
Bases de données avancées

Concurrence d'accès et reprise

Dan VODISLAV

CY Cergy Paris Université
Master Informatique M1
Cours BDA

Plan

- La notion de transaction
- Les problèmes de la concurrence
 - Problèmes de cohérence
 - Problèmes d'annulation
- Contrôle de concurrence
- Algorithmes
 - Verrouillage à deux phases
 - Verrouillage hiérarchique
 - Estampillage
 - Niveaux d'isolation
- Reprise après panne
- Concurrence et reprise dans Oracle

La notion de transaction

- Modèle de base de données
 - On ne considère pas un modèle de données en particulier (ex. relationnel)
 - Base de données = ensemble d'articles nommés, contenant n'importe quel type de valeur
 - Modèle général, compatible avec n'importe quel type de BD
 - Ex. x:5, y: "toto", z: 2.78*
 - Opérations possibles sur les articles: lecture, écriture, création, etc.
 - Plusieurs programmes qui s'exécutent en même temps et font des opérations sur la BD
- On se limite pour l'instant à deux opérations de base
 - Lecture: $v = Lire(x)$
 - Écriture: $Ecrire(x, v)$

Programme et transaction

- Programme
 - L'exécution d'un programme peut produire une séquence d'opérations
 - On ne s'intéresse qu'aux opérations de lecture/écriture dans la BD
 - La séquence peut être découpée en plusieurs sous-séquences, chacune pouvant être validée (*Commit*) ou annulée (*Rollback*)
- Exemple: programme de débit d'un compte

```
Débit (Article solde, int montant){  
    int temp = Lire(solde);  
    if (temp < montant) Rollback;  
    else {Ecrire(solde, temp-montant); Commit;}  
}
```
- Séquences possibles à l'exécution
 - *Lire*(solde), *Rollback*
 - *Lire*(solde), *Ecrire*(solde), *Commit*

Transactions

- Transaction
 - Séquence d'opérations sur la BD produite par l'exécution d'un programme
 - Terminée par validation (*Commit*) ou annulation (*Rollback*)
 - La transaction a une *cohérence logique*
 - La séquence a une signification logique
 - Elle part d'un état cohérent de la BD et la transforme en un autre état cohérent
- Relation programme – transaction
 - Une transaction provient de *l'exécution* d'un programme
 - Une exécution de programme produit une séquence d'opérations, qui peut être découpée en plusieurs transactions
 - Toute opération de l'exécution fait partie d'une transaction
 - Des exécutions différentes d'un même programme peuvent produire des séquences d'opérations (donc aussi des transactions) différentes
 - Plusieurs exécutions d'un même programme peuvent être présentes en même temps dans le système

Propriétés ACID

- Propriétés des transactions que le SGBD doit assurer
 - "Contrat" transactionnel entre l'utilisateur et le SGBD
- Quatre propriétés
 - *Atomicité*: une transaction s'exécute soit en totalité soit pas du tout
 - *Cohérence*: une transaction respecte les contraintes d'intégrité de la BD
 - *Isolation*: une transaction ne voit pas les effets des autres transactions qui s'exécutent en même temps
 - *Durabilité*: les effets d'une transaction validée ne sont jamais perdus

Atomicité, cohérence

- **Atomicité**: une transaction s'exécute soit en totalité soit pas du tout
 - Une exécution partielle de la transaction est inacceptable
 - Une transaction qui ne peut pas se terminer doit être annulée
 - L'annulation (*Rollback*) peut venir:
 - du programme: l'annulation du débit si solde insuffisant
 - du système: blocage ou incohérence dans l'exécution concurrente
 - Annulation → annulation de toutes les modifications (écritures)
- **Cohérence**: une transaction respecte les contraintes d'intégrité de la BD
 - Les contraintes expriment la cohérence de la BD
 - Contraintes sur les données: *compte.solde* ≥ 0 , en permanence
 - Contraintes sur les opérations: dans un transfert entre deux comptes *source.solde* + *dest.solde* = constant
 - Une transaction mène la BD d'un état cohérent vers un autre état cohérent
 - Pendant la transaction l'état peut être incohérent!

Isolation, durabilité

- **Isolation**: une transaction ne voit pas les effets des autres transactions qui s'exécutent en même temps
 - Les écritures des autres transactions en cours sur les données manipulés par la transaction en question ne sont pas visibles
 - C'est comme si chaque transaction s'exécutait toute seule dans le SGBD
 - Conséquence: l'exécution de plusieurs transactions concurrentes est équivalente à une exécution séparée, en série des transactions
- **Durabilité**: les effets d'une transaction validée ne sont jamais perdus
 - On ne doit pas annuler une transaction validée

Les problèmes de la concurrence

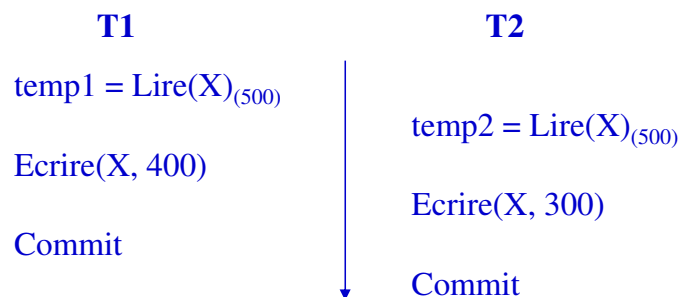
- Concurrence de transactions
 - Plusieurs transactions actives (pas terminées) en même temps
 - Vision client-serveur
 - Les transactions/programmes = clients qui envoient des demandes au SGBD (demandes de lecture ou d'écriture d'articles)
 - Le SGBD = serveur qui reçoit ces demandes et les exécute
 - Le SGBD exécute les opérations *en séquence*, pas en parallèle!
 - Concurrence = *entrelacement* des opérations des transactions
- Concurrence parfaite: *toute opération est exécutée dès son arrivée*
 - Problème: certains entrelacements ne sont pas corrects
- Contrôle de concurrence
 - Modifier l'entrelacement pour produire une exécution correcte des transactions
 - Retardement des opérations, perte de performances

Phénomènes indésirables

- Perte d'une mise à jour

Exemple: compte X avec un solde de 500 euros initialement

- T1: débit de 100 euros, T2: débit de 200 euros
- Ordre d'exécution:



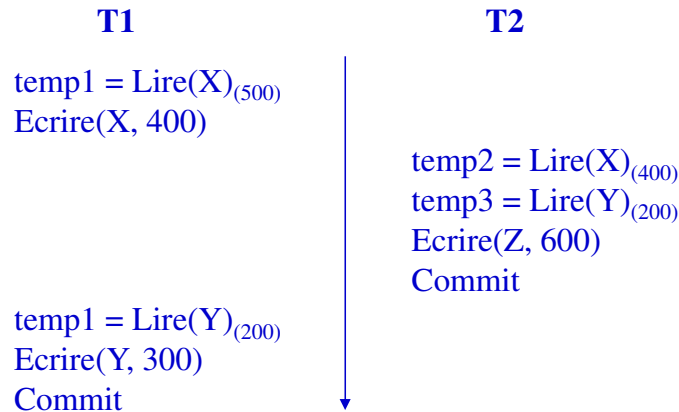
- Résultat: solde de 300 euros au lieu de 200 euros
 - Perte de la première mise à jour
- Isolation non respectée: dans T2, la lecture trouve X=500, mais avant l'écriture X=400 (lecture non répétable)

Phénomènes indésirables (suite)

- Analyse incohérente

Exemple: comptes X (500 euros) et Y (200 euros), total Z

- T1: transfert de 100 euros de X vers Y; T2: écriture de la somme X+Y dans Z



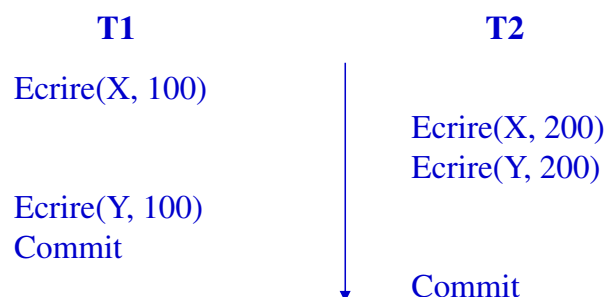
- La somme X+Y=700 avant et après le transfert, mais au milieu du transfert
→Le total calculé par T2 est incohérent
- Isolation non respectée: T2 voit la modification de X faite par T1

Phénomènes indésirables (suite)

- Écritures incohérentes

Exemple: comptes X et Y, qui doivent toujours avoir le même solde (X=Y)

- T1: initialiser X et Y avec 100 euros; T2: pareil, mais avec 200 euros



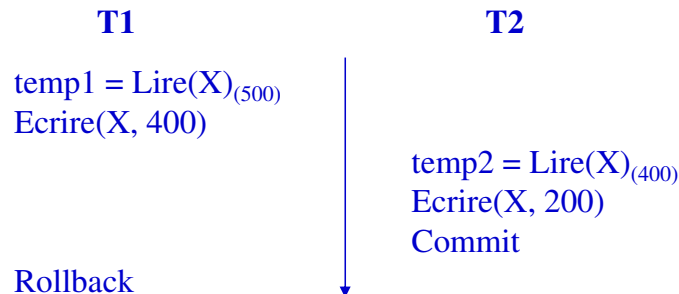
- A la fin X=200 et Y=100, bien que T1 et T2 respectent la contrainte X=Y
→Les écritures sont incohérentes, les contraintes d'intégrité ne sont pas respectées
- Isolation non respectée: T1 peut voir la modification de X faite par T2
– Écriture d'une valeur non-validée ("écriture sale")

Phénomènes indésirables (suite)

- Dépendances non-validées

Exemple: compte X avec un solde de 500 euros initialement

- T1: débit de 100 euros finalement annulé; T2: débit de 200 euros



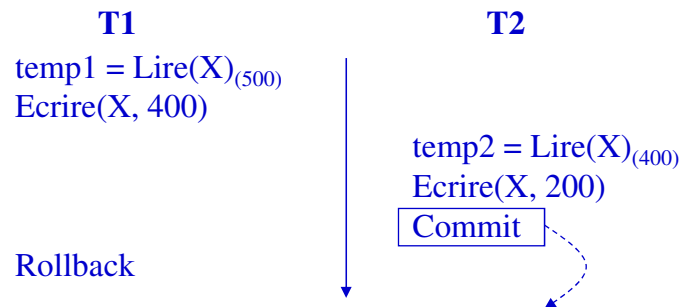
- T2 utilise la valeur X=400 écrite par T1, qui est finalement annulée
→L'annulation de T1 implique l'annulation de T2 aussi (annulation en cascade)
- Durabilité non respectée: T2, qui est validée, doit être annulée
– Cause: lecture d'une valeur non validée ("lecture sale")

Le problème des annulations

- Annulation d'une transaction T →
 - Annulation des effets de T dans la BD (annulation des écritures de T)
 - Annulation des transactions qui utilisent les valeurs écrites par T (annulations en cascade à cause des dépendances non validées)
- L'exemple des dépendances non validées montre que l'annulation d'une transaction peut avoir des effets indésirables
- Catégories d'exécutions par rapport aux annulations
 - *Non recouvrable*: permet l'annulation en cascade de transactions validées
 - *Recouvrable*: permet l'annulation en cascade, mais seulement de transactions non-validées
 - *Sans annulations en cascade*: évite les annulations en cascade
 - *Stricte*: évite les annulations en cascade et permet l'annulation des écritures à l'aide d'un journal d'écritures avant

Recouvrabilité et annulations en cascade

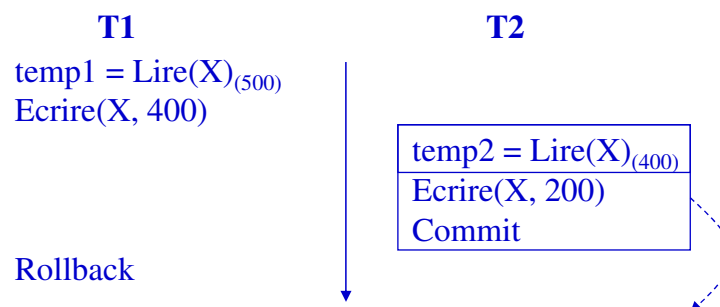
- La cause des annulations en cascade: les "lectures sales"
 - Question: quand risque-t-on d'annuler une transaction déjà validée?



- Réponse: quand la transaction qui fait la lecture sale est validée avant la fin de celle qui a fait l'écriture
- Exécution recouvrable: pas d'annulation de transaction validée
 - Méthode: retarder le *Commit* des transaction ayant fait une lecture sale
- Remarque: le caractère recouvrable ou non d'une exécution ne dépend pas de la présence explicite d'un *Rollback* !

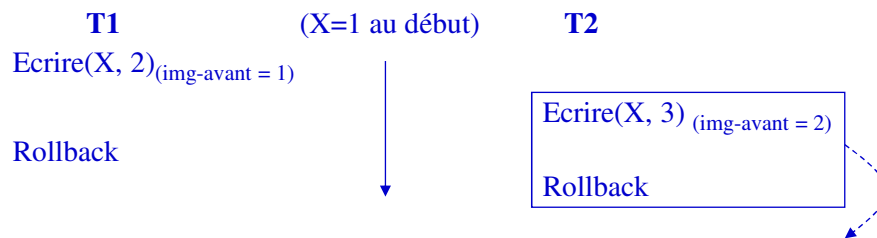
Éviter les annulations en cascade

- Pour éviter les annulations en cascade → éviter les lectures sales
 - Méthode: retarder la lecture après la fin de la transaction qui a fait l'écriture



Exécution stricte

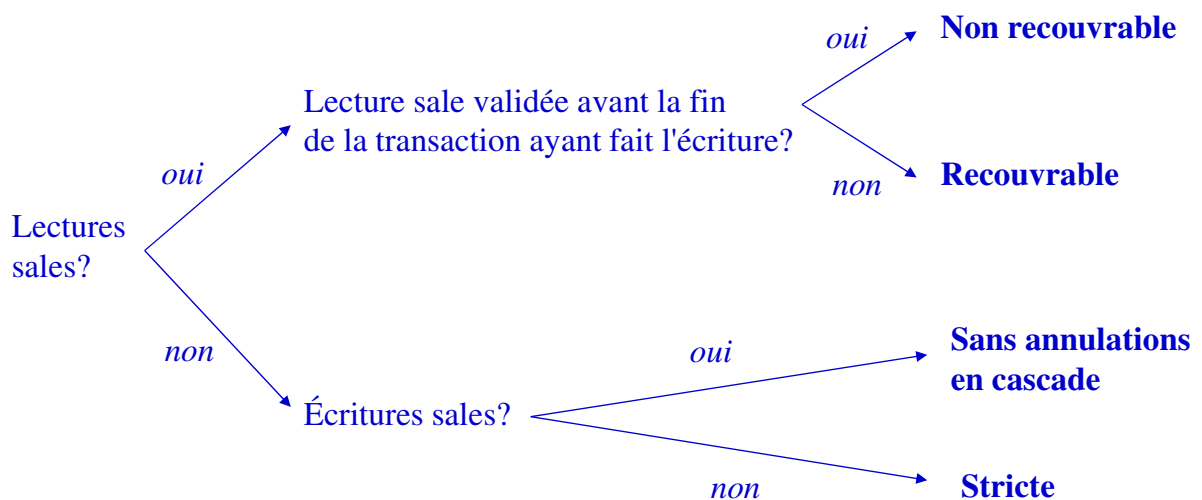
- Pour annuler une transaction il faut surtout annuler ses écritures
 - Méthode simple: utilisation d'un *journal des images-avant*
 - Avant chaque écriture on marque dans le journal la valeur modifiée
 - À l'annulation on met dans l'article son image-avant au début de la transaction
- Problème: cette méthode ne marche pas toujours!



- Le premier *Rollback* restaure $X=1$: erreur, ça ne tient pas compte de *Ecrire(X, 3)*
- Le second *Rollback* restaure $X=2$: erreur, valeur déjà annulée
- Solution: éviter les écritures sales en retardant la seconde écriture

Résumé des problèmes d'annulation

- Étant donnée une exécution, comment savoir à quelle catégorie par rapport aux annulations elle appartient?



Contrôle de concurrence

- Objectif: produire une exécution correcte, en respectant les propriétés ACID
 - Isolation : les transactions ne doivent pas voir les effets des autres transactions concurrentes
 - l'exécution doit être équivalente à une exécution séparée (en série) des transactions
 - Exécution *sérialisable* = exécution équivalente à une exécution en série *quelconque* des transactions
 - Plusieurs exécutions en série sont possibles ($n!$ pour n transactions), n'importe laquelle est acceptable
 - Que veut dire exactement "exécution équivalente"?
- Objectif contrôle de concurrence = produire des exécutions *sérialisables* en modifiant l'entrelacement des transactions
- Attention, seul l'entrelacement doit être modifié, les opérations d'une même transaction doivent préserver leur ordre relatif !

Opérations commutables / en conflit

- Opérations commutables / en conflit
 - Opérations A et B *commutables*: l'exécution A;B (A suivi de B) donne le même résultat que B;A
 - "Donner le même résultat":
 - Les valeurs finales des articles sont les mêmes
 - Les valeurs obtenues par lecture sont les mêmes
 - Opérations *en conflit*: qui ne sont pas commutables
- Remarques
 - Les opérations *d'une même transaction* ne peuvent pas changer de position relative → la question de la commutativité/conflit ne se pose pas
 - Deux opérations sur *des articles différents* sont toujours commutables
- Opérations de lecture et écriture sur un même article
 - Seules les lectures sont commutables
 - Les combinaisons *Lire-Ecrire*, *Ecrire-Lire* et *Ecrire-Ecrire* sont en conflit

Équivalence des exécutions

- Intuition: permuter deux opérations commutables dans une exécution produit une exécution équivalente
- Deux exécutions sont équivalentes si l'on peut passer de l'une à l'autre par une suite de permutations d'opérations commutables
- Méthode pour vérifier l'équivalence de deux exécutions
 - Vérifier que les exécutions portent sur les mêmes opérations/transactions
 - Trouver toutes les paires d'opérations en conflit (les mêmes dans les deux exécutions)
 - Vérifier que toutes ces paires ont le même ordre des opérations dans les deux exécutions

Notation

- Transaction T_i
 - Séquence d'opérations terminée par Commit/Rollback
 - L'indice i identifie la transaction et ses opérations
- Opérations: lire, écrire, commit, rollback
 - $l_i[x]$: lecture de l'article x dans la transaction T_i
 - $e_i[x]$: écriture de l'article x dans la transaction T_i
 - c_i : validation (commit) de la transaction T_i
 - r_i : annulation (rollback) de la transaction T_i
- Remarque: on ne s'intéresse pas aux valeurs lues/écrites
- Exemple: deux transactions de débit et leur exécution concurrente
 - $T_1: l_1[x] e_1[x] c_1$ (débit compte x) ; $T_2: l_2[y] e_2[y] c_2$ (débit compte y)
 - $H: l_1[x] l_2[y] e_1[x] e_2[y] c_2 c_1$ (une exécution concurrente de T_1 et T_2)

Exemple d'équivalence d'exécutions

- Transactions: $T_1: l_1[x] e_1[y] e_1[x] c_1$ $T_2: e_2[x] l_2[y] e_2[y] c_2$
- Exécutions
 - $H_1: \underline{l_1[x]} e_2[x] \underline{e_1[y]} l_2[y] \underline{e_1[x]} e_2[y] \underline{c_1} c_2$
 - Conflits: $l_1[x] - e_2[x], e_2[x] - e_1[x], e_1[y] - l_2[y], e_1[y] - e_2[y]$
 - $H_2: \underline{l_1[x]} \underline{e_1[y]} e_2[x] \underline{e_1[x]} \underline{c_1} l_2[y] e_2[y] c_2$
 - Conflits: $l_1[x] - e_2[x], e_2[x] - e_1[x], e_1[y] - l_2[y], e_1[y] - e_2[y]$
 - $H_1 \equiv H_2$
 - $H_3: e_2[x] l_2[y] \underline{l_1[x]} e_2[y] \underline{e_1[y]} c_2 \underline{e_1[x]} \underline{c_1}$
 - Conflits: $e_2[x] - l_1[x], e_2[x] - e_1[x], l_2[y] - e_1[y], e_2[y] - e_1[y]$
 - H_3 et H_1 ne sont pas équivalentes
 - $H_4: e_2[x] l_2[y] e_2[y] c_2 \underline{l_1[x]} \underline{e_1[y]} \underline{e_1[x]} \underline{c_1}$ (exécution en série)
 - Conflits: $e_2[x] - l_1[x], e_2[x] - e_1[x], l_2[y] - e_1[y], e_2[y] - e_1[y]$
 - $H_4 \equiv H_3$ → H_3 est sérialisable

Théorème de sérialisabilité

- Pour savoir si une exécution est sérialisable → la méthode précédente n'est pas pratique
 - Il faut prouver l'équivalence avec une exécution en série des transactions
 - ... mais il faut trouver laquelle parmi les $n!$ possibilités
- Graphe de sérialisation d'une exécution
 - Nœuds: transactions
 - Arcs: pour chaque paire d'opérations conflictuelles A_i avant B_j il y a un arc $T_i \rightarrow T_j$
- **Théorème:** une exécution H est sérialisable \Leftrightarrow son graphe de sérialisation ne contient pas de cycle
 - Dans les exemples précédents, pour les exécutions H_1, H_2, H_3, H_4 :

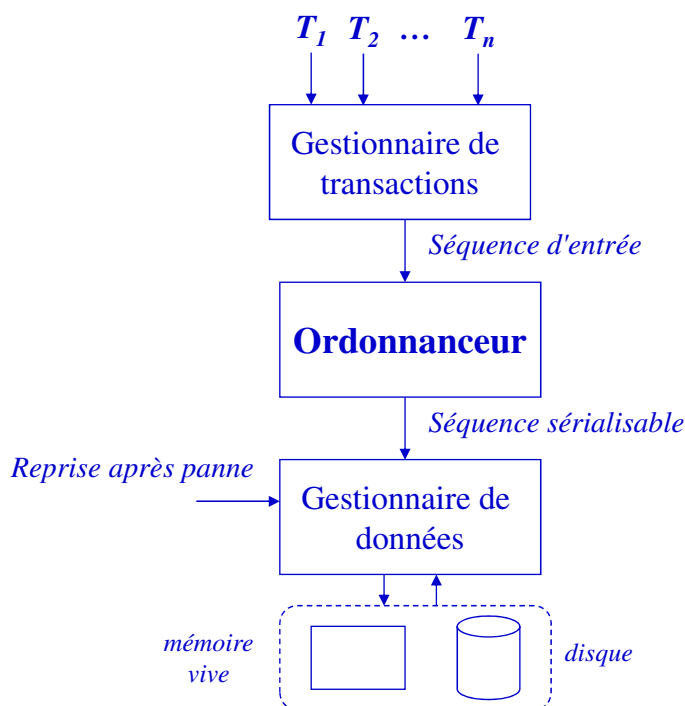
$$H_1, H_2: T_1 \longleftrightarrow T_2$$

$$H_3, H_4: T_1 \longleftarrow T_2$$

Algorithmes de contrôle de concurrence

- L'objectif du contrôle de concurrence: produire des exécutions sérialisables
 - Théorème de sérialisabilité : insuffisant, permet seulement de savoir si une exécution est sérialisable ou non
- Algorithmes de contrôle de concurrence
 - Modifient l'entrelacement des transactions pour rendre l'exécution sérialisable → réordonnancement des opérations
 - *Ordonnanceur* ("scheduler"): module logiciel qui exécute l'algorithme de contrôle de concurrence
 - Le SGBD reçoit une séquence d'exécution d'entrée que l'ordonnanceur modifie pour rendre l'exécution concurrente correcte
- Propriétés d'annulation
 - Les propriétés d'annulation sont *orthogonales* à la sérialisabilité → on peut avoir toutes les combinaisons: sérialisable mais pas recouvrable, stricte mais pas sérialisable, etc.
 - L'ordonnanceur essaie d'assurer en plus de la sérialisabilité la meilleure propriété d'annulation possible dans le cadre de son algorithme

Place de l'ordonnanceur dans le système



Verrouillage à deux phases

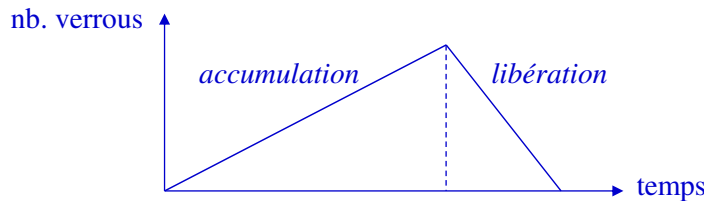
- L'algorithme de contrôle de concurrence le plus utilisé
 - Stratégie pessimiste: essaie de prévenir les incohérences en mettant en attente des opérations
- Verrous
 - Un verrou associé à chaque article, accordé aux transactions
 - Objectif: marquer les opérations réalisés sur l'article par les transactions afin de ne laisser passer que des opérations commutables
 - Une opération en conflit avec des opérations déjà acceptées des transactions en cours → bloquée au verrou
 - Verrou composé
 - Sous-verrou de lecture → partageable (les lectures sont commutables)
 - Sous-verrou d'écriture → exclusif (les écritures sont en conflit avec tout)

Algorithme de verrouillage

- L'ordonnanceur reçoit l'opération $o_i[x]$ de la transaction T_i
 - Si le verrou de x est déjà donné à une opération en conflit avec $o_i[x]$
 - opération bloquée en attente de la libération du verrou
 - Lorsqu'une opération est bloquée, toute sa transaction est bloquée!
 - Sinon, on donne à la transaction T_i le sous-verrou de x qui correspond à o
 - opération acceptée
- Un sous-verrou détenu par une transaction est libéré au plus tard à la fin de la transaction
 - Lorsqu'un sous-verrou est libéré, une des opérations en attente de ce verrou est choisie pour le prendre
 - Généralement on respecte l'ordre de blocage
 - Plusieurs opérations peuvent être débloquées si le verrou est partagé

Algorithme de verrouillage (suite)

- **Règle des deux phases:** une transaction qui libère un verrou ne pourra plus en demander d'autres pour ses opérations
 - Dans la vie d'une transaction il y a deux phases
 - Une première *d'accumulation de verrous*
 - Une seconde de *libération de verrous*



- **En pratique: libération des verrous à la fin de la transaction**
 - Cette variante produit aussi des *exécutions strictes* (empêche les lectures et les écritures sales)

Exemple de verrouillage

- **Séquence d'opérations reçue:** $l_1[x] l_2[y] e_1[y] c_1 e_2[y] c_2$
 - Conflits: $l_2[y] - e_1[y]$, $e_1[y] - e_2[y]$ forment un cycle → non sérialisable
- **Exécution**
 - $l_1[x]$: *acceptée*, sous-verrou de lecture sur x donné à T_1
 - $l_2[y]$: *acceptée*, sous-verrou de lecture sur y donné à T_2
 - $e_1[y]$: *bloquée* (conflit avec $l_2[y]$), le sous-verrou de lecture sur y donné à T_2 empêche de donner le sous-verrou d'écriture sur y à T_1
 - c_1 : *bloquée*, car le blocage de $e_1[y]$ a bloqué toute la transaction T_1
 - $e_2[y]$: *acceptée*, sous-verrou d'écriture sur y donné à T_2
 - c_2 : *acceptée*, fin de T_2 et libération de tous les verrous pris par T_2
 - déblocage de $e_1[y]$ et c_1 qui sont acceptées
- **Résultat:** $l_1[x] l_2[y] e_2[y] c_2 e_1[y] c_1$
 - Conflits: $l_2[y] - e_1[y]$, $e_2[y] - e_1[y]$ pas de cycle → sérialisable

Interblocage

- Provoqué par l'attente circulaire de verrous entre transactions
- Exemple: $l_1[x] e_2[y] e_2[x] e_1[y] c_1 c_2$
 - $l_1[x]$: T_1 obtient le sous-verrou de lecture sur x
 - $e_2[y]$: T_2 obtient le sous-verrou d'écriture sur y
 - $e_2[x]$: T_2 bloquée par T_1 , en attente du sous-verrou d'écriture sur x
 - $e_1[y]$: T_1 bloquée par T_2 , en attente du sous-verrou d'écriture sur y

→ *interblocage ("deadlock")*
- Pour éviter l'interblocage → annulation de transactions, afin de libérer des verrous

Éviter l'interblocage

- Détection
 - *Durée limite* ("timeout"): annulation d'une transaction qui dépasse la durée limite
 - Problème: paramétrage fin nécessaire pour la durée limite
 - *Graphe d'attente*: représente l'attente de verrous entre transactions
 - Interblocage = cycle dans le graphe d'attente
 - Annulation d'une transaction qui brise le(s) cycle(s)

Verrouillage hiérarchique

- Variante du verrouillage à deux phases utilisée dans les BD relationnelles
 - La différence: opérations à *plusieurs niveaux de granularité* des données
Ex. champ \in article \in table \in base de données
 - des actions sur des données différentes peuvent être en conflit
- Solution "naïve": utiliser un seul niveau
 - Niveau fin: trop de verrous, gestion lourde
 - Niveau grossier: peu de verrous, mais beaucoup de blocages
- Verrouillage hiérarchique → des verrous à tous les niveaux, chaque opération choisit le niveau approprié

Principes du verrouillage hiérarchique

- Verrouillage descendant: implicite
 - Un verrou à un niveau est implicitement valable pour toutes les données de niveau inférieur incluses
 - Ex.* Un verrou en écriture sur une table est valable pour tous les articles de la table et pour tous les champs de ces articles
- Verrouillage ascendant: verrous d'intention
 - Pour réaliser une opération à un niveau, il faut aussi vérifier s'il n'y a pas conflit avec les verrous des niveaux supérieurs
 - Ex.* Lorsqu'on veut écrire dans un article, il faut voir s'il n'y a pas des verrous en lecture/écriture sur la table
 - Méthode: utilisation de verrous supplémentaires dits *d'intention*
 - Règle: avant de réaliser une opération à un niveau, il faut demander le verrou d'intention pour l'opération au niveau supérieur

Types de verrous hiérarchiques

- Cinq types de verrous hiérarchiques
 - **S** ("shared", partagé, en lecture)
 - **X** ("exclusive", exclusif, en écriture)
 - **IS** (intention de lecture)
 - **IX** (intention d'écriture)
 - **SIX** (lecture et intention d'écriture)
 - Cas fréquent: lecture table (S) pour écrire quelque articles (IX) = update SQL
- Matrice de conflits entre verrous

	X	SIX	IX	S	IS
X	oui	oui	oui	oui	oui
SIX	oui	oui	oui	oui	<i>non</i>
IX	oui	oui	<i>non</i>	oui	<i>non</i>
S	oui	oui	oui	<i>non</i>	<i>non</i>
IS	oui	<i>non</i>	<i>non</i>	<i>non</i>	<i>non</i>

Exemple de verrouillage hiérarchique

- Table **Compte** (numéro, titulaire, solde), deux niveaux: article et table
- Transactions
 - T_1 : débit du compte numéro 7 $\rightarrow I_1[x] e_1[x] c_1$
 - T_2 : lecture de tous les comptes de la table $\rightarrow I_2[R] c_2$
 - T_3 : initialisation du compte numéro 3 $\rightarrow e_3[y] c_3$
- Ordre de réception: $I_1[x] I_2[R] e_3[y] e_1[x] c_1 c_2 c_3$
 - $I_1[x]$: obtient **IS**(R) et **S**(x) \rightarrow acceptée
 - $I_2[R]$: obtient **S**(R) (pas de conflit avec **IS**(R)) \rightarrow acceptée
 - $e_3[y]$: n'obtient pas **IX**(R) (conflit avec **S**(R)) \rightarrow bloquée, T_3 bloquée
 - $e_1[x]$: n'obtient pas **IX**(R) (conflit avec **S**(R)) \rightarrow bloquée, T_1 bloquée
 - c_1 : bloquée, car T_1 bloquée
 - c_2 : acceptée et verrous de T_2 libérés (**S**(R))
 - $e_3[y]$: obtient **IX**(R) (pas de conflit avec **IS**(R)) \rightarrow acceptée
 - $e_1[x]$: obtient **IX**(R) (pas de conflit avec **IS**(R), **IX**(R)) \rightarrow acceptée
 - c_1 : acceptée
 - c_3 : acceptée
- Résultat: $I_1[x] I_2[R] c_2 e_3[y] e_1[x] c_1 c_3$

Le problème des objets fantômes

- Problème de concurrence dans les BD dynamiques
 - Opérations de création et suppression de données
 - Cas usuel: T_1 consulte toutes les données d'un certain type pendant que T_2 crée une donnée de même type
- Exemple:
 - Tables **Compte** (numéro, titulaire, solde) et **Cumul** (titulaire, total)
 - T_1 : compare la somme des soldes des comptes de Dupont avec le cumul
 - T_2 : ajoute un nouveau compte pour Dupont

$\text{Lire}_1(\text{Compte}[5])_{(100)}, \text{Lire}_1(\text{Compte}[8])_{(300)}, \text{Lire}_1(\text{Compte}[14])_{(600)}$
 $\text{Inserer}_2(\text{Compte}[10, 'Dupont', 500])$
 $\text{Lire}_2(\text{Cumul}['Dupont'])_{(1000)}$
 $\text{Ecrire}_2(\text{Cumul}['Dupont'], 1500), \text{Commit}_2$
 $\text{Lire}_1(\text{Cumul}['Dupont'])_{(1500)}, \text{Commit}_1,$

→ T_1 trouve la somme des comptes de Dupont (1000) différente du cumul (1500)
- Similaire à l'analyse incohérente, mais produit par une insertion et non pas par une écriture

Solution au problème des objets fantômes

- Le problème: le verrouillage à deux phases ne fonctionne pas!
 - L'insertion par T_2 n'est pas bloquée par le verrouillage, car elle agit sur un article différent de ceux lus par T_1
- Solution: traiter l'insertion comme une opération sur *l'ensemble d'articles* et non pas sur un seul article
 - Dans l'exemple: verrou sur l'ensemble des comptes de Dupont
 - Lecture de tous les comptes de Dupont → lecture de l'ensemble
 - Insertion compte → écriture de l'ensemble
- En pratique: *le verrouillage hiérarchique* règle le problème!
 - Ensemble d'articles → donnée de niveau supérieur (table)
 - Lecture de tous les comptes de Dupont → lecture de la table *Compte*
 - Insertion compte → écriture de la table *Compte*
 - Inconvénient: pas très précis, car on verrouille toujours *toute la table*

Estampillage

- Algorithme plus simple que le verrouillage
 - Approche optimiste: on ne bloque pas des opérations, ... mais on doit annuler des transactions si des incohérences risquent d'apparaître
- Estampille: valeur associée à une transaction, indiquant l'ordre de démarrage des transactions
 - Produite par un compteur, par une horloge, etc.
- Idée générale: les opérations en conflit doivent passer dans l'ordre des estampilles
 - L'exécution est sérialisable, le graphe ne peut pas avoir de cycle
- Conclusions
 - Plus simple à réaliser que le verrouillage (pas de gestion de verrous)
 - Plus restrictif: impose un ordre préétabli
 - Risque d'annulations massives, annulations en cascade

Exemple d'estampillage

- Séquence reçue: $l_1[x] e_2[x] l_3[x] l_2[x] e_1[x] \dots$
 - Estampille = l'indice de la transaction
 - $l_1[x]$: acceptée
 - $e_2[x]$: en conflit avec $l_1[x]$, mais estampille plus grande → acceptée
 - $l_3[x]$: en conflit avec $e_2[x]$, mais estampille plus grande → acceptée
 - $l_2[x]$: pas de conflit avec les opérations antérieures → acceptée
 - $e_1[x]$: conflit avec $e_2[x]$ (et aussi $l_3[x]$, $l_2[x]$), estampille plus petite → rejet
 - On dit que $e_1[x]$ arrive *en retard*
 - Rejet de $e_1[x]$ → rejet de toute la transaction T_1 (atomicité) → la transaction est relancée avec *une nouvelle estampille* (pourquoi?)
- Résultat: $l_1[x] e_2[x] l_3[x] l_2[x] e_1[x] r_1 l_4[x] e_4[x] \dots$

Niveaux d'isolation

- Idée: relâcher les contraintes d'isolation afin d'augmenter la concurrence
 - L'isolation totale assure la cohérence de toute transaction
 - Certaines transactions peuvent accepter un niveau moindre d'isolation
- SQL: quatre niveaux d'isolation - évitent des phénomènes indésirables
 - *Read uncommitted*: les écritures bloquent les écritures, mais pas les lectures
 - Permet des *lectures sales* (de valeurs non validées)
 - *Read committed*: les écritures bloquent aussi les lectures
 - Pas de lectures sales, mais permet la *lecture non répétable* (les lectures n'empêchent pas les écritures)
 - *Repeatable read*: les lectures bloquent les écritures
 - Lecture répétable, mais permet des *objets fantômes*
 - *Serializable*: isolation totale, évite aussi les objets fantôme
- Commande: SET TRANSACTION ISOLATION LEVEL xxx
- Remarque: la norme SQL ne prévoit pas de verrouillage explicite, mais seulement l'utilisation des niveaux d'isolation

Exemple niveaux d'isolation

- Deux transactions sur un article x (au début $x = 1$)
 - T_1 : $l_1[x]$ $l_1[x]$ c_1 - deux consultations successives de x
 - T_2 : $l_2[x]$ $e_2[x]$ $e_2[x]$ c_2 - ajoute 1 à x à deux reprises
 - Séquence reçue: $l_1[x]$ $l_2[x]$ $e_2[x]$ $l_1[x]$ $e_2[x]$ c_2 c_1
- Read uncommitted $\rightarrow l_1[x]_{(1)}$ $l_2[x]_{(1)}$ $e_2[x]_{(2)}$ $\underline{l_1[x]}_{(2)}$ $e_2[x]_{(3)}$ c_2 c_1
 - Pas de retard, lectures sales – peut être acceptable si T_1 n'écrit pas
- Read committed $\rightarrow l_1[x]_{(1)}$ $l_2[x]_{(1)}$ $e_2[x]_{(2)}$ $e_2[x]_{(3)}$ c_2 $\underline{l_1[x]}_{(3)}$ c_1
 - Évite lecture sale – lecture retardée après la validation de l'écriture
- Repeatable read $\rightarrow l_1[x]_{(1)}$ $l_2[x]_{(1)}$ $\underline{l_1[x]}_{(1)}$ c_1 $e_2[x]_{(2)}$ $e_2[x]_{(3)}$ c_2
 - En plus: écritures des autres transactions retardées après la fin de T_1
- Serializable
 - En plus: insertions/suppressions des autres transactions retardées après T_1

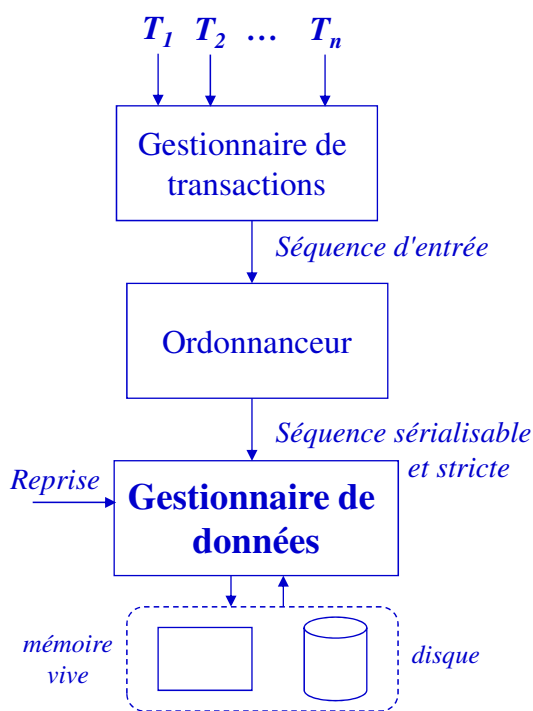
Point de sauvegarde

- Point de sauvegarde ("savepoint"): notion prévue dans SQL
 - Validation partielle d'une transaction
 - *Savepoint s*
 - Principale utilisation: faire une *annulation partielle* de la transaction
 - *Rollback to s*
 - Utile pour des "transactions longues", où en cas d'échec on veut garder une partie du travail réalisé
- Exemple: réservation d'un voyage
 - Réservation avion
 - Savepoint* avion
 - Réservation voiture chez Loueur1
 - si échec alors
 - Rollback to* avion
 - Réservation voiture chez Loueur2
 - si échec alors *Rollback*
 - Commit*

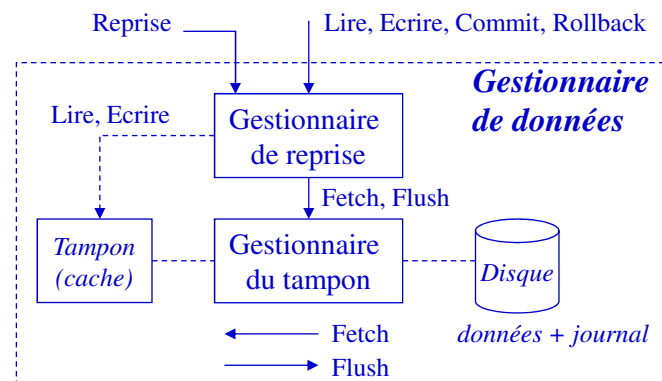
Reprise après panne

- Types de pannes
 - *De transaction*: annulation d'une transaction
 - *De système*: perte du contenu de la mémoire vive
 - *De support*: perte du contenu du disque
- Hypothèses
 - On s'intéresse aux pannes de système
 - opération *Reprise* qui doit reconstituer un état cohérent de la BD
 - L'ordonnanceur produit des exécutions sérialisables et *strictes*
 - On considère toutes les opérations *au niveau des articles*, même si les échanges avec le disque se font au niveau des pages

Architecture



- Gestionnaire de données
 - Gestionnaire du tampon (cache)
 - Échanges entre mémoire vive et disque
 - Gestionnaire de reprise
 - Pilote l'exécution des opérations des transactions + *Reprise*
 - Utilise le gestionnaire du tampon



Objectif de la reprise

- Au moment de la panne, l'état de la BD peut être incohérent
 - Pendant l'exécution d'une transaction: état incohérent possible
- État validé de la BD = dernière valeur validée pour chaque article de la BD
 - Dernière valeur validée pour un article = dernière valeur écrite dans l'article par une transaction validée
- Objectif → *ramener la BD à son état validé avant la panne*
- Comment?
 - Assurer l'*atomicité*: annuler dans la BD les transactions non validées au moment de la panne
 - Assurer la *durabilité*: reproduire dans la BD l'effet des transactions validées au moment de la panne
- Pour assurer la reprise → *journal* à stocker sur disque

Gestionnaire du tampon

- Tampon (cache) divisé en cellules (slots)
 - Une cellule peut être vide ou contenir un article
 - En réalité les cellules stockent *des pages*, contenant des articles
 - Bit de consistance: article du tampon consistant ou non avec le disque
 - Répertoire du cache: correspondance article → cellule

Tampon (cache)

n°	bit consist.	valeur article
1	1	"Marie"
2	–	–
3	0	15
...

Répertoire du cache

article	n° cellule
x	3
y	1
...	...

Opérations dans le tampon

- **Flush** (c) : écriture du contenu de la cellule c sur disque
 - si c inconsistante **alors** copier c sur disque et rendre c consistante
 - sinon** rien
- **Fetch** (x) : transfert de l'article x du disque vers le tampon
 - si tampon plein **alors**
 - vider une cellule après l'avoir rendue consistante avec *Flush*
 - sélectionner c une cellule vide
 - copier x du disque dans c et rendre c consistante
 - mettre à jour le répertoire du cache avec (x, c)
- Politique du pour choisir la cellule à vider: LRU, FIFO, etc.
- **Lire** et **Ecrire**: toujours dans le tampon
 - *Lire*(x): si x n'est pas dans le tampon, alors *Fetch*(x) d'abord
 - *Ecrire*(x):
 - si x n'est pas dans le tampon, alors on lui alloue une cellule c dans le tampon
 - écrire la valeur de x dans c et rendre c inconsistante
 - éventuellement *Flush*(c), selon l'algorithme du gestionnaire de reprise

Journalisation

- Objectif: garder sur disque une trace des écritures, pour pouvoir au moment de la reprise:
 - Refaire les écritures validées
 - Annuler les écritures non validées
- Journal
 - Liste ordonnée des opérations de mise à jour jusqu'au moment courant
 - Ensembles de transactions validées, annulées, en cours au moment courant
- Exemple de journal
 - $[T_1, x, 4], [T_2, y, 3], [T_1, z, 1], [T_2, x, 8], [T_3, y, 5], [T_4, x, 2], [T_3, z, 6]$
 - *validé* = $\{T_1, T_4\}$
 - *annulé* = $\{T_2\}$
 - *encours* = $\{T_3\}$

Types de journal

- Journaux d'images après et d'images avant
 - *Journal d'images après*: valeurs après modification (valeurs écrites)
 - Pour refaire des écritures
 - *Journal d'images avant*: valeurs avant modification
 - Pour annuler des écritures

Remarques:

 - On peut utiliser un journal d'images après pour trouver une image avant
 - On peut fusionner les deux types de journal en un seul journal
- Journaux physiques et logiques
 - *Journal physique*: on stocke les valeurs écrites
 - *Journal logique*: on stocke des opérations de plus haut niveau, avec leurs paramètres
 - Pour les images avant: on stocke l'opération inverse
 - Taille plus petite, mais plus difficile à réaliser et à interpréter

Recyclage de l'espace utilisé par le journal

- Journal physique d'images après, avec entrées de type $[T_i, x, v]$
- Une entrée $[T_i, x, v]$ peut être recyclée si
 - Soit T_i annulée ($T_i \in \text{annulé}$),
 - Soit T_i validée ($T_i \in \text{validé}$), mais une autre T_j validée a écrit x par la suite
- Explication
 - À la reprise, les écritures annulées ne sont plus nécessaires
 - À la reprise, seule la dernière écriture validée sur x est nécessaire

Gestionnaire de reprise

- Rôles du gestionnaire de reprise (GR)
 - Réalisation des opérations dans la BD
 - Validation et annulation des transactions dans la BD
 - Gestion de la journalisation
 - Gestion des échanges tampon – disque
 - Réalisation de la reprise
- Deux paramètres qui définissent l'algorithme du GR
 - Forcer ou non l'écriture sur disque des modifications dans le tampon
 - La façon d'enregistrer les écritures dans le journal

Politiques d'écriture sur disque

- Écrire des valeurs non validées sur disque
 - Permettre (politique "*Steal*")
 - Annuler à la reprise sur disque les écritures non validées
 - Plus rapide en fonctionnement normal, mais plus coûteux à la reprise
 - Cas typique: le GR ne force jamais le Flush
 - Question: comment les valeurs non validées se retrouvent sur disque?
 - Ne pas permettre (politique "*No-steal*")
 - Reprise plus rapide, mais plus coûteux en fonctionnement normal
- Écrire toutes les valeurs validées sur disque au *Commit*
 - Pas de contrainte (politique "*No-force*")
 - Répéter à la reprise des écritures validées
 - Cas typique: le GR qui ne force jamais le Flush
 - Toujours (politique "*Force*")
 - Plus coûteux en fonctionnement normal, mais pas de répétition à la reprise
- Quatre types de GR: les combinaisons *Steal/No-steal* – *Force/No-force*

Algorithme *Steal* / *No-force*

- Le plus performant en fonctionnement normal
 - Toutes les opérations se font dans le tampon, le GR ne demande jamais de *Flush* → minimise les entrées/sorties
 - Écriture sur disque → lorsque le gestionnaire du tampon vide une cellule
 - Conséquences
 - On ne contrôle pas ce qui est écrit sur disque → des valeurs non validées peuvent se retrouver sur disque (*steal*)
 - Les valeurs validées restent dans le tampon, pas d'écriture sur disque de ces valeurs au *Commit* (*no-force*)
- La reprise demande à la fois *annulation* et *répétition* d'écritures
- Hypothèses
 - Journal physique d'images après/avant avec listes de transactions
 - Écriture dans le journal juste avant l'écriture dans le tampon

Steal / No-force : opérations

- *Ecrire* (T_i, x, v)
 $encours = encours \cup \{T_i\}$
 $journal = journal + [T_i, x, v]$
si x n'est pas dans le tampon **alors** allouer cellule pour x dans le tampon
 $cellule(x) = v$
- *Lire* (T_i, x)
 $encours = encours \cup \{T_i\}$
si x n'est pas dans le tampon **alors** $Fetch(x)$
return $cellule(x)$
- *Commit* (T_i)
 $validé = validé \cup \{T_i\}$
 $encours = encours - \{T_i\}$
- *Rollback* (T_i)
pour chaque x écrit par T_i **répéter**
si x n'est pas dans le tampon **alors** allouer cellule pour x dans le tampon
 $cellule(x) = image-avant(x, T_i)$
fin répéter
 $annulé = annulé \cup \{T_i\}$
 $encours = encours - \{T_i\}$

Steal / No-force : reprise

- *Reprise* ($$)
marquer toutes les cellules du tampon comme étant vides
 $écriture-répétée = \{ \}$ //articles restaurés par répétition
 $écriture-annulée = \{ \}$ //articles restaurés par annulation
pour chaque $[T_i, x, v] \in journal$ à partir de la fin **répéter**
si $x \notin écriture-répétée \cup écriture-annulée$ **alors**
si x pas dans le tampon **alors** allouer cellule pour x dans le tampon
si $T_i \in validé$ **alors** //répétition de l'écriture
 $cellule(x) = v$
 $écriture-répétée = écriture-répétée \cup \{x\}$
sinon //annulation de l'écriture
 $cellule(x) = image-avant(x, T_i)$
 $écriture-annulée = écriture-annulée \cup \{x\}$
fin si
si $écriture-répétée \cup écriture-annulée = BD$ **alors stop boucle**
fin répéter

Algorithme Steal / Force

- La différence avec Steal / No-force : force au *Commit* l'écriture sur disque des valeurs validées
 - La reprise ne demande pas de *répétition* d'écritures
 - *Ecrire, Lire* et *Rollback*: pareil que pour Steal / No-force
 - *Commit*: en plus
 - pour chaque** x écrit par T_i **répéter**
 - si** x est dans le tampon **alors** $Flush(x)$
 - fin répéter**
 - *Reprise*: pareil, sauf que *écriture-répétée* et la branche "**si** $T_i \in \text{validé}$ **alors**" n'existent pas

Algorithme No-steal / No-force

- Ne permet pas des écritures non-validées sur disque
 - La reprise ne demande pas d'*annulation* d'écritures
- Problème: le tampon a un fonctionnement peu contrôlable
 - Idée: ne pas écrire dans le tampon, mais *utiliser le journal*
- Opérations
 - *Ecrire*: ajoute juste $[T_i, x, v]$ au journal
 - *Lire*: si T_i a déjà écrit x alors chercher la valeur dans le journal, sinon dans la BD
 - Exécution stricte → si T_i a écrit x alors les lectures/écritures de x par d'autres transactions sont retardées après la fin de T_i
 - *Commit*: chaque x écrit par T_i est calculé à partir du journal et écrit dans le tampon (no-force)
 - *Rollback*: rien à annuler, juste ajouter T_i à *annulé*
- Algorithme No-steal / Force : similaire, mais au *Commit* on écrit les valeurs validées aussi sur disque (avec *Flush*)

Concurrence et reprise dans Oracle

- Contrôle de concurrence *multi-version*
 - Technique qui améliore la concurrence en diminuant les blocages
 - Idée: pouvoir accéder à des versions antérieures des données
 - Comment? → en utilisant le journal
 - Mode de fonctionnement
 - Ordonnancement des transactions de type estampillage
 - Chaque version de la donnée dans le journal possède une estampille
 - SCN = system change number
 - Une lecture $l_i[x]$ *n'attend jamais!*
 - Elle peut utiliser la dernière version de x qui respecte le niveau d'isolation
 - Dernière valeur validée (niveau « read committed »)
 - Valeur en début de la transaction (niveau « serializable »)
 - Une écriture $e_i[x]$ n'est pas bloquée par les lectures
 - Cela crée tout simplement une nouvelle version de x
- Plus de concurrence, pas de blocage lecture-écriture ou écriture-lecture

Niveaux d'isolation

- Isolation avec le contrôle multi-version
 - La lecture peut choisir la dernière version validée
 - facile à assurer le niveau "read committed"
 - Les lectures d'une transaction peuvent utiliser la même version (celle du début)
 - facile à assurer la lecture répétable dans une transaction
- Oracle n'offre que *deux niveaux d'isolation* + niveau *Read-Only*
 - Read committed
 - Toute lecture choisit la dernière version *validée* avant son estampille
 - Une écriture $e_i[x]$ est bloquée par une écriture non validée $e_j[x]$, mais sera débloquée à la fin de T_j par validation ou annulation
 - Serializable
 - Toutes les lectures d'une transaction voient *l'état validé de la BD au début de la transaction* + les modifications apportées par la transaction elle-même
 - Une écriture $e_i[x]$ est bloquée par une écriture non validée $e_j[x]$, et
 - Sera débloquée si T_j est annulée, ou
 - Produira une exception si T_j est validée

Verrous

- Verrous hiérarchiques à deux niveaux:
 - Ligne (article): X (pas de S, les lectures ne bloquent pas)
 - Table: IS, IX, S, SIX, X
- Verrouillage implicite
 - SELECT ... FROM ... WHERE: aucun verrou
 - INSERT/UPDATE/DELETE: X (ligne), IX (table)
 - SELECT ... FOR UPDATE: X (ligne), IS (table)
- Verrouillage explicite : seulement au niveau des tables
 - LOCK TABLE *nom* IN *type* MODE
 - *type*: ROW SHARE (IS), ROW EXCLUSIVE (IX), SHARE (S), SHARE ROW EXCLUSIVE (SIX), EXCLUSIVE (X)

Reprise dans Oracle

- Reprise: pour les pannes de support ou pannes système
- Algorithme Steal / No-force
- Journalisation
 - Journal d'images après (redo log)
 - Journal d'images avant (undo segment)
- Reprise en deux étapes
 - Répétition des écritures (journal d'images après)
 - Annulation des écritures non validées (journal d'images avant)
- Retrouver des versions plus anciennes des données
 - Flashback: reconstitution de la BD à un moment donné du passé
 - CREATE TABLE *nom* AS SELECT ... AS OF TIMESTAMP *t*