CONTROLLO DELLA CONCORRENZA

1. Introduzione

Un criterio per classificare i sistemi di gestione di basi di dati è il numero di utenti che possono usare il sistema concorrentemente. Un sistema è *singolo-utente* se può essere usato da al più un utente alla volta ed è *multiutente* se molti utenti possono usarlo contemporaneamente; la maggior parte dei sistemi di gestione di basi di dati è del secondo tipo.

Se il sistema di calcolo ha più CPU allora è possibile il simultaneo processamento di due programmi da parte di due diverse CPU; tuttavia la maggior parte della teoria del controllo della concorrenza nelle basi di dati è stata sviluppata per sistemi di calcolo con una sola CPU. In tali sistemi i programmi sono eseguiti concorrentemente in modo *interleaved* (interfogliato): la CPU può eseguire un solo programma alla volta, tuttavia il sistema operativo permette di eseguire alcune istruzioni di un programma, sospendere quel programma, eseguire istruzioni di un altro programma e quindi ritornare ad eseguire istruzioni del primo. In tal modo l'esecuzione concorrente dei programmi è interleaved; ciò consente di tenere la CPU occupata quando un programma deve effettuare operazioni di I/O.

In un sistema di gestione di basi di dati multiutente la principale risorsa a cui i vari programmi accedono concorrentemente è la base di dati. L'esecuzione di una parte di un programma che rappresenta un'unità logica di accesso o modifica del contenuto della base di dati è detta *transazione*. Ci sono sistemi (ad esempio le basi di dati statistici) in cui gli utenti effettuano solo interrogazioni ma non modifiche; in tali sistemi l'esecuzione concorrente di più transazioni non crea problemi. Al contrario nei sistemi in cui vengono effettuate da più utenti sia operazioni di lettura che di scrittura (un tipico esempio di sistemi di questo tipo è costituito dai sistemi per la prenotazione di posti sui voli) l'esecuzione concorrente di più transazioni può provocare problemi se non viene controllata in qualche modo.

Prima di esaminare alcuni dei problemi che possono sorgere quando l'esecuzione concorrente di più transazioni non è controllata, introduciamo il concetto di schedule (piano di esecuzione) di un insieme di transazioni. Dato un insieme T di transazioni uno *schedule* S di T è un ordinamento delle operazioni nelle transazioni in T tale che per ogni transazione T in T se o_1 e o_2 sono due operazioni in T tali che o_1 precede o_2 in T allora o_1 precede o_2 in S (in altre parole uno schedule deve conservare l'ordine che le operazioni hanno all'interno delle singole transazioni). Qualsiasi schedule ottenuto permutando le transazioni in T è detto *seriale*.

Consideriamo le seguenti transazioni

T_1
read(X)
X:=X-N
write(X)
read(Y)
Y:=Y+N
write(Y)

T_2
read(X)
X:=X+M
write(X)

e i seguenti schedule di $\{T_1, T_2\}$

T_1	T_2
read(X)	
X:=X-N	
	read(X)
	X:=X+M
write(X)	
read(Y)	
	write(X)
Y:=Y+N	
write(Y)	

T_1	T_2
read(X)	
X:=X-N	
write(X)	
	read(X)
	X:=X+M
	write(X)
read(Y)	
T_1 *fallisce*	

Nel primo caso l'aggiornamento di *X* prodotto da T_1 viene perso in quanto T_2 legge il valore di X prima che l'aggiornamento prodotto da T_1 sia stato reso permanente. Nel secondo caso T_2 legge e aggiorna il valore di *X* dopo che l'aggiornamento prodotto da T_1 è stato reso permanente, ma prima che venga ripristinato il vecchio valore di *X* in conseguenza del fallimento di T_1 .

Consideriamo ora una transazione T_3 che somma i valori di X e di Y. Il seguente schedule di $\{T_1,T_3\}$ fa sì che la somma prodotta da T_3 sia la somma del valore di X dopo che X è stato aggiornato da T_1 e del valore di Y prima che sia stato aggiornato da T_1.

T_1	T_3
	somma:=0
read(X)	
X:=X-N	
write(X)	
	read(X)
	somma:=somma+X
	read(Y)
	somma:=somma+Y
read(Y)	
Y:=Y+N	
write(Y)	

In tutti e tre i casi visti siamo portati a considerare gli schedule non corretti in quanto i valori prodotti non sono quelli che si avrebbero se le due transazioni fossero eseguite nel modo "naturale" cioè sequenzialmente.

In generale possiamo osservare che l'esecuzione naturale e, quindi, intuitivamente corretta di un insieme di transazioni è quella sequenziale; la possibilità di eseguire concorrentemente un insieme di transazioni, come si è detto, è introdotta nei sistemi per motivi di efficienza. Pertanto tutti gli schedule seriali sono corretti e uno schedule non seriale è corretto se è *serializzabile*, cioè se è "equivalente" ad uno schedule seriale. Sorge quindi la necessità di definire un concetto di equivalenza di schedule.

La più semplice definizione di equivalenza potrebbe essere basata sul confronto del risultato: due schedule sono equivalenti se producono lo stesso stato finale. Tale definizione non è però soddisfacente in quanto due schedule potrebbero produrre lo stesso stato finale solo per alcuni valori iniziali. Consideriamo ad esempio le due transazioni

T_1
read(X)
X:=X+5
write(X)

T_2
read(X)
*X:=X*1.5*
write(X)

e gli schedule

T_1	T_2
read(X)	
	read(X)
X:=X+5	
	*X:=X*1.5*
	write(X)
write(X)	

T_1	T_2
read(X)	
	read(X)
	*X:=X*1.5*
X:=X+5	
write(X)	
	write(X)

Tali schedule producono gli stessi valori solo se il valore iniziale di *X* è 10; ma producono valori diversi in tutti gli altri casi. Problemi di questo tipo potrebbero essere evitati sfruttando proprietà algebriche che garantiscano che il risultato è lo stesso indipendentemente dai valori iniziali delle variabili; tuttavia tale soluzione richiederebbe dei costi inaccettabili (che non sono giustificati dallo scopo che si vuole raggiungere). Pertanto si fa l'assunzione più restrittiva che *due valori sono uguali solo se sono prodotti da esattamente la stessa sequenza di operazioni*. Quindi, ad esempio, date le due transazioni

T_1
read(X)
X:=X+N
write(X)

T_2
read(X)
X:=X-M
write(X)

i due schedule

T_1	T_2
read(X)	
X:=X+N	
write(X)	
	read(X)
	X:=X-M
	write(X)

T_1	T_2
	read(X)
	X:=X-M
	write(X)
read(X)	
X:=X+N	
write(X)	

non sono considerati equivalenti.

Oltre alla definizione di equivalenza, un altro elemento che ha influenza sulla complessità del problema di decidere se uno schedule è serializzabile (cioè se è equivalente ad uno schedule seriale) è costituito dal fatto che il valore calcolato da una transazione per ogni dato sia dipendente o meno dal vecchio valore di quel dato. Nel primo caso il problema della serializzabilità può essere risolto in tempo polinomiale con un semplice algoritmo su grafi; nel secondo caso il problema risulta essere NP-completo.

Nella pratica è difficile testare la serializzabilità di uno schedule. Infatti l'ordine di esecuzione delle operazioni delle diverse transazioni è determinato in base a diversi fattori: il carico del sistema, l'ordine temporale in cui le transazioni vengono sottomesse al sistema e le loro priorità. Pertanto è praticamente impossibile determinare in anticipo come le operazioni saranno interleaved, cioè in quale ordine verranno eseguite; d'altra parte, se prima si eseguono le operazioni e poi si testa la serializzabilità dello schedule, i suoi effetti devono essere annullati se lo schedule risulta non serializzabile. Inoltre quando le transazioni vengono sottomesse al sistema in modo continuo è difficile stabilire quando uno schedule comincia e quando finisce. Quindi l'approccio seguito nei sistemi è quello di determinare metodi che garantiscano la serializzabilità di uno schedule eliminando così la necessità di dover testare ogni volta la serializzabilità di uno schedule. Uno di tali metodi consiste nell'imporre dei *protocolli*, cioè delle regole, alle transazioni in modo da garantire la serializzabilità di ogni schedule. Questi protocolli usano tecniche di *locking* (cioè di controllo dell'accesso ai dati) per prevenire l'accesso concorrente ai dati. Altri metodi di controllo usano i *timestamp* delle transazioni, cioè degli identificatori delle transazioni che vengono generati dal sistema e in base ai quali le operazioni delle transazioni possono essere ordinate in modo da assicurare la serializzabilità.

2. Item

Tutte le tecniche per la gestione della concorrenza richiedono che la base di dati sia partizionata in *item*, cioè in unità a cui l'accesso è controllato. Le dimensioni degli item devono essere definite in base all'uso che viene fatto della base di dati in modo tale che in media una transazione acceda a pochi item. Ad esempio se la transazione tipica su una base di dati relazionale è la ricerca di una

tupla mediante un indice, è appropriato trattare le tuple come item; se invece la transazione tipica consiste nell'effettuazione di un join di due relazioni, è opportuno considerare le relazioni come item. Le dimensioni degli item usate da un sistema sono dette la sua granularità. Una granularità grande permette una gestione efficiente della concorrenza; una piccola granularità può invece sovraccaricare il sistema, ma consente l'esecuzione concorrente di molte transazioni.

3. Tecniche di locking per il controllo della concorrenza

Queste tecniche fanno uso del concetto di lock. Un *lock* è un privilegio di accesso ad un singolo item. In pratica è una variabile associata all'item il cui valore descrive lo stato dell'item rispetto alle operazioni che possono essere effettuate su di esso. Un lock viene richiesto da una transazione mediante un'operazione di *locking* e viene rilasciato mediante un'operazione di *unlocking*; fra l'esecuzione di un'operazione di locking su un certo item *X* e l'esecuzione di un'operazione di unlocking su *X* diciamo che la transazione mantiene un lock su *X*. Sono stati studiati diversi tipi di lock; in ogni caso si assume che una transazione debba effettuare un'operazione di locking ogni volta che deve leggere o scrivere un item e che l'operazione agisca come primitiva di sincronizzazione, cioè se una transazione richiede un lock su un item su cui un'altra transazione mantiene un lock, la transazione non può procedere finchè il lock non viene rilasciato dall'altra transazione. Inoltre si assume che ciascuna transazione rilascia ogni lock che ha ottenuto. Uno schedule è detto *legale* se obbedisce a queste regole.

3.1 Lock binario

Un lock binario può assumere solo due valori *locked* e *unlocked*. Le transazioni fanno uso di due operazioni *lock(X)* e *unlock(X)*; la prima serve per richiedere l'accesso all'item *X*, la seconda per rilasciare l'item *X* consentendone l'accesso ad altre transazioni. Se una transazione richiede l'accesso ad un item *X* mediante un *lock(X)* e il valore della variabile è *locked* la transazione viene messa in attesa, altrimenti viene consentito alla transazione l'accesso ad *X* e alla variabile associata ad *X* viene assegnato il valore *locked*. Se una transazione rilascia un item *X* mediante un *unlock(X)*, alla variabile associata ad *X* viene assegnato il valore *unlocked*; in tal modo se un'altra transazione è in attesa di accedere ad *X* l'accesso gli viene consentito.

Consideriamo di nuovo le due transazioni dell'esempio precedente e vediamo come l'uso dei lock può prevenire il problema dell'"aggiornamento perso". Le transazioni T_1 e T_2 risultano modificate nel modo seguente

T_1
lock(X)
read(X)
X:=X-N
write(X)
unlock(X)
lock(Y)

T_2
lock(X)
read(X)
X:=X+M
write(X)
unlock(X)

read(Y)
Y:=Y+N
write(Y)
unlock(Y)

Uno schedule legale di T_1 e T_2 è il seguente

T_1	T_2
lock(X)	
read(X)	
X:=X-N	
write(X)	
unlock(X)	
	lock(X)
	read(X)
	X:=X+M
	write(X)
	unlock(X)
lock(Y)	
read(Y)	
Y:=Y+N	
write(Y)	
unlock(Y)	

Il più semplice modello per le transazioni è quello che considera una transazione come una sequenza di operazioni di *lock* e *unlock*. Si assume che ogni operazione di *lock* su un item *X* implica la lettura di *X* e ogni operazione di *unlock* di un item *X* implica la scrittura di *X*. Il nuovo valore dell'item viene calcolato da una funzione che è associata in modo univoco ad ogni coppia *lock-unlock* ed ha per argomenti tutti gli item letti (locked) dalla transazione prima dell'operazione di *unlock*. I valori che un item assume durante l'esecuzione di una transazione sono formule costruite applicando le funzioni suddette ai valori iniziali degli item. Due schedule sono *equivalenti* se le formule che danno i valori finali per ciascun item sono le stesse per i due schedule.

Consideriamo ad esempio le due transazioni seguenti

T_1
lock(X)
unlock(X) $f_1(X)$
lock(Y)
unlock(Y) $f_2(X,Y)$

T_2
lock(Y)
unlock(Y) $f_3(Y)$
lock(X)
unlock(X) $f_4(X,Y)$

e il seguente schedule S di $\{T_1, T_2\}$

T_1	T_2
lock(X)	
unlock(X)	
	lock(Y)
	unlock(Y)
lock(Y)	
unlock(Y)	
	lock(X)
	unlock(X)

Se indichiamo con X_0 e Y_0 i valori iniziali di X e Y, abbiamo che il valore finale per X prodotto da S è dato dalla formula $f_4(f_1(X_0), Y_0)$. D'altra parte il valore finale per X prodotto dallo schedule seriale T_1, T_2 è dato dalla formula $f_4(f_1(X_0), f_2(X_0,Y_0))$, mentre il valore finale per X prodotto dallo schedule seriale T_2, T_1 è dato dalla formula $f_1(f_4(X_0, Y_0))$; pertanto S non è serializzabile in quanto non è equivalente a nessuno schedule seriale di $\{T_1, T_2\}$.

Consideriamo ora le due transazioni seguenti

T_1
lock(X)
unlock(X) $f_1(X)$
lock(Y)
unlock(X) $f_2(X,Y)$

T_2
lock(X)
unlock(X) $f_3(X)$
lock(Y)
unlock(Y) $f_4(X,Y)$

e il seguente schedule S di $\{T_1, T_2\}$

T_1	T_2
lock(X)	
unlock(X)	
	lock(X)
	unlock(X)
lock(Y)	
unlock(Y)	
	lock(Y)
	unlock(Y)

Se di nuovo indichiamo con X_0 e Y_0 i valori iniziali di X e Y, abbiamo che il valore finale per X e Y prodotti da S sono dati rispettivamente dalle formule $f_3(f_1(X_0))$ e $f_4(f_1(X_0), f_2(X_0,Y_0))$. Poiché tali formule coincidono con quelle prodotte dallo schedule seriale T_1, T_2 lo schedule S è serializzabile.

La serializzabilità di uno schedule in questo semplice modello, può essere testata mediante un semplice algoritmo su grafi diretti. L'idea su cui si basa tale algoritmo è la seguente. Dato uno schedule, per ogni item si esamina l'ordine in cui le varie transazioni fanno un lock su quell'item; se lo schedule è serializzabile questo ordine deve essere consistente con quello di uno schedule seriale; pertanto se l'ordine imposto sulle transazioni da un certo item è diverso da quello imposto da un altro item, lo schedule non è serializzabile.

L'algoritmo lavora nel modo seguente.

Algoritmo 1
Dato uno schedule S

- crea un grafo diretto G (*grafo di serializzazione*) i cui nodi corrispondono alle transazioni; in tale grafo c'è un arco diretto da T_i a T_j se in S T_i esegue un *unlock(X)* e T_j esegue il successivo *lock(X)* (il significato intuitivo dell'esistenza di un arco da T_i a T_j è che T_j legge il valore per X scritto da T_i e quindi se esiste uno schedule seriale equivalente ad S in tale shedule T_i deve precedere T_j)
- verifica se G ha un ciclo. Se G ha un ciclo allora S non è serializzabile; altrimenti esiste un ordinamento S' delle transazioni tale che T_i precede T_j se c'è in G un arco diretto da T_i a T_j; tale ordinamento può essere ottenuto applicando a G il procedimento noto come *ordinamento topologico* consistente nell'eliminare ricorsivamente da un grafo diretto i nodi che non hanno archi entranti (l'ordine di eliminazione dei nodi fornisce lo schedule seriale S')

Il seguente teorema prova la correttezza dell'algoritmo.

Teorema 1. L'Algoritmo 1 determina correttamente se uno schedule è serializzabile.
Dim. Cominciamo con il dimostrare che se il grafo di serializzazione G contiene un ciclo allora S non è serializzabile. Supponiamo, per assurdo, che G contenga un ciclo $T_1, T_2, \ldots, T_k, T_1$ e che esista uno schedule seriale S' equivalente ad S. Sia T_i, $1 \le i \le k$, la transazione del ciclo che compare per prima in S' e sia X l'item che causa la presenza in G dell'arco da T_{i-1} a T_i. Il valore finale per X prodotto da S' è dato da una formula in cui compare una funzione f associata ad una coppia *lock-unlock* in T_{i-1} e almeno uno degli argomenti di f è una formula in cui compare una funzione g associata ad una coppia *lock-unlock* in T_i. D'altra parte in S l'effetto di T_{i-1} su X precede quello di T_i su X; quindi nella formula che dà il valore finale per X prodotto da S la funzione f compare più internamente di g (f è applicata prima di g). Pertanto S' ed S non sono equivalenti (contraddizione).
Mostriamo ora che se G non ha cicli allora S è serializzabile. A tal fine definiamo *profondità* di una transazione T la lunghezza del più lungo cammino in G da un qualsiasi nodo a T.

Sia *S'* lo schedule seriale costruito dall'algoritmo; mostreremo per induzione sulla profondità delle transazioni che ogni transazione *T* per ogni item su cui effettua un'operazione di *lock* legge in *S* lo stesso valore che legge in *S'* (e quindi per ogni item su cui effettua un'operazione di *lock* produce in *S* lo stesso valore che produce in *S'*).

Base dell'induzione. Se per una transazione *T* la profondità è *0* vuol dire che in *G* non ci sono archi entranti in *T*; pertanto in *S T* legge solo valori iniziali. D'altra parte in *S' T* viene prima di qualsiasi transazione che effettua un'operazione di *lock* su un item su cui *T* effettua un'operazione di *lock*. Infatti, se *X* è un item su cui *T* effettua un'operazione di *lock* e T_{i1}, T_{i2},…, T_{ik} è la sequenza in *S* delle transazioni che effettuano un'operazione di lock su X, T_{i1}, T_{i2},…, T_{ik} è un cammino in *G*; poiché *T* deve comparire in tale cammino e in *G* non ci sono archi entranti in *T*, si deve avere $T=T_{i1}$ e quindi nessun altro nodo del cammino può essere eliminato dal procedimento di ordinamento topologico prima di *T*.

Induzione. Cominciamo con il mostrare che ogni transazione per ogni item su cui effettua un'operazione di *lock* legge sia in *S* che in *S'* il valore prodotto da una stessa transazione. Supponiamo, per assurdo, che esistano una transazione *T* e un item *X* tali che *T* legge in *S* il valore di *X* prodotto da una transazione *T'* e in *S'* il valore prodotto da un'altra transazione *T''*. Sia T_{i1}, T_{i2},…, T_{ik} la sequenza in *S* delle transazioni che effettuano un'operazione di *lock* su *X*; T_{i1} , T_{i2} ,…, T_{ik} è un cammino in *G* in cui *T'* compare immediatamente prima di *T*. Anche *T''* compare in tale cammino, e quindi deve comparire prima di *T'*. Poiché in *S' T''* deve seguire *T'*, *S'* non può essere ottenuto da *G* mediante il procedimento di ordinamento topologico (contraddizione).

Sia *T* una transazione che ha profondità *d*, *d*>0, in *G*; per quanto visto *T*, per ogni item su cui effettua un'operazione di *lock*, legge sia in *S* che in *S'* il valore prodotto da una stessa transazione *T'*. Poiché *T'* ha profondità minore di *d* per l'ipotesi induttiva T', per ogni item su cui effettua un'operazione di *lock*, legge sia in *S* che in *S'* il valore prodotto da una stessa transazione e, quindi, produce sia in *S* che in *S'* lo stesso valore. Pertanto *T*, per ogni item su cui effettua un'operazione di *lock*, legge sia in *S* che in *S'* lo stesso valore. ♦

Diciamo che una transazione obbedisce al protocollo di locking a due fasi, o più semplicemente che *è a due fasi*, se prima effettua tutte le operazioni di *lock* (*fase di locking*) e poi tutte le operazioni di *unlock* (*fase di unlocking*). Mostreremo che il protocollo di locking a due fasi garantisce la serializzabilità; più precisamente, si ha che:

Teorema 2. Sia T un insieme di transazioni. Se ogni transazione in T è a due fasi allora ogni schedule di T è serializzabile.

Dim. Sia S uno schedule di T. Supponiamo, per assurdo, che S non sia serializzabile. Per il Teorema 1, il grafo di serializzazione *G* di *S* contiene un ciclo T_1, T_2 ,…, T_k , T_1; ciò significa che T_{i+1}, *i=1,… k-1*, effettua un'operazione di *lock*, su un item su cui T_i ha effettuato un'operazione di *unlock* e che T_1 effettua un'operazione di *lock*, su un item su cui T_k ha effettuato un'operazione di *unlock*. Pertanto in *S* T_1 effettua un'operazione di *lock* dopo aver effettuato un'operazione di *unlock* e quindi T_1 non è a due fasi (contraddizione). ♦

Mostreremo ora che se una transazione T_1 non è a due fasi, esiste sempre una transazione T_2 tale che esiste uno schedule di $\{T_1, T_2\}$ che non è serializzabile. Infatti, se T_1 non è a due fasi effettua un'operazione di *lock* dopo aver effettuato un'operazione di *unlock*:

T_1
⋮
unlock(X)
⋮
lock(Y)
⋮

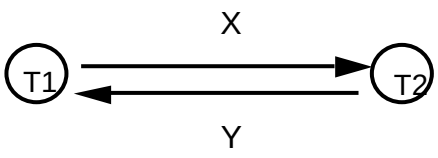
Sia T_2 la seguente transazione

T_2
lock(X)
lock(Y)
unlock(X)
unlock(Y)

Il seguente schedule di $\{T_1, T_2\}$ non è serializzabile:

T_1	T_2
⋮	
unlock(X)	
⋮	
	lock(X)
	lock(Y)
	unlock(X)
	unlock(Y)
⋮	
lock(Y)	
⋮	

Infatti il suo grafo di serializzazione



contiene un ciclo. Naturalmente possono esistere schedule di transazioni che non sono a due fasi che sono serializzabili. Tuttavia poiché è normale non conoscere l'insieme di transazioni con cui

una transazione può essere eseguita concorrentemente siamo costretti a richiedere che tutte le transazioni siano a due fasi perché sia garantita la serializzabilità di ogni schedule.

4. **Deadlock e livelock**

In un sistema che utilizza i lock per il controllo della concorrenza si possono verificare delle situazioni anomale note con il nome di deadlock e livelock.

Un *livelock* si verifica quando una transazione aspetta indefinitamente che gli venga garantito un lock su un certo item. Ad esempio, consideriamo il caso di una transazione *T* in attesa di effettuare un lock su un item *X* su cui un'altra transazione T_1 mantiene un lock; se quando T_1 rilascia X un'altra transazione T_2 ottiene un lock su *X*, *T* deve rimanere in attesa; poiché questa situazione può ripetersi un numero indefinito di volte, *T* può rimanere indefinitamente in attesa di ottenere un lock su *X*.

Un *deadlock* si verifica quando ogni transazione in un insieme T è in attesa di ottenere un lock su un item sul quale qualche altra transazione in T mantiene un lock.

Entrambi i problemi si possono presentare in un qualsiasi sistema in cui i processi possono essere eseguiti concorrentemente e quindi sono stati largamente studiati nell'ambito dei sistemi operativi. Il primo problema può essere risolto con una strategia *first came- first served*; in altre parole il sistema mantiene memoria delle successive richieste di lock in modo che, quando un item *X* viene rilasciato, viene garantito un lock su *X* alla transazione che per prima ha richiesto un lock su *X*. Un'altra possibile strategia può essere quella di eseguire le transazioni in base alle loro priorità e di aumentare la priorità di una transazione all'aumentare del tempo in cui rimane in attesa in modo che possa ad un certo punto assumere la priorità più alta ed essere eseguita.
Per risolvere il secondo problema possono essere seguiti due approcci. Un approccio è preventivo in quanto cerca di evitare il verificarsi di situazioni di stallo adottando opportuni protocolli; l'altro invece si preoccupa di risolvere le situazioni di stallo quando si verificano. Il sussistere di una situazione di stallo può essere rilevato mantenendo un gra*fo di attesa*, cioè un grafo i cui nodi sono le transazioni e in cui c'è un arco $T_1 \rightarrow T_2$ se la transazione T_1 è in attesa di ottenere un lock su un item sul quale T_2 mantiene un lock; se in tale grafo c'è un ciclo si sta verificando una situazione di stallo che coinvolge le transazioni nel ciclo; per risolverla occorre che almeno una transazione nel ciclo sia *rolled-back* (cioè la transazione viene abortita, i suoi effetti sulla base di dati vengono annullati ripristinando i valori dei dati precedenti l'inizio della sua esecuzione e, infine, tutti i lock mantenuti dalla transazione vengono rilasciati) e quindi venga fatta ripartire.

5. **Protocollo di locking a due fasi stretto**

Oltre al verificarsi di un deadlock, ci sono altri motivi per cui una transazione deve essere abortita, ad esempio: perché ha cercato di effettuare un accesso non autorizzato oppure perché ha cercato di eseguire una divisione per 0. Il punto in cui una transazione non può più essere abortita per uno dei suddetti motivi, cioè il punto in cui ha ottenuto tutti i lock che gli sono necessari e ha effettuato tutti

i calcoli nell'area di lavoro, viene detto *punto di commit* della transazione; una transazione che ha raggiunto il suo punto di commit viene detta *committed* altrimenti viene detta *attiva*. Quando una transazione raggiunge il suo punto di commit effettua un'*operazione di commit* (nei sistemi reali tale operazione prevede lo svolgimento di diverse azioni, ma per i nostri scopi serve solo a marcare il punto di commit della transazione). I dati scritti da una transazione sulla base di dati prima che abbia raggiunto il punto di commit vengono detti *dati sporchi*. Il fatto che un dato sporco possa essere letto da qualche altra transazione può causare un effetto di *roll-back a cascata*.

Consideriamo il seguente schedule

T_1	T_2
wlock(X)	
read(X)	
X:=X-N	
write(X)	
unlock(X)	
	rlock(X)
	wlock(Z)
	read(X)
	read(Z)
	Z:=Z+X
	write(Z)
	commit
	unlock(X)
	unlock(Z)
wlock(Y)	
read(Y)	
Y:=Y+N	
write(Y)	
commit	
unlock(Y)	

Se la transazione T_1 viene abortita dopo che ha letto Y, il valore di X scritto da T_1 è un dato sporco; infatti, quando T_1 viene abortita è necessario annullare gli effetti di T_1 sulla base di dati ripristinando il vecchio valore di X (quello letto da T_1). Ma allora è necessario annullare anche gli effetti di T_2 sulla base di dati in quanto il valore di Z prodotto da T_2 è calcolato a partire dal valore di X scritto T_1 . Pertanto sia T_1 che T_2 devono essere rolled-back.

Per evitare questo fenomeno detto *rollback a cascata* occorre impedire alle transazioni di leggere dati sporchi. Ciò può essere ottenuto adottando un protocollo di locking a due fasi stretto.

Una transazione soddisfa il *protocollo a due fasi stretto* se:

1. non scrive sulla base di dati fino a quando non ha raggiunto il suo punto di commit
2. non rilascia un lock finchè non ha finito di scrivere sulla base di dati.

La condizione 1 garantisce che se una transazione è abortita allora non ha modificato nessun item nella base di dati; la condizione 2 garantisce che quando una transazione legge un item scritto da un'altra transazione quest'ultima non può essere abortita (quindi nessuna transazione legge dati sporchi).

Perché la transazione T_1 nell'esempio precedente soddisfi il protocollo di locking a due fasi stretto deve essere modificata nel modo seguente.

T_1
wlock(X)
read(X)
X:=X-N
wlock(Y)
read(Y)
Y:=Y+N
commit
write(X)
write(Y)
unlock(X)
unlock(Y)

I protocolli di locking a due fasi stretti possono essere classificati in *protocolli conservativi* e *protocolli aggressivi*; i primi cercano di evitare il verificarsi di situazioni di stallo, i secondi cercano di processare le transazioni il più rapidamente possibile anche se ciò può portare a situazioni di stallo e, quindi, alla necessità di abortire qualche transazione.

La versione più conservativa di un protocollo di locking a due fasi stretto è quella che impone ad una transazione di richiedere all'inizio tutti gli item di cui può avere bisogno. Lo scheduler permette alla transazione di procedere solo se tutti i lock che ha richiesto sono disponibili, altrimenti la mette in una coda di attesa. In tal modo non possono verificarsi deadlock, ma possono ancora verificarsi livelock. Per evitare anche il verificarsi di livelock si può impedire ad una transazione *T* di procedere (anche se tutti i lock da essa richiesti sono disponibili) se c'è un'altra transazione che precede *T* nella coda che è in attesa di ottenere un lock richiesto da *T*. E' facile vedere che se si adotta un protocollo in cui tutti i lock sono richiesti all'inizio e uno scheduler che garantisce ad una transazione T tutti i lock richiesti se e solo se:

1. tutti i lock sono disponibili
2. nessuna transazione che precede *T* nella coda è in attesa di un lock richiesto da *T*

allora non si possono verificare né dealdock né livelock. Infatti per la condizione 1 nessuna transazione che mantiene un lock può essere in attesa di un lock mantenuto da un'altra transazione; inoltre, la condizione 2 garantisce che dopo un tempo finito ogni transazione raggiunge la testa della coda e quindi viene eseguita.
Il protocollo conservativo esaminato presenta due inconvenienti. Infatti l'esecuzione di una transazione può essere ritardata dal fatto che non può ottenere un lock anche se molti passi della transazione potrebbero essere eseguiti senza quel lock; inoltre una transazione è costretta a richiedere un lock su ogni item che potrebbe essergli necessario anche se poi di fatto non l'utlizza. Ad esempio, se una transazione deve effettuare una ricerca su un file con indice sparso e gli item sono i blocchi, la transazione deve effettuare un lock su ogni blocco del file principale e del file indice anche se poi accederà soltanto ad alcuni blocchi del file indice e ad un solo blocco del file principale.

La versione più aggressiva di un protocollo di locking a due fasi stretto è quella che impone ad una transazione di richiedere un lock su un item immediatamente prima di leggerlo o scriverlo. Se le transazioni soddisfano tale protocollo possono verificarsi situazioni di stallo; un modo per evitare ciò è quello di definire un ordinamento sugli item e di imporre alle transazioni di richiedere i lock in accordo a tale ordinamento. In tal modo non possono verificarsi deadlock. Infatti, sia T un insieme di transazioni (che richiedono i lock in accordo all'ordinamento fissato sull'insieme degli item) tale che ogni transazione T_k è in attesa di un item A_k e mantiene un lock su almeno un item (si osservi che se una transazione T in T non soddisfa la seconda condizione allora l'insieme $T - \{T\}$ è ancora un insieme di transazioni in stallo) e sia A_i il primo item nell'ordinamento fissato sull'insieme degli item; allora la transazione T_i che è in attesa di A_i non può mantenere un lock su alcun item (contraddizione). Tuttavia il metodo di ordinare gli item non è molto praticabile perché non sempre una transazione può scegliere l'ordine in cui richiedere i lock.

Un protocollo aggressivo è adatto a situazioni in cui la probabilità che due transazioni richiedano un lock su uno stesso item è bassa (e quindi è bassa la probabilità che si verifichi una situazione di stallo), in quanto evita al sistema il sovraccarico dovuto alla gestione dei lock (decidere se garantire un lock su un dato item ad una data transazione, gestire la tavola dei lock, mettere le transazioni in una coda o prelevarle da essa). Un protocollo conservativo è invece adatto a situazioni opposte in quanto evita al sistema il sovraccarico dovuto alla gestione dei deadlock (rilevare e risolvere situazioni di stallo, eseguire parzialmente transazioni che poi vengono abortite, rilascio dei lock mantenuti da transazioni abortite).

6. **Controllo della concorrenza basato sui timestamp**

I metodi per il controllo della concorrenza basati sui timestamp ordinano le transazioni in base ai loro timestamp. Il timestamp è assegnato ad una transazione dallo scheduler quando la transazione ha inizio. A tale scopo lo scheduler può o gestire un contatore (ogni volta che ha inizio una transazione tale contatore viene incrementato e il suo valore è il timestamp della transazione)

oppure usare l'orologio interno della macchina (in tal caso il timestamp è l'ora di inizio della transazione).

In tale approccio al controllo della concorrenza, uno schedule è serializzabile se è equivalente allo schedule seriale in cui le transazioni compaiono ordinate in base al loro timestamp.

Dato uno schedule occorre verificare che, per ciascun item acceduto da più di una transazione, l'ordine con cui le transazioni accedono all'item non viola la serializzabilità dello schedule. A tale scopo vengono associati a ciascun item *X* due timestamp:

- il *read timestamp* di *X*, denotato con *read_TS(X)*, che è il più grande fra tutti i timestamp di transazioni che hanno letto con successo *X*;
- il *write timestamp* di *X*, denotato con *write_TS(X)*, che è il più grande fra tutti i timestamp di transazioni che hanno scritto con successo *X*.

Ogni volta che una transazione *T* cerca di eseguire un *read(X)* o un *write(X)*, occorre confrontare il timestamp *TS(T)* di *T* con il read timestamp e il write timestamp di *X* per assicurarsi che l'ordine basato sui timestamp non è violato. L'algoritmo per il controllo della concorrenza opera nel modo seguente:

1. ogni volta che una transazione *T* cerca di eseguire l'operazione *write(X*):
   a) se *read_TS(X)>TS(T), T* viene rolled back; infatti in tal caso qualche transazione con un timestamp maggiore di *TS(T)* (cioè una transazione che segue *T* nell'ordinamento basato sui timestamp) ha già letto il valore di *X* prima che *T* abbia potuto scriverlo, violando in tal modo l'ordinamento basato sui timestamp;
   b) se *write_TS(X)>TS(T)*, l'operazione di scrittura non viene effettuata; infatti in tal caso qualche transazione *T'* con un timestamp maggiore di *TS(T)* (cioè una transazione che segue *T* nell'ordinamento basato sui timestamp) ha già scritto il valore di *X* e quindi l'esecuzione dell'operazione di scrittura provocherebbe la perdita di tale valore; si noti che nessuna transazione *T''* con *TS(T')>TS(T'')>TS(T)*, può aver letto *X* altrimenti al passo precedente T sarebbe stata rolled back;
   c) se nessuna delle condizioni precedenti è soddisfatta allora l'operazione di scrittura è eseguita e *TS(T)* diventa il nuovo valore di *write_TS(X)*

2. ogni volta che una transazione *T* cerca di eseguire l'operazione *read(X*):
   a) se *write_TS(X)>TS(T), T* viene rolled back; infatti in tal caso qualche transazione con un timestamp maggiore di *TS(T)* (cioè una transazione che segue *T* nell'ordinamento basato sui timestamp) ha già scritto il valore di *X* prima che *T* abbia potuto leggerlo, violando in tal modo l'ordinamento basato sui timestamp;
   b) se *write_TS(X)≤TS(T)*, allora l'operazione di lettura è eseguita e se *read_TS(X) <TS(T), TS(T)* diventa il nuovo valore di *read_TS(X)*.

Consideriamo ad esempio le due transazioni seguenti

T_1
read(X)
X;=X+10
write(X)

T_2
read(X)
X;=X+5
write(X)

con i seguenti timestamp: $TS(T_1)=110$, $TS(T_2)=100$. Consideriamo il seguente schedule di $\{T_1, T_2\}$

T_1	T_2
	read(X)
read(X)	
	X;=X+5
X;=X+10	
write(X)	
	(*) write(X)

Quando T_2 cerca di seguire (*) viene abortita.

Consideriamo ora le due transazioni seguenti

T_1
read(Y)
X;=Y+10
write(X)

T_2
read(Y)
X;=Y+5
write(X)

con i seguenti timestamp: $TS(T_1)=110$, $TS(T_2)=100$. Consideriamo il seguente schedule di $\{T_1, T_2\}$

T_1	T_2
	read(Y)
read(Y)	
	X;=Y+5
X;=Y+10	
write(X)	
	(*) write(X)

In questo caso l'operazione (*) non viene eseguita.