Tecnologia di un Database Server (centralizzato)

Gestione della concorrenza

Angelo Montanari

Dipartimento di Matematica e Informatica
Università di Udine

Controllo della concorrenza: introduzione

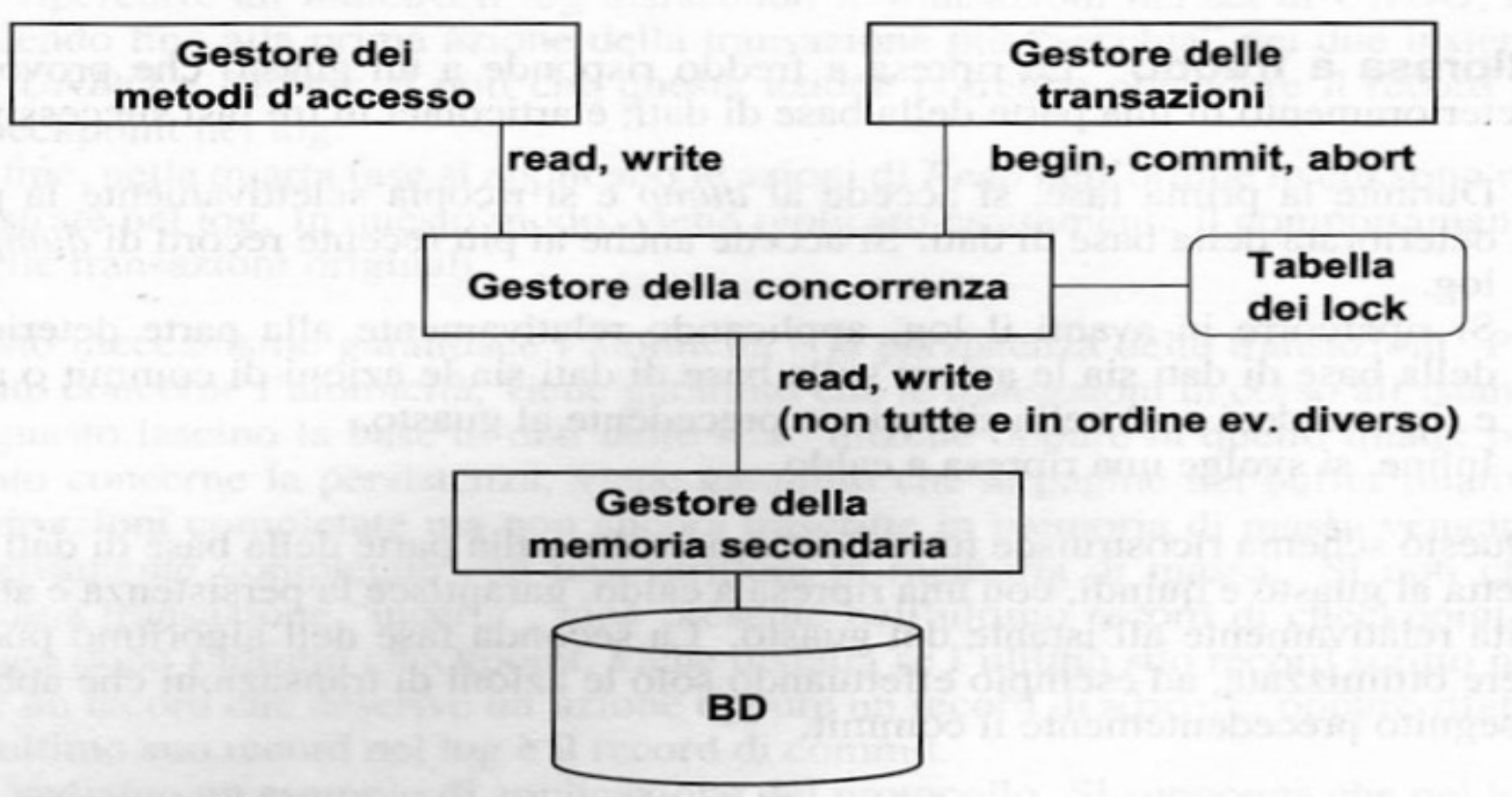
In generale, un DBMS deve servire diverse applicazioni e rispondere a più richieste provenienti da utenti diversi

Unità di misura del carico applicativo di un DBMS: numero di **transazioni per secondo** (tps). Tale valore può variare da decine/centinaia di tps a migliaia di tps, passando da sistemi informativi bancari o finanziari a sistemi di gestione di carte di credito o sistemi di prenotazione delle grandi compagnie aeree

Conseguenza: è impensabile un'esecuzione (fisicamente) seriale delle transazioni!

Nota bene: utente e sistema possono definire in modo diverso ciò che è transazione (ad esempio, una transazione per il sistema può corrispondere a più transazioni per l'utente)

Controllo della concorrenza: architettura



Nota bene: lettura/scrittura dei dati = lettera/scrittura dei corrispondenti blocchi/pagine

Anomalie delle transazioni concorrenti

Le transazioni concorrenti possono causare diversi **tipi di anomalie**:

- perdita di aggiornamento;
- lettura sporca;
- aggiornamento fantasma;
- letture inconsistenti;
- inserimento fantasma.

Per illustrare le prime due anomalie, faremo riferimento alle seguenti due semplici transazioni (dalla medesima struttura):

$$t_1 : r(x), x := x + 1, w(x)$$
$$t_2 : r(x), x := x + 1, w(x)$$

Anomalie: perdita di aggiornamento

Transazione 1	Transazione 2
bot	
$r_1(x)$	
$x := x + 1$	
	bot
	$r_2(x)$
	$x := x + 1$
	$w_2(x)$
	commit
$w_1(x)$	
commit	

Ad ogni operazione di lettura/scrittura associamo un indice che identifica la transazione cui appartiene.

Anomalie: lettura sporca

Transazione 1	Transazione 2
bot	
$r_1(x)$	
$x := x + 1$	
$w_1(x)$	
	bot
	$r_2(x)$
	$x := x + 1$
	$w_2(x)$
	commit
abort	

Anomalie: aggiornamento fantasma

Transazione 1	Transazione 2
bot	
$s := 0$	
$r_1(x); r_1(y)$	
$s := s + x$	
$s := s + y$	
	bot
	$r_2(z); z := z + 10000$
	$r_2(y); y := y - 10000$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s := s + z$	
commit	

Anomalie: letture inconsistenti

Tale anomalia si presenta quando una transazione esegue solo operazioni di lettura, ma ripete più volte, in istanti successivi, la lettura del medesimo dato. Si vuole garantire che una transazione che accede due (o più) volte alla base di dati trovi esattamente lo stesso valore per ciascun dato letto e non risenta dell'effetto di altre transazioni.

Transazione 1	Transazione 2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x := x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

Anomalie: inserimento fantasma

Consideriamo una transazione che valuta un valore aggregato relativo all'insieme di tutti gli elementi che soddisfano un dato predicato di selezione (esempio: voto medio studenti del secondo anno di un certo corso di laurea).

Se tale valore aggregato viene calcolato due volte e tra la prima e la seconda volta viene inserito un nuovo elemento che soddisfa il predicato di selezione, il valore aggregato restituito nei due casi può essere diverso.

Si noti come per evitare tale anomalia non sia sufficiente far riferimento solo ai dati già presenti nella base di dati (l'anomalia dipende da eventuali **inserimenti** intervenuti tra le due valutazioni).

Formalizzazione della nozione di transazione

Definizione formale di transazione in cui le istruzioni iniziale (begin transaction) e finale (end transaction) vengono omesse, si prescinde dall'esito finale della transazione (commit/abort). Alle volte si assume anche che una transazione non legga/scriva più volte il medesimo dato.

Definizione. Una **transazione** è una sequenza di azioni di lettura e scrittura.

Letture/scritture eseguite dalla stessa transazione sono contraddistinte dal medesimo indice (indice che identifica la transazione).

Si noti come in tale formalizzazione non compaiono le operazioni di manipolazione dei dati eseguite dalle transazioni. Dal punto di vista della teoria del controllo della concorrenza, infatti, ogni transazione è un oggetto sintattico, del quale si conoscono unicamente le azioni di ingresso/uscita (lettura/scrittura).

Esempio. transazione $t_1 : r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$

La nozione di schedule

Dato che più transazioni vengono eseguite in modo concorrente, le operazioni di ingresso/uscita vengono richieste da varie transazioni in istanti successivi (**modello interleaving**).

Uno **schedule** è una sequenza di operazioni di ingresso/uscita relative ad un dato insieme di transazioni concorrenti.
Formalmente, uno schedule S_1 è una sequenza del tipo:

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

dove $r_1(x)$ rappresenta la lettura dell'oggetto x da parte della transazione t_1 e $w_2(z)$ rappresenta la scrittura dell'oggetto z da parte della transazione t_2.

Le operazioni compaiono nello schedule nell'ordine temporale di esecuzione sulla base di dati.

Controllo della concorrenza: lo scheduler

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 dalle transazioni.

Per il momento, assumiamo che l'esito (commit/abort) delle transazioni sia noto a priori. Ciò consente di rimuovere dallo schedule tutte le transazioni abortite:

schedule = **commit-proiezione**

Si noti che tale assunzione non consente di trattare alcune anomalie (lettura sporca).

La nozione di schedule seriale

Obiettivo: individuare opportune condizioni da imporre agli schedule per garantire che l'esecuzione delle corrispondenti transazioni sia corretta.

Uno schedule S si dice **seriale** se per ogni transazione t, tutte le azioni di t compaiono in S in sequenza, senza essere inframezzate da azioni di altre transazioni.

Esempio di schedule seriale S_2 in cui le transazioni t_0, t_1, e t_2 vengono eseguite in sequenza:

$$S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$$

La nozione di schedule serializzabile

Definiremo l'esecuzione di uno schedule (commit-proiezione) S_i corretta quando produce lo stesso resultato prodotto da un qualunque schedule seriale S_j delle stesse transazioni.

In questo caso, diremo che lo schedule S_i è **serializzabile**.

Questione: cosa vuol dire produrre lo stesso risultato?

Diverse risposte a tale questione hanno dato origine a diversi criteri di serializzabilità.

Equivalenza di vista

L'equivalenza di vista è caratterizzata in termini della relazione *legge* e dell'insieme delle *scritture finali*

La relazione **legge**

La relazione *legge* lega coppie di operazioni di lettura e scrittura: un'operazione di lettura $r_i(x)$ legge da un'operazione di scrittura $w_j(x)$ ($legge(r_i(x), w_j(x))$) se e solo se (i) $w_j(x)$ precede $r_i(x)$ e (ii) non vi è alcun $w_k(x)$ compreso tra $w_j(x)$ e $r_i(x)$.

L'insieme delle **scritture finali**

Un'operazione di scrittura $w_i(x)$ viene detta una scrittura finale se è l'ultima scrittura dell'oggetto x che appare nello schedule

Equivalenza e serializzabilità rispetto alle viste

Coppie di schedule **equivalenti** rispetto alle **viste**.

Due schedule S_i e S_j vengono detti equivalenti rispetto alle viste ($S_i \approx_V S_j$) se possiedono la stessa relazione *legge* e le stesse *scritture finali*.

Schedule **serializzabili** rispetto alle **viste**.

Uno schedule viene detto serializzabile rispetto alle viste se è equivalente rispetto alle viste ad un generico schedule seriale.

Denotiamo con **VSR** l'insieme degli schedule serializzabili rispetto alle viste.

Esempi

Consideriamo i seguenti schedule S_3, S_4, S_5, e S_6.

$S_3 : \ w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$

$S_4 : \ w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$

$S_5 : \ w_0(x)\ r_2(x)\ w_2(x)\ r_1(x)\ w_2(z)$

$S_6 : \ w_0(x)\ r_2(x)\ w_2(x)\ w_2(z)\ r_1(x)$

Lo schedule S_3 è equivalente rispetto alle viste allo schedule seriale S_4 (ed è, quindi, serializzabile rispetto alle viste). S_5 non è, invece, equivalente rispetto alle viste allo schedule seriale S_4, ma è equivalente rispetto alle viste allo schedule seriale S_6 (ed è, quindi, serializzabile rispetto alle viste).

Problemi con l'equivalenza rispetto alle viste

Controesempio. E' interessante osservare come i seguenti schedule, corrispondenti alle anomalie di perdita di aggiornamento, letture inconsistenti e di aggiornamento fantasma, non siano serializzabili rispetto alle viste:

$S_7 : r_1(x)\ r_2(x)\ w_2(x)\ w_1(x)$ $\qquad$ $S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$

$S_9 : r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$

Problema: la **complessità**. Il problema di determinare se due schedule dati sono equivalenti rispetto alle viste ha complessità lineare. Il problema di determinare se un dato schedule è equivalente ad uno schedule seriale (qualsiasi) è, invece, NP-hard (è necessario confrontare lo schedule con tutti gli schedule seriali).

Soluzione: definire una condizione di equivalenza più ristretta, che non copra tutti i casi di equivalenza tra schedule coperti dell'equivalenza di vista, ma che sia utilizzabile nella pratica (complessità inferiore).

Equivalenza rispetto ai conflitti

L'equivalenza rispetto ai conflitti si basa sulla nozione di azioni in conflitto.

Un'azione a_i si dice **in conflitto** con un'azione a_j (coppia ordinata di azioni in conflitto (a_i, a_j)) se

(i) $i \neq j$;

(ii) a_i e a_j operano sullo stesso oggetto;

(iii) almeno una delle due azioni è una scrittura.

Si distinguono conflitti del tipo lettura-scrittura (rw e wr) e conflitti del tipo scrittura-scrittura (ww).

Equivalenza e serializzabilità rispetto ai conflitti

Coppie di schedule **equivalenti** rispetto ai **conflitti**.

Due schedule S_i e S_j vengono detti equivalenti rispetto ai conflitti ($S_i \approx_C S_j$) se i due schedule presentano le medesime operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi gli schedule.

Schedule **serializzabili** rispetto ai **conflitti**.

Uno schedule viene detto serializzabile rispetto ai conflitti se è equivalente rispetto ai conflitti ad un generico schedule seriale.

Denotiamo con **CSR** l'insieme degli schedule serializzabili rispetto ai conflitti.

Legame tra VSR e CSR

E' possibile dimostrare che la classe degli schedule CSR è **strettamente contenuta** nella classe degli schedule VSR (la condizione di serializzabilità rispetto ai conflitti è condizione sufficiente, ma non necessaria, per la serializzabilità rispetto alle viste):

- tutti gli schedule in CSR appartengono anche a VSR;
- esistono degli schedule appartenenti a VSR che non appartengono a CSR.

Il grafo dei conflitti

Per stabilire se uno schedule è serializzabile rispetto ai conflitti è possibile utilizzare il **grafo dei conflitti**.

I nodi del grafo sono le transazioni e vi è un arco orientato da un nodo t_i ad un nodo t_j se (e solo se) esiste almeno una coppia di azioni in conflitto a_i e a_j, con a_i che precede a_j.

Proposizione: uno schedule è in CSR se e solo se il suo grafo dei conflitti è aciclico.

L'analisi di ciclicità di un grafo ha una complessità lineare nella dimensione del grafo.

Dimostrazione della proposizione

Appartenenza a CSR implica aciclicità.

Per definizione, se uno schedule S appartiene a CSR, S è equivalente rispetto ai conflitti ad uno schedule seriale S_0. Sia $t_1, t_2, \ldots, t_n$ la sequenza delle transazioni in S_0. Dato che tutte le coppie in conflitto compaiono nello stesso ordine in S e S_0, nel grafo dei conflitti di S vi sono soltanto archi (t_i, t_j), con $i < j$, e, quindi, il grafo risulta privo di cicli (un ciclo richiede la presenza di almeno un arco (t_i, t_j), con $i > j$).

Aciclicità implica appartenenza a CSR.

Se il grafo di S è aciclico, allora esiste un ordinamento topologico dei nodi (una numerazione dei nodi tale che il grafo contiene solo archi (i, j) con $i < j$). Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è equivalente a S, perchè per ogni conflitto (i, j), si ha che $i < j$.

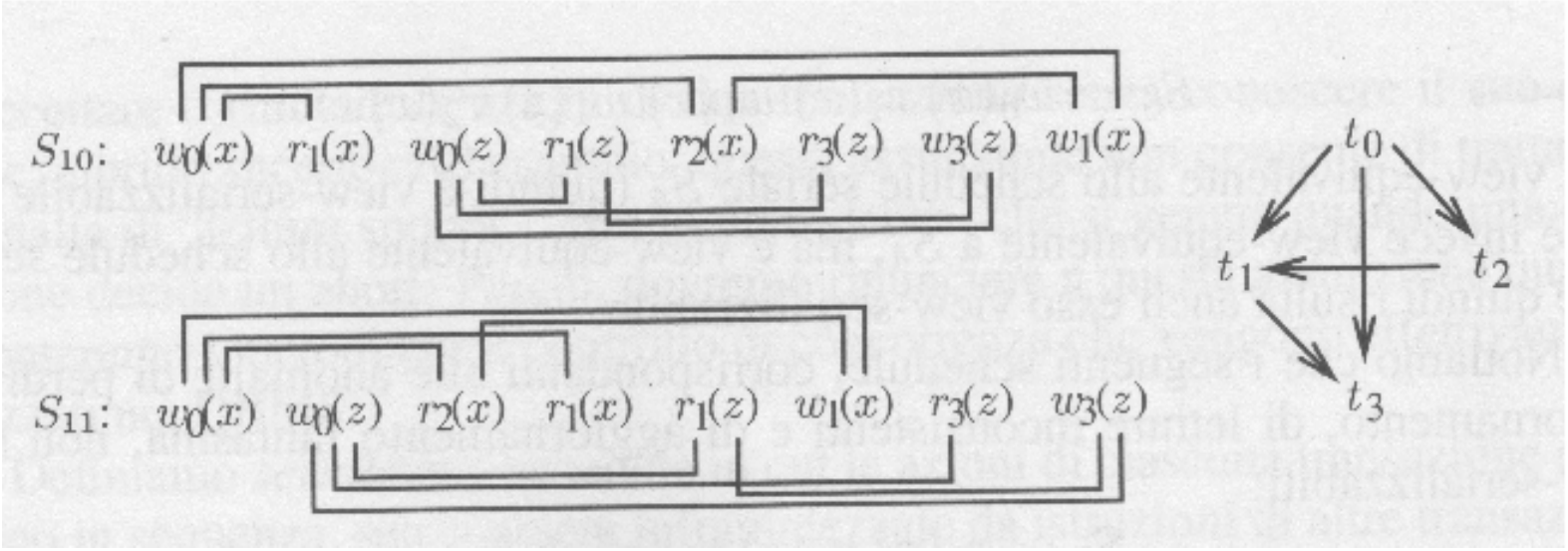
Un'applicazione

Sia dato lo schedule:

$S_{10} : w_0(x)\ r_1(x)\ w_0(z)\ r_1(z)\ r_2(x)\ r_3(z)\ w_3(z)\ w_1(x)$

E' possibile mostrare (algoritmo di ordinamento topologico applicato al corrispondente grafo dei conflitti) che S_{10} è equivalente rispetto ai conflitti allo schedule seriale S_{11}:

$S_{10} : w_0(x)\ w_0(z)\ r_2(x)\ r_1(x)\ r_1(z)\ w_1(x)\ r_3(z)\ w_3(z)$



Limiti

Nonostante la complessità lineare dell'analisi di aciclicità (nella dimensione del grafo), la nozione di serializzabilità rispetto ai conflitti risulta ancora **troppo onerosa** dal punto di vista pratico.

Ad esempio, in un sistema con 100 tps e transazioni 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.

Inoltre, il grafo dei conflitti continua a modificarsi dinamicamente, rendendo difficoltose le decisioni dello scheduler.

Infine, 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 distribuito).

Locking a due fasi

Idea (**locking**): proteggere le operazioni di lettura e scrittura tramite l'esecuzione di opportune primitive (read_lock, write_lock e unlock).

Lo scheduler (**lock manager**) riceve una sequenza di richieste di esecuzione di tali primitive da parte delle transazioni e ne determina la correttezza mediante l'ispezione di un'opportuna struttura dati.

Vincoli che caratterizzano le transazioni ben formate:

- ogni **operazione di lettura** deve essere preceduta da un read_lock e seguita da un unlock (in questo caso si parla di **lock condiviso**, perché possono essere attivi contemporaneamente più lock di questo tipo);
- ogni **operazione di scrittura** deve essere preceduta da un write_lock e seguita da un unlock (in questo caso si parla di **lock esclusivo**, perché non può coesistere con altri lock, condivisi o esclusivi, sul medesimo dato).

Osservazioni

- Le regole di precedenza non sono strette: **possibile ritardo** tra l'esecuzione di un lock (di lettura o scrittura) e l'esecuzione della corrispondente operazione (di lettura o scrittura).
- In alcuni sistemi è disponibile un **unico tipo di lock** (lock esclusivo).
- E' possibile **incrementare il livello di lock**, ossia passare da un lock condiviso (per l'esecuzione di un'operazione di lettura) ad un lock esclusivo (per l'esecuzione di un'operazione di scrittura).

Il comportamento dello scheduler

Le transazioni sono in genere automaticamente ben formate (lock / unlock inseriti in modo trasparente all'applicazione).

La politica dello scheduler è basata sulla **tabella dei conflitti**.

Lo scheduler riceve richieste di lock dalle transazioni e concede/rifiuta di concedere i lock sulla base dei lock precedentemente concessi ad altre transazioni.

Quando viene concesso il lock su una certa **risorsa** ad una data transazione, si dice che la risorsa è **acquisita** dalla transazione. Nel momento dell'unlock, la risorsa viene **rilasciata**.

Quando una richiesta di lock viene rifiutata, la transazione richiedente viene messa in **stato di attesa**. Tale attesa può terminare quando la risorsa ritorna disponibile.

Per tener traccia dei lock già concessi e gestire le nuove richieste, lo scheduler utilizza opportune tabelle dei conflitti (una per ogni risorsa).

Le tabelle dei conflitti

Oltre che dal tipo di lock, ogni richiesta di lock è caratterizzata da due **parametri**:

(i) l'identificativo della transazione che effettua la richiesta;

(ii) la risorsa cui la richiesta si riferisce.

La **politica** del **lock manager** è descritta dalla tabella dei conflitti sotto riportata, dove le righe identificano le richieste, le colonne lo stato della risorsa, il primo valore nella cella l'esito della richiesta e il secondo valore lo stato della risorsa dopo l'eventuale esecuzione della primitiva.

Richiesta	Stato risorsa		
	libero	*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 / *dipende*	OK / *libero*

Interpretazione della tabella

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

Un **unlock** libera una risorsa solo se non vi sono altre transazioni che operano in lettura su di essa; altrimenti, la risorsa rimane bloccata in lettura. Per tener conto del numero di transazioni che bloccano in lettura una risorsa, va utilizzato un **contatore** che viene opportunamente incrementato / decrementato.

Un ingrediente aggiuntivo (essenziale)

La politica proposta è sufficiente **solo** a garantire la **mutua esclusione** (analogia con i meccanismi di controllo utilizzati nell'ambito dei sistemi operativi).

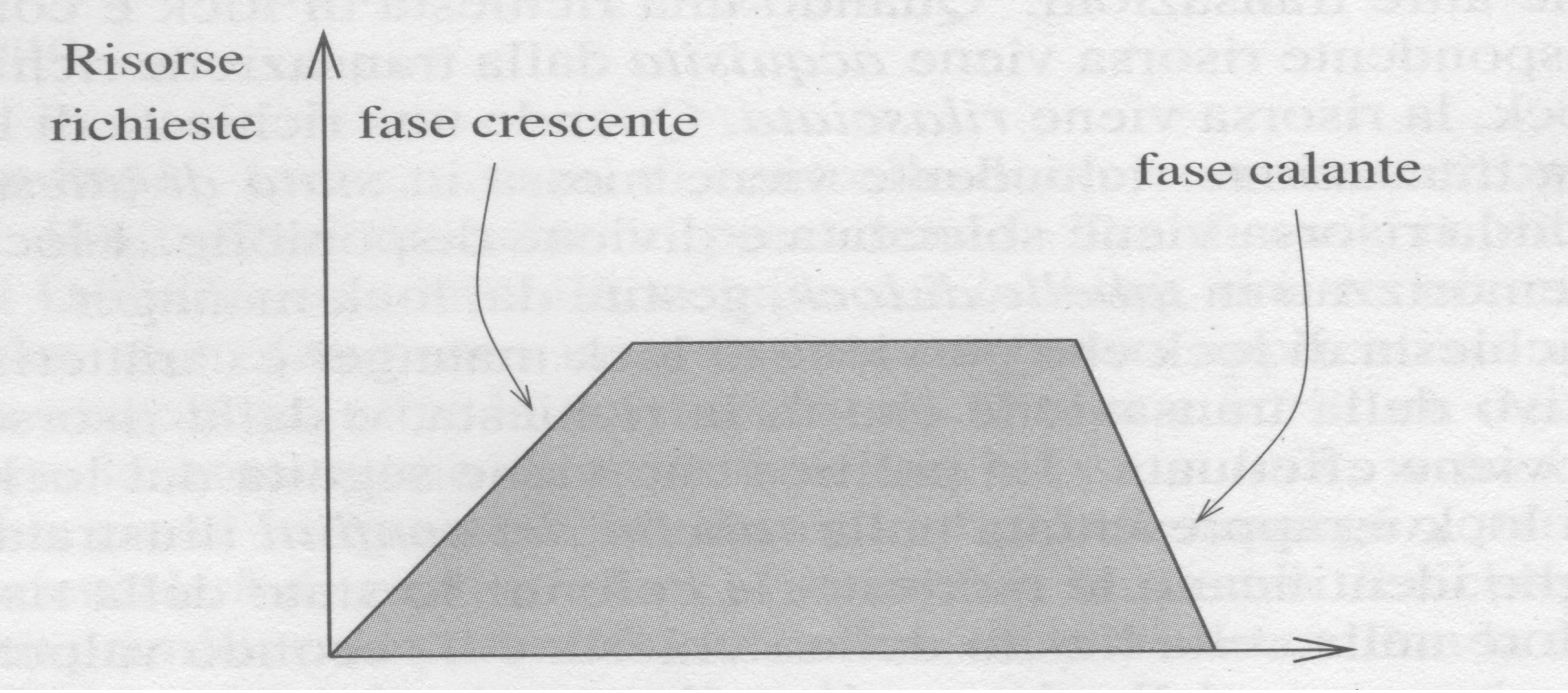
Per garantire che le transazioni seguano uno schedule serializzabile, occorre imporre un vincolo sull'ordinamento delle richieste di lock formulate da una transazione:

una transazione, dopo aver rilasciato un lock, non può acquisirne altri (**locking a 2 fasi**, abbreviato 2PL)

Conseguenza: nell'esecuzione di ogni transazione si possono distinguere 2 fasi, una in cui si acquisiscono progressivamente i lock per le risorse cui si deve accedere (**fase crescente**) e una in cui si rilasciano progressivamente i lock (**fase calante**).

Rappresentazione del protocollo 2PL

In figura viene fornita una rappresentazione grafica del comportamento richiesto dal protocollo 2PL, dove l'ascissa rappresenta il tempo e l'ordinata il numero di risorse possedute da una transazione durante la sua esecuzione.



Legame tra 2PL e CSR - 1

Sia 2PL la classe contenente gli schedule che soddisfano le condizioni imposte dal protocollo 2PL.

La classe **2PL** è **contenuta** nella classe **CSR**.

Assumiamo, per assurdo, che esista uno schedule S tale che $S \in 2PL$ e $S \notin CSR$. Da $S \notin CSR$ segue che il grafo dei conflitti per S contiene un ciclo $t_1, t_2, \ldots, t_n, t_1$. Se esiste un arco (conflitto) tra t_1 e t_2, significa che esiste una risorsa x su cui le due transazioni operano in modo conflittuale. Affinché t_2 possa procedere, è necessario che t_1 rilasci il lock su x e t_2 lo acquisisca. Analogamente, se esiste un arco (conflitto) tra t_n e t_1, significa che esiste una risorsa y su cui le due transazioni operano in modo conflittuale. Affinché t_1 possa procedere, è necessario che t_n rilasci il lock su x e t_1 lo acquisisca. La transazione t_1 non rispetta, quindi, il protocollo 2PL.

Legame tra 2PL e CSR - 2

Esistono schedule in **CSR**, ma **non** in **2PL**.

$S_{12} : r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$

La transazione t_1 deve cedere un lock esclusivo sulla risorsa x per consentire alla transazione t_2 di accedervi, prima in lettura e poi in scrittura, e successivamente richiedere un lock esclusivo sulla risorsa y.

Si noti che t_1 non può anticipare il lock su y in modo da effettuarlo prima del rilascio di x in quanto dovrebbe comunque rilasciare la risorsa y (per poi riacquisirla) per consentirne l'utilizzo da parte di t_3.

Al contrario, lo schedule è serializzabile rispetto ai conflitti (è equivalente allo schedule seriale t_3, t_1, t_2).

2PL e anomalie

E' facile vedere che 2PL risolve le anomalie di perdita di aggiornamento, di aggiornamento fantasma e di letture inconsistenti. Vediamo il caso dell'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

Legenda

Lo stato di ogni risorsa è descritto come libero, bloccato in lettura o bloccato in scrittura.

Si noti come, per effetto del 2PL, le richieste di lettura di y e di z da parte della transazione t_1 vengano messe in attesa. La transazione t_1 può, quindi, procedere solo nel momento in cui la risorsa y (risp., z) viene rilasciata dalla transazione t_2.

Al termine dell'esecuzione, la variabile s contiene il valore corretto della somma $x + y + z$.

Lettura sporca: 2PL stretto

Per risolvere il problema della **lettura sporca**, occorre rimuovere l'assunzione di commit-proiezione.

Per garantire l'assenza di tale anomalia (rimossa l'assunzione di commit-proiezione), occorre imporre una restrizione al protocollo 2PL, che porta al cosiddetto **2PL stretto** (locking a due fasi stretto):

i lock di una transazione possono essere rilasciati solo dopo aver correttamente effettuato le operazioni di commit

In 2PL stretto i lock vengono rilasciati solo al termine della transazione, dopo che ciascun dato è stato portato nel suo stato finale (le azioni di rilascio dei lock seguono l'azione di commit, esplicitamente indicata nello schedule).

Questa versione di 2PL viene usata nei sistemi commerciali.

Inserimento fantasma: lock di predicato

Fin qui, i lock sono stati definiti con riferimento agli oggetti *presenti* nella base di dati.

Per evitare gli **inserimenti fantasma**, è necessario che i lock possano essere definiti anche con riferimento a condizioni (o predicati) di selezione, impedendo non solo l'accesso ai dati coinvolti, ma anche la scrittura di nuovi dati che soddisfano il predicato.

I **lock di predicato** sono realizzati nei sistemi relazionali con l'ausilio degli indici o, nel caso non siano disponibili, bloccando intere relazioni.

Controllo basato sui timestamp

Metodo molto semplice, ma (almeno nella versione base) meno efficace del precedente.

Definizione. Un **timestamp** è un identificatore associato ad ogni evento che definisce un ordinamento (temporale) totale sugli eventi.

Nei sistemi centralizzati, il timestamp viene generato leggendo il valore dell'orologio di sistema nel momento in cui è avvenuto l'evento (si può estendere anche ai sistemi distribuiti).

Controllo della concorrenza basato sui timestamp (**metodo TS**):

- a ogni transazione si associa un timestamp che rappresenta il momento di inizio transazione;
- si accetta uno schedule solo se riflette l'ordinamento seriale delle transazioni definito dai valori dei loro timestamp.

Il metodo TS

Si osservi come TS imponga una serializzazione delle transazioni (rispetto a ciascuna risorsa/oggetto) sulla base dell'ordine di acquisizione dei timestamp.

Ad ogni oggetto x vengono associati due **parametri**:

- $WTM(x)$: il timestamp della transazione che ha eseguito l'ultima scrittura di x;
- $RTM(x)$: il timestamp più grande fra i timestamp delle transazioni che hanno letto x.

Politica dello scheduler

Allo scheduler arrivano richieste di accesso agli oggetti della forma $r_t(x)$ (la transazione con timestamp t chiede di leggere x) e $w_t(x)$ (la transazione con timestamp t chiede di scrivere x). Lo scheduler concede o meno di eseguire l'operazione secondo la seguente **politica**:

- $r_t(x)$: se $t < WTM(x)$ la transazione viene uccisa, altrimenti la richiesta viene accettata e $RTM(x)$ viene posto pari al massimo fra il valore corrente e t;
- $w_t(x)$: se $t < WTM(x)$ o $t < RTM(x)$ la transazione viene uccisa, altrimenti la richiesta viene accettata e $WTM(x)$ viene posto pari a t.

Interpretazione ed esempio

Interpretazione: una transazione non può leggere o scrivere un dato scritto da una transazione con timestamp superiore e non può scrivere un dato che è stato già letto da una transazione con timestamp superiore.

Esempio. Si assuma che $RTM(x) = 7$ e $WTM(x) = 5$. Vediamo le risposte dello scheduler a fronte di una specifica sequenza di richieste.

Richieste	Risposte	Nuovi valori
$r_6(x)$	ok	
$r_7(x)$	ok	
$r_9(x)$	ok	$RTM(x) = 9$
$w_8(x)$	no	t_8 uccisa
$w_{11}(x)$	ok	$WTM(x) = 11$
$r_{10}(x)$	no	t_{10} uccisa

Limiti del metodo TS

Il metodo TS comporta l'**uccisione** di un gran numero di transazioni.

Inoltre, esso opera sotto l'assunzione di **commit-proiezione**. Per rimuovere quest'ipotesi, occorre *bufferizzare* le scritture, ossia effettuarle e trascriverle in memoria di massa solo dopo il commit. Ciò comporta che le letture da parte di altre transazioni dei dati memorizzati nel buffer e in attesa di commit vengano a loro volta messe in attesa del commit della transazione scrivente (meccanismi di attesa analoghi a quelli del 2PL stretto).

Varianti del metodo TS

Uso della **pre-write**. Simile ad una richiesta di scrittura di un oggetto, una pre-write è una segnalazione anticipata allo scheduler dell'intenzione di effettuare una scrittura. Essa consente allo scheduler di ritardare letture che, se effettuate, provocherebbero un esito negativo della scrittura.

Uso delle **multiversioni**. Consiste nel mantenere diverse copie degli oggetti della base di dati. Ogni volta che una transazione scrive un oggetto, la vecchia copia non viene persa, ma viene creata una nuova n-esima copia $WTM_n(x)$. Al contrario, viene mantenuto un unico $RTM(x)$. Ciò consente di non rifiutare alcuna richiesta di lettura: ogni richiesta di lettura viene diretta alla versione appropriata dei dati, tenuto conto del timestamp della transazione (le copie vengono rilasciate quando diventano inutili, ossia quando non esistono più transazioni interessate a leggere il loro valore).

Uso delle multiversioni: regole riviste

Sia $WTM_n(x)$ il timestamp della transazione che per ultima ha aggiornato il valore di x e per ogni $WTM_k(x)$, con $k \leq n$, disponibile, sia x_k il corrispondente valore di x.

Le regole che definiscono il comportamento dello scheduler diventano le seguenti:

- $r_t(x)$: una lettura è sempre accettata. Il valore $\overline{x}$ letto è determinato nel modo seguente. Se $t > WTM_n(x)$, allora $\overline{x} = x_n$; altrimenti, $\overline{x} = x_i$, con $\mathrm{WTM}_i(x) \leq t < WTM_{i+1}(x)$;
- $w_t(x)$: se $t < WTM_n(x)$ o $t < RTM(x)$ la transazione viene uccisa; altrimenti la richiesta viene accettata e $WTM_{n+1}(x)$ viene posto uguale a t.

Nei sistemi commerciali..

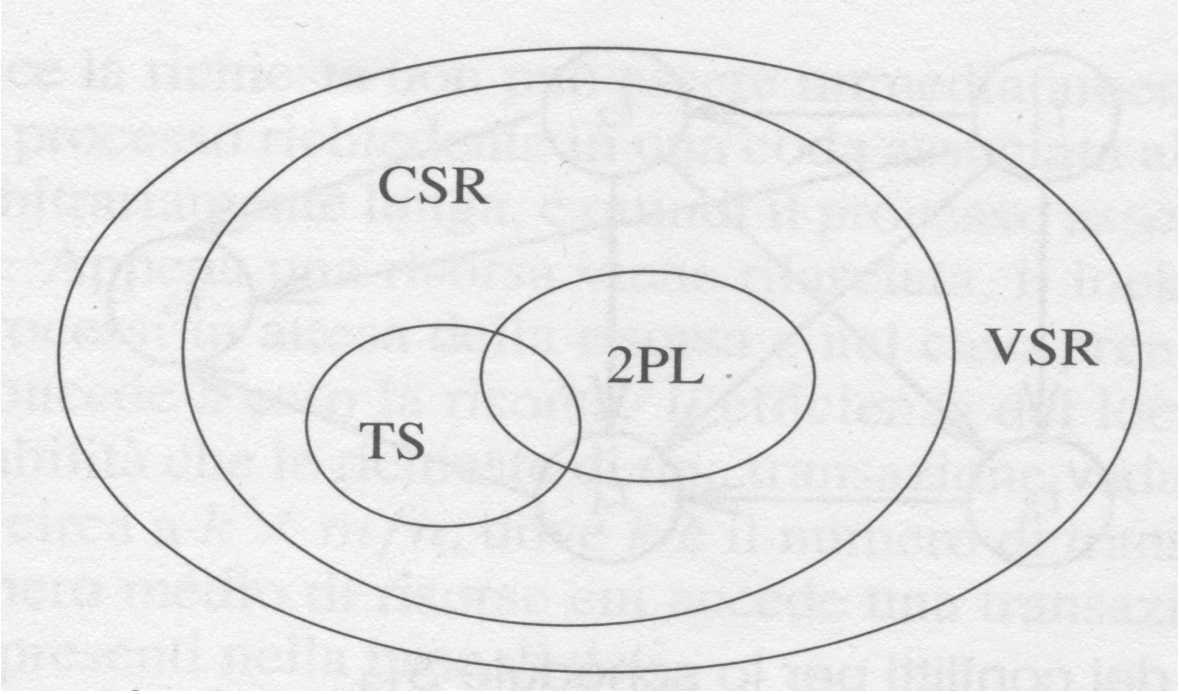
L'idea di mantenere più versioni di un dato, introdotta nel contesto dei metodi basati su timestamp, è stata estesa ad altri metodi, incluso il metodo 2PL.

Un caso di particolare interesse è quello che si limita a mantenere **due copie** (copia corrente e copia precedente).

Ciò consente alle transazioni in lettura di accedere alla copia precedente (ciò evita loro, in molti casi, l'uccisione).

Relazione fra i metodi VSR, CSR, 2PL e TS

La classe VSR è la più generale e include propriamente CSR, che, a sua volta, include propriamente sia 2PL sia TS. 2PL e TS hanno intersezione non vuota, ma né TS è incluso in 2PL né 2PL (risp., 2PL stretto) è incluso in TS.



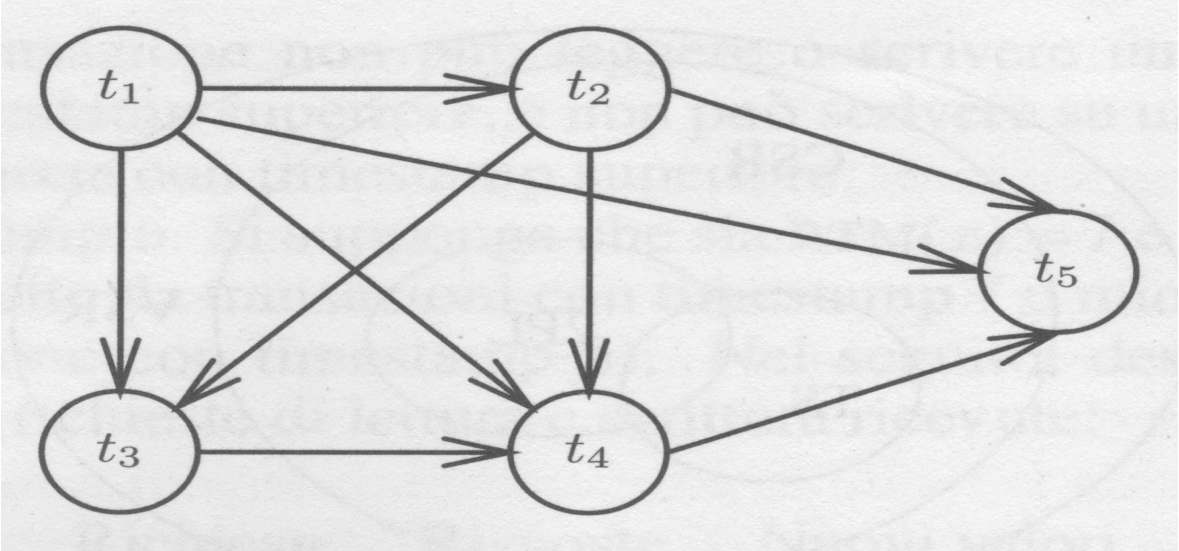
Legame tra TS e 2PL - 1

Esistono schedule che stanno **in TS**, ma **non in 2PL**.

Consideriamo il seguente schedule (interpretiamo gli indici delle transazioni come timestamp):

$S_{13} : r_1(x)\ w_2(x)\ r_3(x)\ r_1(y)\ w_2(y)\ r_1(v)\ w_3(v)\ r_4(v)\ w_4(y)\ w_5(y)$

Il corrispondente grafo dei conflitti è aciclico: $S_{13} \in$ **CSR** (ordinamento seriale: t_1, t_2, t_3, t_4, t_5).



Legame tra TS e 2PL - 2

Lo schedule $S_{13} \notin 2PL$: t_2 prima rilascia x, per consentirne la lettura da parte di t_3, poi acquisisce y, una volta rilasciato da t_1 (si noti ancora una volta che t_2 non può anticipare l'acquisizione di y).

Al contrario, lo schedule $S_{13} \in TS$, in quanto le transazioni operano sui diversi oggetti nell'ordine indotto dai timestamp.

Esistono schedule che stanno **in 2PL**, ma **non in TS**.

$S_{14} : r_2(x)\ w_2(x)\ r_1(x)\ w_1(x)$

Infine, esistono schedule che stanno sia in **in 2PL**, sia **in TS**.

$S_{14} : r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)$

Differenze tra 2PL e TS

- In 2PL, le transazioni sono poste in attesa; in TS, vengono uccise e riavviate.
- In 2PL, l'ordine di serializzazione è imposto dai conflitti; in TS, è imposto dai timestamp (ordine di attivazione).
- La necessità di attendere l'esito della transazione comporta l'allungarsi del tempo di blocco in 2PL (passaggio da 2PL a 2PL stretto) e la creazione di tempi di attesa in TS.
- Il metodo 2PL può presentare il problema del blocco critico (deadlock).
- Il restart impiegato da TS in generale costa più dell'attesa imposta da 2PL.

I DBMS commerciali utilizzano varianti di queste tecniche che cercano di limitarne gli inconvenienti, specie dal punto di vista delle prestazioni.

Meccanismi per la gestione dei lock

Ruolo del **lock manager**: viene invocato da tutti i processi che intendono accedere alla base di dati.

Tali processi dovranno eseguire le operazioni di *read_lock*, *write_lock* e *unlock*, caratterizzate dai seguenti parametri:

read_lock(T, x, errorcode, timeout)

write_lock(T, x, errorcode, timeout)

unlock(T, x)

dove T è l'identificativo della transazione, x l'elemento per il quale si richiede o si rilascia il lock, *errorcode* è un valore restituito dal lock manager (0 se la richiesta è stata soddisfatta, diverso da 0 altrimenti), ***timeout*** è l'intervallo massimo di tempo che il chiamante è disposto ad aspettare per acquisire il lock sulla risorsa.

Possibili situazioni - 1

Si possono presentare 3 diverse situazioni:

Richieste immediatamente accolte. Un processo chiede una risorsa e la richiesta può essere immediatamente soddisfatta. Il lock manager assegna la risorsa al processo e aggiorna lo stato della risorsa nelle sue tabelle interne (il ritardo introdotto dal lock manager sul tempo di esecuzione della transazione è molto modesto).

Richieste che non possono essere immediatamente accolte. Il processo viene inserito in una coda associata alla risorsa richiesta. L'attesa può essere arbitrariamente lunga. Il processo associato alla transazione viene sospeso. Quando la risorsa viene rilasciata, viene assegnata dal lock manager al primo processo in coda (se non vi sono processi in attesa, la risorsa rimane libera). La probabilità che si verifichi un conflitto è direttamente proporzionale al numero di transazioni attive e al numero medio di risorse cui accede ogni transazione, e inversamente proporzionale al numero di risorse presenti nella base di dati.

Possibili situazioni - 2

Scatto del timeout (di una richiesta non ancora soddisfatta). Quando scatta il timeout e la richiesta è ancora insoddisfatta, la transazione può eseguire un rollback (suicidio), generalmente seguito da una ripartenza della transazione. In alternativa, la transazione può decidere di proseguire, chiedendo nuovamente il lock (un fallimento nella richiesta di lock non comporta un rilascio delle altre risorse acquisite in precedenza dalla transazione).

Le **tabelle di lock** vengono accedute molto di frequente e per tale ragione risiedono di norma in memoria centrale (riduzione dei tempi di accesso). Struttura delle tabelle: a ciascun oggetto si associano due bit di stato (per rappresentare i 3 possibili stati dell'oggetto), più un contatore per tener traccia del numero di processi in attesa.

Granularità dei lock

A quali risorse/oggetti si vuole accedere, se necessario in mutua esclusione? Intere tabelle? Insiemi di tuple? Singole tuple? Campi di singole tuple? In diversi sistemi è possibile specificare diversi livelli di lock (**granularità dei lock**).

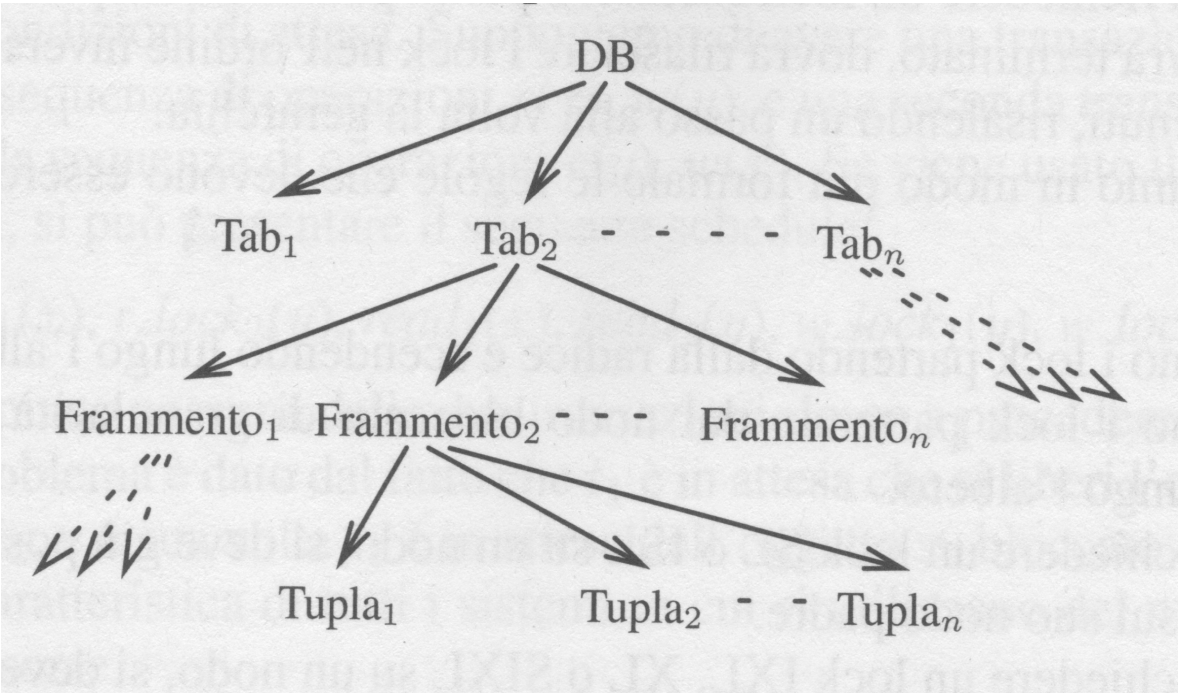
La scelta del/i livello/i di lock è fatta dal progettista dell'applicazione o dall'amministratore della base di dati.

Come scegliere il **livello giusto**?

Un livello troppo elevato (livello di tabella) riduce il parallelismo e aumenta la probabilità di occorrenza di un conflitto. Un livello troppo basso aumenta il rischio di un fallimento dopo lo svolgimento di una notevole quantità di lavoro. In generale, transazioni diverse possono avere requisiti molto diversi dal punto di vista dei lock (transazioni che effettuano modifiche localizzate vs. transazioni che devono accedere a insiemi molto ampi di oggetti, ad esempio, consuntivi).

Il lock gerarchico

Il **lock gerarchico** estende il protocollo di lock tradizionale permettendo alle transazioni di definire in modo molto efficiente i lock di cui hanno bisogno, operando al livello prescelto della gerarchia (dal lock per l'intera base di dati al lock per una singola tupla).



Primitive di richiesta di lock

- **XL**: lock esclusivo (è il write_lock del protocollo normale);
- **SL**: lock condiviso (è il read_lock del protocollo normale);
- **ISL**: intenzione di lock condiviso (intenzione di bloccare in modo condiviso uno dei nodi che discendono dal nodo corrente);
- **IXL**: intenzione di lock esclusivo (intenzione di bloccare in modo esclusivo uno dei nodi che discendono dal nodo corrente);
- **SIXL**: lock condiviso e intenzione di lock esclusivo (blocca il nodo corrente in modo condiviso e dichiara l'intenzione di bloccare in modo esclusivo uno dei nodi che discendono dal nodo corrente).

Come funziona il lock gerarchico?

Esempio. Sia data la gerarchia mostrata in precedenza.

Per bloccare in scrittura una tupla della tabella, occorrerà innanzitutto eseguire un **IXL** a livello di base di dati.

Accettata tale richiesta, occorrerà eseguire un **IXL** prima a livello di relazione e poi a livello della partizione contenente la tupla desiderata.

Una volta che quest'ultimo sarà stato concesso, verrà eseguito un **XL** a livello della singola tupla.

Terminata l'esecuzione, la transazione dovrà rilasciare i vari lock in ordine inverso a quello con cui sono stati acquisiti (risalendo passo passo la gerarchia).

In modo più formale..

Le regole che devono essere rispettate dal protocollo:

1. si richiedono i lock partendo dalla radice e scendendo lungo l'albero;
2. si rilasciano i lock partendo dal nodo bloccato di granularità più piccola e risalendo lungo l'albero;
3. per poter richiedere un lock **SL** o **ISL** su un nodo, si deve già possedere un lock **ISL** o **IXL** sul suo nodo padre;
4. per poter richiedere un lock **IXL**, **XL** o **SIXL** su un nodo, si deve già possedere un lock **SIXL** o **IXL** sul suo nodo padre;
5. le regole che il lock manager utilizza per accettare o rifiutare le richieste di lock sono riportate nella tabella delle compatibilità.

Tabella delle compatibilità

Compatibilità tra le modalità di lock in presenza di gerarchie:

Richiesta	Stato risorsa				
	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Blocco critico

Il **blocco critico** (stallo / deadlock) è il problema tipico dei sistemi concorrenti in cui si introducono condizioni di attesa.

Esempio. Siano date 2 transazioni t_1 e t_2. Supponiamo che t_1 debba eseguire in sequenza le operazioni $r(x)$ e $w(y)$ e che t_2 debba eseguire in sequenza le operazioni $r(y)$ e $w(x)$. Col protocollo 2PL si può presentare il seguente schedule:

$$rlock_1(x)\ rlock_2(y)\ r_1(x)\ r_2(y)\ w_1(y)\ w_2(x)$$

In tale situazione, nessuna delle 2 transazioni riesce a procedere e il sistema risulta bloccato (t_1 rimane in attesa che si liberi y, bloccato da t_2, e t_2 che si liberi x, bloccato da t_1).

In generale, la probabilità del blocco critico è **bassa**, ma **non nulla**.

Come evitare/risolvere un blocco critico?

Tre sono le tecniche comunemente usate:

- timeout
- rilevamento (deadlock detection)
- prevenzione (deadlock prevention)

Uso dei 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 stato di deadlock viene tolta dallo stato di attesa e di norma abortita. Tecnica molto semplice, usata dalla gran parte dei sistemi commerciali.

Come scegliere il timeout?

La **scelta** del **timeout** non è banale.

Un **valore troppo elevato** del timeout tende a risolvere tardi i blocchi critici, dopo che le transazioni coinvolte hanno trascorso inutilmente diverso tempo in attesa.

Un **valore troppo basso** del timeout rischia di intepretare come blocchi critici anche situazioni in cui una transazione sta attendendo la disponibilità di una risorsa destinata a liberarsi, uccidendo la transazione e sprecando il lavoro da essa già svolto.

Rilevamento dei blocchi critici

Non vengono posti vincoli al comportamento del sistema, ma viene controllato tutte le volte che lo si ritiene necessario il contenuto delle tabelle di lock, al fine di rilevare eventuali situazioni di deadlock.

Tale **controllo** può essere effettuato periodicamente o alla scadenza dei timeout di attesa delle transazioni.

Il controllo consiste nell'analisi delle relazioni di attesa fra le varie transazioni e nella verifica dell'esistenza di un ciclo. La ricerca di cicli in un grafo, specie se effettuata periodicamente, risulta abbastanza efficiente. Per tale ragione, alcuni sistemi commerciali utilizzano tale tecnica.

Prevenzione dei blocchi critici

Approccio alternativo: **prevenire** l'insorgenza di un blocco critico.

Una possibile tecnica: richiedere il **lock** di **tutte le risorse** necessarie alla transazione in una sola volta.

Problemi: rigidità/inefficienza (spesso le transazioni non conoscono a priori le risorse cui vogliono accedere).

Un'altra tecnica: utilizzo dei **timestamp**.

Tutte le transazioni acquisiscono un timestamp. Una transazione t_i rimane in attesa di una risorsa acquisita da una transazione t_j solo se i timestamp i, j stanno in una determinata relazione (ad esempio, solo se $i < j$).

In tal modo, circa il 50% delle richieste che generano un conflitto rimane in coda in attesa, mentre nel restante 50% dei casi la transazione viene uccisa.

Politiche interrompenti e non interrompenti

In generale, come scegliere le transazioni da uccidere (vittime)? Due possibili opzioni: politiche interrompenti (preemptive) e politiche non interrompenti.

- **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).
- **Politiche non interrompenti**: una transazione può essere uccisa solo nel momento in cui effettua una nuova richiesta.

Un esempio di politica interrompente

Criterio: uccidere le transazioni che hanno svolto meno lavoro (*si spreca meno lavoro*).

Il problema del **blocco individuale**: può accadere che una transazione, all'inizio della propria elaborazione, acceda ad un oggetto al quale accedono molte altre transazioni. In tal modo, essa può trovarsi sempre in conflitto con altre transazioni e, avendo effettuato poco lavoro, essere ripetutamente uccisa. Non c'è un blocco critico (deadlock), ma un blocco individuale (**starvation**).

Possibile **soluzione**: mantenere invariato il timestamp delle transazioni abortite e fatte ripartire, dando in questo modo priorità crescente alle transazioni più anziane.

In pratica..

La tecnica basata sulla prevenzione dei blocchi critici non viene utilizzata nei DBMS commerciali:

mediamente si uccide una transazione ogni due conflitti, ma la probabilità di insorgenza di un blocco critico è molto minore della probabilità di un conflitto

Gestione della concorrenza in SQL:1999

Le funzionalità per il controllo della concorrenza possono introdurre dei ritardi nell'esecuzione delle transazioni:

- attività del lock manager;
- transazioni poste in attesa.

In particolare, i lock di predicato, utilizzati per impedire gli inserimenti fantasma, possono alle volte bloccare gli accessi a intere tabelle.

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

Approccio generale: il programmatore / utente evoluto può rinunciare ad alcuni aspetti del controllo della concorrenza per aumentare le prestazioni

Più precisamente, per ciascuna transazione è possibile specificare un **livello di isolamento** scelto fra i seguenti:

- serializable (default);
- repeatable read;
- read committed;
- read uncommitted.

Confronto fra i livelli

Il **livello serializable** garantisce tutti i requisiti di isolamento che possono essere richiesti (2PL stretto e lock di predicato).

Gli altri tre livelli consentono di rinunciare, per le sole operazioni di lettura, ad alcuni requisiti di isolamento: migliori prestazioni, ma possibilità di inconsistenze.

Per le operazioni di scrittura sono richiesti lock esclusivi, che vengono mantenuti fino al commit /abort (2PL stretto): l'anomalia di perdita di aggiornamento (l'unica che coinvolge due transazioni che scrivono entrambe) è sempre evitata.

Livelli repeatable read e read committed

Repeatable read applica il 2PL stretto anche nel caso di lock di lettura (che vengono applicati a livello di tupla).

Evita tutte le anomalie ad eccezione dell'inserimento fantasma (non impedisce l'inserimento di nuove tuple).

Read committed richiede lock condivisi per effettuare le letture (ma non impone il vincolo delle 2 fasi).

Poiché le scritture rispettano il 2PL stretto, non può leggere dati intermedi ed evita, quindi, oltre alla perdita di aggiornamento, le letture sporche.

Non evita, invece, le altre anomalie.

Livello read uncommitted

Read uncommited non pone alcun vincolo sui lock (in molti sistemi è usato per le sole transazioni read-only): la transazione non emette lock per la lettura e, per leggere, non rispetta i lock esclusivi posti da altre transazioni.

Se usato con transazioni read-only, non produce informazioni scorrette.

Una transazione di livello read uncommited può presentare tutte le anomalie, ad eccezione della perdita di aggiornamento