

Gestione della concorrenza

P. Rullo

Esecuzione concorrente di transazioni

- Un DBMS è un sistema che supporta l'esecuzione di più transazioni che operano su dati condivisi – sistema multiutente
- L'esecuzione seriale sarebbe troppo inefficiente
- Esempi:
 - Sistema di prenotazione di compagnie di treni, aerei, ecc.
 - BD bancarie
 - ...
- L'esecuzione concorrente consente di aumentare il throughput del sistema (numero di transazione per unità di tempo)

Esecuzione concorrente di transazioni

- Esecuzione concorrente: operazioni di una transazione si alternano ad operazioni di altre transazioni

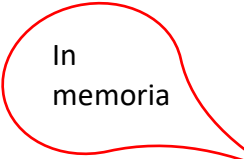
T1	T2
R(X)	
	R(X)
	W(X)
	Commit
R(Y)	
W(Y)	
Commit	

Il controllo della concorrenza: perché serve?

- L'esecuzione concorrente di più transazioni può generare risultati scorretti
- Infatti, durante l'esecuzione di una transazione, la BD può passare attraverso stati (temporaneamente) inconsistenti; se altre transazioni hanno accesso a tali stati, i risultati prodotti potrebbero essere scorretti
- Il controllo della concorrenza serve per evitare che si generino risultati scorretti, mantenendo un certo grado di concorrenza

Il controllo della concorrenza: perché serve?

- Sul conto corrente cointestato al sig. Rossi e alla moglie ci sono 100€
- Il marito fa un versamento di 20 €
- Contemporaneamente, la moglie preleva 80€



T1 (marito)	T2 (moglie)
r(X)	
X=X+20	r(X)
	X=X-80
	w(X)
w(X)	Commit
Commit	

Valore di X		
BD	T1	T2
100	100	
100	120	100
100	120	20
100	120	20
20	120	20
120	120	20

- Risultato finale X = 120€, invece di X=40! - come se il prelievo della moglie non fosse mai avvenuto
- Anomalia: LOST UPDATE
- Problema: T2 legge un valore *provvisorio* di X che T1 ha letto per modificare, ma la modifica non è ancor avvenuta

Il controllo della concorrenza: perché serve?

- Una transazione legge due volte lo stesso dato e, senza averlo modificato, trova valori diversi, ad es., due valori diversi del saldo sul cc

T1	T2
r(X)	
...	r(X)
	X=X-80
	w(X)
r(X)	Commit
Commit	

- T1 legge X=100 dalla BD;
- T2 scrive X=20 nella BD
- T2 rilegge X dalla BD e trova il valore 20
- Anomalia: LETTURE IRRIPETIBILI

Il controllo della concorrenza: perché serve?

Schedule concorrente S1

T1	T2
	r(X)
r(X)	X=X-20
	w(X)
	r(y)
	y=y+20
	w(y)
r(Y)	
Z=X+Y	

- In una BD, i due oggetti X e Y rispettano il vincolo di integrità $X+Y=100$
- T2 modifica X e Y nel rispetto del vincolo
- In T1 il vincolo risulta alterato: $X+Y=120$
- Anomalia: AGGIORNAMENTO FANTASMA

Equivalenza e Serializzabilità

- Una esecuzione concorrente è corretta se produce un risultato uguale a quello prodotto da una qualsiasi esecuzione seriale delle stesse transazioni
- Due esecuzioni delle stesse transazioni sono *equivalenti* se producono (sempre) gli stessi risultati
- Una esecuzione concorrente è *serializzabile* se è equivalente a qualche esecuzione seriale delle stesse transazioni
- Quindi, una esecuzione serializzabile è corretta

Equivalenza e Serializzabilità

Schedule concorrente S1

T1	T2
r(X)	
X=X-20	
w(X)	
	r(X)
	X=X*1,1
	w(X)
r(y)	
y=y+20	
w(y)	

Schedule seriale S2

T1	T2
r(X)	
X=X-20	
w(X)	
r(y)	
y=y+20	
w(y)	
	r(X)
	X=X*1,1
	w(X)

- T1: trasferisce 20€ dal cc X al cc Y
- T2: incrementa del 10% il saldo di X durante il trasferimento fondi
- S1: concorrente
- S2: seriale T1 < T2
- E' facile vedere che S1 e S2 producono (sempre) lo stesso risultato, in quanto in entrambi i casi T2 legge il valore definitivo di X prodotto da T1
- Quindi, S1 e S2 sono equivalenti
- Essendo S2 seriale, S1 è *serializzabile*

Equivalenza e Serializzabilità

- Sul conto corrente cointestato al sig. Rossi e alla moglie ci sono 100€
- Il marito fa un versamento di 20 €
- Contemporaneamente, la moglie preleva 80€

T1 (Rossi)	T2 (moglie)
r(X)	
X=X+20	r(X)
	X=X-80
	w(X)
w(X)	Commit
Commit	

Valore di X		
BD	T1	T2
100	100	
100	120	100
100	120	20
100	120	20
20	120	20
120	120	20

- Risultato finale X = 120€
- Diverso da quello prodotto da entrambi gli schedule seriali
 - T1 < T2: X = 40
 - T2 < T1: X = 40
- L'esecuzione concorrente di T1 e T2 non è serializzabile

Equivalenza e Serializzabilità

Schedule concorrente S1
(aggiornamento fantasma)

T1	T2
	r(X)
r(X)	X=X-20
	w(X)
	r(y)
	y=y+20
	w(y)
r(Y)	
Z=X+Y	

Schedule concorrente S2

T1	T2
	r(X)
	X=X-20
	w(X)
r(X)	
	r(y)
	y=y+20
	w(y)
r(Y)	
Z=X+Y	

- S1 NON è serializzabile – T1 legge il valore di X non ancora modificato da T2 ed il valore di Y modificato da T2. Quindi non può essere equivalente a nessuna delle due esecuzioni seriali $T1 < T2$ e $T2 < T1$
- S2 è serializzabile – equivalente a $T2 < T1$ in quanto T1 legge i valori di X e Y prodotti da T2

Grafo delle precedenze e c-serializzabilità

Grafo delle precedenze:

- Un nodo per ogni transazione
- Un arco orientato $(T1, T2)$ se
 - Nella transazione $T1$ c'è una $r(X)$ e in $T2$ una $w(X)$, e $r(X)$ precede $w(X)$
 - Nella transazione $T1$ c'è una $w(X)$ e in $T2$ una $r(x)$ o una $w(X)$, e la $w(X)$ in $T1$ precede l'operazione in $T2$
- Lo schedule è *c-serializzabile* (conflict-serializzabile) se il grafo è *aciclico*
- Se uno schedule è c-serializzabile è anche serializzabile (quindi corretto)

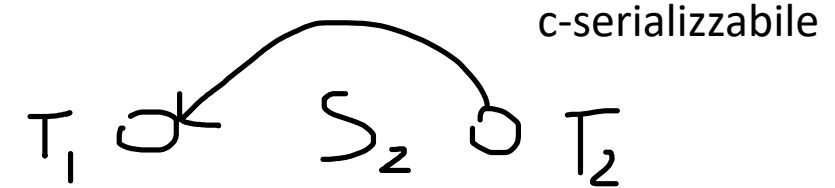
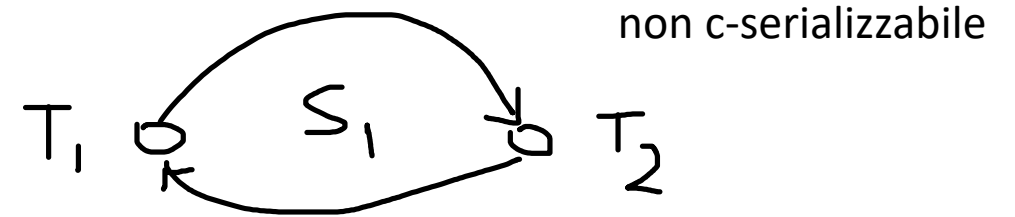
Grafo delle precedenze e c-serializzabilità

Schedule concorrente S1
(aggiornamento fantasma)

T1	T2
	r(X)
r(X)	X=X-20
	w(X)
	r(y)
	y=y+20
	w(y)
r(Y)	
Z=X+Y	

Schedule concorrente S2

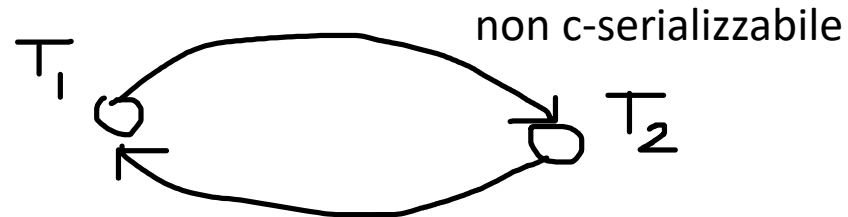
T1	T2
	r(X)
	X=X-20
	w(X)
r(X)	
	r(y)
	y=y+20
	w(y)
r(Y)	
Z=X+Y	



Grafo delle precedenze e c-serializzabilità

T1	T2
r(X)	
X=X-20	
w(X)	
	r(X)
	X=X*1,1
	w(X)
	r(y)
	y=y*1,1
	w(y)
r(y)	
y=y+20	
w(y)	

- T1 trasferisce 20€ da X a Y
- Mentre ciò avviene, T2 incrementa del 10% sia X che Y
- Lo schedule S non è serializzabile: T1 precede T2 rispetto alla variabile X, mentre T2 precede T1 rispetto a Y
- Quindi S non può produrre lo stesso risultato né di $T1 < T2$ né di $T2 < T1$
- Il problema è che T2 legge un valore provvisorio di Y



Come controllare la concorrenza

- Per evitare i problemi legati alla concorrenza, i DBMS usano comunemente la tecnica dei *lock* (blocchi) – la gestione del grafo delle precedenze sarebbe troppo costosa
- Per eseguire una operazione read o write è necessario acquisire prima un lock sulla risorsa cui si vuole accedere
 - *Read_lock(X)*: lock *condiviso* necessario per leggere l'oggetto X
 - *Write_lock(X)*: lock *esclusivo* necessario per modificare l'oggetto X
- Per rilasciare una risorsa X precedentemente acquisita si utilizza
 - *Unlock(X)*
- I lock e gli unlock non sono visibili al programmatore (vengono implicitamente generati)

Il Protocollo 2PL

1. Il protocollo 2PL (2-phase locking) prevede che dopo un'operazione di unlock non possa più essere effettuata una operazione di lock
2. Se gli unlock vengono fatti tutti dopo il COMMIT, si parla di *2PL stretto*

Il Protocollo 2PL

T1
r_lock(X);
r(X)
w_lock(X);
w(X)
r(X)
w(X)
unlock(X)
Commit

T2
r_lock(X);
r(X)
w_lock(X);
w(X)
unlock(X)
w_lock(Y)
r(Y)
w(Y)
unlock(X)
Commit

T3
r_lock(X);
r(X)
w_lock(X);
w(X)
r(X)
w(X)
Commit
unlock(X)

- T1 è una transazione 2PL *ben formata*
- T2 *non è ben formata*, in quanto vi è un w_lock preceduto da un unlock
- In T3 il 2PL è stretto (unlock dopo il commit)

Il Sistema di Gestione della Concorrenza

- Il Sistema di Gestione della Concorrenza (SGC) è un componente del DBMS che implementa la politica di gestione dei lock
- Quando una transazione deve fare una operazione su X, invia al SGC una richiesta di lock su X; in particolare
 - Una richiesta di `r_lock(X)` se l'operazione è `read(X)`
 - Una richiesta di `w_lock(X)` se l'operazione è `write(X)`

Il Sistema di Gestione della Concorrenza

T
r_lock(X)
r(X)
X=X-20
w_lock(X)
w(X)
....
unlock(X)

- Il SGC risponde alle richieste di lock secondo il seguente schema
- Il SGC è un automa a stati finiti

Su X *un'altra* transazione ha un lock di tipo

	no lock	r_lock	w_lock
r_lock	SI	SI	NO
w_lock	SI	NO	NO
unlock	errore	SI/NO	SI

T manda al SGC
una richiesta di
lock o unlock su X

Il Protocollo 2PL genera schedule c-serializzabili

S1

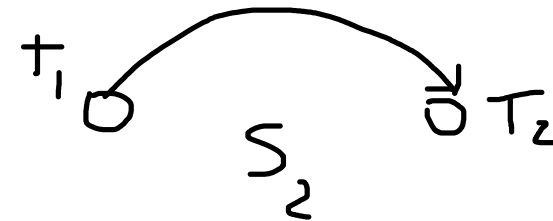
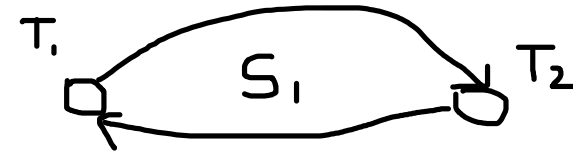
T1	T2
r(X)	
w(X)	
	r(Z)
	w(Z)
	r(X)
	w(X)
r(X)	
w(X)	

Lo schedule generato
dall'applicazione del 2PL (stretto) è
c-serializzabile – quindi corretto

S2

T1	T2
r_lock(X);	
r(X)	
w_lock(X);	
w(X)	r_lock(Z)
	r(Z)
	w_lock(Z)
	w(Z)
	r_lock(X)
r(X)	wait
w(X)	wait
unlock(X)	wait
	r(X)
	w(X)

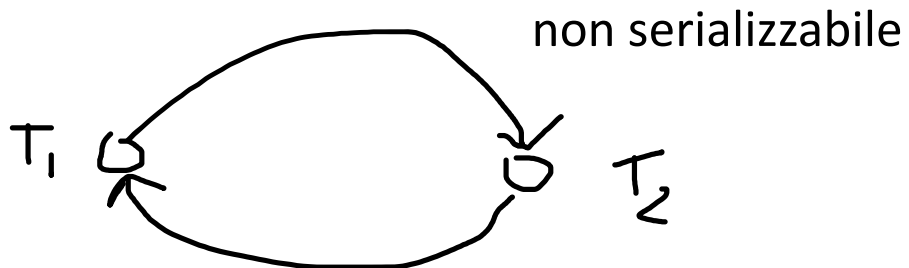
- S1 non è c-serializzabile
- S2, risultante dalla applicazione del 2PL, è c-serializzabile



Il Protocollo 2PL genera schedule c-serializzabili

Lecture irripetibili

T1	T2
r(X)	
...	r(X)
	X=X-80
	w(X)
r(X)	Commit
Commit	



T1	T2
rlock(X)	
r(X)	
	rlock(X)
...	r(X)
	w_lock(X)
r(X)	wait
unlock(X)	wait
Commit	w(x)



- La modifica di X è posticipata
- In tal modo, la T1 legge lo stesso valore di X

Il Problema della Lettura Sporca (dirty read)

- La transazione T1 aggiorna X nella BD, poi fa rollback annullando l'aggiornamento nella BD
- La transazione T2 legge X prima del rollback della T1; quindi legge un valore *provvisorio* di X – che poi non sarà più presente nella BD

T1	T2
r(X)	
X=X+20	
w(X)	
	r(X)
abort
...	

- T1 accredita 20€ sul cc X, portandolo, ad es., a 120€;
- T2 legge 120€ e su tale valore opera, ad es., una trattenuta in percentuale. Ma 120€ è un valore provvisorio, poi riportato a 100, quindi l'addebito risulta più alto del dovuto (cioè, di quanto sarebbe stato se applicato a 100€)

Il Problema della Lettura Sporca (dirty read)

- Il 2PL *stretto* risolve questo problema

2PL

T1	T2
r_lock(X)	
r(X)	
w_lock(X)	
w(X)	
unlock(X)	
	r_lock(X)
	r(X)
abort	...

2PL stretto

T1	T2
r_lock(X)	
r(X)	
w_lock(X)	
w(X)	
	r_lock(X)
	wait
	wait
abort	wait

- Nel 2PL stretto, l'unlock avviene dopo il Commit
- Quindi se T1 va in abort (prima del Commit), su X risulta un w_lock, e pertanto T2 NON può leggere il valore provvisorio di X (essendo in wait)
- Infatti T2 legge il valore di X prodotto da T1 solo se questa ha raggiunto il punto di Commit
- Altrimenti T2 legge il valore di X ripristinato dopo il rollback

Il Deadlock

- L'applicazione del 2PL può generare deadlock (abbraccio mortale)

Lost update

T1	T2
r(X)	
	r(X)
	w(X)
w(X)	Commit
Commit	

T1	T2
r_lock(X)	
r(X)	r_lock(X)
	r(x)
	w_lock(X);
w_lock(X)	wait
wait	wait
...

- T1 blocca T2 in scrittura
- T2 blocca T1 in scrittura
- T1 attende che T2 rilasci X, e viceversa
- Abbraccio mortale!

Come prevenire il Deadlock

- Utilizzare direttamente il w_lock, senza passare per il r_lock – quando ciò è possibile
- Ciò a scapito della efficienza, in quanto si rischia di bloccare troppi oggetti inutilmente

T1	T2
r(X)	
	r(X)
	w(X)
w(X)	Commit
Commit	

T1	T2
w_lock(X)	
r(X)	
	w_lock(X)
w(X)	wait
Commit	wait
unlock	
	r(X)
	w(X)
	Commit

Come risolvere una situazione di deadlock

- Time-out
 - Ad ogni richiesta di lock è associato un tempo massimo di attesa
 - Scaduto il tempo, la richiesta viene negata e la transazione 'uccisa'
- E' la tecnica più semplice e diffusa