Bases de données (avancées)

Cours: Fabrice.Jouanot@imag.fr
TDs/TPs: Clement Charpentier / Fabrice Jouanot

Support: http://imag-moodle.e.ujf-grenoble.fr
(code: M1INFO15)

Programme du cours

- Gestion des transactions
- Gestion des contraintes d'intégrités
  - BDs dynamiques & déclencheurs
  - Application à Oracle
- Optimisation de requêtes
  - Optimisation algébrique, optimisation du stockage et des structures d'accès
  - Application à Oracle: tuning
- Modèle de données avancées
  - BDs objets en bref
  - Relationnel Object (application à Oracle)
- Accès à une BD
  - JDBC
  - Framework de persistance: Hibernate ORM
- Bases de données distribuées
  - Introduction
  - conception

Références

- Database System concepts, H. F. korth
- Systèmes de bases de données, T. connolly & C. Begg
- « Bases de données » de Gardarin (PDF)
- Cours en ligne de S. Abiteboul
- etc.

1. Gestion des transactions

Comment garantir la cohérence et la fiabilité tout en favorisant les accès concurrents aux données ?

Introduction

- La propriété fondamentale d'un SGBD est de garantir l'intégrité des données.
- Mais autoriser les accès concurrents est nécessaire pour un SI.
- 3 fonctions importantes et liées faisant référence à la notion de transaction:
  - Garantir un état cohérent
  - Contrôler la concurrence
  - Reprise après panne.

Un exemple très simplifié

- On considère un système de gestion de stades basés sur trois tables :
  Stade(nomStade, NbPlaces),
  Match(Affiche, NomStade, Prix, PlacesPrises)
  Client(NoClient, Solde).

- et disposant d'une fonctionnalité de réservation de places pour un client.

Exemple: Réservation de places

```
Places (Client C, Match M, NbPlaces N)
begin
        Lire le match M (tuple de la table match)
        Lire le stade S (tuple de la table stade)

        if ( (S.nbplaces – M.PlacesPrises) >= N )
        begin
                Lire le client C (tuple de la table client)
                M.PlacesPrises += N
                C.solde -= N * M.Prix
                Ecrire le match M (tuple modifié de la table match)
                Ecrire le client C (tuple modifié de la table client)
        end
end
```

Exemple: scénario concurrent

- Soit l'histoire suivante représentant deux exécutions concurrentes de la fonction de réservation pour le même match et le même client :
  - T1=Places(C, M, 100)
  - T2=Places(C, M, 200)

T1	L(M)
T1	L(S)
T1	L(C)
T1	E(M)
T2	L(M)
T2	L(S)
T2	L(C)
T2	E(M)
T1	E(C)
T2	E(C)

- Si on suppose que le stade dispose de 1000 places, sans résa pour le match au début de l'exécution: Quel est l'état de la base à la fin ?

Trace de l'histoire

T1	L(M)	=> 0 places réservées
T1	L(S)	=> 1000 places dans le stade
T1	L(C)	=> soit X le solde du client
T1	E(M)	=> 100 places réservées par le client
T2	L(M)	=> 100 places réservées
T2	L(S)	=> 1000 places dans le stade
T2	L(C)	=> soit X le solde du client
T2	E(M)	=> 300 places réservées par le client
T1	E(C)	=> X=X – 100*prix
T2	E(C)	=> X=X – 200*prix

- 300 places ont été réservées, mais 200 ont été comptabilisées au client !

Notion de transaction

- **Une transaction est une opération ou une suite d'opérations qui lit ou met à jour le contenu de la base de données.**
  - C'est une unité logique de travail
  - Formé par un programme, une partie de programme ou une commande

Ex: une suite de commande SQL peut former une transaction composée de plusieurs opérations.

2 opérations	1 opération
Select count(*) from T1	
	Update T1 Set attr=select count(*) from T2
Select count(*) from T1	

Fonctionnement d'une transaction

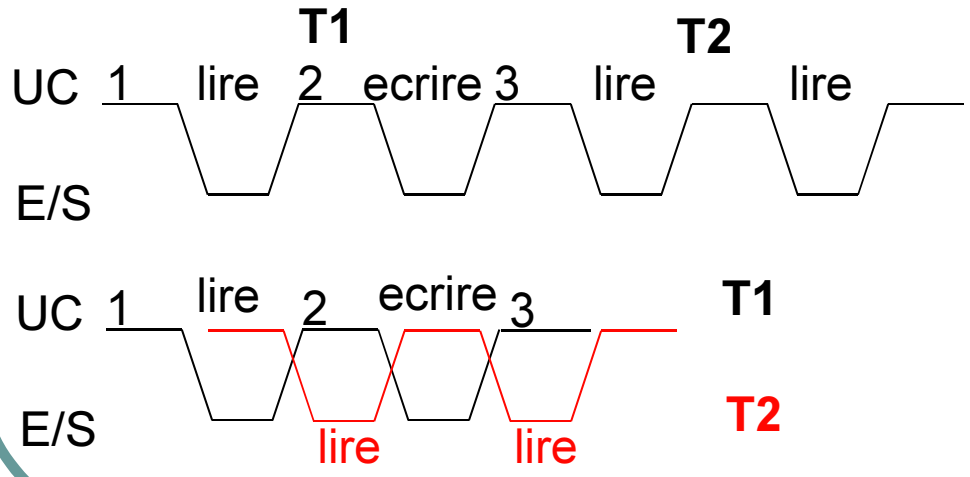
- Une transaction est constituée de 2 types d'opérations:
  - Lecture, notée lire(x) ou L(x) ou R(x)
  - Ecriture, notée ecrire(x) ou E(x) ou W(x)
- Une transaction possède 2 fins possibles
  - Achèvement avec succès, elle est validée ou confirmée (committed)
  - En échec, elle est annulée (aborted) car la BD doit rester dans un état cohérent et stable. L'annulation consiste en une opération de "roll back" pour remonter dans le dernier état cohérent.

Notion de concurrence

- La concurrence d'accès dans un environnement multi-utilisateurs permet une meilleure utilisation des ressources.
  - Simultanéité des accès
  - Mais compétition sur les données
- Les opérations d'E/S sont interfoliées pour assurer un accès concurrent aux données.

Exécution série vs. parallèle

T1:	T2:
| 1	|
Lire(x)	Lire(x)
| 2	|
Ecrire(x)	Lire(y)
| 3	|



Exécution en série

Exécution en parallèle

Problèmes liés à la concurrence

- L'entrelacement (interfoliage) des opérations de plusieurs transactions est un **ordonnancement** lorsque l'ordre des opérations de chaque transaction en concurrence est préservée.
- Des transactions a priori correctes peuvent produire des résultats incohérents en fonction de l'ordonnancement choisi.
- 3 problèmes peuvent apparaître:
  - Perte de mise à jour (écriture sale)
  - Lecture sale
  - Lecture non reproductible (et pare extension fantôme)

Perte de mise à jour (O1)

T1	T2
Début transaction	Début transaction
lire(solde)	lire(solde)
solde=solde-10	solde=solde+100
ecrire(solde)	ecrire(solde)
Validation	validation

- Une opération apparemment validée se trouve écrasée par une autre.
  - T1 et T2 démarrent presque au même temps
  - T1 et T2 lisent la même valeur de solde
  - La dernière écriture de T1 écrase celle de T2

<u>solde=solde+100 est perdue</u>

Lecture sale (O2)

T1	T2
	Début transaction
	lire(solde)
	solde=solde+100
	ecrire(solde)
Début transaction	…
lire(solde)	Annulation
solde=solde-10	
ecrire(solde)	
Validation	

- Une transaction lit les données écrites par une autre transaction non encore validée (O1 => O2)
  - T2 modifie la valeur de solde
  - T1 lit la valeur de solde modifiée et réalise sa tâche en se basant sur cette valeur
  - T2 annule entre-temps ces propres opérations.

solde=solde+100-10

Lecture non reproductible (O3)

T1	T2
Début transaction	Début transaction
lire(solde)	lire(solde)
solde=solde-10	solde=solde+100
…	ecrire(solde)
lire(solde)	validation
…	
Validation	

- Une transaction relit une valeur lu précédemment mais modifiée, entre-temps, par une autre transaction.
  - T1 lit la valeur de solde
  - T2 modifie la valeur de solde et valide
  - T1 relit la valeur de solde qui est différente de sa première lecture.

solde=100

Propriétés ACID assurées par un système de gestion des transactions

- **Atomicité**: "tout ou rien". Une transaction forme une entité indivisible, soit exécutée, soit annulée dans sa totalité.
- **Cohérence**: une transaction fait passer la BD d'un état cohérent à un autre état cohérent. La responsabilité de cohérence est cependant partagée:
  - Le SGBD vérifie les CI de schéma, et certaines CI métiers (déclencheurs des BD dynamiques)
  - Le développeur doit assurer la cohérence pour les CI d'application (créditer un compte erroné).

Propriétés ACID assurées par un système de gestion des transactions

- **Isolation**: les transactions s'exécutent de manière indépendante les unes des autres. Les effets des transactions incomplètes ne doivent pas être visibles par les autres transactions.
- **Durabilité**: les effets d'une transaction complètement achevée et validée sont inscrits de manière durable dans la BD. Ils ne peuvent être altérés par une défaillance.

Un SGBD possède donc un système de contrôle de concurrence et de reprise après panne capable de respecter ses propriétés + une gestion avancée des CI.

Gestion de transactions sous Oracle

- Valider une transaction: la commande **Commit** indique la fin d'une transaction et le début d'une autre. Une transaction est l'ensemble des opérations entre 2 commit.
- Annuler une transaction: la commande **RollBack** permet d'annuler toutes les opérations d'une transaction en remontant jusqu'au dernier point de sauvegarde. En l'absence de point de sauvegarde, la restauration remonte jusqu'au dernier commit.
  - savepoint Nom_du_point_de_sauvegarde
- Mode AutoCommit: Oracle est par défaut en mode **autocommit**, toutes les opérations sont systématiquement validées. Aucune transaction n'est possible dans ce mode (1 transaction = 1 opération).
  - Set autocommit on/off pour basculer d'un mode à l'aute

Ordonnancement et sérialisation

- Un ordonnancement est une séquence d'opérations d'un ensemble de transactions concurrentes qui préserve l'ordre des opérations dans chacune des transactions.
- **Un ordonnancement sériel** exécute les opérations de chaque transaction de manière consécutive, sans aucune opération interfoliée d'autres transactions.

T1	T2
	Début transaction
	lire(solde)
	solde=solde+100
	ecrire(solde)
	Validation
Début transaction	
lire(solde)	
solde=solde-10	
ecrire(solde)	
Validation	

Capacité de sérialisation

- Solution simple pour éviter les interférences (conflits) entre transactions: sérialiser les transactions.
  - Chaque transaction attend que la précédente ait validé pour commencer.
  - Cela contredit toute forme de simultanéité et de parallélisme.
- La capacité de sérialisation d'un ordonnancement reste donc intéressante
  - Si on trouve des ordonnancements non sériels dont le résultat est équivalent à une exécution sériel,
  - L'ordonnancement est dit cohérent.

Ordonnancement cohérent

- Les ordonnancements O1, O2 et O3 précédents ne sont pas sérialisables:
  - Les résultats produits ne sont équivalents ni à T1 puis T2, ni T2 puis T1.
  - Ces ordonnancements sont incohérents !

T1	T2
Début transaction	
lire(solde)	
solde=solde-10	
ecrire(solde)	
…	
	Début transaction
	…
	lire(solde)
	solde=solde+100
	ecrire(solde)
lire(solde)	
Validation	
	Validation

Conflits de sérialisation

- Le type et l'ordre des opérations influe directement sur la capacité de sérialisation:
  - 2 transactions qui lisent uniquement des données ne sont jamais en conflits
  - 2 transactions qui lisent et/ou écrivent des données différentes ne sont jamais en conflits
  - 2 transactions qui lisent et/ou écrivent des données dont un sous-ensemble est commun peuvent entrer en conflits.
- Une transaction Ti peut être en conflit avec Tj si:
  - Soit Lec(T) l'ensemble des objets lus par T et Ecr(T) l'ensemble des objets écrits par T, alors
  - $Ecr(Ti) \cap Lec(Tj) \neq \emptyset$
  - Ou $Ecr(Ti) \cap Ecr(Tj) \neq \emptyset$
  - Ou $Lec(Ti) \cap Ecr(Tj) \neq \emptyset$
- Comment identifier un conflit ?

Graphe de dépendance

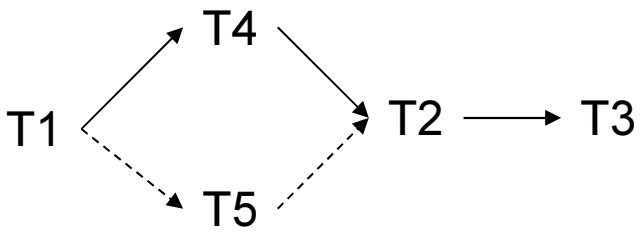
- Un graphe de dépendance permet de tester la capacité de sérialisation d'un ordonnancement: il décrit l'ordre des dépendances entre les opérations.
- Un graphe de dépendance est un graphe orienté (N,A) avec N un ensemble de nœuds et A un ensemble d'arcs dirigés:
  - Créer un nœud pour chaque transaction
  - Créer un arc Ti → Tj, si Tj lit la valeur d'un élément écrit par Ti
  - Créer un arc Ti → Tj, si Tj écrit une valeur d'un élément après qu'il a été lu par Ti
  - Créer un arc Ti → Tj, si Tj écrit une valeur d'un élément après qu'il a été écrit par Ti
  - Un arc est étiqueté avec la donnée lue/écrite (la cause de la dépendance).

  Aucun arc n'est ajouté si Tj lit la valeur d'un élément lu par Ti.

Analyser un graphe de dépendance

- Si une flèche Ti→Tj existe dans le graphe d'un ordonnancement O, alors dans tout ordonnancement sériel O' équivalent à O, Ti doit apparaître avant Tj.
- Un graphe de dépendance peut contenir plusieurs ordonnancements sériels O' équivalent à l'ordonnancement O de départ.

T1: Ecrire(A = 100)
Valider T1
T4 : Lire(A)
T5 : Lire(A)
Valider T5
Valider T4
T2 :Ecrire(A = 50)
Valider T2
T3 : Ecrire(A = 200)
Valider T3

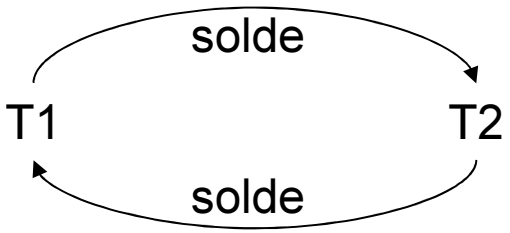


- Le graphe montre deux chemins pour aller de T1 à T2
  - Les positions T4 et T5 sont interchangeables
  - Équivalent à 2 ordonnancements
    sériels T1, T4, T5, T2, T3
    ou T1, T5, T4, T2, T3

Graphe de dépendance cyclique

- Si le graphe de dépendance montre la présence d'un cycle, l'ordonnancement n'a pas la capacité de sérialisation. Il est dit incohérent et des transactions sont en conflits.
- Si on considère O1 par exemple:

T1	T2
	Début transaction
Début transaction	lire(solde)
lire(solde)	solde=solde+100
solde=solde-10	ecrire(solde)
ecrire(solde)	validation
Validation	

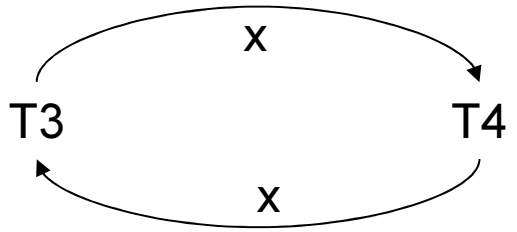


O1 n'est pas cohérent, il n'est équivalent à aucun ordonnancement sériel (ni T1 puis T2, ni T2 puis T1)

Problème des écritures non contraintes Limite du graphe de dépendance

- **Si une transaction écrit un objet sans l'avoir lu au préalable, il n'existe pas d'algorithme efficace pour déduire la capacité de sérialisation.**
- Prenons un exemple simple de deux transactions T3 et T4:

T3	T4
Lire(x)	
	Ecrire(x)
Ecrire(x)	



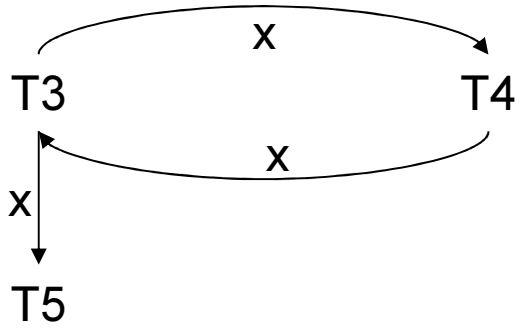
- Le graphe de dépendance est cyclique, il trouve le conflit.
- Le résultat n'est équivalent ni à T3 puis T4, ni T4 puis T3, donc l'ordonnancement n'est pas cohérent.

On croit que ça marche dans tous les cas…

Problème des écritures non contraintes

- Ajoutons une transaction T5:

T3	T4	T5
Lire(x)		
	Ecrire(x)	
Ecrire(x)		
		Ecrire(x)



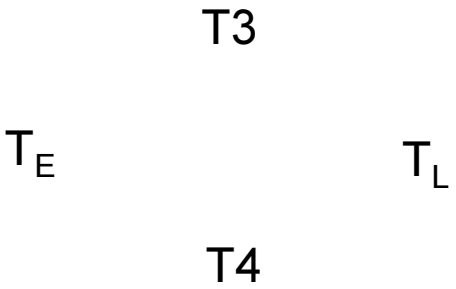
- Le graphe de dépendance est cyclique, donc l'ordonnancement n'a pas de capacité de sérialisation a priori.
- Cependant l'arc T4→T3 ne devrait pas être ajouté au graphe
  - Les écritures produites par T3 et T4 ne sont utilisées par aucune transaction: elles sont inutiles et remplacée par celle de T5
  - **L'ordonnancement sériel équivalent est T3, T4, T5**

Besoin d'un nouveau modèle de graphe

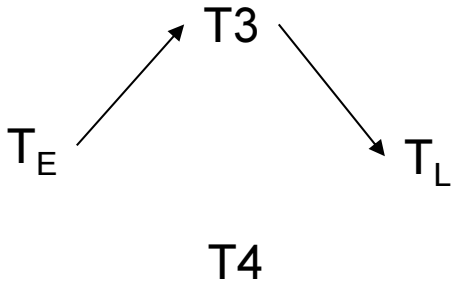
- Soit l'ordonnancement O et O' son équivalent sériel, avec Tj qui lit un objet précédemment écrit par Ti:
  - Si O est sérialisable alors Ti précède Tj dans O'
  - Si $\exists$Tk qui effectue une écriture dans O, alors Tk est avant Ti ou après Tj, sinon Tj ne lit pas la valeur écrite par Ti et $\neg$(O $\equiv$ O')
- Il est impossible d'exprimer la possibilité d'ajouter ou non un arc avec un graphe de dépendance…
- Il faut l'étendre à un graphe de dépendance labellisé (ce label ne doit pas être confondue avec la donnée lue/écrite).

Graphe de dépendance labellisé 1/2

1. Créer un nœud pour chaque transaction
2. Créer un nœud T_E pour une transaction factice en début d'ordonnancement contenant une opération d'écriture pour chaque donnée accédée
3. Créer un nœud T_L pour une transaction factice en fin d'ordonnancement contenant une opération de lecture pour chaque donnée accédée



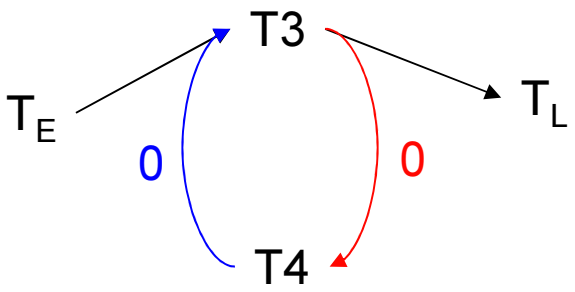
4. Créer un arc Ti $\rightarrow^0$ Tj si Tj lit la valeur d'une donnée écrite par Ti
5. Supprimer tous les arcs dirigés vers Tj pour lesquels il n'existe pas de chemin de Ti vers T_L



Graphe de dépendance labellisé 2/2

6. Pour chaque donnée lu par Tj, qui a été écrite par Ti, et que Tk écrit (Tk $\neq T_E$):
   a) Si $Ti=T_E$ et $Tj \neq T_L$, alors $Tj \rightarrow^0 Tk$
   b) Si $Ti \neq T_E$ et $Tj=T_L$, alors $Tk \rightarrow^0 Ti$
   c) Si $Ti \neq TE$ et $Tj \neq TL$, alors créer deux arcs $Tk \rightarrow^x Ti$ et $Tj \rightarrow^x Tk$ où x un entier positif encore non utilisé dans le graphe.

- Le cas 6.c généralise les règles précédentes: si Ti écrit une donnée que Tj lira, alors Tk qui écrit la même donnée doit soit précéder Ti, soit succéder Tj.



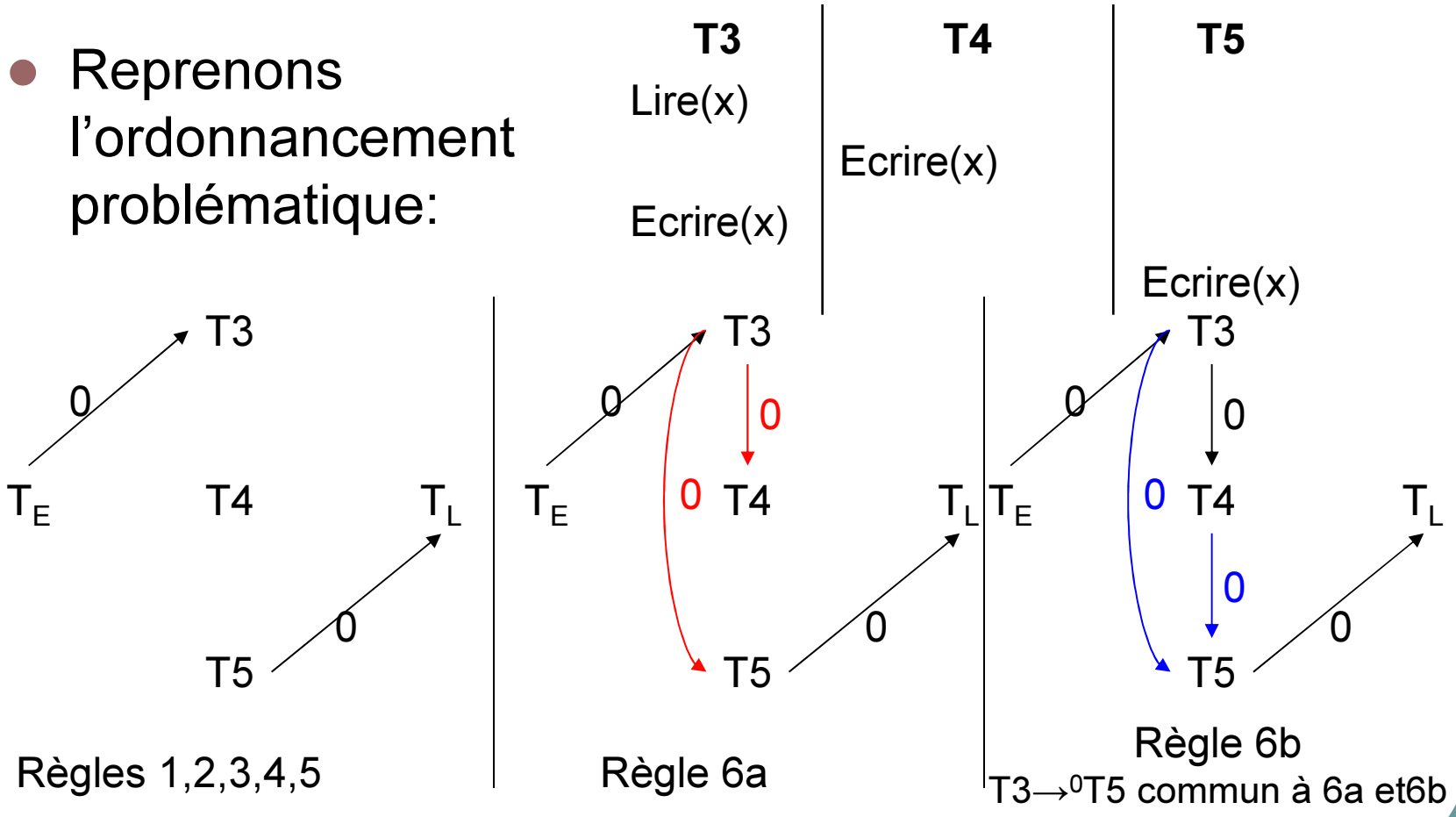
On identifie un cyle, donc a priori le graphe est incohérent ?

Analyse du graphe de dépendance labellisé

- Si le graphe ne contient aucun cycle, l'ordonnancement est sérialisable.
- Si le graphe possède un cycle, on ne peut conclure directement:
  - Si le graphe est étiqueté uniquement avec des arcs $\rightarrow 0$, alors l'ordonnancement n'est pas sérialisable
  - La génération d'une paire d'arcs par la règle 6c produit 2 graphes possibles. Si au moins l'un des graphes est acyclique: l'ordonnancement est sérialisable.
  - Si m paires d'arcs sont générées par la règle 6c alors il faut étudier 2^m graphes.

Exemple de graphe labellisé

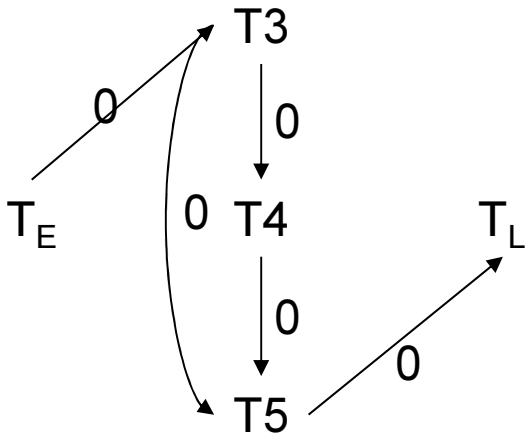
- Reprenons l'ordonnancement problématique:



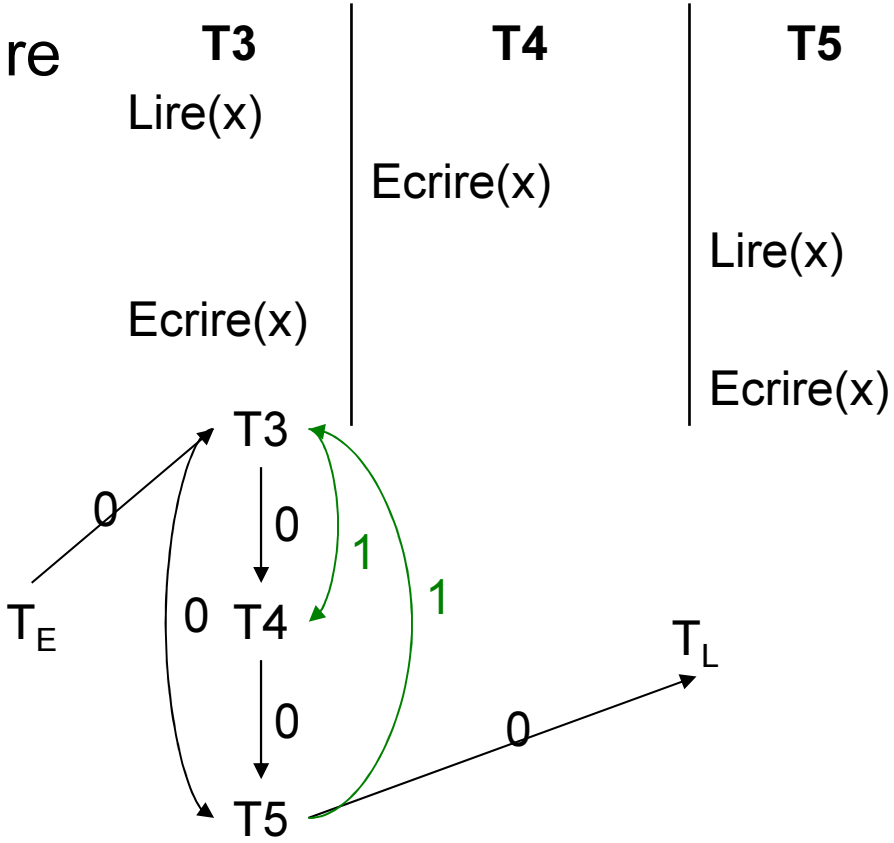
Pas de cycle, donc ordonnancement sérialisable !

Exemple complexe de graphe labellisé 1/2

- Considérons une lecture dans T5, la règle 6c devrait s'appliquer:



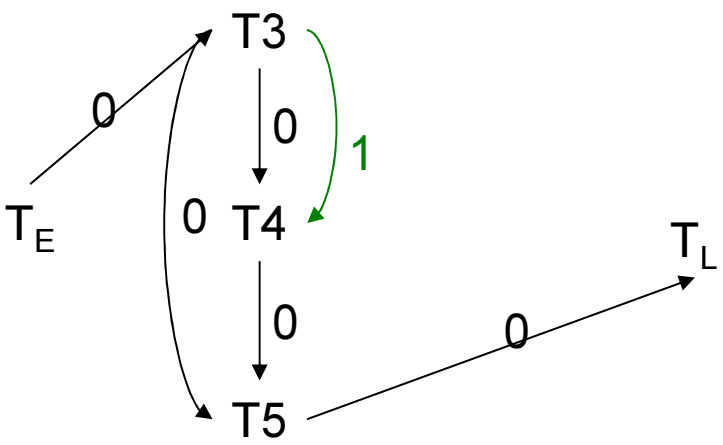
Règles 1,2,3,4,5,6a,6b



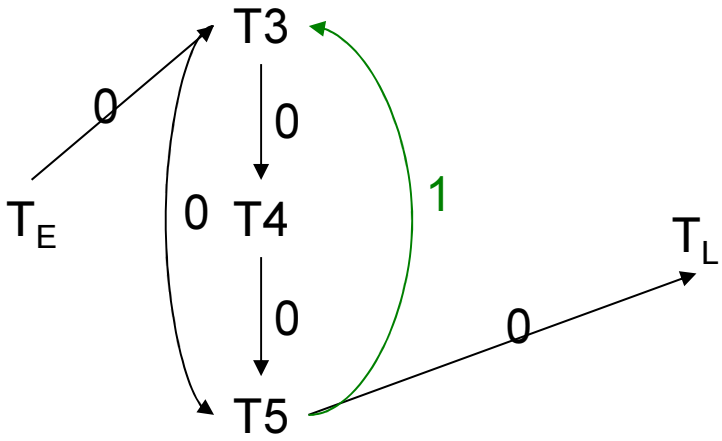
Règle 6c: T4 écrit, ensuite T5 lit, T3 ?

Exemple complexe de graphe labellisé 1/2

- Nous obtenons un graphe pour chaque arc de la paire labellisé 1:



Pas de cycle, donc <u>sérialisable</u>:
T3, T4, T5



cyclique, donc <u>incohérent</u> !

- Au moins 1 graphe est sérialisable, donc l'ordonnancement est cohérent.

Objectif: produire des ordonnancements sérialisables

- Un SGBD ne teste pas la capacité de sérialisation d'un ordonnancement: trop coûteux !
- Des protocoles permettent d'assurer l'obtention d'ordonnancement sérialisable:
  - Protocoles de verrouillages
  - Protocoles d'estampillage
  - Méthodes optimistes

Méthodes de verrouillage

- Lorsqu'une transaction accède à une BD, un verrou est susceptible de bloquer l'accès à d'autres transactions.
- On distingue 2 types de verrous:
  - Partagé: si une transaction dispose d'un verrou partagé sur une donnée, elle peut la lire mais pas la modifier = verrou en lecture (VL).

    Plusieurs transactions peuvent se partager un même verrou en lecture.
  - Exclusif: si une transaction dispose d'un verrou exclusif sur une donnée, elle peut la lire et la modifier = verrou en écriture (VE).

    Une seule transaction peut obtenir un verrou Exclusif, les autres transactions doivent attendre la libération du verrou pour accéder à la donnée (en posant un nouveau verrou).

Protocole de verrouillage

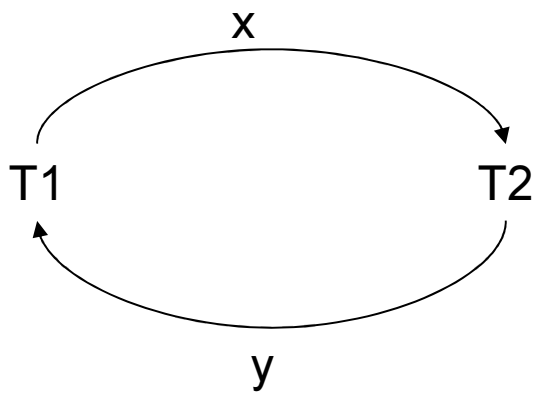
- Toute transaction devant accéder à une donnée doit obtenir un verrou sur la donnée: soit VE(x) , soit VL(x)
- Si la donnée est déjà verrouillée par une autre transaction avec un verrou non compatible (seul VL est compatible avec VL) la transaction doit attendre la libération du verrou.
- Une transaction conserve un verrou tant qu'elle ne le libère pas:
  - Explicitement avec une commande de libération notée L(x).
  - Implicitement avec une terminaison de transaction: validation (commit) ou annulation (abort).

- **Insuffisant pour garantir la sérialisation des ordonnancements…**

Exemple d'ordonnancement incohérent par verrouillage

T1	T2
VE(x)	
Lire(x)	
x=x+100	
Ecrire(x)	
L(x)	
	VE(x)
	Lire(x)
	x=x*1.1
	Ecrire(x)
	L(x)
	VE(y)
	Lire(y)
	y=y*1.1
	Ecrire(y)
	L(y)
	Valider
VE(y)	
Lire(y)	
y=y-100	
Ecrire(y)	
L(y)	
Valider	

- Initialement x=100, y=400
  - T1, puis T2 donne x=220 et y=330
  - T2, puis T1 donne x=210 et y=340
- L'ordonnancement donne x=220 et y=340: donc **PAS SERIALISABLE**



Verrouillage à deux phases

- Pour garantir la sérialisation d'un ordonnancement on ajoute un protocole pour contrôler le verrouillage: V2P ou 2PL (2 phase-locking)
- Une transaction suit le protocole de verrouillage à deux phases si toutes les opérations de verrouillage précédent la première opération de déverrouillage de cette transaction.
  - Phase 1: la transaction acquière des verrous
  - Phase 2: la transaction libère les verrous.
  - Une transaction qui libère un verrou ne peut plus en acquérir.
- **Si dans un ordonnancement toutes les transactions respectent le protocole V2P alors cet ordonnancement est sérialisable.**

Résolution de la MAJ perdue

T1	T2
Début transaction	Début transaction
VE(solde)	VE(solde)
wait	lire(solde)
wait	solde=solde+100
wait	ecrire(solde)
wait	VL(prime)
wait	si (prime>500) …
lire(solde)	Validation (L(solde), L(prime))
solde=solde-10	
ecrire(solde)	
Validation (L(solde))	

- T1 et T2 en V2P
  - T1 attend que T2 relache son verrou sur la donnée
  - On retombe sur une exécution sérielle (T2 puis T1).

Résolution de la lecture sale

T1	T2
	Début transaction
	VE(solde)
	lire(solde)
	solde=solde+100
	ecrire(solde)
Début transaction	…
VE(solde)	Annulation (L(solde))
wait	
wait	
lire(solde)	
solde=solde-10	
ecrire(solde)	
Validation	

- T1 et T2 en V2P
  - T1 attend que T2 relache son verrou sur la donnée
  - On retombe sur l'exécution de T1.

Problème des annulations en cascade 1/2

T1	T2	T3
Début transaction		
VE(x)		
lire(x)		
VE(y)		
lire(y)		
x=y-x		
ecrire(x)		
L(x)	Début transaction	
…	VE(x)	
	lire(x)	
	x=x+100	
	ecrire(x)	
	L(x)	
	…	
Annulation		
		Début transaction
		VE(x)
		lire(x)
	Annulation	…
		Annulation

Problème des annulations en cascade 2/2

- Dépendances des transactions
  - T1 a écrit x avant de libérer son verrou,
  - T2 récupère la donnée de x modifié par T1, et écrit une valeur modifiée de x avant de libérer son verrou sur x,
  - T3 lit la valeur modifiée par T2 après avoir posé son verrou
- Si T1 est annulée
  - T2 dépend de T1 donc T2 doit être annulée
  - T3 dépend de T2 donc T3 doit être annulée
- Risque d'annulation en cascade: destruction important du travail accompli.
- Solution: repousser les libérations de verrous à la fin de la transaction
  - C'est le verrouillage rigoureux en deux phases
  - On peut aussi retarder uniquement les verrous exclusifs, c'est le verrouillage strict en deux phases.

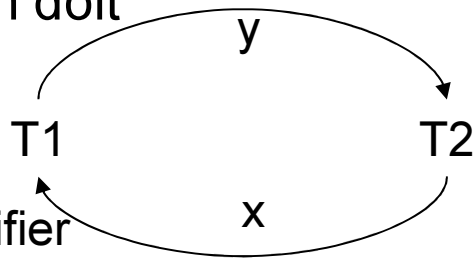
Problème des interblocages

T1	T2
Début transaction	
VE(x)	Début transaction
lire(x)	VE(y)
x=x-10	lire(y)
ecrire(x)	y=y+100
VE(y)	ecrire(y)
wait	VE(x)
wait	wait
…	wait
	…

- T1 attends que T2 libère y et T2 attends que T1 libère x
- **Stituation de blocage ou interblocage** (deadlock): impasse générée par deux transactions (ou plus) qui attendent mutuellement que des verrous se libèrent.
- Résolution simple: délai imparti pour l'acquisition d'un verrou
  - Une transaction a un délai maximum pour obtenir un verrou
  - Sinon la transaction est annulée.

Résolution de l'interblocage

- La solution générale est la construction d'un graphe d'attentes:
  - Un nœud par transaction
  - Un arc Ti → Tj est ajouté si la transaction Ti attend pour verrouiller un élément déjà verrouillé par Tj.
- Si un cycle est détecté: au moins une transaction doit être annulée
  - Soit la plus récente,
  - Soit celle ayant modifié le moins de données
  - Soit celle ayant encore le plus de données à modifier
- Un SGBD choisit la méthode la moins couteuse
  - En terme d'informations à conserver pour choisir sa victime
  - En terme d'annulation à réaliser
  - Il évite la famine, c'est-à-dire ne pas choisir toujours la même victime.



Stratégie de prévention d'interblocages

- Deux transactions Ta et Tb, on considère que Ta veut accéder à une donnée verrouillée par Tb:
  - Wait-die (attendre - mourir)
    - Si Ta plus vieille que Tb, alors Ta est mise en attente et Tb finit (commit ou rollback), sinon Ta est tuée (rollback).
    - Les transactions jeunes sont prioritaires tant qu'elles ne sont pas gênées par des plus vieilles.
    - Les transactions plus vieilles peuvent attendre longtemps
  - Wound-wait (blesser- attendre)
    - Si Ta plus vieille que Tb alors Ta tue Tb (rollback) et s'empare de sa ressource, sinon Ta attend.
    - Les transactions les plus vieilles sont exécutées rapidement (et les jeunes attendent).

Autre technique: Estampillage

- Verrouillage très employé dans les SGBD centralisés mais peu adaptés au SGBD répartis sur plusieurs sites (1 SGBD par sites qui forme globalement un système réparti).

- Les approches distribuées utilisent une technique à base d'**estampilles** pour gérer les transactions.
  - Une estampille est un identificateur temporel permettant d'ordonner des objets chronologiquement.
  - Le verrouillage assure qu'**un** ordonnancement soit équivalent à un ordonnancement sériel
  - L'estampillage assure qu'un ordonnancement soit équivalent à **l'**ordonnancement sériel qui correspond à l'ordre chronologique des estampilles des transactions.

Principe de l'estampillage

- Gestion de la concurrence
  - Toutes les transactions s'exécutent simultanément (pas de verrou)
  - Un conflit = une transaction demande à lire un élément qui a déjà été écrit par une transaction plus récente **ou** une transaction demande à écrire un élément qui a déjà été lu ou mis à jour par une transaction plus récente.
  - En cas de conflit: rejets des demandes conflictuelles sinon l'exécution simultanée ne serait plus équivalente à l'exécution chronologiques des transactions.
  - Solution: tuer la transaction "trop vieille" qui fait la demande et la relancer avec une nouvelle estampille.

- Tuer une transaction
  - Revient à défaire ou annuler une transaction, donc risque d'annulation en cascade.
  - Effectuer physiquement les écriture lors de la fin normale (COMMIT) de la transaction: rien à faire en cas d'annulation (ROLLBACK)

Deux types d'estampilles

- De transaction, noté e(T): identificateur unique créé par le SGBD et précisant le moment de démarrage d'une transaction (implémenté sous la forme d'un compteur logique)
- De donnée où chaque donnée possède deux estampilles:
  - LMAX qui est l'estampille de la transaction la plus jeune qui a lu cet élément (sans avoir été tuée lors de cette lecture),
  - EMAX qui est l'estampille de la transaction la plus jeune qui a écrit avec succès l'élément (sans avoir été tuée lors du COMMIT).
  - Le système ordonne l'exécution des transactions en comparant ces deux valeurs pour tout élément lu ou écrit avec l'estampille de la transaction.

Protocole de gestion des conflits avec estampilles

- soit une transaction T qui demande à lire un élément E

si e (T) ≥ EMAX(E)
    alors **/*OK, la lecture est effectuée */**
        LMAX(E):=Max(LMAX(E), e (T))
    sinon /*conflit */
        Relancer la transaction T avec une nouvelle estampille.

- soit une transaction T qui demande à écrire un élément E

si e(T) ≥ LMAX(E) et e(T) ≥ EMAX(E)
    alors **/*OK, la mise à jour est effectuée */**
        EMAX(E):=e (T)
    sinon **/*conflit */**
        Relancer la transaction T avec une nouvelle estampille.

Exemple d'estampillage

T1	T2
Lire(B)	
	Lire(B)
	B=B-50
	Ecrire(B)
Lire(A)	
	Lire(A)
Afficher(A+B)	
	A=A+50
	Ecrire(A)
	Afficher(A+B)

- On suppose e(T1)<e(T2)
- Montrer que cet ordonnancement est valide…

Exemple d'estampillage

- Démonstration:

  e(T1)>=EMAX(B) [première écriture, EMAX(B)=0] donc
  LMAX(B):=Max(LMAX(B),e(T1)):=Max(0,e(T1)):=e(T1)

  e(T2)>=EMax(B) (première écriture, EMAX(B)=0) donc
  LMAX(B):=Max(LMAX(B), e(T2):=Max(e(T1), e(T2)):=e(T2)

  e(T2)>=LMAX(B) [e(T2)] et e(T2)>=EMAX(B) [0] donc
  EMAX(B):=e(T2)

  e(T1)>=EMAX(A) donc LMAX(A):=e(T1)

  e(T2)>=EMAX(A) donc LMAX(A):=e(T2)

  e(T2)>=LMAX(A) donc EMAX(A):=e(T2)

Approche Optimiste

- Réservé aux cas où les conflits sont rares: beaucoup de lecture et très peu d'écritures
  - Vérification d'un conflit lors de la validation d'une transaction
  - Annulation si conflit, donc peu être très coûteux si risque de conflits important !
- Protocole optimiste = 2 ou 3 phases
  1. Lecture (jusqu'à validation): les MAJ de données sont appliquées sur des copies locales
  2. Validation: détection des conflits (estampille par exemple)
  3. Écriture: copies locales recopiées sur la base.

Gestion de transactions dans un SGBD

- Elle repose sur 2 commandes
  - Validation = commande Commit
  - Annulation = commande Rollback
- Deux types de transaction
  - Read only
  - Read write
- Quatre niveaux d'isolation sont définies pour relâcher ou consolider les Pbs de lecture sale, lecture non reproductible et lecture fantôme
  - Chaque transaction peut choisir son niveau d'isolation
  - Le niveau d'isolation modifie la visibilité des écritures des autres et de ses propres écritures par les autres.

Niveaux d'isolation 1/2

- Serializable
  - La transaction suit le protocole 2PL strict
  - La transaction ne voit que les modifications validées au début de son exécution et ses propres modifications => synchronisation nécessaire à sa validation.
  - Une même requête (ré-exécutée) donnera le même résultat car les modifications sont bloquées pour les autres transactions.
- Repeatable read
  - Une même requête donnera le même résultat, mais les autres transactions peuvent modifier les mêmes données (non bloquées)
  - Fantômes possibles

Niveaux d'isolation 2/2

- Read commited
  - Chaque requête de la transaction ne voit que les modifications validées avant son exécution.
  - L'état de la base peut changer entre deux requêtes.
  - fantômes et lecture non reproductibles possibles
  - Ecriture sale possible
- Read uncommitted
  - La transaction peut lire des objets écrits par une autre transaction (pas de verrous en lecture)
  - lectures sales possibles
  - Proche du mode read only d'oracle.

Niveaux d'isolation: résumé

	Lecture sale	Lecture non reproductible	fantôme
Read Uncommited	Oui	Oui	Oui
Read commited	Non (mais ecriture sale)	Oui	Oui
Repeatable Read	Non	Non	Oui
Serializable	Non	Non	Non

Exemple d'isolation

- Voir Document Resa-spectacle
  - Description du schéma
  - Description des transactions (SQL)

- Trouver pour chaque niveau d'isolation un ordonnancement entre deux transactions qui démontrent une incohérence:
  - READ UNCOMMITTED => Lecture sale
  - READ COMMITTED => Ecriture sale
  - REPEATABLE READ => fantôme
  - SERIALIZABLE => Dead lock

Niveau d'isolation sous Oracle

- Deux modes pour Oracle:
  - `Set transaction isolation level read committed`, chaque instruction au sein de la transaction ne voit que les données validées.
  - `Set transaction isolation level serializable`, chaque instruction au sein de la transaction ne voit que les données validées avant le début de la transaction. ATTENTION, une transaction "sérialisable" peut être annulé par le système si elle ne peut être sérialisé !
  - Des verrous niveau enregistrements sont mis en œuvre.
- Un troisième mode: `set transaction read only`, définit une transaction en lecture seule qui voit les données validées avant le début de la transaction (et interdit les modifications)

Gestion des verrous sous Oracle

- Oracle dispose deux types de verrous
  - Logique sur les tables ou les lignes
  - Physique sur les segments, fichiers et pages.
- Au niveau logique, on trouve:
  - Verrous lignes: seules les lignes à modifier d'une table que modifie une transaction sont verrouillées.
  - Verrous tables: toutes les lignes de la table que modifie une transaction sont verrouillées.
  - Verrous dictionnaire: protègent la description des structures de données (schéma).
  - Verrous internes (loquets): protègent les objets systèmes.
  - Etc.
- Oracle intercepte les interblocages et annule l'ordre à la source de celui-ci (erreur !).

Manipulation des verrous sous Oracle

L'utilisateur a la possibilité de définir sa propre stratégie en conformité avec le verrouillage implicite géré par le SGBD.

`LOCK TABLE {nom_table|nom_vue IN`

- **`ROW SHARE`** verrou (RS), les lignes à modifier sont verrouillées et les autres transactions peuvent mettre à jour les données non concernées par le verrou.
- **`ROW EXCLUSIVE`** verrou (RX), aucune transaction ne peut demander de verrous (S) ou (X, SRX)
- **`SHARE UPDATE`** verrou (RS)
- SHARE verrou partagé (S), les autres transactions peuvent poser des verrous (S) sur la ressource mais pas exclusif (X ou SRX). Si plusieurs transactions ont posé un verrou S, aucune ne peut faire de modification.
- **`SHARE ROW EXCLUSIVE`** verrou (SRX), les autres transactions peuvent obtenir des verrous (RS)
- **`EXCLUSIVE`** verrou exclusif (X), toutes les lignes de la tables sont verrouillées, la consultation reste possible pour les autres transactions mais ne peuvent poser de verrous.

`MODE}`

`[NOWAIT]` empêche d'attendre si la ressource n'est pas disponible (rend la main à l'utilisateur).

Résumé de la compatibilité des verrous Oracle

Verrou obtenu	Verrou **impossibles** pour les autres	Verrou **possibles** pour les autres
RS	X	RX, RX, S, SRX
RX	S, SRX, X	RX, RS
S	RX, SRX, X	RS, S
SRX	RX, S, SRX, X	RS
X	RS, RX, S, SRX, X	-