Transactions et Concurrence d'accès dans les bases de données

BDD Avancées

Nicolas Travers
DUT Informatique – 2° année

CNAM
nicolas.travers (at) cnam.fr

Plan

Problématique

- Une Base de Données - Plusieurs accès simultanés
  - Opérations : Lecture (LR), Écriture (LMD)
  - Programmes (ensemble d'opérations) : Transaction
  - Problème : Mêmes données accédées

Exemple

- **Programme** : Réservation de place d'avion
  1. Consultation des vols disponibles
  2. Consultation des places libres
  3. Réservation d'une place
  4. Paiement
  5. Confirmation
- Variante : Annulation, Réserver plusieurs places...
- Problème : Deux utilisations du programme sur le même vol. Qui peut réserver sa place ? Quelles sont les places disponibles ?

Transaction

Transaction

Séquence d'opérations de lectures et d'écritures sur des données dont l'ordre est immuable et se terminant par une validation ou une annulation.

- Opérations :
  - Lecture : $r_i[x]$ (read)
  - Écriture : $w_i[x]$ (write)
  - Numéro de transaction "$_i$"
  - Données à lire/écrire "x"
- Validation / Annulation des effets de la transaction
  - Validation : Commit (c_i)
  - Annulation : Abort (a_i)
- Exemple :
  - T_1 : $r_1[x]$ $r_1[y]$ $w_1[y]$ c_1
  - T_2 : $r_2[z]$ $r_2[x]$ a_2

Concurrence

Histoire

Une *Histoire* $\mathcal{H}$ est une séquence d'opérations provenant de plusieurs transactions.

Histoire concurrentielle

$\mathcal{H}$ est dites concurrentiel si les opérations simultanées sur les mêmes données.

- $\mathcal{H}$ concurrente : $r_1[x]\ r_2[z]\ r_1[y]\ w_1[y]\ r_2[y]\ c_1\ a_2$
- $\mathcal{H}$ non-concurrente : $r_1[x]\ r_2[z]\ r_1[y]\ w_1[y]\ r_2[z]\ c_1\ a_2$

Validation / Annulation

- **Commit** : Validation des effets de la transaction dans la base de données
  - Les opérations d'écritures doivent être effectives et visibles par les autres transactions
- **Abord** (ou Rollback) : Annulation des effets de la transaction
  - Annule tout effet d'écriture sur les données ou les transactions concurrentes

Propriétées ACID

- **A***tomicité* : Transaction exécutée en totalité (pas d'exécution partielle)
  - une transaction interrompue doit être annulée (Abort)
- **C***ohérence* : respect des contraintes d'intégrité sur les données
  - solde(compte) ≥ 0 ; solde(source) + solde(dest) = *const*
  - une transaction modifie la BD d'un état initial cohérent à un état final cohérent
  - pendant la transaction, l'état peut être incohérent !
- **I***solation* : Pas d'échanges entre les transactions
  - la transaction s'exécute comme si elle était seule
  - objectif : exécution concurrente des transactions équivalente à une exécution en série (non-concurrente)
- **D***urabilité* : les effets d'une transaction validée par Commit sont permanents
  - on ne doit pas annuler une transaction validée

Mise-à-jour de la BD

Deux variantes :

- *immédiate* : modification immédiate de la BD
  - Modifications directement visibles et utilisables
  - ⚠ Beaucoup de concurrence
- *différée* : chaque transaction travaille sur des copies, avec mise-à-jour de la BD à la fin de la transaction
  - Pas de concurrence
  - ⚠ Modifications appliquées ou annulation de la transaction (relancée)

Types de problèmes

Exécution d'une histoire peut provoquer des incohérences

1. Lecture sale
   - Dépendances non validées
   - Lecture non validés
2. Écriture sale
3. Annulation en cascade
4. Exécution stricte
5. Tuple fantôme

Lectures Sales - Dépendance non-validées

Opérations de crédits (sur le compte x)
P1 : *Crédit(x, 100)* $\rightarrow$ T_1 : $r_1[x]$ $w_1[x]$ c_1
P2 : Crédit(x, 50) $\rightarrow$ T_2 : r_2[x] w_2[x] c_2
au début, x=200

$$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ c_2\ a_1$$

BD	P1	P2
x = 200		

Concurrence Intro Pb MAJ Ctrl Isolation Algo

Lectures Sales - Dépendance non-validées

Opérations de crédits (sur le compte x)
P1 : *Crédit(x, 100)* $\rightarrow$ T_1 : $r_1[x]$ $w_1[x]$ c_1
P2 : Crédit(x, 50) $\rightarrow$ T_2 : r_2[x] w_2[x] c_2
au début, x=200

$$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ c_2\ a_1$$

BD	P1	P2
x = 300	x=300	

Lectures Sales - Dépendance non-validées

Opérations de crédits (sur le compte x)
P1 : *Crédit(x, 100)* $\rightarrow$ T_1 : $r_1[x]$ $w_1[x]$ c_1
P2 : Crédit(x, 50) $\rightarrow$ T_2 : r_2[x] w_2[x] c_2
au début, x=200

$$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ c_2\ a_1$$

BD	**P1**	**P2** Commit
x = 350	x=300	x=350

Lectures Sales - Dépendance non-validées

Opérations de crédits (sur le compte x)
P1 : *Crédit(x, 100)* $\rightarrow$ T_1 : $r_1[x]$ $w_1[x]$ c_1
P2 : Crédit(x, 50) $\rightarrow$ T_2 : r_2[x] w_2[x] c_2
au début, x=200

$$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ w_2[x]\ c_2\ a_1$$

BD	**P1** Rollback	**P2** Commit
x = 200	x=300	x=350

- Mise à jour w_2[x] perdue (écrasée par l'annulation a_1)
- Solution : Retarder le "commit/rollback" c_2 (après le a_1)

$\Rightarrow$ `READ UNCOMMITTED`

Lectures Sales - Lecture non-validée

Opérations de transfert d'argent (50 de x vers y), et de calcul de somme (z = x + y)
P1 : *Transfert* T_1 : $r_1[x]$ $w_1[x]$ $r_1[y]$ $w_1[y]$ c_1
P2 : *Somme* T_2 : $r_2[x]$ $r_2[y]$ $w_2[z]$ c_2
Au début, x=200, y=100, z=0

$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ r_2[y]\ w_2[z]\ c_2\ r_1[y]\ w_1[y]\ c_1$

BD	P1	P2
x = 200 y = 100 z = 0	x=200	

Lectures Sales - Lecture non-validée

Opérations de transfert d'argent (50 de x vers y), et de calcul de somme (z = x + y)
P1 : *Transfert* T_1 : $r_1[x]$ $w_1[x]$ $r_1[y]$ $w_1[y]$ c_1
P2 : *Somme* T_2 : $r_2[x]$ $r_2[y]$ $w_2[z]$ c_2
Au début, x=200, y=100, z=0

$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ r_2[y]\ w_2[z]\ c_2\ r_1[y]\ w_1[y]\ c_1$

BD	P1	P2
x = 150 y = 100 z = 0	x=150	

Lectures Sales - Lecture non-validée

Opérations de transfert d'argent (50 de x vers y), et de calcul de somme (z = x + y)
P1 : *Transfert* T_1 : $r_1[x]$ $w_1[x]$ $r_1[y]$ $w_1[y]$ c_1
P2 : *Somme* T_2 : $r_2[x]$ $r_2[y]$ $w_2[z]$ c_2
Au début, x=200, y=100, z=0

$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ r_2[y]\ w_2[z]\ c_2\ r_1[y]\ w_1[y]\ c_1$

BD	P1	P2
x = 150 y = 100 z = 250	x=150	x=150 y=100 z=250

Lectures Sales - Lecture non-validée

Opérations de transfert d'argent (50 de x vers y), et de calcul de somme (z = x + y)
P1 : *Transfert* T_1 : $r_1[x]$ $w_1[x]$ $r_1[y]$ $w_1[y]$ c_1
P2 : *Somme* T_2 : $r_2[x]$ $r_2[y]$ $w_2[z]$ c_2
Au début, x=200, y=100, z=0

$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ r_2[y]\ w_2[z]\ c_2\ r_1[y]\ w_1[y]\ c_1$

BD	P1	P2
x = 150 y = 100 z = 250	x=150 y=100	x=150 y=100 z=250

- Incohérence de lecture d'une donnée modifiée mais non-validée ("lecture sale") - $w_1[x]$ $r_2[x]$
- Solution possible : retarder la "lecture" $r_2[x]$ après le commit c_1

⇒ `READ COMMITTED`
⚠ Annulation en cascade (T_2 annulée si a_1)

Lectures Sales - Lecture non-validée

Opérations de transfert d'argent (50 de x vers y), et de calcul de somme (z = x + y)
P1 : *Transfert* T_1 : $r_1[x]$ $w_1[x]$ $r_1[y]$ $w_1[y]$ c_1
P2 : *Somme* T_2 : $r_2[x]$ $r_2[y]$ $w_2[z]$ c_2
Au début, x=200, y=100, z=0

$\mathcal{H} = r_1[x]\ w_1[x]\ r_2[x]\ r_2[y]\ w_2[z]\ c_2\ r_1[y]\ w_1[y]\ c_1$

BD	P1	P2
x = 150 y = 150 z = 250	x=150 y=150	x=150 y=100 z=250

- Incohérence de lecture d'une donnée modifiée mais non-validée ("lecture sale") - $w_1[x]$ $r_2[x]$
- Solution possible : retarder la "lecture" $r_2[x]$ après le commit c_1

⇒ `READ COMMITTED`
⚠ Annulation en cascade (T_2 annulée si a_1)

Écritures sales

Opérations de crédits (sur le compte x)
P1 : *Crédit(x, 100)* → T_1 : $r_1[x]$ $w_1[x]$ c_1
P2 : Crédit(x, 50) → T_2 : $r_2[x]$ $w_2[x]$ c_2
Au début, x=200

$\mathcal{H} = r_1[x]\ r_2[x]\ w_1[x]\ w_2[x]\ c_1\ c_2$

BD	P1	P2
x = 200	x=200	x=200

Écritures sales

Opérations de crédits (sur le compte x)
P1 : *Crédit(x, 100)* $\rightarrow$ T_1 : $r_1[x]$ $w_1[x]$ c_1
P2 : Crédit(x, 50) $\rightarrow$ T_2 : r_2[x] w_2[x] c_2
Au début, x=200

$$\mathcal{H} = r_1[x]\ r_2[x]\ w_1[x]\ w_2[x]\ c_1\ c_2$$

BD	P1	P2
x = 300	x=300	x=200

Écritures sales

Opérations de crédits (sur le compte x)
P1 : *Crédit(x, 100)* $\rightarrow$ T_1 : $r_1[x]$ $w_1[x]$ c_1
P2 : Crédit(x, 50) $\rightarrow$ T_2 : r_2[x] w_2[x] c_2
Au début, x=200

$$\mathcal{H} = r_1[x]\ r_2[x]\ w_1[x]\ w_2[x]\ c_1\ c_2$$

BD	P1	P2
x = 250	x=300	x=250

- Mise à jour w_1[x] perdue (écrasée par w_2[x]), lecture r_1[x] n'est plus répétable après le w_2[x]
- Solution : retarder l'écriture w_2[x] après "commit/rollback" c_1

$\Rightarrow$ `REPEATABLE READ`

Objets Fantômes

Opérations de sommes des comptes (x,y) et d'ajout d'un compte (z) pour une même personne
P1 : *Somme* $\rightarrow$ T_1 : $r_1[x]$ $r_1[y]$ c_1
P2 *:* Création $\rightarrow$ T_2 : $w_2[z]$ c_2

$$\mathcal{H} = r_1[x]\ r_1[y]\ w_2[z]\ c_2\ c_1$$

- Pas de concurrence directe
- Concurrence logique (liste des comptes de la personne : x, y, z, à la fin de T_2)
- Le tuple 'z' n'existe pas au début de T_1
- Solution : Isolation logique ou globale des données

$\Rightarrow$ `SERIALIZABLE`

Réordonnancement des transactions

- *Solution* : Algorithmes de réordonnancement des opérations
- Contrainte de temps et d'ordre à respecter
  - Une opération déjà effectuée ne peut être déplacée
  - La séquence d'opération d'une transaction doit être respectée
  - ⇒ La nouvelle histoire doit être équivalente

⇒ Retarder les opérations causant les problèmes

- *Objectif* : un maximum de concurrence, donc un minimum de retards
- *Idéal* : Produire une histoire *équivalente et sérialisable*

Sérialisabilité

Exécution sérialisable

$\mathcal{H}$ est sérialisable si elle est équivalente à une exécution en série *quelconque* des transactions.
⇒ Les effets (lectures/écritures) de l'histoire sont identiques à une exécution en série.

Exemple :
T_1= crédit sur x ; T_2= transfert de y vers x
Série : $\mathcal{H}_1 = r_1[x]\ w_1[x]\ c_1\ r_2[y]\ w_2[y]\ r_2[x]\ w_2[x]\ c_2$
Sérialisable : $\mathcal{H}_2 = r_1[x]\ r_2[y]\ w_1[x]\ w_2[y]\ r_2[x]\ c_1\ w_2[x]\ c_2$

Équivalence d'histoires

Equivalence de deux histoires

1. Les transactions sont identiques
2. La liste des conflits sont équivalentes
3. Le graphe de sérialisation est acyclique

Conflits

p_i[x] et **q_j[y]** sont en conflit $\Longleftrightarrow$

- i$\neq$j, x=y (transactions différentes, même enregistrement)
- p_i[x] q_j[x] n'a pas le même effet que q_j[x] p_i[x]

Graphe de sérialisation

- *Noeuds* : transactions T_i validées dans $\mathcal{H}$
- *Arcs* : si conflits p_i[x]-q_j[x] $\Rightarrow$ arc $T_i \rightarrow T_j$

Équivalence d'histoires : Exemple

- $\mathcal{H}_1$= r_1[x] w_1[x] c_1 r_2[y] w_2[y] r_2[x] w_2[x] c_2
  - T_1 : r_1[x] w_1[x] c_1, T_2 : r_2[y] w_2[y] r_2[x] w_2[x] c_2
  - Conflits : r_1[x]-w_2[x], w_1[x]-r_2[x], w_1[x]-w_2[x]
- $\mathcal{H}_2$= r_1[x] r_2[y] w_1[x] w_2[y] c_1 r_2[x] w_2[x] c_2
  - T_1 : r_1[x] w_1[x] c_1, T_2 : r_2[y] w_2[y] r_2[x] w_2[x] c_2
  - Conflits : r_1[x]-w_2[x], w_1[x]-r_2[x], w_1[x]-w_2[x]
- $\mathcal{H}_3$= r_1[x] r_2[y] w_2[y] r_2[x] w_1[x] c_1 w_2[x] c_2
  - T_1 : r_1[x] w_1[x] c_1, T_2 : r_2[y] w_2[y] r_2[x] w_2[x] c_2
  - Conflits : r_1[x]-w_2[x], r_2**[x]**-w_1**[x]**, w_1[x]-w_2[x]
- $\mathcal{H}_4$= r_1[x] w_2[y] r_2[y] w_1[x] c_1 r_2[x] w_2[x] c_2
  - T_1 : r_1[x] w_1[x] c_1, T_2 : w_2**[y]** r_2**[y]** r_2[x] w_2[x] c_2

Équivalence d'histoires : Exemple

- $\mathcal{H}_1$= $r_1[x]$ $w_1[x]$ c_1 $r_2[y]$ $w_2[y]$ $r_2[x]$ $w_2[x]$ c_2
  - T_1 : $r_1[x]$ $w_1[x]$ c_1, T_2 : $r_2[y]$ $w_2[y]$ $r_2[x]$ $w_2[x]$ c_2
  - Conflits : $r_1[x]$-$w_2[x]$, $w_1[x]$-$r_2[x]$, $w_1[x]$-$w_2[x]$
- $\mathcal{H}_2$= $r_1[x]$ $r_2[y]$ $w_1[x]$ $w_2[y]$ c_1 $r_2[x]$ $w_2[x]$ c_2
  - T_1 : $r_1[x]$ $w_1[x]$ c_1, T_2 : $r_2[y]$ $w_2[y]$ $r_2[x]$ $w_2[x]$ c_2
  - Conflits : $r_1[x]$-$w_2[x]$, $w_1[x]$-$r_2[x]$, $w_1[x]$-$w_2[x]$
- $\mathcal{H}_1$ et $\mathcal{H}_2$ sont équivalentes
- Graphe de sérialisation : $T_1 \rightarrow T_2$
  - $\mathcal{H}_1$ est une exécution en série

  $\Rightarrow$ $\mathcal{H}_2$ est sérialisable

- Comment gérer le niveau de concurrence dans un SGBD ?

⇒ Définir le niveau d'isolation du système

- `SET TRANSACTION ISOLATION LEVEL` < *level* >

Niveaux d'isolation

1. `SERIALIZABLE`
   - Protection : lectures sales, lecture répétable, objets fantômes
2. `REPEATABLE READ` (lecture répétable)
   - Les données lues ne sont pas modifiables par d'autres
   - Ne lit que des valeurs validées (pas de lecture sale)
   - Protection : lectures sales, lecture répétable, objets fantômes
3. `READ COMMITTED` (lecture de valeurs validées)
   - Ne lit que des valeurs validées
   - Protection : lectures sales, lecture répétable, objets fantômes
4. `READ UNCOMMITTED` (lecture de valeurs non-validées)
   - Protection : lectures sales, lecture répétable, objets fantômes
   - Préférable pour transactions READ ONLY
   - Annulations en cascade possibles

Verrouillage à deux phases

- **Objectif :** Retarder toutes les opérations conflictuelles
- Gestion de verrous au niveau de l'ordonanceur
  - Lecture : Verrou partagé (compatible avec d'autres lectures)
  - Ecriture : Verrou exclusif (incompatible avec les autres transactions)
  - Verrou relâché après "commit" ou "rollback"
  - Blocage des opérations en attente de verrous

Verrouillage à deux phases : Algorithme

1. Arrivée de l'opération $p_i[x]$ à l'ordonnanceur
2. La transaction T_i est-elle en attente ?
   - Oui ⇒ $p_i[x]$ est mis en attente
3. $p_i[x]$ est une opération de :
   - Lecture
     - Y a t-il un verrou *exclusif* sur 'x' ? (par T_j)
     - Oui ⇒ $p_i[x]$ est mise en attente
     - Non ⇒ Verrou Partagé sur 'x' par T_i, $p_i[x]$ est exécutée
   - Écriture
     - Y a t-il un verrou sur 'x' ? (par T_j)
     - Oui ⇒ $p_i[x]$ est mise en attente
     - Non ⇒ Verrou Exclusif sur 'x' par T_i, $p_i[x]$ est exécutée
   - *Commit* ou *Rollback*
     - Tous les verrous de T_i sont relâchés
     - On reprend les opérations en attente par ordre chronologique

Verrouillage à deux phases : Exemple

$\mathcal{H}_1$: $r_1[x]\ r_2[y]\ w_1[y]\ c_1\ w_2[y]\ c_2$

I	Verrous		Exécution			En attente
	x	y	T_1	T_2	T_3	
(1)	rl_1		$r_1[x]$			
(2)	rl_1	rl_2		$r_2[y]$		
(3)	rl_1	rl_2				$w_1[y]$
(4)	rl_1	rl_2				$w_1[y]\ c_1$
(5)	rl_1	wl_2		$w_2[y]$		$w_1[y]\ c_1$
(6)	rl_1			c_2		$w_1[y]\ c_1$
(7)	rl_1	wl_1	$w_1[y]$			c_1
(8)			c_1			

rl : read lock, verrou partagé (en lecture).
wl : write lock, verrou exclusif (en écriture).

Verrouillage à deux phases : Effets

\+ Plus d'incohérences possibles

\- Retardement abusif des transactions concurrentielles

\- Modification d'histoires sérialisables

\- Possibilité d'interblocages (*deadlock*)

Verrouillage à deux phases : Interblocage

T_1 : $r_1[x]$ $w_1[y]$ c_1
T_2 : $w_2[y]$ $w_2[x]$ c_2

Ordre de réception : $r_1[x]$ $w_2[y]$ $w_2[x]$ $w_1[y]$

- T_1 obtient verrou pour $r_1[x]$, T_2 pour $w_2[y]$
- $w_2[x]$ attend $r_1[x]$, $w_1[y]$ attend $w_2[y]$
  $\Rightarrow$ interblocage de T_1 et de T_2

Solution : "timeout", graphe d'attente

Estampillage

- **Objectif :** Ne rien retarder, valider si c'est possible
- Donner une estampille aux transactions et aux données
  - Estampilles ordonnées (temporel)
  - $T_i \rightarrow e(T_i)$
  - Deux opérations en conflit doivent s'exécuter suivant l'ordre des estampilles
    Si $p_i[x]$ puis $q_j[x] \Leftrightarrow e(T_i) < e(T_j)$

Estampillage : Algorithme

1. Arrivée de l'opération $p_i[x]$ à l'ordonnanceur
2. Les paires d'opérations conflictuelles s'exécutent dans l'ordre des transactions (de leurs estampilles)
3. Si $p_i[x]$ arrive après $q_j[x]$ et $e(T_i) < e(T_j)$
   $\Rightarrow$ $p_i[x]$ rejetée, ainsi que T_i
4. Si $p_i[x]$ peut être exécutée (pas en retard)
   $\Rightarrow$ Exécutée réellement dès que les transactions conflictuelles (sur 'x') sont validées

Estampillage : Exemple

$$r_1[x]\ w_2[x]\ r_3[x]\ r_2[x]\ w_1[x]\ c_1\ c_2\ c_3$$

$$e(T_1) < e(T_2) < e(T_3)$$

1. $r_1[x]$ acceptée
2. $w_2[x]$ conflit avec $r_1[x]$, test : $2 > 1 \rightarrow$ acceptée
3. $r_3[x]$ conflit avec $w_2[x]$, test : $3 > 2 \rightarrow$ acceptée
4. $r_2[x]$ aucun conflit $\rightarrow$ acceptée
5. $w_1[x]$ conflit avec $r_2[x]$, test : $1 < 2 \rightarrow$ rejetée

$\Rightarrow$ Exécution finale : $r_1[x]\ w_2[x]\ r_3[x]\ r_2[x]\ a_1\ c_2\ c_3$

Estampillage : Effets

\+ Exécution optimiste
\+ Produit des exécutions sérialisables
\- Si beaucoup de conflits, beaucoup d'annulations

Contrôle dans différents SGBD

- *Oracle*, *MySQL (InnoDB)* : Contrôle **multi-version**
  - Plusieurs versions de chaque donnée
  - Ordonnancement de type estampillage
  - 1 version → Transaction d'écriture
  - 1 donnée → Dernière transaction de lecture

  ⇒ une lecture n'attend jamais ! (dernière version respectant l'ordre)

  ⇒ une écriture → nouvelle version ou annulation (si ne respecte pas l'ordre)
- *SQL Server* (Microsoft), *DB2* (IBM) : Au choix
  - Optimiste : Multi-version
  - Pessimiste : Verrouillage à deux phases

Démonstration

- Exemple de concurrence sous MySQL
- Moteur InnoDB
- Deux consoles (deux transactions)
- Mode *autocommit* OFF