Corso di Laurea
in
Ingegneria Informatica
"*Basi di dati*"
a.a. 2019-2020

Docente: Gigliola Vaglini
Docente laboratorio SQL: Francesco Pistolesi

1

Lezione 8

8.2.Gestione delle transazioni: controllo della concorrenza

2

Controllo di concorrenza

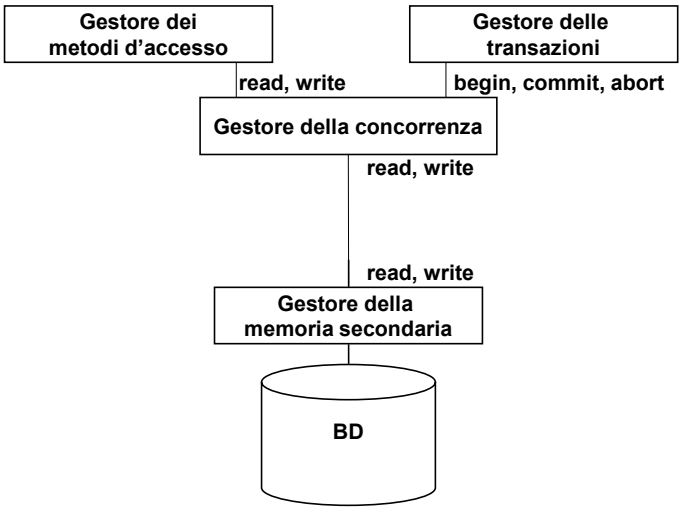
- La concorrenza è fondamentale: decine o centinaia di transazioni al secondo, non possono essere seriali

Problema

- Anomalie causate dall'esecuzione concorrente, che quindi va governata

3

Architettura del controllore della concorrenza



4

Perdita di aggiornamento

- Due transazioni identiche:
  - t 1 : $r(x)$, $x = x + 1$, $w(x)$
  - t 2 : $r(x)$, $x = x + 1$, $w(x)$
- Inizialmente x=2; dopo un'esecuzione seriale x=4
- Un'esecuzione concorrente:

t_1	t_2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
$x = x + 1$	
	$x = x + 1$
$w_1(x)$	
commit	
	$w_2(x)$
	commit

- Un aggiornamento viene perso: x=3

5

Lettura sporca

t_1	t_2
bot	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	bot
	$r_2(x)$
abort	
	commit

- Aspetto critico: t_2 ha letto uno stato intermedio ("sporco") e lo può comunicare all'esterno

6

Letture inconsistenti

- t_1 legge due volte:

t_1	t_2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

  - t_1 legge due valori diversi per x !

7

Aggiornamento fantasma

- Assumere ci sia un vincolo $y + z = 1000$;

t_1	t_2
bot	
$r_1(y)$	
	bot
	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = y + z$	
commit	

- $s = 1100$: t_1 vede un aggiornamento non completo

8

Inserimento fantasma

t_1	t_2
`bot`	
"legge gli stipendi degli impiegati del dip A e calcola la media"	
	`bot`
	"inserisce un impiegato in A"
	`commit`
"legge gli stipendi degli impiegati del dip A e calcola la media"	
`commit`	

9

Anomalie

- Perdita di aggiornamento W-W
- Lettura sporca R-W (o W-W) con abort
- Letture inconsistenti R-W
- Aggiornamento fantasma R-W
- Inserimento fantasma R-W su dato "nuovo"

10

Schedule

- Dato che più transazioni vengono eseguite in modo concorrente, le operazioni di R/W vengono richieste da transazioni differenti in interleaving. Uno schedule è una sequenza di R/W relative all'insieme delle transazioni concorrenti in un certo istante. Formalmente, uno schedule S1 è una sequenza
- S1:r1(x)r2(z)w1(x)w2(z)
- Dove r1(x) rappresenta la lettura dell'oggetto x da parte della transazione t1 e w2(z) rappresenta la scrittura dell'oggetto z da parte della transazione t2. Le operazioni compaiono nello schedule nell'ordine temporale di esecuzione sulla base di dati.

11

Schedule

- Sequenza di operazioni di lettura/scrittura di transazioni concorrenti
- Esempio:

$$S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$$

12

Controllo di concorrenza

- Il controllo della concorrenza è eseguito dallo scheduler, che tiene traccia di tutte le operazioni eseguite sulla base di dati dalle transazioni e decide se accettare o rifiutare le operazioni che vengono via via richieste.
- Per il momento, assumiamo che l'esito (commit/abort) delle transazioni sia noto a priori
  - In questo modo possiamo rimuovere dallo schedule tutte le transazioni abortite; schedule =commit-proiezione
  - Si noti che tale assunzione non consente di trattare alcune anomalie (lettura sporca).

13

Controllo di concorrenza

- *Obiettivo*: evitare le anomalie
- *Soluzione:Scheduler* (sistema che accetta o rifiuta, anche tramite riordino, le operazioni richieste dalle transazioni)

14

Schedule seriale

- Uno schedule S si dice seriale se, per ogni transazione t, tutte le azioni di t compaiono in sequenza, senza essere inframezzate da azioni di altre transazioni.
- In S2 le transazioni t0, t1, e t2vengono eseguite in sequenza:
  - S2:r0(x)r0(y)w0(x)r1(y)r1(x)w1(y)r2(x)r2(y)r2(z)w2(z)

15

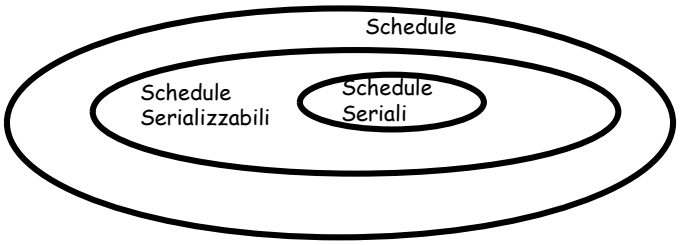
Schedule serializzabile

- L'esecuzione di uno schedule (commit-proiezione) Si è corretta quando produce lo stesso risultato di un qualunque schedule seriale Sj definito dalle stesse transazioni di Si. In questo caso, lo schedule Si è detto serializzabile.
- E' richiesta una nozione di equivalenza fra schedule

16

Idea base

- Individuare classi di schedule serializzabili la cui proprietà di serializzabilità sia verificabile a costo basso



17

- Esiste la relazione legge-da tra le operazionoi $r_j(x)$ e $w_j(x)$ presenti in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è nessun $w_k(x)$ ($k \neq j$) tra di loro.
- $w_i(x)$ in S è detta ***scrittura finale*** *su x se* è l'ultima scrittura sull'oggetto x in S

18

View-Serializzabilità

- Due schedule sono ***view-equivalenti*** ($S_i \approx_V S_j$) se hanno la stessa relazione *legge-da* e le stesse scritture finali su ogni oggetto.
- Uno schedule S è ***view-serializzabile*** se è view-equivalente ad un qualche schedule seriale
- L'insieme degli schedule view-serializzabili è indicato con **VSR**

19

View serializzabilità: esempi

- Consideriamo i seguenti schedule
  S3:*w0(x)r2(x)r1(x)w2(x)w2(z)*
- S4:*w0(x)r1(x)r2(x)w2(x)w2(z)*
- S5:*w0(x)r2(x)w2(x)r1(x)w2(z)*
- S6:*w0(x)r2(x)w2(x)w2(z)r1(x)*
- S3è view-equivalente allo schedule seriale S4 (quindi, è view-serializzabile).
- S5 non è view-equivalente allo schedule seriale S4, ma lo è allo schedule seriale S6 (ed è quindi anch'esso view-serializzabile)

20

View serializzabilità: esempi cont.

- S_3 : $r_1(x)$ $r_2(x)$ $w_1(x)$ $w_2(x)$ (perdita di aggiornamento)
  S_4 : $r_1(x)$ $r_2(x)$ $w_2(x)$ $r_1(x)$ (letture inconsistenti)
  S_5 : $r_1(x)$ $r_1(y)$ $r_2(z)$ $r_2(y)$ $w_2(y)$ $w_2(z)$ $r_1(z)$
  (aggiornamento fantasma)
  - S_3, S_4, S_5 non view-serializzabili, non view-equivalenti a nessun schedule seriale

21

View serializzabilità: verifica

- Problema:
- Complessità:
  - la verifica della view-equivalenza di due schedule:
    - polinomiale
  - decidere la view-serializzabilità di uno schedule:
    - problema NP-completo (è necessario confrontare lo schedule con tutti gli schedule seriali).
- Non è utilizzabile in pratica

22

Soluzione

- Definire una condizione di equivalenza più ristretta, che non copra tutti i casi di equivalenza tra schedule coperti della view-equivalenza, ma che sia utilizzabile nella pratica (la procedura di verifica abbia cioè una complessità inferiore).

Conflitti tra operazioni

- Definizione preliminare:
  - Un'operazione a_i è in *conflitto* con un'altra a_j ($i \neq j$), se operano sullo stesso oggetto e almeno una di esse è una scrittura. Due casi:
    - conflitto *read-write* (*rw* o *wr*)
    - conflitto *write-write* (*ww*).

Conflict-serializzabilità

- *Schedule conflict-equivalenti* ($S_i \approx_C S_j$): includono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi
- Uno schedule è *conflict-serializable* se è conflict-equivalente ad un qualche schedule seriale
- L'insieme degli schedule conflict-serializzabili è indicato con **CSR**

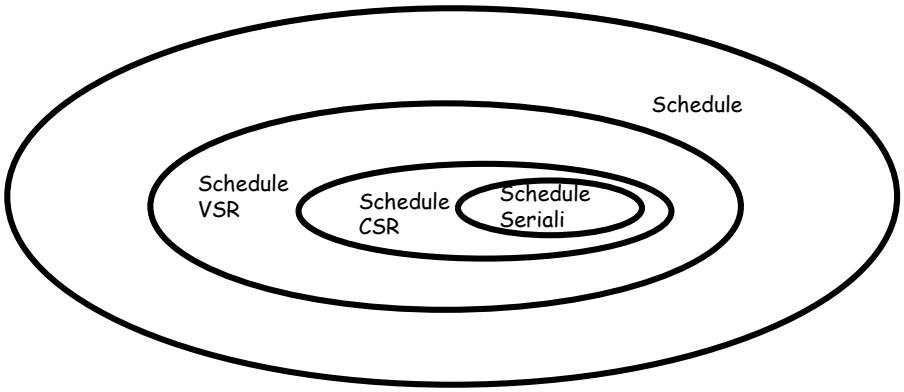
25

VSR e CSR

- Si può dimostrare che la classe degli schedule CSR è strettamente contenuta nella classe degli schedule VSR (la serializzabilità rispetto ai conflitti è condizione sufficiente, ma non necessaria, per la view-serializzabilità):
  - ogni schedule conflict-serializable è anche view-serializable;
    - CSR implica VSR
  - esistono degli schedule appartenenti a VSR che non appartengono a CSR

26

CSR e VSR



27

Verifica di conflict-serializzabilità

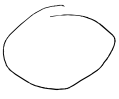
- Per mezzo del **grafo dei conflitti**:
  - un nodo per ogni transazione t_i
  - un arco (orientato) da t_i a t_j se c'è almeno un conflitto fra un'operazione a_i e un'operazione a_j tale che a_i precede a_j
- Teorema
  - **Uno schedule è in CSR se e solo se il grafo è aciclico**

28

Grafo dei conflitti

- S= r1(x)w2(x)r3(x)r1(y)w2(y)r1(v)w3(v)r4(v)w4(y)w5(y)
- x y v
- r1 r1 r1
- w2 w2 w3
- r3 w4 r4
- w5

29



30

Esempio 1

- S: r1(y) w3(z) r1(z) r2(z) w3(x) w1(x) w2(x) r3(y)

a) S è VSR o CSR? se è serializabile mostrare uno schedule seriale equivalente

31

- X w3 w1 w2
- Y r1 r3
- Z w3 r1 r2

32

- S è CSR e quindi anche VSR

Esempio 2

- Dire se i seguenti due schedule sono view-equivalenti o conflict-equivalenti o nessuna delle due cose.
-

S1=w2(x) r2(x) w1(x) r1(x) w2(y) r2(y) w1(x) w2(z)

S2= w1(x) r1(x) w2(x) r2(x) w1(x) w2(y) r2(y) w2(z)

-

- View-equivalenti ma non conflict-equivalenti
- VSR ma non CSR, schedule seriale equivalente T2T1

35

- S1: r1(x) r2(x) w1(x) r2(y) w1(y) w2(z) r3(z) r1(z) w3(x) w1(z) w3(z) r3 (y) w3(y)
- S2: r1(x) r2(x) w1(x) r2(y) w1(y) w2(z) r1(z) w1(z) r3(z) w3(x) w3(z) r3 (y) w3(y)

36

- Non conflict-equivalenti e non view-equivalenti
- S1 Non CSR e non VSR
- S2 VSR e non CSR

37

- $r_2(x)\ r_1(x)\ r_2(y)\ w_2(y)\ w_1(z)\ w_3(z)\ r_1(z)\ r_3(z)\ w_1(x)\ r_2(y)\ w_3(y)$
- $r_1(x)\ r_2(y)\ w_2(y)\ r_2(x)\ w_1(z)\ w_3(z)\ r_3(z)\ w_1(x)\ r_1(z)\ r_2(y)\ w_3(y)$

38

- Conflict-equivalenti
- Non CSR e non VSR

Conflict-serializabilità: verifica

- La conflict-serializabilità è più rapidamente verificabile (l'algoritmo, con opportune strutture dati richiede tempo lineare), ma necessita della costruzione del grafo dei conflitti ad ogni richiesta di scrittura

Però

- In un sistema con 100 tps e che mediamente accedono a 10 pagine e durano 5 secondi, vanno gestiti in ogni istante grafi con 500 nodi, tenendo traccia dei 5000 accessi delle 500 transizioni attive.
- Il grafo dei conflitti continua a modificarsi dinamicamente, rendendo difficoltose le decisioni dello scheduler.
- Quindi non è praticabile mantenere il grafo, aggiornarlo e verificarne l'aciclicità on-line ad ogni richiesta di operazione.
- N.B. la tecnica risulta del tutto impraticabile nel caso di basi di dati distribuite (il grafo deve essere ricostruito a partire da archi riconosciuti dai diversi server del sistema).

41

In pratica

- In pratica, si utilizzano tecniche di scheduling che
  - garantiscono la conflict-serializzabilità a priori senza dover costruire il grafo

42

Lock

- Principio:
  - Tutte le letture sono precedute da un *lock* e seguite da *unlock*
  - Tutte le scritture sono precedute da un *lock* e seguite da *unlock*
- Il *lock manager* (parte dello scheduler) riceve queste richieste dalle transazioni e le accoglie o rifiuta

43

Lock condiviso e esclusivo

- Per aumentare la concorrenza è possibile avere lock di tipo diverso, condiviso o esclusivo, usati in momenti diversi sulla stessa risorsa.
- Per una lettura si richiede un lock condiviso (contatore delle letture), quando serve scrivere si richiede il lock esclusivo.

44

Upgrading e downgrading dei lock

- Principio:
  - Tutte le letture sono precedute da *r_lock* (lock condiviso) e seguite da *unlock*
  - Tutte le scritture sono precedute da *w_lock* (lock esclusivo) e seguite da *unlock*
- Quando una transazione prima legge e poi scrive un oggetto, può:
  - richiedere subito un lock esclusivo
  - chiedere prima un lock condiviso e poi uno esclusivo (*lock upgrade*)

45

Transazioni ben formate rispetto al locking

- Ogni read è preceduta da un *r_lock* e seguita da un *unlock*
- Ogni write è preceduta da un *w_lock* e seguita da un *unlock*

46

Comportamento dello scheduler

- La politica dello scheduler è basata sulla tavola dei conflitti. Il lock manager riceve richieste di lock dalle transazioni e concede/rifiuta di concederli sulla base dei lock precedentemente concessi ad altre transazioni.
  - Quando viene concesso il lock su una risorsa ad una transazione, si dice che la risorsa è acquisita dalla transazione.
  - Nel momento dell'unlock, la risorsa viene rilasciata.

47

Gestione dei lock

- Tavola dei conflitti (permette di realizzare la politica per la gestione dei conflitti)

Richiesta	Stato della risorsa		
	free	*r_locked*	*w_locked*
r_lock	OK / r_locked	OK / r_locked	NO/ w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends (1)	OK / free

(1) Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata solo quando il contatore scende a zero

48

Gestione dei lock cont.

- I tre No presenti nella tabella corrispondono ai conflitti che si possono presentare: richiesta di lettura (risp., scrittura) su una risorsa già acquisita in scrittura e richiesta di scrittura su una risorsa già acquisita in lettura. Solo richieste di lettura su una risorsa già acquisita in lettura possono essere sempre accettate.

Gestione dei lock (cont)

- Se la risorsa non è concessa, la transazione richiedente è posta in attesa (eventualmente in coda), fino a quando la risorsa non diventa disponibile
- Tale attesa può terminare quando la risorsa ritorna disponibile.

Ordine di lock e unlock

- Viene garantita la mutua esclusione sulla risorsa, ma non la serializzabilità.
- E' possibile imporre un ordine alle richieste di acquisizione e rilascio di una risorsa che automaticamente garantiscano la serializzabilità?

- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- unlock(B);
- w1_lock(A);
- r1(A);
- A:=A+50;
- w1(A);
- unlock(A);
- commit

51

Locking a due fasi

- Usato da quasi tutti i sistemi
- Garantisce "a priori" la conflict-serializzabilità
- Due regole:
  - "proteggere" tutte le letture e scritture con lock
  - un vincolo sulle richieste e i rilasci dei lock:
    - **una transazione, dopo aver rilasciato un lock, non può acquisirne altri finchè tutti quelli che ha acquisito non sono stati rilasciati**

52

- Una transazione attraversa una prima fase di acquisizione di ciò che le serve
- Poi comincia a rilasciare e non può acquisire altro

53

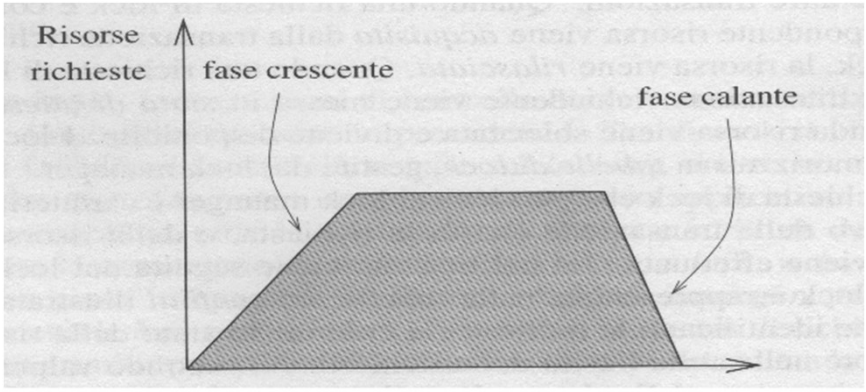
Two phase locking

- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- w1_lock(A);
- r1(A);
- unlock(B);
- A:=A+50;
- w1(A);
- unlock(A);
- commit

- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- unlock(B);
- w1_lock(A);
- r1(A);
- A:=A+50;
- w1(A);
- unlock(A);
- commit

54

Rappresentazione del 2PL



55

Upgrading e downgrading dei lock

- L'upgrade si può fare solo nella fase di acquisizione dei lock,
- il downgrade nella fase di rilascio.

56

2PL e CSR

- Ogni schedule 2PL e' anche conflict serializzabile, ma non è vero il viceversa
  - 2PL implica CSR

57

2PL e CSR

- La classe di schedule 2PLè contenuta nella classe CSR.
- Prova: Assumiamo, per assurdo, che esista uno schedule S tale che S∈2PL e S∉CSR. Da S ∉ CSR segue che il grafo dei conflitti per S contiene un ciclo t1, t2, : : : , tn, t1. Se esiste un arco (conflitto) tra t1 e t2, significa che esiste una risorsa x su cui si verifica il conflitto: t2 può procedere solo se t1 rilascia il lock su x così che t2 lo può acquisire. Così avanti fino al conflitto tra tn e t: t1 deve acquisire il lock rilasciato da tn, ma t1 ha già rilasciato un lock per farlo acquisire da t2 e quindi t1 non rispetta il 2PL.

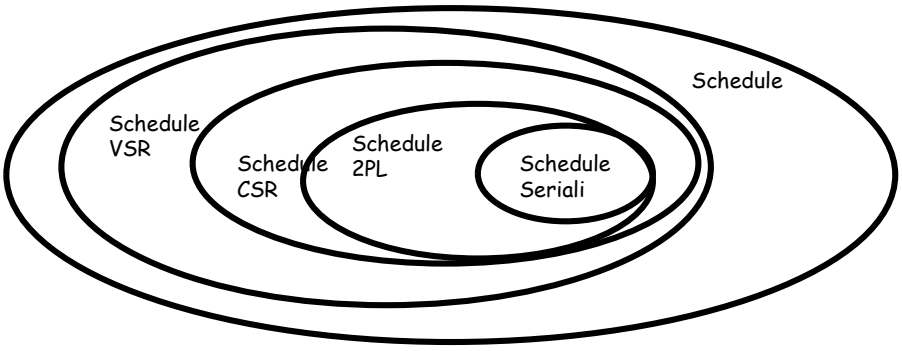
58

2PL e CSR

- Esistono schedule in CSR, ma non in 2PL.
  - S:r1(x)w1(x)r2(x)w2(x)r3(y)w1(y)
- La transazione t1 deve cedere un lock esclusivo sulla risorsa x per consentire alla transazione t2 di accedervi, prima in lettura e poi in scrittura, e successivamente richiedere un lock esclusivo sulla risorsa y. Si noti che t1 non può anticipare la richiesta del lock su y prima del rilascio di x in quanto dovrebbe comunque rilasciare la risorsa y (per poi riacquisirla) per consentirne l'utilizzo da parte di t3.
- S è però conflict-serializzabile (è conflict-equivalente allo schedule seriale t3; t1; t2).

59

CSR, VSR e 2PL



60

Le anomalie

- E' facile vedere che 2PL risolve le anomalie di perdita di aggiornamento, di aggiornamento fantasma e di letture inconsistenti.

Aggiornamento fantasma

t_1	t_2	x	y	z
		free	free	free
$r\_lock_1(x)$		1:read		
$r_1(x)$				
	$w\_lock_2(y)$		2:write	
	$r_2(y)$			
$r\_lock_1(y)$			1:wait	
	$y = y - 100$			
	$w\_lock_2(z)$			2:write
	$r_2(z)$			
	$z = z + 100$			
	$w_2(y)$			
	$w_2(z)$			
	commit			
	$unlock_2(y)$		1:read	
$r_1(y)$				
$r\_lock_1(z)$				1:wait
	$unlock_2(z)$			1:read
$r_1(z)$				
$s = x + y + z$				
commit				
$unlock_1(x)$		free		
$unlock_1(y)$			free	
$unlock_1(z)$				free

Però 2PL presenta altre anomalie

- Il fallimento di una transazione che ha scritto una risorsa deve causare il fallimento di tutte le transazioni che hanno letto il valore scritto (letture sporche)
- Attese incrociate (o deadlock): due transazioni detengono ciascuna una risorsa e aspettano la risorsa detenuta dall'altra. In generale, la probabilità di deadlock è bassa, ma non nulla.

63

Cascading rollbacks

- begin(T1);
- w1_lock(A);
- r1(A);
- r1_lock(B);
- r1(B);
- w1(A);
- unlock(A);
- abort

begin(T2);
w2_lock(A);
r2(A);
w2(A);
unlock(A);…

begin(T3);
r3_lock(A);
r3(A);…

Quando T1 fallisce, il fallimento si deve trasmettere a T2 e T3

64

Deadlock

• begin(T1)	
• w1_lock(B);	
• r1(B);	
• B:=B-50;	
• w1(B);	begin(T2)
	r2_lock(A);
	r2(A);
	r2_lock(B);
	wait T1
• w1_lock(A);	
• wait T2	
• r1(B);	
• unlock (B)	

65

Locking a due fasi stretto (rigoroso)

- Condizione aggiuntiva:
  - **I lock possono essere rilasciati solo dopo il commit**
- elimina il rischio di letture sporche e quindi di rollback in cascata

66

Strict two phase locking

- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- w1_lock(A);
- r1(A);
- A:=A+50;
- w1(A);
- commit
- unlock(B);
- unlock(A);

- begin (T1)
- w1_lock(B);
- r1(B);
- B:=B-50;
- w1(B);
- unlock(B);
- w1_lock(A);
- r1(A);
- A:=A+50;
- w1(A);
- unlock(A);
- commit

67

Controllo di concorrenza basato su timestamp

- Tecnica alternativa al 2pL
- **Timestamp**:
  - identificatore che definisce un ordinamento totale sugli eventi di un sistema
- Ogni transazione ha un timestamp che rappresenta l'istante di inizio della transazione
- Uno schedule è accettato solo se riflette l'ordinamento seriale delle transazioni indotto dai timestamp

68

Dettagli

- Lo scheduler ha due contatori RTM(x) e WTM(x) per ogni oggetto
- Lo scheduler riceve richieste di letture e scritture (con indicato il timestamp della transazione):
  - *read*(x,*ts*):
    - se *ts* < WTM(x) allora la richiesta è respinta e la transazione viene uccisa;
    - altrimenti, la richiesta viene accolta e RTM(x) è posto uguale al maggiore fra RTM(x) e *ts*
  - *write*(x,*ts*):
    - se *ts* < WTM(x) o *ts* < RTM(x) allora la richiesta è respinta e la transazione viene uccisa,
    - altrimenti, la richiesta viene accolta e WTM(x) è posto uguale a *ts*
- Vengono uccise molte transazioni

69

	Risposta	Nuovo Valore
read(x,1)	Ok	RTM(x)=1
write(x,1)	Ok	WTM(x)=1
read(z,2)	Ok	RTM(z)=2
read(y,1)	Ok	RTM(y)=1
write(y,1)	Ok	WTM(y)=1
read(x,2)	Ok	RTM(x)=2
write(z,2)	Ok	WTM(z)=2

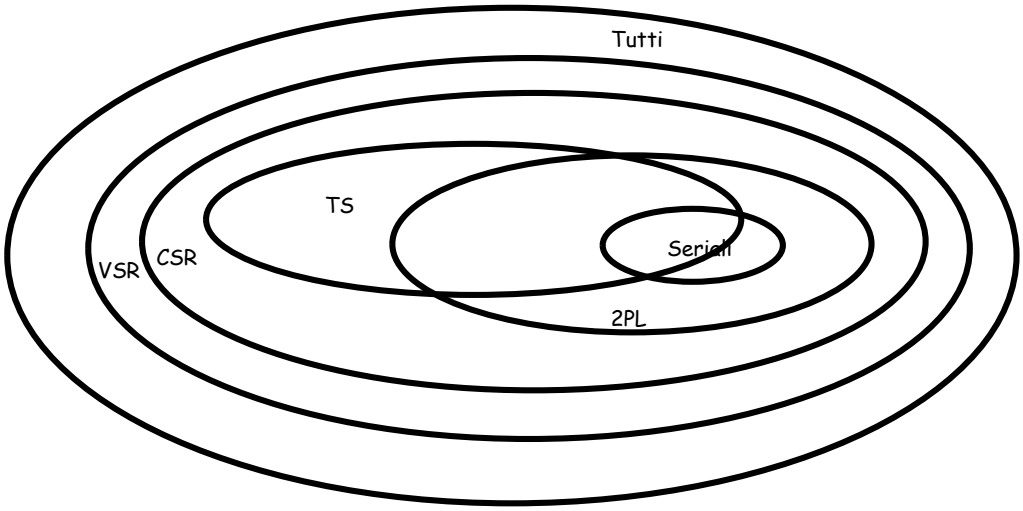
70

2PL vs TS

- Gli schedule TS sono automaticamente CSR: corrispondono ad una esecuzione seriale (quella in cui le transazioni sono eseguite nell'ordine in cui sono iniziate)
- Ma 2PL e TS sono incomparabili

71

CSR, VSR, 2PL e TS



72

Attenzione

- L'ordine seriale delle transazioni è fissato prima che le operazioni vengano richieste, tutti gli altri ordinamenti non sono accettati.
- Quando T1 comincia prima di T2, potrebbe essere abilitato uno schedule 2PL o CSR equivalente ad uno seriale T2 T1; col TS non è possibile, al limite T1 viene uccisa e poi fatta ripartire dopo T2.

73

2PL vs TS

- In 2PL le transazioni sono poste in attesa quando non è possibile acquisire un lock, in TS uccise e rilanciate
  - Le ripartenze sono di solito più costose delle attese:
  - conviene il 2PL
- 2PL può causare deadlock, TS no
  - mediamente si uccide una transazione ogni due conflitti, ma la probabilità di insorgenza di deadlock è molto minore della probabilità di un conflitto
  - conviene il 2PL

74

Risoluzione dello stallo

- Uno stallo corrisponde ad un ciclo nel grafo delle attese
- Tre tecniche di risoluzione

1. Timeout. Le transazioni rimangono in attesa di una risorsa per un tempo prefissato. Se, trascorso tale tempo, la risorsa non è ancora stata concessa, alla richiesta di lock viene data risposta negativa. In tal modo una transazione in potenziale statodi deadlock viene tolta dallo stato di attesa e di norma abortita.Tecnica molto semplice, usata dalla gran parte dei sistemicommerciali

problema: scelta dell'intervallo

2. Rilevamento dello stallo
   - ricerca di cicli nel grafo delle attese
3. Prevenzione dello stallo
   - Prevenzione: uccisione di transazioni "sospette"

75

Come scegliere il timeout

- Un valore troppo elevato tende a risolvere tardi i blocchi critici, dopo che le transazioni coinvolte hanno trascorso diverso tempo in attesa.
- Un valore troppo basso rischia di intepretare come blocchi anche situazioni in cui una transazione sta attendendo la disponibilità di una risorsa destinata a liberarsi, uccidendo la transazione e sprecando il lavoro già svolto.

76

Come scegliere la transazione da uccidere

- Politiche interrompenti: un conflitto può essere risolto uccidendo la transazione che possiede la risorsa (in tal modo, essa rilascia la risorsa che può essere concessa ad un'altra transazione).
  - Criterio aggiuntivo: uccidere le transazioni che hanno svolto meno lavoro (si spreca meno).
- Politiche non interrompenti: una transazione può essere uccisa solo nel momento in cui effettua una nuova richiesta.

77

Problema aggiuntivo dal criterio aggiuntivo

- Una transazione, all'inizio della propria elaborazione, accede ad un oggetto richiesto da molte altre transazioni, così è sempre in conflitto con altre transazioni e, essendo all'inizio del suo lavoro, viene ripetutamente uccisa.
- Non c'è deadlock, ma starvation.
  - Possibile soluzione: mantenere invariato il timestamp delle transazioni abortite e fatte ripartire, dando in questo modo priorità alle transazioni più anziane.

78

Esempio 1 cont.

- S: r1(y) w3(z) r1(z) r2(z) w3(x) w1(x) w2(x) r3(y)

b) Mostrare l'esecuzione delle operazioni in S quando
   1. È applicato il two phase locking stretto
   2. È applicato il protocollo basato su time-stamp

79

2PL stretto

r1(y)	r1_lock(y)
w3(z)	w3_lock(z)
r1(z)	T1 wait T3
r2(z)	T2 wait T3
w3(x)	w3_lock(x)
r3(y)	r3_lock(y)
commit T3	unlock (y, z, x) dequeue T1, T2
r1(z)	r1_lock(z)
r2(z)	r2_lock(z)
w1(x)	w1_lock(x)
commit T1	unlock (z, x)
w2(x)	w2_lock(x)
commit T2	unlock (z, x)

80

Time-stamp

Assumiamo che una transazione abortita venga fatta ripartire immediatamente con un nuovo time-stamp

r1(y)	RTM(y)=1
w3(z)	WTM(z)=3
r1(z)	abort, restart T1 come T4
r4(y)	RTM(y)=4
r4(z)	RTM(z)=4
r2(z)	abort, restart T2 come T5
r5(z)	RTM(z)=5
w3(x)	WTM(x)=3
r3(y)	RTM(y)=4
w4(x)	WTM(x)=4
w5(x)	WTM(x)=5

Letture e scritture delle transazioni

- In SQL:1999, le transazioni sono partizionate in transazioni read-only e transazioni read-write(read-write è il default).
- Le transazioni read-only non possono modificare né il contenuto né lo schema della base di dati e vengono gestite coi soli lock condivisi (read lock)

Livelli di isolamento in SQL:1999 (e JDBC)

- Per le transazioni **`read-only`** il livello di isolamento può essere scelto per ogni transazione
  - **`read uncommitted`** permette letture sporche, letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`read committed`** evita letture sporche ma permette letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`repeatable read`** evita tutte le anomalie esclusi gli inserimenti fantasma
  - **`serializable`** evita tutte le anomalie
- Nota:
  - la perdita di aggiornamento è sempre evitata

83

Livelli di isolamento: implementazione

- **`read uncommitted`**:
  - nessun lock in lettura (e non rispetta i lock altrui)
- **`read committed`**:
  - lock in lettura (e rispetta quelli altrui), ma senza 2PL
- **`repeatable read`**:
  - 2PL anche in lettura
- **`serializable`**:
  - 2PL

84

Transazioni read/write

- Sulle scritture si ha sempre il 2PL stretto (e quindi si evita la perdita di aggiornamento)