

t0	t1
lire(x, 1)	
	lire(x, 1)
écrire(x, 2)	
	écrire(y, 2)

Exécution concurrente de : t0 || t1

t0	t1
	lire(x, 1)
	écrire(y, 2)
lire(x, 1)	
écrire(x, 2)	

Exécution en série de t1;t0

Dans les deux cas:

les transactions lisent les mêmes valeurs, ReadSet??

et produisent le même état de la base, WriteSet??

Exemple: exécution non view sérialisable

t0	t1
lire(s, 600)	
	lire(s, 600) écrire(s, s-300)
écrire(s, s-200)	

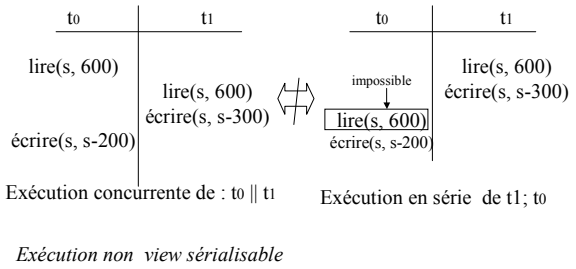
Exécution concurrente de : t0 || t1

t0	t1
	lire(s, 600) écrire(s, s-200)
	impossible lire(s, 600)
	écrire(s, s-300)

Exécution en série de t0; t1

Exécution non view sérialisable ??

Exemples



Exécution concurrente de : t0 || t1

Exécution en série de t1; t0

Exécution non view sérialisable

Est-il possible d'implanter le critère de view serialisabilité ?

- Trouver si une exécution est view sérialisable est un problème NP-complet
- Exemple:
  - Tj lit une valeur x écrite par Ti => toutes les opérations de Tj suivent les opérations de Ti (il y a une contrainte: Ti suivie par Tj)
  - Tk écrit x alors Tk avant Ti ou après Tj (sinon, Tj doit lire la valeur écrite par tk) (pas de contrainte..)
  - Deux choix possibles..
  - Le nb de choix ????
- Plusieurs choix; explosion combinatoire..

Conflict sérailisabilité

- Imposer un ordre d'exécution entre les transactions:
  - pas seulement dans le cas ou une transaction lit une valeur écrite par une autre transaction..
  - Mais aussi lorsqu'il y a une écriture concurrentes
  - ou lecture et écriture…
- Idée: opérations en conflit ou commutativité entre les opérations..

Opérations en conflit

- Deux opérations sont en conflit :
  - si leur résultats dépend de leur ordre d'exécution (cas opérations de lecture/écriture)..
  - ou la valeur finale de la base dépend de leur ordre d'exécution (opérations d'écriture..)
- Pour deux opérations op1 et op2 issues de deux transactions t1 et t2, on a 5 cas possibles..

Opérations en conflit

- op1 et op2 sur des objets différents: pas de conflit…
- op1 et op2 sont des opérations de lectures: pas de conflit..
- écriture précède une lecture: conflit
- lecture précède une écriture: conflit..
- op1 et op2 sont d'écritures sur le même objet: conflit

Résumons

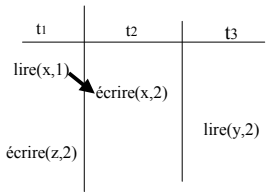
	Lire	écrire
Lire	NO	Yes
écrire	Yes	Yes

Deux opérations sont en conflits si elles opèrent sur le même objet et une parmi elles est une opération d 'écriture

Conflict sérailisabilité

- Une exécution correcte pour un ensemble des transactions est une exécution pour laquelle :
  - il existe une exécution en série pour le même ensemble des transactions dans ces deux exécutions l'ordre des opérations en conflits est le même..

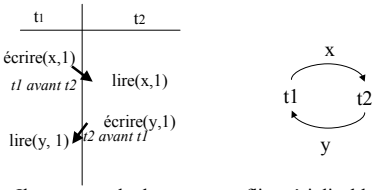
Exemple



T1||t2||t3 <=> t1;t3;t2

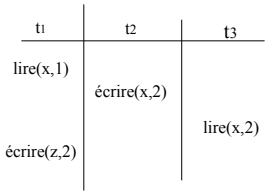
Graphe de dépendance (graphe de précédence)

- Une exécution est conflict sérialisable si son graphe de dépendance est acyclique



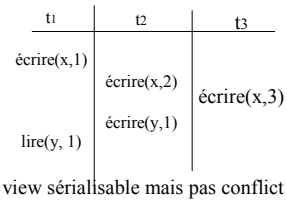
Il y a un cycle donc pas conflict sérialisable

Exemple: exécution conflict sérialisable



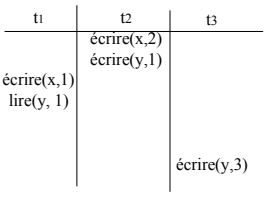
Conflict sérialisabilité

- Une exécution Conflict sérialisable est aussi view sérialisable mais le contraire n'est pas vrai



view sérialisable mais pas conflict sérialisable:

dans exécution en série équivalente: t1 doit précéder t2 et t2 doit précéder t1

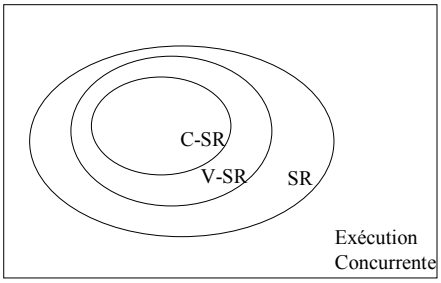


Exécution en série

Conclusion

- Une exécution concurrente est correcte si elle est équivalente à une exécution en série..
- Critères de correction:
  - Sérialisabilité (View séralisable) : NP-Complet
  - Conflict Sérialisable: graphe de dépendance..

Conclusion



Conflict sérialisabilité

- Protocoles (algorithmes) pour contrôler les accès concurrents et assurer les conflicts sérialisabilité des transactions:
  - **Protocole de verrouillage à deux phases: 2PL (two phases locking protocol).**
  - **Timestamp ordering protocole..**
  - Optimistic protocole...
  - serialization graph testing..

Sécurité de fonctionnement: recouvrement (Failure Atomicity)

- 2 types d'échecs
  - échec transaction: l'utilisateur veut annuler la transaction, division par zéro, erreur !!
  - Echec système: coupure d'électricité, crash disque !!!

N'oubliez pas une transaction est atomique (toute ou rien..)

Echec de transaction

t1
lire(x, 1)
écrire(x, 5)
écrire(y, 25)
abort ← annuler

- **Abort**: annuler les effets de la transaction, comme si la transaction n'a jamais existait !!
- Pour l'opération lire: rien à faire..
- Pour l 'opération écrire: il faut l'annuler «undo», pour retrouver l'état de la base avant la transaction (*before-image*)
- La technique utilisée consiste à tenir le journal de modification *log*

Journalisation des transactions

- Pour permettre de **défaire** et \ou **refaire** il faut connaître les actions effectuées sur la BD et conserver les anciennes et les nouvelles valeurs des objets
- Tenir un JOURNAL de toutes les opérations sur la BD (le « LOG »)..
- Synchroniser les E/S sur ce journal avec les E/S sur la base..
- Le JOURNAL est un fichier, il est supposé survivre à une panne (copie, etc..)

Journalisation des transactions

- Un élément du journal contient:
  - id de transaction,
  - début de transaction,
  - opération sur la bd (écrire !!!!)
  - ancienne valeur de l 'objet (image « avant »)
  - nouvelle valeur de l 'objet (image « après »)
  - validation d 'une transaction
  - point de sauvegarde (save point)
- **Principe**: «C'est le journal qui fait foi »

(« WRITE AHEAD LOG »: on écrit sur le journal)

Exemple

t1
begin
lire(x, 1)
écrire(x, 5)
écrire(y, 25)
abort

Un extrait du log:

t1
begin
écrire, 1, 5
écrire, 3, 25
rollback

Exemple

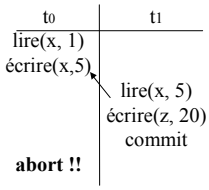
t1
lire(x, 1)
écrire(x, 5)
save point p1
écrire(y, 25)
abort

Un extrait du log:

t1
begin
écrire, 1, 5
save transction p1
écrire, 3, 25
rollback

Echec de transaction

- Problème est un peu plus complexe !!!
- Scénario:



Annuler t1 => il faut annuler ses effets de la base :
(prendre *before-image* de t1)

Echec de transaction

- Problèmes:
  - t2 observe une valeur de x qui n'existe plus!!
  - t2 est terminée, il ne plus possible de défaire quoi que soit (une transaction est durable !!!!)
- Il faut fixer la règle suivante:

***Règle 1*: si une transaction t2 lit un objet modifié (un résultat intermédiaire) par une autre transaction t1 alors t2 ne peut pas se terminer terminer avant t1..**

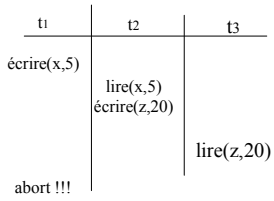
Les exécutions qui vérifient cette règles sont dites ***recouvrable***...

Synthèse

- Transaction est annulée=>
  - ses effets sur la base sont annulés en restaurant *before-image* pour les objets modifiés par la transaction..
  - Ses effets sur les autres transactions (celles qui ont lu des objets modifiés par la transaction) en les annulant *abort*

  => *cascade aborts !!*

Annulation en cascade



t1 est annulée => t2 doit être annulée => doit être annulée

Pour éviter ce phénomène utiliser l'**isolation** de transaction

Eviter les annulations en cascade

- Les transactions sont isolées:
  - une transaction ne peut pas voir les résultats intermédiaires des autres transactions..

Règle 2:

une transaction t_2 peut lire un résultat d'une autre transaction t_1 seulement si t_1 est terminée (commit)

Les exécutions qui respectent cette règle sont «avoid cascading aborts»

Echec de transaction

- Les deux règles précédentes + *before-image* ne permettent pas d'établir la cohérence de la base en cas d 'échec..
- Exemple: valeur initiale de x est 0

t_1	t_2
écrire(x,1)	
	écrire(x,2)
	commit
abort !!!	

Le before-image de t1 entraîne une mise à jour perdue:
les modifications effectuées par t2 sont perdues…
Et si j 'annule t2 puis t1??

Echec de transaction

- **Règle 3** :
  - Si une transaction t1 écrit une valeur x alors aucune autre transaction ne peut écrire x tant que t1 n 'est pas terminée ou annulée..

Résumons

On peut résumer les trois règles par:

si une transaction t1 écrit une valeur x alors aucune autre transaction ne peut lire ou écrire x tant que t1 n 'est pas terminés ou annulée

- Les exécutions qui respectent cette règle sont ***exécution stricte***
- Le protocole utilisé pour assurer ce type d'exécution est : 2PL strict

Echec système

- Une base de données utilise:
  - Mémoire secondaire, non volatile (disque)
  - Mémoire centrale, volatile (*database cache*)..
- Idée: pendant son exécution, une transaction lit et modifie les données dans la cache et à sa terminaison, l'idéal est que la transaction écrit ses modifications sur le disque..
- Problème: que se passe t-il en cas du panne système?
  - assurer l'atomicité de transactions ..

Echec système

- Solution: Utiliser le journal *log*
  - les effets de transactions qui étaient dans l 'état commit au moment de panne doit être incorporées dans la base (**do** opération)..
  - Les effets des transactions qui étaient dans l 'état abort au moment de défaillance doit être éliminer de la base (**undo** opération)

Echec système

- En cas d'arrêt du système
  - log contient *begin* sans commit: annuler la transaction. La commande **undo** restaure les objets modifié par la transaction à leur valeur initiale.;
  - log contient *begin* et *commit*: valider la transaction. La commande **redo** remet les objets modifié par la transaction à leur nouvelle valeur ..

2PL

- Basé sur le principe de verrouillage..
- Deux types de verrous:
  - partageables en lecture: V-partage(x)
  - exclusif en écriture: V-exclusif(x)
- Granularité (taille de données verrouillée) de verrou: tuple, table, base, page...
- **Table de comptabilité:**

	V-partage	V-exclusif
V-partage	Yes	no
V-exclusif	no	no

2PL

- Transaction bien formée si:
  - Avant de LIRE x elle a au moins un V-partage(x)
  - Avant d'ECRIRE x elle a un V-exclusif(x)
  - Aucun objet ne reste verrouillé après la FIN de la transaction
- Transaction à deux phases (2PL):
  - La règle de 2 phases: une transaction a deux phases:
    - *Growing* phase (verrouillage croissant) : elle obtient les locks mais pas de déverrouillage...
    - *shinking* phases (verrouillage décroissant): elle libère les locks mais pas de nouveaux verrouillages
    - n 'effectue aucune LOCK après avoir exécutée un UNLOCL

2PL

- L'exécution concurrente des transactions à 2PL est conflict sérialisable
- Exemple:

t1	t2
V-partage(A)	
lire(A, 5)	
	V-exclusif(C)
	écrire(C,100)
V-partage(B)	V-partage(B)
lire(B,10)	lire(B,10)
unlock(A)	unlock(B)
unlock(B)	unlock(C)

Problème de 2PL: Iterblocage (deadlock)

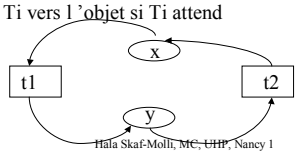
- Le verrouillage entraîne le problème suivant:

t1	t2
V(x) OK	
	V(y) OK
V(x) attendre	
	V(y) attendre

t1 attendre t2 et t2 attend t1 => interblocage

Détection d'interblocage

- Construire le graphe Qui attend Quoi (ou Qui)
- (on peut utiliser aussi le graphe de précédence)
- Nœuds= transactions et objets
- Arcs:
  - de l'objet vers Ti si Ti a verrouillé l 'objet
  - de Ti vers l 'objet si Ti attend



Gestion des interblocages

- 3 méthodes :
  - Temporisation :
  - Détection-guérison:
  - Prévention :

Problème des objets Fantômes

- Relation EMP(no, dept, sal)

No	dept	sal
e1	d1	50
e2	d1	45
e3	d2	25
e4	d2	30
e5	d2	48

	T1	T2
A1	select max(sal) from EMP where dept= «d1»	
B1		insert into EMP values(e6,d1,52)
B2		delete from EMP where ne= e5
A2	select max(sal) from EMP where dept = «d2»	

Problème des objets Fantômes

Exécution 1	Exécution 2	Exécution 3
A1 A2 B1 B2	B1 B2 A1 A2	A1 B1 B2 A2
max d1=50 max d2: 48	max d1 = 52 max d2= 30	max d1= 50 max d2= 30

• Pas de sérilialisation ….
• T1 ne peut pas verrouiller des n-uplets n'existent pas
• verrou sur **toute** la relation pour éviter insertions et suppressions de n-uplets

SQL et Transactions

- Début de transaction implicite (une instruction SQL)
- Fin de transaction
  - implicit (après chaque instruction, ..)
  - explicite: COMMIT pour valider, ROLLBACK pour annuler
- Type et degré d'isolation d 'une transaction
  - Action explicite de verrouillage et libération (Lock, Unlock)
  - Granule de verrouillage ? Tuple, relation, index, page, segment..

SQL et Transctions

- SQL2: SET TRANSACTION <mode><isolation>
- Mode: READ ONLY: lecture seulement, pas de mise à jour de la BD
- READ WRITE
- Niveau d 'isolation???

Niveaux d'isolation

Niveau d 'isolation	Lecture impropre	Lecture non reproductible	Fantômes
Level 0: *READ UNCOMMITED DATA*	Possible	possible	Possible
Level 1: *READ COMMITED DATA*	NON	Possible	Possible
Level 2: *REPEATABLE READ*	NON	NON	Possible
Level 3: *SERIALIZABLE*	NON	NON	NON

NB: pas de mise à jour perdue

SQL2: Isolation level

- Serializable:
  - T verrouille au début et libère à la fin 2PL strict.
  - Verrous sur ensemble d 'objets pour éviter les fantômes..
- Repeatable Read:
  - T lit seulement les modifications de transactions ayant validé (commit).
  - Aucun objet lu ou modifié par T n'est pas modifié par une autre transaction
  - verrou sur objet individuel et non sur collection, ce qui peut entraîner les fantômes..

SQL2: Isolation level

- Read Committed Data:
  - T lit seulement les mises à jour de transactions ayant validé commit).
  - Aucun objet modifié par T n 'est pas modifié par une autre transition
  - mais un objet lu par T peut avoir été modifié par une autre transaction..
  - Verrous exclusifs pour écrire et ne pas les libérer avant la fin
  - Verrous de lecture libérés de suite..

SQL2: Isolation level

- Read Uncommitted Data:
  - T peut lire des objets modifiés par une autre transaction
  - pas de verrous en lecture et mode «READ ONLY»

Scénario d 'exécution ..

Transactions sous Oracle

- Niveaux 1 et 3
- Undo segment
- Exemple:

Les transactions dans le SGBD sybase

select **@@isolation**
// the current isolation level.…

- Modifier le niveau d'isolation
  set transaction isolation level 0
  // allows read uncommitted data
  set transaction isolation level 1
  // allows read uncommitted
  set transaction isolation level 3
  // serializable

Quelques variables utiles

Choix de mode de transaction

set chained on

// begin transaction is executed impicitly before: update,delete, open, fetch select…

select **@@tranchained**

// @@tranchained= 0 -> unchained mode (explicite mode)

// @@tranchained= 1 -> chained mode (implicite mode)

Quelques variables utiles

@@transtate: elle donne l 'état de la transaction

select @@transtate

// @@transtate = 0 -> transaction in progess..

// @@transtate = 1 -> transaction succeded (committed)

// @@transtate = 2 -> statement aborted (only previous state has aborted no effect on transaction)

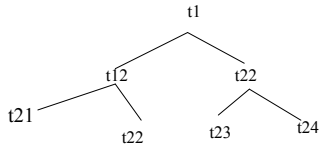
// @@transtate = 3 -> transaction aborted (rolledback)

Conclusion

- La gestion de transaction est assurée par un moniteur transactionnel
- Le moniteur, le journal, le verrou est transparents pour l'utilisateur..
- Tout se passe comme si l'utilisateur était seul avec sa base de données..

Modèle de transactions avancés

- Transactions imbriquées (Nested transactions) (Moss85)
- Avantage ??
  - Plus de parallélisme +
  - limiter l'effets de l 'annulation d 'une transaction..
- si t1 est annulée => toutes ses filles sont annulées
- l'annulation d'une fille n'a pas d 'impact sur la transaction mère …



Transaction imbriquée dans Sybase

```
begin tran
 select @@trancount
 /*trancount = 1*/
    begin transction
    select @@trancount
    /*trancount = 2*/
    a1
          begin transction
               select @@trancount
                 /*trancount = 3*/
          commit tran
    commit tran
commit tran
               select @@trancount
                 /*trancount = 0*/
```

Variable globale **@@trancount** numéro de transaction utilise avec les nested transactions

Transaction et procédures stockées

begin tran

update titles set….

insert into tiles…

exec myproc

delete titles where

```
create proc myproc
as
 begin tran
statements
if … rollback tran
else commiy tran
```

si myproc est annulée alors **update** et **insert** sont annulée mais delete est exécutée