

Transactions Concurrence d'accès

HLIN605

Pascal Poncelet
LIRMM
Pascal.Poncelet@lirmm.fr
<http://www.lirmm.fr/~poncelet>



Problème

- Dans votre session :

LIRE (A)
A := A+100
ECRIRE (A)
LIRE (B)
B := B +100
ECRIRE (B)



2

Problème

Dans la session 1

LIRE (A)
A := A+100
ECRIRE (A)
LIRE (B)
B := B +100
ECRIRE (B)

Dans la session 2

LIRE (A)
A := A+100
ECRIRE (A)
LIRE (B)
B := B +100
ECRIRE (B)

A et B sont partagés

Dans la session 3

LIRE (A)
A := A+100
ECRIRE (A)
LIRE (B)
B := B +100
ECRIRE (B)



3

Problème

- Nécessité d'exécuter les programmes et requêtes de façon concurrente
- Garantir que l'exécution simultanée de plusieurs programmes se fait correctement
 - Cohérence de la base de données
 - Exécution correcte des programmes



4

Problème

- Régler les conflits lecture / écriture
- Garder de très bonnes performances
- Eviter les blocages
- De nombreux contextes
 - Annuaire téléphonique (accès en lecture)
 - Systèmes de réservation de billets (accès en lecture et mise à jour)



5

Les problèmes de concurrence

- Perte d'opérations

$T_1 : a := \text{LIRE}(A);$
 $T_2 : b := \text{ECRIRE}(A);$
 $T_2 : b := b + 1;$
 $T_2 : A := \text{ECRIRE}(b);$

$T_1 : a := a * 2 ;$
 $T_1 : \text{ECRIRE}(A) ;$
 Valeur de A pour T_1 ?

Cette anomalie résulte de la dépendance ECRITURE-
ECRITURE entre actions de plusieurs transactions
qui s'interfèrent



6

Les problèmes de concurrence

- Introduction d'incohérence

$A = B$ (A DOIT TOUJOURS ETRE EGAL A B)

```

T1 : A:=A*2;
      T2 : A:=A+1;
      T2 : B:=B+1;
T1 : B:=B*2;

```

Test : A=5, B=5

T₁ : A:=A*2; // A = 10

T₂ : A:=A+1; // A=11

T₂ : B:=B+1; //B=6

T₁ : B:=B*2; // B=12 et A=11 !!!

7

Les problèmes de concurrence

- Vision SGBD

```

T1 : A:=A*2;
      T2 : A:=A+1;
      T2 : B:=B+1;
T1 : B:=B*2;

```

T₁ L(A), E(A) L(B), E(B)
T₂ L(A), E(A), L(B), E(B)

Séquence = <T₁L(A), T₁E(A), T₂L(A), T₂E(A), T₂L(B), T₂E(B), T₁L(B), T₁E(B)>

8

Transaction

- **Transaction** : unité de programme exécutée sur un SGBD
- Une transaction débute souvent par un programme d'application
 - DEBUT TRANSACTION
 - Accès à la base de données (lectures, écritures)
 - Calculs en mémoire centrale
 - FIN TRANSACTION
 - **COMMIT** (si la transaction est bonne) ou **ROLLBACK** (s'il y a une erreur)

9

Transaction

- Une transaction = ensemble d'instructions séparant un **COMMIT** ou un **ROLLBACK** du **COMMIT** ou du **ROLLBACK** suivant
- Une transaction T_i fait l'état de la base d'un état consistant à un autre état consistant
 - A noter que pendant l'exécution de T_i soit la transaction dans son intégralité fonctionne soit les opérations doivent être annulées (**ROLLBACK**)
- Chaque transaction T_i est composée d'une séquence d'actions $\langle a_1, a_2, \dots, a_{n_i} \rangle$



10

Transaction

- Une exécution simultanée (Histoire) des transactions $\{T_1, T_2, \dots, T_n\}$ est une séquence d'actions
 - $H = \langle a_{i_1j_1}, a_{i_2j_2}, \dots, a_{i_kj_k} \rangle$ telle que $a_{ij} < a_{i_{j+1}}$ pour tout i et tout j et quelque soit a_{ij} de T_1, \dots, T_n , a_{ij} est dans H
 - C'est une séquence d'actions complète respectant l'ordre des actions des transactions
- Exemple :

$H = \langle T_1L(A), T_1E(A), T_2L(A), T_2E(A), T_2L(B), T_2E(B), T_1L(B), T_1E(B) \rangle$



11

Transaction

- Quand une transaction est validée (par commit) toutes les opérations sont validées et on ne peut plus en annuler aucune : **les mises à jour sont définitives**
- Quand une transaction est annulée (par **ROLLBACK** ou par une panne), on annule toutes les opérations depuis le dernier commit ou **ROLLBACK** ou depuis le premier ordre SQL s'il n'y a eu ni **COMMIT** ni **ROLLBACK**



12

Transaction

- Problèmes à éviter :
 - Lecture impropre (*dirty read*)
 - Ecriture impropre (*dirty write*)
 - Lecture non répétable (*non repeatable read*)
 - Données fantômes (*phantom read*)



13

Transaction

- *Ecriture impropre* : la mise à jour d'une transaction est écrasée par une autre transaction
 - T₁ crédite le compte A de 100 euros. T₂ débite le compte A de 50 euros. Au départ A = 500. Si T₁ et T₂ sont appliquées correctement A = 550.
- | | |
|-------------------------|------------------------|
| T ₁ (crédit) | T ₂ (débit) |
| Lire (A); {A = 500} | Lire (A); {A=500} |
| A := A+100; {A=600} | A:=A-50; {A=450} |
| Ecrire (A); {A=600} | Ecrire (A); {A=450} |
- La valeur finale de A = 450. Le crédit de T₁ n'est pas pris en compte



14

Transaction

- *Ecriture impropre* : un rollback annule la transaction
 - T₁ crédite le compte A de 1000 euros. T₂ lit la valeur de A qui est supérieure à 500 donc trouve que la provision est suffisante et modifie le débit du compte.
- | | |
|-------------------------|----------------------------|
| T ₁ (crédit) | T ₂ (débit) |
| LIRE (C) | |
| C := C+1000 | |
| ECRIRE (C) | |
| | LIRE (C) |
| | IF C > 500 THEN D := D+500 |
| | ECRIRE (D) |
| ROLLBACK | |



La contrainte Débit <= crédit n'est plus vérifiée. Le solde peut être négatif₁₅

Transaction

- *Lecture impropre* : un rollback annule la transaction

– T₁ veut faire un virement entre deux comptes A et B. L'opération commence par débiter 1000 du compte A.

T₁ (crédit) T₂ (débit)
LIRE (A)
A := A-1000
ECRIRE (A)

LIRE (A)
LIRE (B)

LIRE (B)
B=B+1000
ECRIRE(B)
COMMIT

En lisant A il voit que A a diminué de 1000.

16

Transaction

- *Lecture impropre* :

- u1 ajoute 10 places et annule sa transaction, u2 souhaite voir 7 places si elles sont disponibles

```

T1:
BEGIN TRANSACTION
UPDATE T_VOL
SET VOL_PLACES_LIBRES=
VOL_PLACES_LIBRES+10
WHERE VOL_ID =2

ROLLBACK

T2:
BEGIN TRANSACTION
IF (SELECT VOL_PLACES_LIBRES FROM T_VOL
WHERE VOL_ID=2)>= 7
THEN
BEGIN
UPDATE T_VOL
SET VOL_PLACES_LIBRES= VOL_PLACES_LIBRES-7
WHERE VOL_ID= 2
INSERT INTO T_CLIENT_VOL VALUES (77,2,7)
COMMIT
END
ELSE
ROLLBACK
  
```

17

Transaction

Temps	T_Vol Vol_id	Vol_ref	Vol_places_libres	T_Client_Vol Cli_id	Vol_id	Vol_places_prises	Transactions
t1	2	AF 121	6				Début transaction pour u2
t2	2	AF 121	6				Début transaction pour u1
t3	2	AF 121	16				
t4	2	AF 121	16				
t5	2	AF 121	6				Fin transaction rollback u1
t6	2	AF 121	-1				
t7	2	AF 121	-1	77	2	7	
t8	2	AF 121	-1	77	2	7	Fin transaction commit u2

Le temps d'un update abandonné, la transaction a lu des informations qu'elle ne devait pas voir et en a déduit qu'elle pouvait réserver surbooking

18

Transaction

- **Lecture non répétable** : si T_2 lit A qui est modifié par T_1 et que T_1 termine la transaction

– Quand T_2 lit A il trouvera une valeur différente de A à la seconde lecture.

T_1 (crédit) T_2 (débit)
Lire (A); {A=500} *Lire* (A); {A=500}
 A := A+100; {A=600} A:=A-50; {A=450}
Ecrire (A); {A=600} *Lire* (A); {A=600}

Dans l'exécution T_1 lit A=500. T_2 lit A=500. T_1 modifie A en 600. Quand T_2 relit A il vaut 600. Ça ne devrait pas être le cas. Dans la même exécution T_2 ne devrait avoir qu'une valeur de A (500 ou 600 mais pas les deux)



19

Transaction

- **Lecture non répétable** :
- U2 souhaite toutes les places d'un vol s'il y en a plus de 4

```

T1:
BEGIN TRANSACTION
UPDATE T_VOL
SET VOL_PLACES_LIBRES=
VOL_PLACES_LIBRES-4
WHERE VOL_ID = 2
COMMIT

T2:
BEGIN TRANSACTION
IF (SELECT VOL_PLACES_LIBRES FROM T_VOL
WHERE VOL_ID=2)>= 4
THEN
BEGIN
UPDATE T_VOL
SET VOL_PLACES_LIBRES= VOL_PLACES_LIBRES-
(SELECT VOL_PLACES_LIBRES FROM T_VOL
WHERE VOL_ID= 2)
WHERE ID=2
INSERT INTO T_CLIENT_VOL VALUES (77, 2,
(SELECT VOL_PLACES_LIBRES
FROM T_VOL WHERE VOL_ID = 2))
COMMIT
END
ELSE
ROLLBACK
  
```



20

Transaction

Temps	T_Vol Vol_id	Vol_ref	Vol_places_libres	T_Client_Vol Cli_id	Vol_id	Vol_places_prises	Transactions
t1	2	AF 121	6				Début transaction pour u2
t2	2	AF 121	6				Début transaction pour u1
t3	2	AF 121	2				
t4	2	AF 121	2				
t5	2	AF 121	2				Fin transaction commit u1
t6	2	AF 121	0				
t7	2	AF 121	0	77	2	2	
t8	2	AF 121	0	77	2	2	Fin transaction commit u2

u2 ne voulait que des places s'il y en avait plus de 4 et il se retrouve avec 2 places



21

Transaction

- Données fantômes** : lecture de valeur non existante de A par T₂

- Si T₁ met à jour A pendant qu'il est lu par T₂ alors si T₁ aborte, T₂ lira une valeur de A qui n'existera pas.

T₁ (crédit) T₂ (débit)

Lire (A); {A=500}

A := A+100; {A=600}

Ecrire (A); {A=600}

Lire (A); {A=600}

A:=A-50; {A=550}

T₁ échoue sa fin Ecrire (A); {A=550}

T₁ modifie A=600. T₂ lit A=600 mais T₁ échoue et son effet est effacé de la base, A est restauré à son ancienne valeur (A=500). A=600 est une valeur non existante mais lu par T₂ (données fantôme)



22

Transaction

- Données fantômes** :
- u2 souhaite n'importe quel vol pas cher pour amener 11 personnes

```

T1:
BEGIN TRANSACTION
INSERT INTO T_VOL VALUES
(5, 'AF111', 125)
COMMIT

T2:
BEGIN TRANSACTION
IF EXISTS (SELECT * FROM T_VOL
WHERE VOL_PLACES_LIBRES >= 11)
THEN
BEGIN
UPDATE T_VOL
SET VOL_PLACES_LIBRES= VOL_PLACES_LIBRES-11
WHERE VOL_PLACES_LIBRES >= 11
INSERT INTO T_CLIENT_VOL VALUES (77, 4, 11)
COMMIT
END
ELSE
ROLLBACK
  
```



23

Transaction

Tem ps	T_Vol Vol_id	Vol_ref	Vol_places_libres	T_Client_Vol Cli_id	Vol_id	Vol_places_pises	Transactions
t1	4	AF 325	258				Début transaction pour u2
t2	4	AF 325	258				Début transaction pour u1
t3	4	AF 325	258				
	5	AF 111	125				
t4	4	AF 325	258				
	5	AF 111	125				
t5	4	AF 325	247				Fin transaction commit u1
	5	AF 111	114				
t6	4	AF 325	247				
	5	AF 111	114				
t7	4	AF 325	247	77	4	2	
	5	AF 111	114				
t8	4	AF 325	247	77	4	2	Fin transaction commit u2
	5	AF 111	114				

Voilà comment un certain été des avions d'Air France volaient à vide
Avec toutes les places réservées. Bug du service de réservation



24

Transaction

- Pour éviter les problèmes :
 - Lecture impropre (*dirty read*)
 - Ecriture impropre (*dirty write*)
 - Lecture non répétable (*non repeatable read*)
 - Données fantômes (*phantom read*)
- Les transactions doivent suivre un certain nombre de propriétés : Propriétés d'une transaction (ACID)



25

Transaction

- **Atomicité**
 - Unité de cohérence : toutes les mises à jour doivent être effectuées ou aucune
- **Cohérence**
 - La transaction doit faire passer la base de données d'un état cohérent à un autre
- **Isolation**
 - Les résultats d'une transaction ne sont visibles aux autres transactions qu'une fois la transaction validée
- **Durabilité**
 - Les modifications d'une transaction validée ne seront jamais perdues



26

Avez vous remarqué ?

- Séquence = <T1L(A), T1E(A), T2L(A), T2E(A), T2L(B), T2E(B), T1L(B), T1E(B)>
- Où L = Lecture et E = Ecriture
- Les transactions n'affectent que les opérations du LMD et non pas le LDD


```
SQL>COMMIT;
SQL>CREATE TABLE R1 (A NUMBER(2));
SQL>CREATE TABLE R2 (B NUMBER(2));
SQL>INSERT INTO R1 VALUES (1);
SQL>INSERT INTO R2 VALUES (2);
SQL>ROLLBACK;
```
- Combien de tables créées ?
- Que donne la requête SELECT * FROM R1;



27

Avez vous remarqué ?

- Séquence = $\langle T_1.L(A), T_1.E(A), T_2.L(A), T_2.E(A), T_2.L(B), T_2.E(B), T_1.L(B), T_1.E(B) \rangle$
- Où L = Lecture et E = Ecriture
- Les transactions n'affectent que les opérations du LMD et non pas le LDD


```
SQL>COMMIT;
SQL>CREATE TABLE R1 (A NUMBER(2));
SQL>CREATE TABLE R2 (B NUMBER(2));
SQL>INSERT INTO R1 VALUES (1);
SQL>INSERT INTO R2 VALUES (2);
SQL>ROLLBACK;
```
- Combien de tables créées ? 2 R1 et R2 (CREATE TABLE : LDD)
- Que donne la requête **SELECT * FROM R1** ; : **aucun résultat** car le **ROLLBACK** fait retourner la transaction dans l'état du dernier **COMMIT**



28

Transaction

- HYPOTHESES
 - Exécution d'une transaction individuelle est correcte
 - Exécution de transactions en série (les unes derrière les autres) est correcte
 - PRINCIPE :
SE RAMENER A UNE EXECUTION DES TRANSACTIONS EN SERIE

Concept de **SERIALISABILITE**



29

Sérialisabilité

- Critère permettant de dire que l'exécution d'un ensemble de transactions (*schedule*) est correct
 - Une exécution est sérielle si toutes les actions des transactions ne sont pas entremêlées. Elle est donc de la forme : $\langle Tp(1), Tp(2), \dots, Tp(n) \rangle$ où p est une permutation de 1, 2, ... n.
- L'exécution des mêmes transactions en série : mêmes valeurs de départ et mêmes valeurs finales



30

Sérialisabilité : exemple

T ₁	T ₂
LIRE (A)	LIRE (A)
A := A+100	A := A*2
ECRIRE (A)	ECRIRE (A)
LIRE (B)	LIRE (B)
B := B+100	B := B*2
ECRIRE (B)	ECRIRE (B)

Au début : A = B = 25
T₁ puis T₂ : A = B = 250
T₂ puis T₁ : A = B = 150

31

Sérialisabilité : exemple

t₀ : A = B = 25

t₁
t₂
t₃
t₄
t₅
t₆
t₇
t₈
t₉
t₁₀
t₁₁
t₁₂

<T ₁ T ₂ T ₁ T ₂ >	
T ₁	T ₂
LIRE (A)	
A := A+100	
ECRIRE (A)	
	LIRE (A)
	A := A*2
	ECRIRE (A)
LIRE (B)	
B := B+100	
ECRIRE (B)	
	LIRE (B)
	B := B*2
	ECRIRE (B)
A=250, B=250	
SERIALISABLE	

32

Sérialisabilité : exemple

t₀ : A = B = 25

t₁
t₂
t₃
t₄
t₅
t₆
t₇
t₈
t₉
t₁₀
t₁₁
t₁₂

<T ₁ T ₂ T ₁ T ₂ >	
T ₁	T ₂
LIRE (A)	
A := A+100	
ECRIRE (A)	
	LIRE (A)
	A := A*2
	ECRIRE (A)
	LIRE (B)
	B := B*2
	ECRIRE (B)
LIRE (B)	
B := B+100	
ECRIRE (B)	
A=250, B=150	
NON SERIALISABLE	

33

Graphe de précédence

- Précédences
 - Techniques basées sur la seule sémantique des opérations de lecture / écriture - Recherche des conflits
 - $T_i(L)$ avant $T_j(L)$: pas de conflit
 - $T_i(L)$ avant $T_j(E)$: conflit
 - $T_i(E)$ avant $T_j(L)$: conflit
 - $T_i(E)$ avant $T_j(E)$: conflit
 - T_i lit O avant T_j écrit \Rightarrow lien T_i précède T_j
 - T_i écrit O avant T_j écrit/lit \Rightarrow lien T_i précède T_j
- Condition de sérialisabilité
 - Le graphe de précédence doit rester sans circuit



34

Graphe de précédence

T_1 : LIRE (A)
 T_2 : LIRE (A)
 T_3 : ECRIRE (B)
 T_1 : LIRE (B)
 T_2 : ECRIRE (A)
 T_3 : LIRE (B)

T_1 L(A) L(B)
 T_2 L(A) E(A)
 T_3 E(B) L(B)

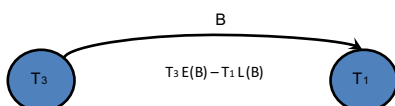


35

Graphe de précédence

T_1 L(A) L(B)
 T_2 L(A) E(A)
 T_3 E(B) L(B)

T_i lit O avant T_j écrit $\Rightarrow T_i$ précède T_j
 T_i écrit O avant T_j écrit/lit $\Rightarrow T_i$ précède T_j



36

Graphe de précédence

T₁ : LIRE (A)
 T₂ : ECRIRE (A)
 T₂ : LIRE (B)
 T₂ : ECRIRE (B)
 T₃ : LIRE (A)
 T₁ : ECRIRE (B)

T₁ L(A) E(B)
 T₂ E(A) L(B) E(B)
 T₃ L(A)

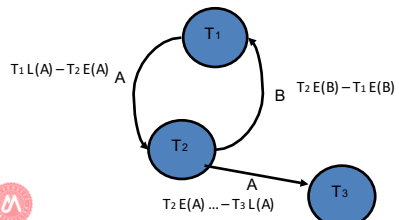


37

Graphe de précédence

T₁ L(A) E(B)
 T₂ E(A), L(B), E(B)
 T₃ L(A)

T_i lit O avant T_j écrit => T_i précède T_j
 T_i écrit O avant T_j écrit/lit => T_i précède T_j



Présence d'un cycle T₁ dépend de celui de T₂ et vice-versa



38

Ordonnancement des transactions

- Exécution en série : une transaction après l'autre
- Exécutions équivalentes :
 - quelque soit la base de données, la première exécution est identique à l'effet de la seconde exécution
 - Vérification : voir l'ordre des ordres de lecture/écriture conflictuelles
- Exécution sérialisable : équivalente à une exécution en série de toutes les transactions

$T_1 = L(A) A := A * 5 E(A) C$
 $T_2 = L(A) A := A + 10 E(A) C$
 $T_1 T_2$ non équivalente à $T_2 T_1$



39

39

Exécution sérialisable

- Deux exécutions sont équivalentes si :
 - Les mêmes actions (lecture/écriture) se retrouvent dans les mêmes transactions
 - Chaque paire d'actions conflictuelles (lecture/écriture ; écriture/lecture ; écriture/écriture) sont ordonnées de la même façon dans les deux exécutions

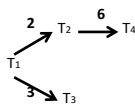


40

Exemple

$T_1 L(A) T_2 E(A) T_1 L(B) T_3 E(B) T_2 L(C) T_4 E(C) T_1 C T_2 C T_3 C T_4 C$

1: $T_1 L(A)$ 2: $T_2 E(A)$ 3: $T_1 L(B)$ 4: $T_3 E(B)$ 5: $T_2 L(C)$ 6: $T_4 E(C)$ $T_1 C T_2 C T_3 C T_4 C$



- Les opérations sont sérialisables dans l'ordre suivant :

$T_1 T_2 T_3 T_4$

$T_1 T_2 T_4 T_3$

$T_1 T_3 T_2 T_4$



41

Approches de gestion de la concurrence

- Objectif : forcer la sérialisabilité, i.e. le graphe de précédence doit être sans cycle
- Problème : les mécanismes utilisant le graphe de sérialisabilité existent mais difficiles à mettre en œuvre car il faut construire tout le graphe en mémoire (coût trop élevé)
- Contrôle continu :
 - vérification au fur et à mesure de la sérialisabilité.
 - Pessimistes car considèrent que les conflits sont fréquents et qu'il faut les traiter le plus tôt possible
- Contrôle par certification :
 - Vérification de la sérialisabilité quand la transaction s'achève
 - Optimistes : considèrent conflits rares et que l'on peut ré-exécuter les transactions qui posent problèmes.



42

Mécanismes de verrouillage

- L'approche la plus fréquente : contrôle continu
- Verrouillage: on bloque l'accès à une donnée dès qu'elle est lue ou écrite par une transaction (« pose de verrou ») et on libère cet accès quand la transaction termine par **COMMIT** ou **ROLLBACK** (« libération du verrou »)
- Permet à une transaction de se réserver l'usage exclusif d'une donnée aussi longtemps que nécessaire
 - Avec un mécanisme de verrouillage, l'accès à une donnée E est encadrée par la paire d'opérateurs :

LOCK E
UNLOCK E



43

Mécanismes de verrouillage

- Le verrouillage définit le type d'opération : lecture, écriture
- Deux modes possibles :
 - Exclusif (*exclusive lock*)
 - Si seule la transaction T peut mettre à jour la donnée E. Une opération d'écriture change la valeur d'une donnée. Une opération de lecture par T₁ et de lecture par T₂ ne sont pas permises
 - Partagé (*shared lock*)
 - Si aucune transaction T ne peut mettre à jour la donnée E mais si des transactions concurrentes peuvent accéder à E. Une opération Lecture ne bloque pas une transaction.



44

Mécanismes de verrouillage

- Ces verrous sont posés de manière automatique par les SGBD
- Attention le verrouillage influence les performances d'une base de données soumise à un très grand nombre de transactions



45

Mécanismes de verrouillage

- Prise en compte du temps
- Le verrou associé à la donnée E peut être de durée variable
 - Courte
 - S'il est maintenu pendant la durée d'une action de T (LIRE, ECRIRE)
 - Longue
 - S'il est maintenu pendant toute la durée de la transaction T



46

Mécanismes de verrouillage

- *Cas de verrous exclusifs de longue durée*

```

Begin T1      Begin T2
LOCK A       LOCK C
LIRE(A)      LIRE(C)
A:=A+10;     // blocage de T2
             // jusqu'à libération
UNLOCK A
End T1       LOCK A
             LIRE(A)
             A:=A+C
             DELETE(C)
             UNLOCK A
             End T2
  
```

La mise à jour de la transaction T₁ n'a pas été perdue. La valeur finale tient compte des modifications de T₁.



47

Un autre exemple

```

T1          T2
LOCK A      LOCK B
LIRE(A)     LIRE(B)
A:=A+100    B:=B*5
ECRIRE(A)   ECRIRE(B)
UNLOCK A    UNLOCK B

LOCK B      LOCK A
LIRE(B)     LIRE A
B:=B+10     A:= A+20
ECRIRE(B)   ECRIRE(A)
UNLOCK B    UNLOCK A
  
```



48

Ca marche toujours ?

```

T1
LOCK A
URE(A) (A=10)
A=A+50
ECRIRE(A) (A=110)
UNLOCK A

T2
LOCK B
URE(B) (B=20)
B=B*5
ECRIRE(B) (B=100)
UNLOCK B

LOCK B
URE(B) (B=100)
B=B+10
ECRIRE(B) (B=110)
UNLOCK B

LOCK A
URE(A) (A=110)
A=A+20
ECRIRE(A) (A=130)
UNLOCK A

```

Les valeurs finales sont A=130 et B=110. Ce n'est pas correct. Une exécution séquentielle aurait produit : A = 130 et B=150. Il manque quelque chose.

49

Protocole en deux phases

- Chaque transaction doit obtenir un verrou partagé P sur un granule avant de le lire, et un verrou exclusif X sur un granule avant d'écrire
- Tous les verrous émis par une transaction sont libérés à la terminaison
- Si une transaction émet un verrou X sur un granule, aucune transaction ne peut obtenir un verrou (P ou X) sur le même granule
- Une transaction ne peut émettre de verrou dès qu'elle commence à libérer ses verrous.

50

Gestion des verrous

- Les demandes de verrouillage/déverrouillage sont gérés par le gestionnaire de verrouillage
- Une table spécifique contenant le nombre de transactions avec verrou, le type de verrou et un pointeur vers la file de demande de verrou
- Verrouillage et déverrouillage sont des opérations atomiques
- Une transaction peut demander à ce qu'un verrou partagé devienne un verrou exclusif

51

Améliorations du verrouillage

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

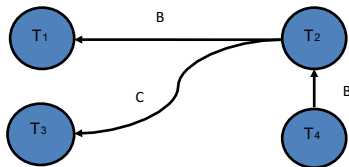


52

Problèmes du Verrouillage

- risque d'interblocage

T_1 L(A) L(B)
 T_2 E(B) E(C)
 T_3 L(C)
 T_4 L(B)

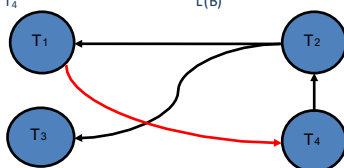


53

Problèmes du Verrouillage

- risque d'interblocage
 - des transactions en attente mutuelles

T_1 L(A) L(B)
 T_2 E(B) E(C)
 T_3 L(C) E(A)
 T_4 L(B)



T1 doit précéder
 T4
 Mais T1 doit être
 Précédé par T4 (via
 T2)



54

Résolution de l'interblocage

- Prévention
 - définir des critères de priorité de sorte à ce que le problème ne se pose pas
 - exemple : priorité aux transactions les plus anciennes
- Détection
 - gérer le graphe des attentes : les nœuds représentent les transactions, il existe un lien entre T_i et T_j si T_i attend que T_j libère un verrou
 - lancer régulièrement un algorithme de détection de circuits
 - choisir une victime qui brise le circuit



55

Mécanismes utilisant les estampilles

- Une estampille est une étiquette de temps que l'on associe :
 - Aux données : valeur de l'horloge au moment de la dernière modification (=numéro de version)
 - Aux transactions : valeur de l'horloge courante
 - Les conflits sont détectés au moment où ils se produisent par comparaison des estampilles
 - En cas de conflit une transaction est annulée



56

Mécanismes utilisant les estampilles

- Principe : attribuer à chaque élément un read-timestamp (RTS) et un write-timestamp (WTS), donner à chaque nouvelle transaction une estampille (TS)

Si l'action a_i de la transaction T_i est en conflit avec l'action a_j de T_j et si $TS(T_i) < TS(T_j)$ alors a_i doit être réalisé avant a_j . Sinon relancer la transaction



57

Mécanismes utilisant les estampilles

- Ce mécanisme garantit que :
 - Pas de mise à jour perdue : Une transaction utilisant une valeur obsolète de la donnée pour effectuer une mise à jour est refusée
 - Pas de lecture impropre : Si une transaction T lit une entité qui a un numéro de version supérieur à l'estampille de T, cette transaction est refusée
 - Contrôle très simple d'ordonnement des accès conformément à l'ordre de lancement des transactions.
 - En cas de désordre : reprendre la transaction ayant créé le désordre



58

Mécanismes utilisant les estampilles

- Cependant
- Les estampilles remplacent en quelque sorte les verrous
- Pas d'attente car reprise des transactions en cas d'accès ne respectant pas l'ordre de lancement des transactions
- => beaucoup trop de reprises
- => difficile à mettre en œuvre dans de grosses applications très concurrentes



59


Conclusions

- Problème difficile
- Généralement dans les SGBD : verrou
- Les solutions « académiques » : in progress
- Prise en compte de la sémantique des application (opérations commutatives (e.g., ajouts d'informations))
- Quid des transactions longues
 - mise à jour d'objets complexes
- Quid du travail coopératif
 - modèles concurrents plutôt que séquentiels



60

- Des questions ?

61
