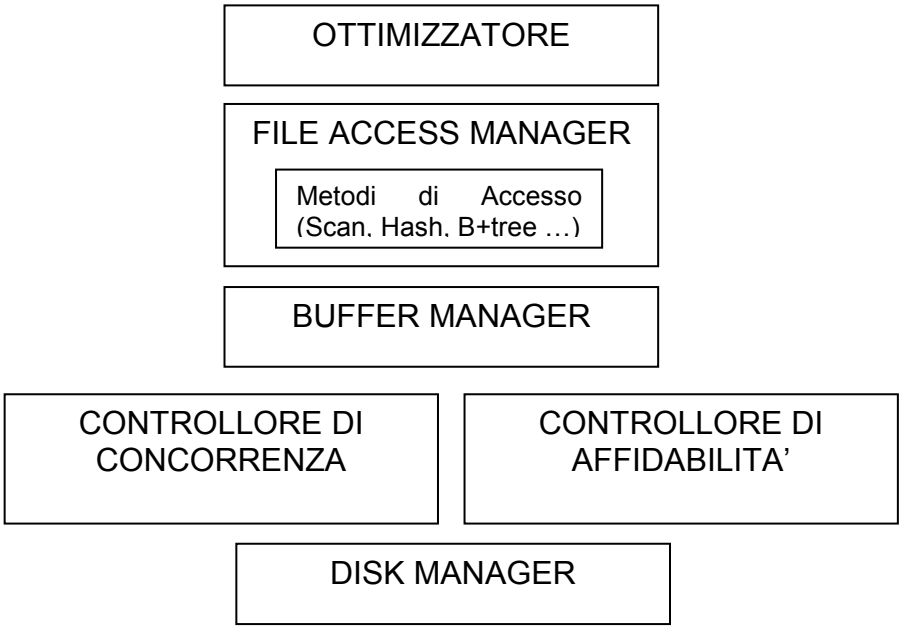
ARCHITETTURA DI UN B.D.M.S.

Parte I – Il controllo di concorrenza

Michele de Nittis

Generalità

A livello astratto un DataBase Management System può essere suddiviso nei seguenti moduli:



L'**Ottimizzatore** è un modulo molto complesso che, ricevuta una query scritta in un linguaggio formale quale SQL:

- Svolge un'analisi lessicale, sintattica e semantica,della query, rifiutandola in caso di riscontro di errori (Parsing);
- Traduce la query corretta in una forma interna e formula le possibili strategie di esecuzione o di accesso ai dati (piani di esecuzione);
- Sceglie la migliore strategia di esecuzione sulla base di un criterio di costo;

Il **Gestore dei Metodi di accesso (FILE ACCESS MANAGER)** esegue gli accessi fisici ai dati secondo la strategia scelta dall'ottimizzatore. Esso fornisce ai livelli superiori le astrazioni dei *file* e dei *record*, nascondendo tutti i dettagli di basso livello.

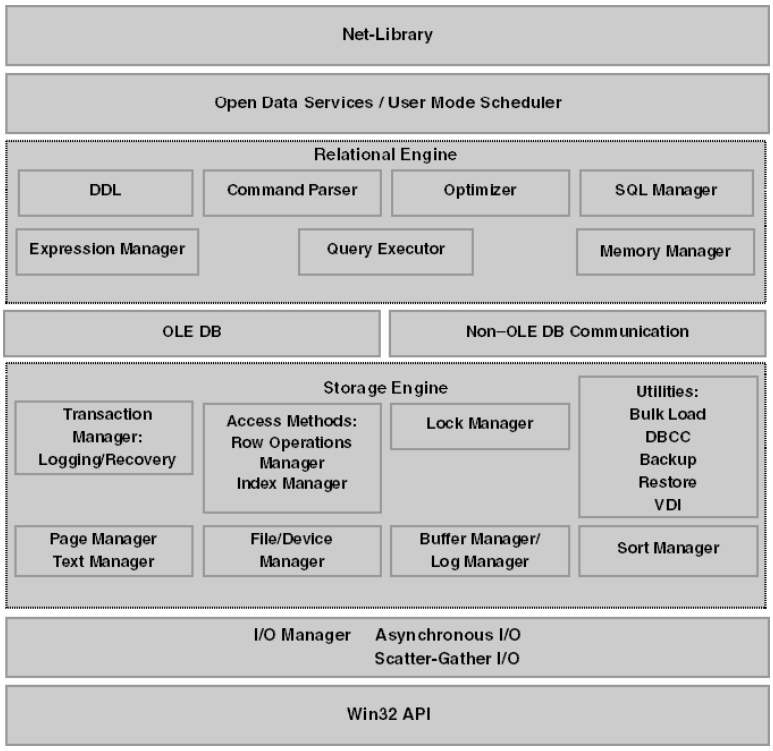
Il **Controllore di affidabilità** garantisce il contenuto della base di dati a fronte di malfunzionamenti e guasti;

Il **Controllore di concorrenza** gestisce gli accessi concorrenti alla base di dati senza che interferiscano tra loro.

Il **Buffer Manager** gestisce la memoria centrale ed il trasferimento dei dati dalla memoria di massa alla memoria centrale e viceversa;

Il **Disk Manager** gestisce lo spazio disco disponibile per il DBMS.

A livello puramente indicativo, si confronti la predetta architettura astratta con quella di un DBMS commerciale (SQL Server 7):



La gerarchia della memoria

La memoria disponibile per un DBMS può essere così classificata;

1. **Memoria Primaria**, o centrale, costituita da memoria R.A.M. e da memoria CACHE. E' una memoria veloce e generalmente volatile che consente l'accesso diretto al dato. Dati i suoi elevati costi, è disponibile in piccole quantità;
2. **Memoria Secondaria**, o di Massa, costituita da memoria disco. E' una memoria lenta e non volatile che consente l'accesso diretto ai dati. I suoi modesti costi ne consentono l'impiego in grossa quantità.
3. **Memoria Terziaria** memoria molto lenta ed economica, disponibile in grandi quantità. Consente l'accesso ai dati unicamente in modo sequenziale. Non è idonea per impieghi operativi ma come magazzino di dati storici o per l'effettuazione di operazioni di back-up.

Transazioni e Sistemi Transazionali

Una transazione identifica una unità elementare di lavoro svolta da una applicazione cui si vogliono associare particolari caratteristiche di correttezza, robustezza e isolamento.[1 - p.300].

Una transazione può essere sintatticamente definita come una sequenza di istruzioni ed interrogazioni (operazioni), formulate attraverso opportuni linguaggi formali, racchiuse tra due comandi: Begin of Transaction (BOT), che ne identifica l'inizio, ed End Of Transaction, che ne identifica la fine. All'interno della definizione della transazione si può trovare una delle seguenti due istruzioni: *Commit* e *Rollback* (o Abort). Una transazione ha esito positivo, cioè produce i propri effetti sulla base di dati, solo dopo l'esecuzione del comando commit. In questo caso si dice che la transazione è andata a *buon fine*. Gli effetti di una transazione vengono completamente annullati se viene eseguita l'istruzione Rollback, indipendentemente dalla sua complessità.

Una Transazione si dice *ben formata* se inizia con un BOT, termina con un EOT ed esegue un commit o un rollback non seguito da ulteriori operazioni.

Un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione delle transazioni viene detto *transazionale*.

Proprietà ACIDE delle transazioni

Una transazione deve rispettare quattro proprietà:

1. **Atomicità (Atomic):** una transazione è un'unità indivisibile di lavoro. Essa produce tutti i suoi effetti sulla base di dati oppure nessuno. In caso di interruzione non sono visibili i risultati intermedi delle singole operazioni componenti e la base di dati viene riportata allo stato precedente il comando BOT. Il modulo che garantisce questa proprietà è il controllore di affidabilità.
2. **Consistenza (Consistency):** l'esecuzione di una transazione non deve violare i vincoli di integrità definiti sulla base di dati. In caso di violazione, il sistema può reagire in modo *immediato* ovvero *differito*. Nel primo caso il sistema verifica la compatibilità di un'operazione con i vincoli della base di dati a tempo d'esecuzione, rimuovendone subito gli effetti prodotti nel caso di violazione, senza imporre l'abort dell'intera transazione. Nel secondo caso il sistema verifica la compatibilità degli effetti della transazione solo dopo il commit e ne impone l'abort in caso di violazione dei vincoli. I moduli che garantiscono questa proprietà sono tutti ad alto livello (Interprete dei comandi, compilatore D.D.L. – vedi schema di figura 2);
3. **Isolamento (Isolation)**: l'esecuzione di una transazione deve essere indipendente dall'esecuzione contemporanea di altre transazioni. Questa proprietà viene garantita dal controllore di concorrenza.
4. **Persistenza (Durability)**: gli effetti prodotti sulla base di dati da una transazione andata a buon fine sono permanenti e non devono essere persi in alcun modo. Questa proprietà è garantita dal controllore di affidabilità.

Il controllo di concorrenza

Il controllo di concorrenza è quell'attività svolta dal DBMS che consente l'esecuzione contemporanea di più transazioni sulla medesima base di dati, senza interferenze reciproche. Esso viene assicurato da diversi moduli del sistema, sinteticamente rappresentati nel *Controllore di Concorrenza*.dello schema di figura 1.

Il controllo di concorrenza interviene al più basso livello di astrazione nell'architettura di un BDMS relativo alle operazioni di ingresso e uscita che attuano il trasferimento di blocchi di memoria, o **pagine**, da quella di massa a quella centrale e viceversa.

Una transazione *T*, dal punto di vista del DBMS, consiste in una sequenza di operazioni elementari, o azioni, di lettura $r_T(x)$ e di scrittura $w_T(x)$ su una o più risorse *x* di una base di dati. Nel caso di esecuzione di più transazioni concorrenti, le relative azioni di I/O sulla base di dati vengono presentate al DBMS in instanti successivi e formano un'unica sequenza di operazioni denominata **schedule**. Uno schedule viene valutato da un apposito modulo del controllo di concorrenza, denominato **Scheduler**, ed accettato o rifiutato secondo che generi o meno delle **anomalie** tipiche dell'esecuzione concorrente.

In realtà lo scheduler non può valutare uno schedule nella sua interezza perché le azioni componenti gli vengono presentate, di volta in volta, a tempo di esecuzione e quindi l'esito finale delle transazioni non gli è noto a priori. Lo scheduler, quindi, deve tenere traccia in proprie strutture interne di tutte le operazioni (azioni) già compiute sulla base di dati dalle transazioni ed accettare o rifiutare, secondo una data politica, quelle che gli verranno progressivamente presentate.

Per lo studio teorico che faremo del controllo di concorrenza, tuttavia, è necessario adottare un modello di scheduler cha valuti schedule completi e formati unicamente da azioni facenti parte di transazioni che effettueranno il **commit**. Questo modello teorico viene denominato **commit-proiezione.**

Le anomalie nell'esecuzione concorrente di più transazioni

L'esecuzione concorrente di più transazioni può causare alcuni problemi di correttezza o **anomalie**. Queste possono essere classificate nelle seguenti quattro categorie:

Perdite di aggiornamento

Si considerino le seguenti due transazioni che effettuano un aggiornamento della medesima risorsa x

1. $t_1(x) = \{r_1(x), x=x+1, w_1(x), commit\}$;
2. $t_2(x) = \{r_2(x), x=x+1, w_2(x), commit\}$;

E' facile verificare che se le azioni componenti vengono presentate nel seguente ordine $\{r_1(x)\ r_2(x)\ w_2(x)\ w_1(x)\}$ si ha la perdita dell'aggiornamento effettuato da una delle due transazioni, in particolare dalla transazionie $t_1(x)$.

x=1	$r_1(x)$	
	x=x+1	
x=1		$r_2(x)$
		x=x+1
x=2		$w_2(x)$
		Commit
x=2	$w_1(x)$	
	Commit	

Nell'esempio considerato ci si sarebbe dovuti aspettare un valore finale pari ad x=3 mentre, per effetto dell'anomalia, il valore finale è x=2.

Letture Sporche:

Questa anomalia si presenta quando una transazione legge il risultato intermedio di un'azione di un'altra transazione operante sulla medesima risorsa, come illustra il seguente esempio in cui s considerano le medesime transazioni dell'esempio precedente:

1. $t_1(x) = \{r_1(x), x=x+1, w_1(x), commit\}$;
2. $t_2(x) = \{r_2(x), x=x+1, w_2(x), commit\}$;

x=1	$r_1(x)$	
	X=x+1	
x=2	$w_1(x)$	
x=2		$r_2(x)$
		x=x+1
x=3		$w_2(x)$
		Commit
	Abort	

Nell'esempio considerato ci si sarebbe dovuti aspettare un valore finale pari ad x=2, a causa del *rollback* effettuato dalla transazione t_1, mentre, per effetto dell'anomalia, il valore finale è x=3.

Letture Inconsistenti:

Questa anomalia si presenta quando una transazione effettua due azioni di lettura di una risorsa in tempi diversi, senza aver effettuato tra esse alcuna operazione di aggiornamento, e rileva valori diversi:

1. $t_1(x) = \{r_1(x), r_1(x), commit\}$;
2. $t_2(x) = \{r_2(x), x=x+1, w_2(x), commit\}$;

x=1	$r_1(x)$	
x=1		$r_2(x)$
		x=x+1
x=2		$w_2(x)$
		Commit
x=2	$r_1(x)$	
	Commit	

Aggiornamento Fantasma:

Questa anomalia si presenta quando tra i valori di un certo numero di risorse esiste un vincolo di integrità che, nonostante venga rispettato singolarmente dalle transazioni che vi operano in concorrenza, viene violato.

Supponendo, ad esempio, che il vincolo di integrità tra le risorse x, y, z della base di dati sia x+y+z=s=1000 e che le transazioni operanti in concorrenza siano:

1. $t_1(x,y,z,s) = \{r_1(x), r_1(y), r_1(z), s=x+y+z, w_1(s)$ commit$\}$;
2. $t_2(x,y,z) = \{r_2(y), y=y-100, r_2(z), z=z+100, w_2(y), w_2(z),$ commit$\}$;

x=500,		
y=400,		
z=100,		
s=1000		
x=500	$r_1(x)$	
y=400	$r_1(y)$	
y=400		$r_2(y)$
		y=y-100
z=100		$r_2(z)$
		z=z+100
y=300		$w_2(y)$
z=200		$w_2(z)$
		commit
z=200	$r_1(z)$	
	s=x+y+z	
s=1100	$w_1(s)$	
	commit	

Si osserva che se le azioni delle due transazioni si presentano come nella tabella di cui sopra si ha una violazione del vincolo da parte della transazione t_1, sebbene entrambe abbiano agito, singolarmente, nel rispetto del vincolo.

La nozione di *Serializzabilità*

Considerando i precedenti quattro esempi, si può facilmente constatare che se le azioni di ciascuna transazione non si interpongono a quelle delle altre, le anomalie non si verificano. Uno schedule formato in questo modo viene detto *seriale*.

*Si definisce **seriale** uno schedule in cui tutte le azioni di tutte le transazioni compaiono in sequenza, senza essere inframmezzate da istruzioni di altre transazioni [ACBT – p.307].*

Il seguente è un esempio di schedule seriale:

S_1:$\{r_0(x)r_0(y)w_0(x)r_1(y)r_1(x)w_1(y)r_2(x)r_2(y)r_2(z)w_2(z)\}$

Prima di continuare deve essere chiaro che gli schedule seriali che si possono formare con le azioni di uno stesso insieme di transazioni producono, in generale, *effetti diversi sulla base di dati*, ma la lasciano, comunque, in uno stato consistente.

L'esecuzione di una *commit-proiezione* si uno schedule S_i è **corretta** quando produce lo stesso risultato prodotto da un qualunque schedule seriale S_j delle stesse

transazioni. Uno schedule che produce lo stesso risultato di uno schedule seriale formato dalle stesse transazioni è detto **serializzabile**.

Per verificare se uno schedule è serializzabile ovvero **equivalente** ad uno schedule seriale, è necessario introdurre una **nozione di equivalenza** tra schedule.

Nozione di View-Equivalenza

Introduciamo due concetti:

1. **Relazione Legge da**: Si dice che un'azione di lettura $r_i(x)$ **legge da** un'operazione di scrittura $w_k(x)$, e si scrive [legge($r_i(x)$, $w_k(x)$)], se $w_k(x)$ precede $r_i(x)$ e non vi è interposta altra operazione di scrittura da parte di altre transazioni sulla medesima risorsa $w_s(x)$.
2. **Scrittura Finale:** Un'azione di scrittura $w_k(x)$ è detta **finale** se è l'ultima operazione di scrittura sull'oggetto x che appare nello schedule.

Due schedule sono detti View Equivalenti se caratterizzati dalle stesse transazioni, possiedono la stessa relazione "Legge da" e le stesse scritture finali.

Uno schedule è view-serializzabile se è view equivalente ad uno schedule seriale.

Esempio:

Lo schedule S_1 è serializzabile perché è View-Equivalente allo schedule seriale S_2:

S_1:{$w_0(x)r_2(x)r_1(x)w_2(x)w_2(z)$}	S_2:{$w_0(x)r_1(x)r_2(x)w_2(x)w_2(z)$}
Legge($r_2(x)$, $w_0(x)$)	Legge($r_2(x)$, $w_0(x)$)
Legge($r_1(x)$, $w_0(x)$)	Legge($r_2(x)$, $w_0(x)$)
$w_2(x)$ = scrittura finale	$w_2(x)$ = scrittura finale
$w_2(z)$ = scrittura finale	$w_2(z)$ = scrittura finale

Esempio:

Lo schedule S_3 è serializzabile perché è View-Equivalente allo schedule seriale S_4:

S_3:{$w_0(x)r_1(x)w_1(x)r_2(x)w_1(z)$}	S_4:{$w_0(x)r_1(x)w_1(x)w_1(z)r_2(x)$}
Legge($r_1(x)$, $w_0(x)$)	Legge($r_1(x)$, $w_0(x)$)
Legge($r_2(x)$, $w_1(x)$)	Legge($r_2(x)$, $w_1(x)$)
$W_1(x)$ = scrittura finale	$W_1(x)$ = scrittura finale
$W_1(z)$ = scrittura finale	$W_1(z)$ = scrittura finale

Verificare che due schedule sono view-equivalenti è un problema di complessità lineare, mentre verificare la serializzabilità di un dato schedule è un problema di tipo NP-Hard perché è necessario trovare tutti gli schedule seriali formati dalle stesse transazioni di quello dato e verificarne, quindi, la view equivalenza con ciascuno di essi. Per questo motivo (cioè la complessità computazionale della verifica di serializzabilità) la nozione di view-equivalenza non è idonea per applicazioni pratiche.

La nozione di Conflict Equivalenza

Introduciamo il concetto di **conflitto:**

Un'azione a_i è in conflitto con un'azione a_j ($i \neq j$) se entrambe operano sullo stesso oggetto ed almeno una di esse è un'operazione di scrittura (write) [ACPT pg. 308].

Esistono, dunque, tre tipi di conflitti: Read-Write, Write-Read e Write-Write.

Uno schedule S_i è conflict-equivalente ad uno schedule S_j se sono composti dalle medesime transazioni ed ogni coppia di operazioni in conflitto è nello stesso ordine in entrambi schedule [ACPT pg. 308]

Uno schedule S_i è **conflict-serializzaabile** se esiste uno schedule seriale ad esso conflict equivalente.

Si può dimostrare che uno schedule conflict serializzabile e anche view-serializzabile, mentre si può facilmente verificare che non è vero il viceversa.

Si osservi, infatti, che i conflitti Write-Read corrispondono alle relazioni "Legge da" e che i conflitti write-write e read-write, qualora non seguiti da altri conflitti, corrispondono alle scritture finali, per cui due schedule conflict-equivalenti possiedono la medesima relazione "Legge da" e le medesime scritture finali.

Esempio:

Nel seguente schedule vi sono 8 conflitti:

S_1:{$w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $r_3(z)$ $w_3(z)$ $w_1(x)$}

1. w_0-$r_1(x)$;
2. w_0-$r_2(x)$;
3. w_0-$w_1(x)$;
4. w_0-$r_1(z)$;
5. w_0-$r_3(z)$;
6. w_0-$w_3(z)$;
7. r_1-$w_3(z)$;
8. r_2-$w_1(x)$;

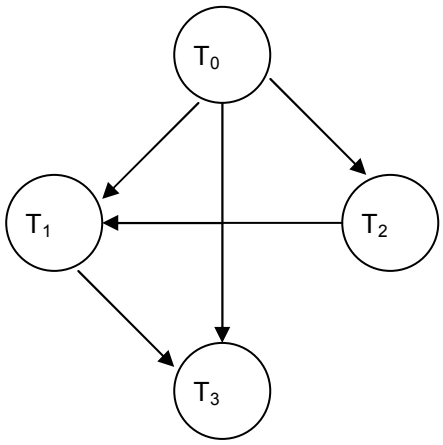
Nel seguente schedule seriale vi sono 8 conflitti, identici a quelli del precedente schedule S_1:

S_s:{$w_0(x)w_0(z)r_2(x)r_1(x)r_1(z)w_1(x)r_3(z)w_3(z)$}

1. w_0-$r_2(x)$ (vedi n° 2 dello schedule S_1);
2. w_0-$r_1(x)$ (vedi n° 1 dello schedule S_1);
3. w_0-$w_1(x)$ (vedi n° 3 dello schedule S_1);
4. w_0-$r_1(z)$ (vedi n° 4 dello schedule S_1);
5. w_0-$r_3(z)$ (vedi n° 5 dello schedule S_1);
6. w_0-$w_3(z)$ (vedi n° 6 dello schedule S_1);
7. r_2-$w_1(x)$ (vedi n° 8 dello schedule S_1);
8. r_1-$w_3(z)$ (vedi n° 7 dello schedule S_1);

Vi è un metodo operativo molto semplice per verificare la conflict-serializzabilità di uno schedule: basta verificare l'aciclicità del grafo dei conflitti, che è un grafo i cui nodi rappresentano le transazioni componenti e gli archi da un nodo 'A' a un nodo 'B' i conflitti tra una azione della transazione A e un'azione della successiva transazione B.

Il grafo dei conflitti relativo alla transazione S_1 è rappresentato nella seguente figura:



Grafo dei conflitti

Locking a due fasi

Il locking a due fasi (2PL) è un particolare protocollo che consente di ottenere facilmente degli schedule serializzabili. Esso è molto diffuso nell'ambito dei DBMS commerciali.

In linea di principio un protocollo basato sul meccanismo dei lock prevede che l'accesso alle risorse della base di dati da parte delle transazioni venga protetto mediante l'esecuzione preventiva di alcune primitive: **r_lock**, **w_lock**, **unlock**. Tutte le transazioni, per accedere in lettura/scrittura ad una risorsa, devono **preventivamente richiedere** ed ottenere, mediante l'esecuzione di una delle predette primitive, il consenso da parte del sistema. Il modulo del sistema che concede, secondo determinate strategie di decisione, il consenso all'accesso ad una risorsa è detto **Lock Manager.** In caso di consenso accordato all'esecuzione di un'operazione su una data risorsa, si dice che la relativa transazione ha **acquisito quella risorsa** o ha acquisito **un lock su quella risorsa** ovvero, dualmente, che il sistema ha **concesso un lock** sulla risorsa stessa. Una transazione che ha richiesto un lock su una determinata risorsa viene posta dal Lock Manager in **stato di attesa (wait)** finché quel lock non le viene concesso.

Affinché un'operazione di lettura/scrittura di una transazione su una risorsa possa essere concessa deve soddisfare alcuni **vincoli** necessari per garantire la corretta esecuzione concorrente di tutte le transazioni attive:

1. ogni operazione di lettura (read) deve essere preceduta (anche con molto anticipo) da una richiesta **r_lock**. Su una medesima risorsa possono essere concessi più r_lock detti, per questo motivo, **lock condivisi**;
2. ogni operazione di scrittura (write) deve essere preceduta da una richiesta **w_lock**. Su una risorsa non può essere concesso più di un w_lock contemporaneamente e per questo motivo esso prende il nome di **lock esclusivo**;
3. terminata un'operazione, la risorsa deve essere disimpegnata per consentire l'accesso da parte di altre transazioni e ciò si consegue eseguendo la primitiva **unlock**. Una risorsa è detta **rilasciata** se il Lock Manager concede un **unlock** per essa.

Il **Lock Manager** concede i lock alle transazioni che ne hanno fatto richiesta controllando il rispetto dei suddetti vincoli del protocollo e decidendo sulla base di una politica, schematizzabile come nella seguente tabella denominata **tabella dei conflitti**.

Richiesta/Stato	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	Errore	ok →[*dipende*]	ok → libero

Ogni volta che il Lock Manager concede o rilascia un lock su una risorsa, lo stato di quella risorsa cambia, come mostra la suddetta tabella dei conflitti. Per determinare lo stato delle risorse, il Lock Manager memorizza tutti i lock concessi/rilasciati in una tabella, denominata **tabella dei lock**.

Nota:
Una transazione che deve prima leggere e poi scrivere sulla medesima risorsa deve prima richiedere il lock condiviso e poi incrementare il livello di protezione sulla risorsa mediante una richiesta di lock esclusivo.

Una transazione che rispetta, nella propria esecuzione, le regole suesposte è detta **ben formata rispetto al lock.**

Il protocollo di lock così formulato non garantisce il livello di serializzabilità necessaria per i sistemi DBMS commerciali e, per questo motivo, si sono introdotte al protocollo delle regole supplementari:

regola del lock a due fasi: una transazione, dopo aver rilasciato un lock su una risorsa, non ne può acquisire altri.

Il nome di questa regola è dovuto al fatto che nell'esecuzione di una transazione si individuano due fasi: nella prima, detta **ascendente**, la transazione acquisisce tutti i lock sulle risorse a cui deve accedere, nella seconda, detta **discendente**, li rilascia.

Un sistema in cui le transazioni sono ben formate rispetto al protocollo del lock e seguono la regola del locking a due fasi, con un Lock Manager che rispetta la politica descritta nella tabella dei conflitti, è un sistema caratterizzato dalla SERIALIZZABILITA' delle proprie transazioni.

Si può dimostrare che uno schedule ben formato rispetto al protocollo 2PL è anche conflict-serializzabile ($2PL \subset CSR$).

Per rimuovere l'ipotesi di commit-proiezione dal protocollo del locking a due fasi descritto precedentemente, si deve introdurre un'ulteriore regola descrittiva:

Regola del lock a due fasi STRETTO: i lock acquisiti da una transazione possono essere rilasciati solo dopo l'effettuazione del commit/abort.

Valori	Transazioni		Stato risorse			Fasi		
	T_1	T_2	x	y	Z			
	$R\_lock_1(x)$		1:read					
X=500	$R_1(x)$	$W\_Lock_2(y)$		2:write				
Y=400		$R_2(y)$						
	$R\_Lock_1(y)$			1:wait				
		Y=y-100						
		$W\_Lock_2(z)$			2:write			
Z=100		$R_2(z)$						
		Z=z+100						
Y=300		$W_2(y)$						
Z=200		$w_2(z)$						
		Commit						
		$Unlock_2(y)$		2:free				
	$R\_Lock_1(y)$			1:read				
Y=300	$R_1(y)$							
	$R\_Lock_1(z)$				1:wait			
		$Unlock_2(z)$			2:free			
	$R\_Lock_1(z)$				1:read			
Z=200	$R_1(z)$							
S=1000	S=x+y+z							
	Commit							
	$Unlock_1(x)$		1:free					
	$Unlock_1(y)$			1:free				
	$Unlock_1(z)$				1:free			

L'esempio della figura sovrastante evidenzia le due fasi delle transazioni T_1 e T_2 e mostra come il protocollo 2PL Stretto consenta di risolvere l'anomalia della modifica fantasma.

Il problema 2PL Stretto risolve anche l'anomalia della lettura sporca.

I protocolli basati su *Timestamps*

Questi protocolli sono più semplici ma meno efficaci dei protocolli 2PL e si basano sui **TimeStamps,** che sono degli identificatori generati e associati a determinati eventi temporali e che definiscono un ordinamento.

I protocolli basati su timestamps associano ad ogni transazione t_i un timestamp $T(t_i)$, che rappresenta il momento in cui essa ha avuto inizio (evento), ed accettano esclusivamente quegli schedule che rispettano l'ordinamento seriale delle transazioni in base al valore del relativo timestamp.

Il protocollo, dunque, **impone che le transazioni siano serializzate in base all'ordine in cui esse acquisiscono il proprio timestamp.**

Il controllo di concorrenza associa ad ogni risorsa x del sistema due valori:

1. **RTM(x)**: rappresenta il maggiore dei timestamp delle transazioni che hanno avuto accesso in **lettura** ad una risorsa x. In pratica indica la transazione 'più giovane' che ha avuto accesso in lettura alla risorsa;
2. **WTM(x)**: rappresenta il timestamp dell'ultima transazione che ha avuto accesso in **scrittura** ad una risorsa x.

Una transazione t_i chiede al controllo di concorrenza di poter accedere ad una risorsa x mediante l'invocazione di due primitive: read(x, t_i) per la lettura, write(x, t_i) per la scrittura. Il controllo di concorrenza decide se concedere o meno l'accesso alla risorsa in base alla seguente politica:

1. read(x, t_i): se **$T(t_i) \geq WTM(x)$** allora il permesso di lettura viene **accordato** e RTM(x) = max(RTM(x), $T(t_i)$), altrimenti viene rifiutato e la transazione

abortisce. Questa regola traduce formalmente il principio che una transazione più anziana non può leggere le modifiche di una transazione più recente.

2. write(x, t_i): se **$T(t_i) \geq WTM(x)$ AND $T(t_i) \geq RTM(x)$** allora il permesso di lettura viene accordato e $WTM(x) = T(t_i)$, altrimenti la transazione abortisce. Questa regola traduce formalmente il principio che una transazione non deve sovrascrivere le modifiche o rendere inconsistenti le letture di transazioni più recenti.

Il problema che si presenta con l'uso di questi protocolli, così come illustrati, è l'eccessiva quantità di transazioni che vengono abortite (non sono contemplati stati di attesa). Inoltre, rimuovendo l'ipotesi di commit-proiezione, questi protocolli non funzionano correttamente (WTM(x) ed RTM(x) sono determinati solo nel caso di commit-proiezione).

Per rimuovere l'ipotesi di commit-proiezione e ridurre il fenomeno dell'abbattimento delle transazioni è possibile utilizzare, per ogni transazione che accede ad una risorsa, un buffer. Ogni transazione legge il valore di una risorsa, apporta le eventuali modifiche nel buffer e solo dopo il commit queste interesseranno la risorsa, secondo l'ordine del proprio timestamp. Le transazioni, per la lettura della risorsa, potranno quindi essere sospese (wait) in attesa del commit delle transazioni con timestamp inferiore.

Esempio:

Chiariamo il suesposto accorgimento con un esempio in cui agiscono su una risorsa x tre transazioni i cui timestamp sono così ordinati: $T(t_1) < T(t_2) < T(t_3)$.

Transazioni dello schedule:	t_1	t_2	t_3
x=5	X_1=6	X_2=9	X_3=11
$RTM(x)= T(t_1)$	Read(x, $T(t_1)$)		
(Attenzione)	X=x+1	Read(x, $T(t_2)$)	
	Write(x, $T(t_1)$)	Wait	
X=6	Commit	Wait	
$WTM(x)=T(t_1)$,$RTM(x)=T(t_1)$		Read(x, $T(t_2)$)	
		X=x+3	Read(x, $T(t_3)$)
X=6		Rollback	Wait
$WTM(x)=T(t_1)$,$RTM(x)=T(t_1)$		(Attenzione)	Wait
			Read(x, $T(t_3)$)
			X=x+5
			Write(x, $T(t_3)$)
X=11			Commit
$WTM(x)=T(t_3)$, $WTM(x)=T(t_3)$			

La transazione t_1 legge l'ultimo valore (5) della risorsa x e, immediatamente dopo, lo legge anche la transazione più recente t_2. La transazione t_1 non potrebbe più aggiornare la risorsa x perché $T(t_1) < RTM(x)=T(t_2)$ e dovrebbe, pertanto, abortire. Per evitare il fallimento della transazione t_1 è possibile farle scrivere la modifica (6) su un apposito buffer x_1, il cui contenuto verrà, poi, copiato nella risorsa x a commit effettuato. Tuttavia in questo modo non verrebbe garantita la serializzabilità dello schedule, perché la lettura della transazione t_2 potrebbe risultare sporca. Per questo motivo si introduce un meccanismo di attesa (wait) per il quale la lettura della risorsa x non può avvenire finché la transazione che per prima vi ha avuto accesso (t_1) non ha effettuato li commit. Per rimuovere, poi, l'ipotesi di commit-proiezione, basta aggiornare, oltre alla risorsa x, i valori di WTM(x) e RTM(x) solo a commit avvenuto (vedi il conflitto tra le transazioni t_2 e t_3).

Per aumentare ulteriormente la probabilità di sopravvivenza delle transazioni si può usare un ulteriore accorgimento, denominato **timestamp con versione** (o timestamp multiversione), che consiste nel mantenere diverse versioni delle risorse della base di dati, una per ciascuna transazione che vi deve accedere in scrittura. Ogni operazione di scrittura su una risorsa x della base di dati comporta la creazione di una nuova versione x_i caratterizzata da un proprio $WTM(x_i)$. Tutte le versioni della medesima risorsa hanno lo stesso RTM(x). Il controllo di concorrenza, in questo caso, accetta le azioni delle transazioni secondo la seguente politica:

Supposte N le versioni della medesima risorsa x

1. read(x,$T(t_k)$) **è sempre accettata:** la versione che viene letta è quella x_j tale che:
   - $x_j \Leftrightarrow WTM(x_j) \leq T(t_k) < WTM(x_{j+1})$
   - $x_N \Leftrightarrow T(t_k) > WTM(x_N)$

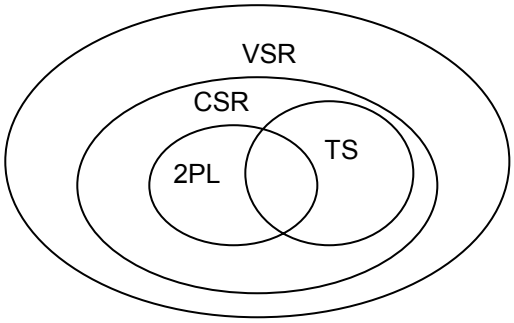
2. write(x,$T(t_k)$) **è rifiutata se** $T(t_k) < RTM(x)$, altrimenti **è accettata**, previa creazione di una nuova versione.

Le versioni vengono rilasciate quando non vi sono più transazioni in lettura interessate al loro valore.

Un'applicazione interessante del protocollo si ha con due versioni della risorsa: quella attuale (x) e quella precedente o vecchia (old_x). Le transazioni più vecchie che intendono leggere una risorsa successivamente alla scrittura da parte di una più recente possono accedere alla versione precedente.

Confronto tra 2PL e TS

Le relazioni tra le classi VSR, CSR, 2PL e TS sono illustrate nella seguente figura:



Confrontando i protocolli 2PL e TS si osserva che:

1. Nel protocollo 2PL le transazioni vengono poste in stato di attesa mentre in quello TS vengono fatte abortire;
2. Nel protocollo 2PL l'ordine della serializzazione è imposto dai conflitti, nel TS dai *timestamp*;
3. la necessità di attendere l'esito di una transazione (commit/abort) provoca nel 2PL l'allungarsi del tempo di blocco e la creazione di condizioni di attesa nel TS [ACBT pg. 316];
4. il protocollo 2PL può presentare il problema del blocco critico (vedi più avanti);

5. l'abort e la reiterazione di una transazione nel protocollo TS costa, in termini di tempo, più dell'attesa nel 2PL.

Il protocollo più diffuso nei DBMS commerciali è il 2PL ed è per questo motivo che, nel seguito, faremo riferimento a questo tipo di controllo di concorrenza.

Il Lock Manager (LM) è un servizio del DBMS a cui possono accedere i vari processi che eseguono le transazioni attraverso l'invocazione delle primitive di richiesta di lock:

- **r_lock(T, X, err, timeout);**
- **w_lock(T,X,err,timeout);**
- **unlock(T,X**);

Il parametro T è l'identificatore della transazione richiedente, X è la risorsa a cui la transazione T deve avere accesso, *err* è un codice di errore che viene restituito dal LM (err=0 se la richiesta viene soddisfatta) e *timeout* è l'intervallo di tempo massimo in cui la transazione T può permanere in stato di attesa per ottenere il lock sulla risorsa, scaduto il quale la transazione abortisce e viene fatta ripartire oppure avanza una nuova richiesta di lock al LM. Il LM può essere visto, in sostanza, come un sistema a CODA.

L'efficienza del LM dipende dalla probabilità che le richieste di lock avanzate al LM da parte delle transazioni vadano in conflitto. Tale probabilità è data da:

$$p = \frac{k * m}{n}$$

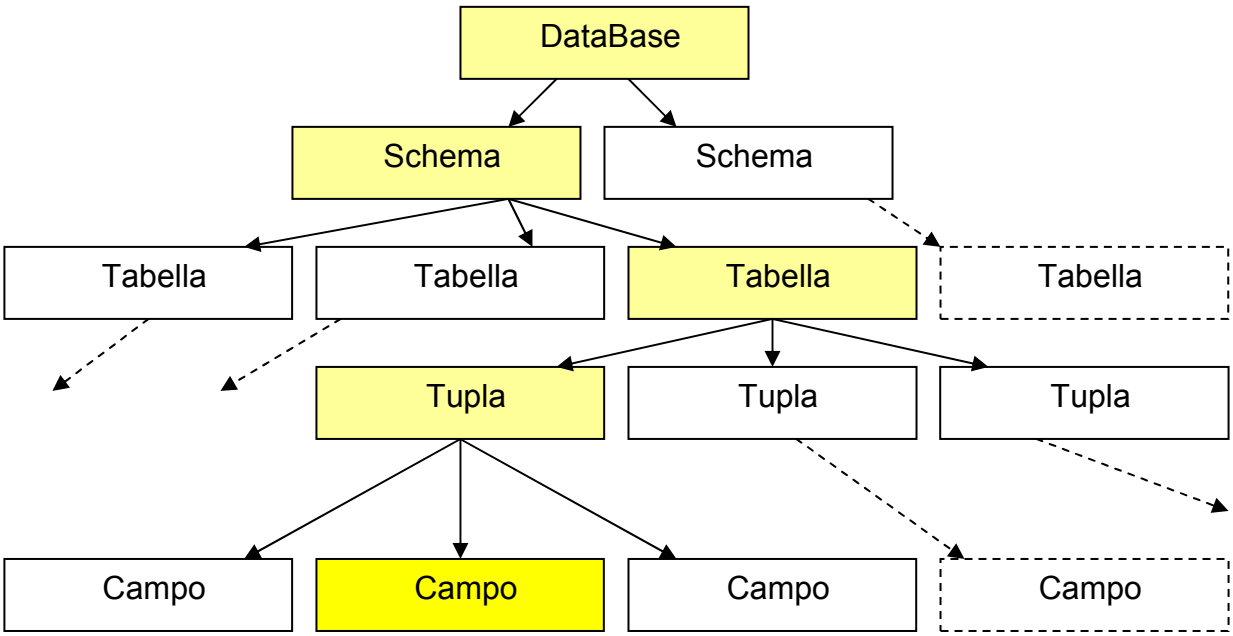
dove:
- k = numero di transazioni operanti nel sistema:
- m = numero medio di risorse a cui accede una transazione;
- n = numero totale di risorse presenti nella base di dati

Ogni volta che il LM concede un lock per una risorsa x, prende nota del lock concesso sulla **tabella di lock** e restituisce il controllo al processo chiamante. Ogni voce della tabella dei lock riporta l'indicazione della risorsa x di riferimento, due bit che ne indicano lo stato (unlocked, r_locked, w_locked) ed il numero dei processi in attesa della concessione del lock. La tabella dei lock, dato il frequente accesso da parte del LM, risiede in memoria centrale.

Locking Gerarchico e Granularità dei Lock

Il protocollo 2PL fa riferimento ad una generica risorsa del database, quale uno schema, una tabella, una tupla, un campo o il database stesso. Esistono, però, precise relazioni tra tutti questi oggetti: una tupla, ad esempio, è formata da più campi ed una tabella è formata da più tuple. Un DBMS deve, quindi, impiegare un protocollo di lock che garantisca il corretto accesso concorrente a tutte le risorse, indipendentemente dal livello di dettaglio o di **granularità**. L'estensione dei protocolli di lock a più livelli di granularità delle risorse è detta **Locking Gerarchico**.

Per comprendere i lock granulari si deve immaginare una gerarchia tra i tipi di risorse strutturata ad albero, come nella seguente illustrazione:



Ovviamente per un'operazione di aggiornamento di un campo non si deve richiedere il lock di tutta la tupla, mentre per un'operazione di ristrutturazione di una tabella si deve poter richiedere un lock esclusivo anche su tutte le sue tuple e sui relativi campi.

Per far il protocollo prevede un'estensione delle primitive del 2PL con ulteriori primitive specifiche per gestire la granularità:

- XL(x) (*exclusive lock*): primitiva di richiesta di un lock esclusivo su una risorsa x;
- SL(x) (*shared lock*): primitiva di richiesta di un lock condiviso su una risorsa x;
- ISL(x) (*intentional shared lock*): primitiva che esprime l'intenzione di ottenere un lock condiviso su una risorsa x che discende da quella corrente;
- IXL(x) (*intentional exclusive lock*): primitiva che esprime l'intenzione di ottenere un lock esclusivo su una risorsa x che discende da quella corrente;
- SIXL (*shared intentional-exclusive lock*): primitiva che richiede un lock condiviso sulla risorsa x ed esprime l'intenzione di ottenere un lock esclusivo su una delle risorse che discendono da quella corrente.

Le regole del protocollo per conseguire un lock granulare al livello i sono:

1. si richiedono i lock partendo dalla radice e scendendo di livello lungo l'albero fino alla risorsa di destinazione;
2. si rilasciano i lock a partire dalla risorsa acquisita fino alla radice;
3. per poter richiedere ed ottenere un lock SL o ISL su una risorsa si **deve** già possedere un lock ISL o IXL sulla risorsa da cui questa discende;
4. per poter richiedere ed ottenere un lock XL, SIXL o IXL su una risorsa si **deve** già possedere un lock IXL o SIXL sulla risorsa da cui questa discende;
5. il Lock Manager decide sulla base della politica illustrata nella seguente tabella:

Richiesta/Stato	ISL	IXL	SL	SIXL	XL
ISL	SI	SI	SI	SI	**NO**
IXL	SI	SI	**NO**	**NO**	**NO**
SL	SI	**NO**	SI	**NO**	**NO**
SIXL	SI	**NO**	**NO**	**NO**	**NO**
XL	**NO**	**NO**	**NO**	**NO**	**NO**

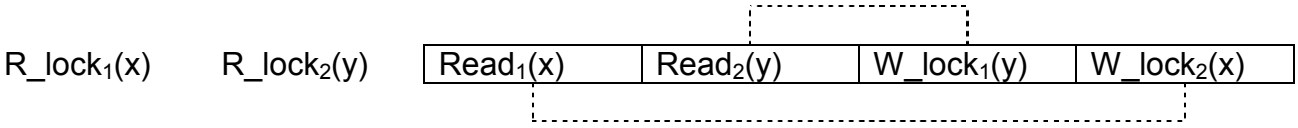
La scelta del giusto livello di granularità è lasciata al progettista: un livello grossolano di granularità può comportare limitazioni al parallelismo, un livello troppo raffinato comporta un sovraccarico computazionale sul LM per l'abbondante quantità di richieste di lock generate.

Il problema del blocco critico (*dead lock*)

Il blocco critico è un fenomeno che si può verificare in contesti dove il controllo di concorrenza impiega un protocollo che prevede degli stati di attesa. **Il blocco critico si verifica quando due o più transazioni sono in attesa reciproca**.

Esempio:

nel seguente schedule vi è un blocco critico:



La probabilità che si verifichi un blocco critico, limitatamente al caso di coppie di transazioni in reciproca attesa, **varia linearmente col numero globale K di transazioni presenti nel sistema e quadraticamente col numero medio m di risorse cui ciascuna transazione deve accedere.**

Per sbloccare un blocco critico si può ricorrere all'uso del *timeout*: allo scadere dell'intervallo di tempo prefissato, la relativa transazione viene fatta abortire. Il valore più opportuno del timeout deve essere scelto considerando che se esso è troppo lungo deteriora le prestazioni generali del sistema, mentre se è troppo breve può far abortire anche transazioni in attesa non coinvolte in blocchi critici.

I blocchi critici si possono **prevenire** mediante diverse tecniche, tra le quali ricordiamo:

1) ogni transazione deve richiede preventivamente, in un'unica soluzione, i lock su tutte le risorse a cui deve accedere. Questa tecnica è difficilmente attuabile perché, fuori dall'ipotesi di commit-proiezione, non sono note a priori le risorse a cui una transazione deve accedere.

2) ogni transazione ottiene, all'avvio, un *timestamp* che viene impiegato, in caso di attesa di rilascio di un lock su una risorsa impegnata da un'altra transazione, per decidere quale delle due debba abortire (allo scadere del timeout). Un criterio di scelta comunemente usato è quello di far abortire le transazioni più recenti che, con molta probabilità, hanno lavorato di meno. In questo modo, tuttavia, si presenta il problema del blocco individuale (*Starvation*) per il quale le nuove transazioni trovano grande difficoltà ad avviarsi. Per ridurre questo fenomeno ed evitare che una transazione abortisca un numero di volte eccessivamente elevato, la si reitera conservando il *timestamp* precedentemente assegnatole.

3) ispezione diretta della **tabella dei lock** alla ricerca di cicli tra le relazioni di attesa.

Controllo di concorrenza in SQL-92

In generale i protocolli usati per il controllo di concorrenza possono causare un abbassamento del livello generale delle prestazioni del sistema. Pertanto, laddove praticabile e conveniente, è possibile rinunciare a qualche garanzia sulle 'anomalie dell'esecuzione concorrente di transazioni' a favore di migliori prestazioni. Per questo motivo molti DBMS commerciali danno la possibilità di definire, diversi livelli di isolamento per le transazioni.

In SQL-92 sono disponibili quattro livelli di isolamento, ciascuno caratterizzato dalla possibilità di proteggere l'esecuzione delle transazioni da uno o più tipi di anomalie:

LIVELLO	PERDITA DI AGGIONAMENTO	LETTURE INCONSISTENTI	AGGIORNAMENTO FANTASMA
READ UNCOMMITTED	**Non protetto**	**Non protetto**	**Non protetto**
READ COMMITTED	**protetto**	**Non protetto**	**Non protetto**
REPETIBLE READ	**Protetto**	**protetto**	**Non protetto**
SERIALIZABLE	**protetto**	**protetto**	**Protetto**

Il minimo livello di protezione si ha con il READ UNCOMMITTED, il massimo con SERIALIZABLE.

In molti DBMS SERIALIZABLE e REPETIBLE READ garantiscono lo stesso livello di protezione, ma SERIALIZZABLE consente la gestione del **lock di predicato**.

Il lock di predicato è un meccanismo di protezione che risolve la seguente anomalia molto particolare: supponiamo di avere due transazioni t_1 e t_2 che insistono su una stessa tabella. La transazione t_1 effettua due aggregazioni consecutive dei valori di un campo delle tuple risultanti da una selezione, mentre la transazione t_2 esegue un inserimento di una nuova tupla nella tabella. Supponiamo che nello schedule l'inserimento della tupla da parte di t_2 avvenga tra le due letture di t_1: è molto probabile che i valori aggregati letti dalla transazione t_1 risultino diversi, anche se la transazione t_2 non ha provocato alcuna delle anomalie viste in precedenza.

Per evitare questa anomalia alcuni DBMS mettono a disposizione di una transazione un particolare lock, il lock di predicato, che impedisce alle altre di compiere azioni che interferiscano con un predicato in esecuzione. Nel nostro esempio la transazione t_1 può richiedere un lock sul predicato di selezione in modo che la transazione t_2 non modifichi il valore aggregato.