



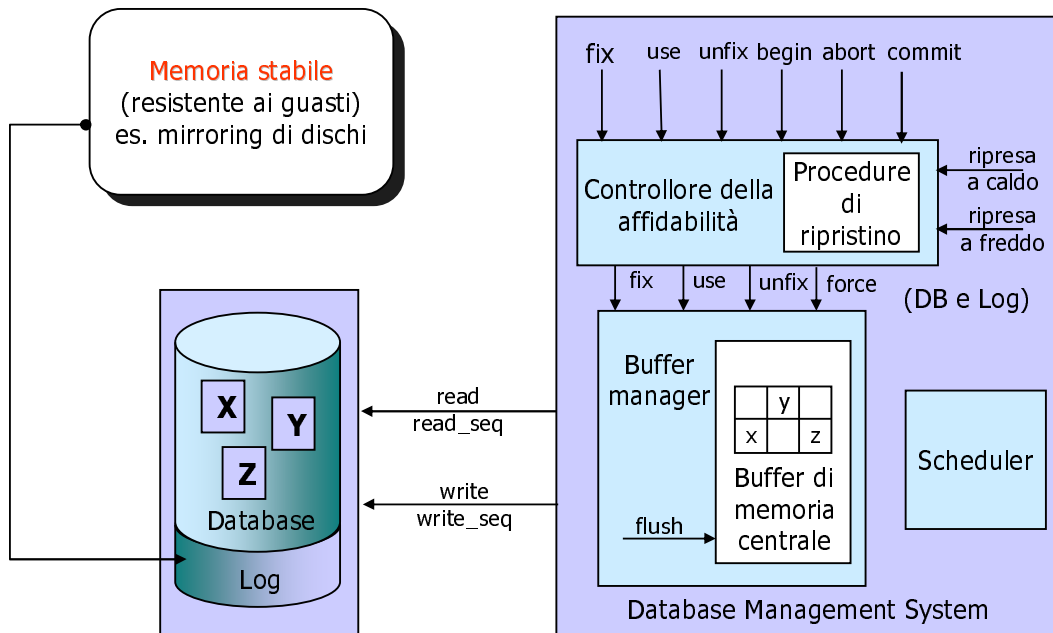
Controllo di affidabilità'



Controllo di affidabilità

- Garantisce l'**atomicità** e la **persistenza** delle transazioni
- Si basa su un file di **log**
 - Il log registra le azioni (scritture) svolte dal DBMS
 - Il log permette di fare l'**undo** o il **redo** delle azioni
- Realizza i comandi transazionali
 - begin transaction (B)
 - commit work (C)
 - rollback work (A)
- Permette il ripristino in caso di malfunzionamenti (ripresa a caldo - ripresa a freddo)

Controllore di affidabilità



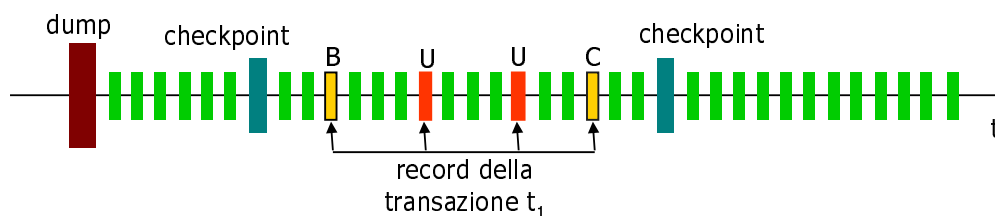
Scarselli Franco

Sistemi per basi di dati 2005-2006

101

Organizzazione del file di log

- Il file di log è un **file sequenziale** su cui vengono registrate le azioni svolte dalle transazioni in ordine temporale



- record di transazione**
 - record di begin (B)
 - record relativi alle operazioni effettuate (Insert - Update - Delete)
 - record di commit (C) o abort (A)
- record di sistema (Dump - Checkpoint)**

Scarselli Franco

Sistemi per basi di dati 2005-2006

102



Record di transazione

- **Begin** B(T) - **Commit** C(T) - **Abort** A(T)
Specificano l'identificativo T della transazione
- **Update** U(T,O,BS,AS)
Specificano la transazione T, l'oggetto O su cui si è fatto l'aggiornamento, il valore BS dell'oggetto prima della modifica (Before State) e quello AS dopo la modifica (After State)
- **Insert** I(T,O,AS)
Specificano la transazione T, l'oggetto O di cui si è fatto l'inserimento e il valore AS dopo l'inserimento
- **Delete** D(T,O,BS)
Specificano la transazione T, l'oggetto O di cui si è effettuata la cancellazione e il valore che aveva prima della cancellazione



Undo e redo

- I record di transazione permettono di disfare (undo) o rifare (redo) le rispettive azioni
- **Undo**
 - si ricopia in O il valore precedente BS
 - per annullare l'insert si cancella l'oggetto O
- **Redo**
 - si ricopia in O il valore AS
 - per ripetere la cancellazione basta cancellare O
- Vale l'**idempotenza**: ripetendo più volte lo stesso undo (redo) si ottiene lo stesso effetto di una sola ripetizione



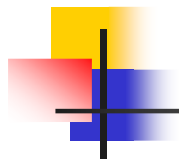
Checkpoint e dump

- **Checkpoint** - $C(T_1, T_2, \dots, T_k)$
 - Operazione periodica per registrare tutte le transazioni attive
 - Si scrivono su disco le pagine relative a transazioni terminate con commit o abort (flush)
 - Non sono accettate operazioni di commit fino al termine del checkpoint
 - Gli effetti delle transazioni che hanno eseguito un commit o un abort sono rese persistenti
 - Si scrive nel log il record di checkpoint che contiene gli identificatori delle transazioni attive T_1, T_2, \dots, T_k
- **Dump** (backup) - DUMP
 - Copia completa della base di dati compiuta offline su supporto affidabile



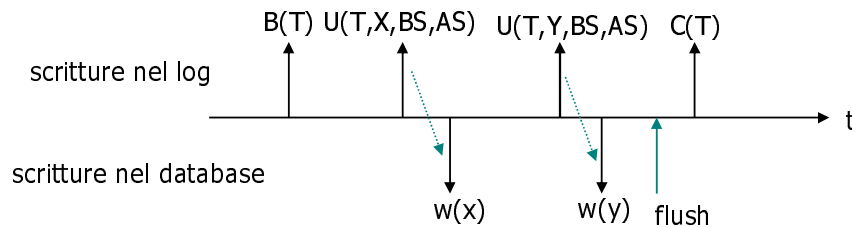
Gestione delle transazioni

- **Regola WAL (Write Ahead Log)**
 - La parte BS deve essere scritta nel log prima di effettuare la corrispondente operazione nella base di dati
 - Viene mantenuto in modo affidabile il valore precedente alla scrittura
 - Permette di fare l'undo delle scritture di una transazione che non ha fatto il commit
 - In pratica il record di log è scritto completamente (BS e AS)
- **Regola di Commit-Precedenza**
 - La parte AS deve essere scritta nel log prima di effettuare il commit
 - Permette di fare il redo delle transazioni che sono completate col commit - la scrittura del record di commit rende effettiva la transazione
 - In pratica il record di log è scritto completamente (BS e AS)

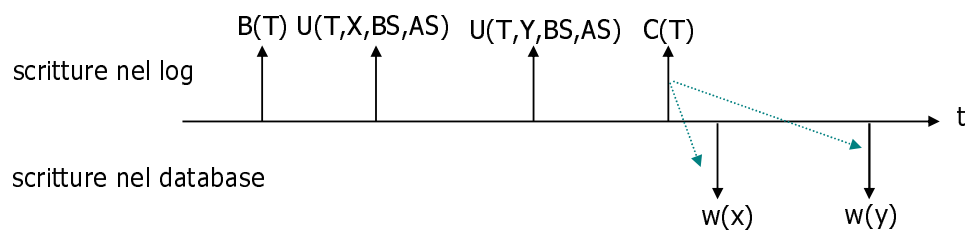


Protocolli di scrittura del log

- flush (force) prima del commit - non richiede il redo

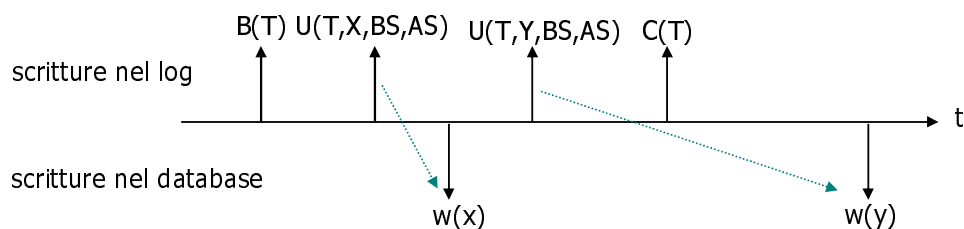


- flush (force) dopo il commit - non richiede undo



Protocolli di scrittura del log

- nessun vincolo sul momento della scrittura rispetto al commit



- Le operazioni di flush sono ottimizzate dal buffer manager
- Richiede redo e undo
- L'uso del file di log rappresenta un **carico aggiuntivo** per il sistema
- Devono essere adottate soluzioni efficienti per la scrittura del log

Tipologie di guasti

■ Guasti di sistema

- Bug del software, cali di tensione
- Perdita del contenuto della memoria centrale (buffer)
- Persistenza dei dati in memoria secondaria (dati e log)

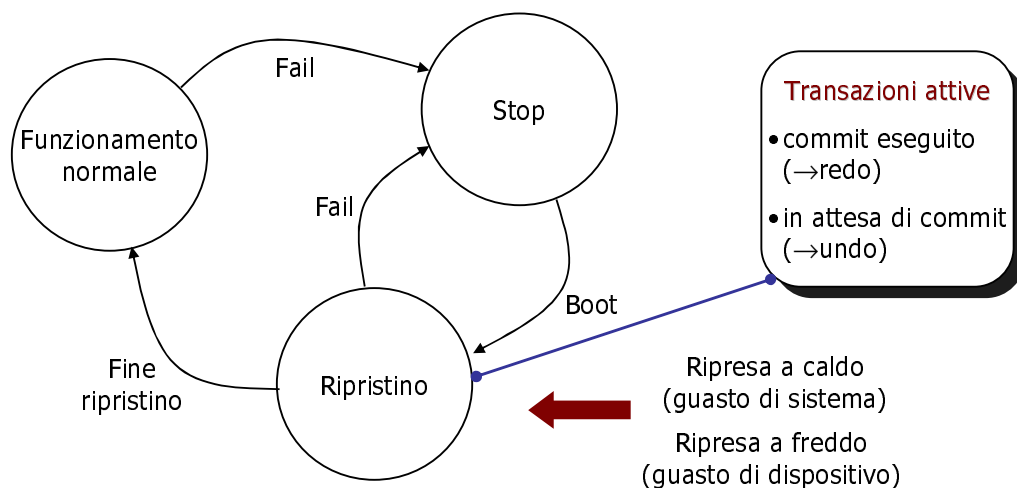
■ Guasti di dispositivo

- Rottura dei dispositivi di memoria di massa
- Si assume che il log sia salvato su memoria stabile (affidabile)
- Si perde il contenuto della base di dati ma non del log
- La perdita del log è un **evento catastrofico** senza recupero!

Modello fail-stop

■ Fail-stop

quando si rileva un guasto il sistema arresta tutte le transazioni





Warm restart

- Si accede all'ultimo blocco del log e si ripercorre il log indietro fino al record di checkpoint
- Si determinano le transazioni da rifare (REDO) e da disfare (UNDO)
 - Inizializzazione
 - $UNDO = \{\text{Transazioni attive al checkpoint}\}$
 - $REDO = \emptyset$
 - Scan del log in avanti
 - $B(T) \Rightarrow UNDO = UNDO \cup \{T\}$
 - $C(T) \Rightarrow REDO = REDO \cup \{T\}$ e $UNDO = UNDO / \{T\}$
 - Un'azione è nell'insieme UNDO anche se la transazione è terminata con un record di abort $A(T)$



Warm restart

- Si ripercorre all'indietro il log fino all'azione più vecchia delle transazioni in UNDO e REDO
 - Si può anche andare oltre il checkpoint (le transazioni attive al checkpoint hanno azioni precedenti)
 - Si disfano le azioni delle transazioni nell'insieme UNDO
 - Si applicano le azioni delle transazioni nell'insieme di REDO nell'ordine in cui sono nel log
- **Atomicità**: viene garantito che le transazioni in corso al momento del guasto non siano lasciate in uno stadio intermedio
- **Persistenza**: le transazioni che hanno fatto il commit sono completate rendendo permanenti le modifiche al database



Un esempio

- File di log che coinvolge le transazioni T_1, T_2, T_3, T_4, T_5

$B(T_1) B(T_2) U(T_2, O_1, B_1, A_1) I(T_1, O_2, A_2) B(T_3) C(T_1) B(T_4)$
 $U(T_3, O_2, B_3, A_3) U(T_4, O_3, B_4, A_4) CK(T_2, T_3, T_4) C(T_4) B(T_5) U(T_3, O_3, B_5, A_5)$
 $U(T_5, O_4, B_6, A_6) D(T_3, O_5, B_7) C(T_5) I(T_2, O_6, A_8) FAIL$

- Il log Si accede al checkpoint precedente al FAIL
 - $UNDO = \{T_2, T_3, T_4\}$ e $REDO = \{\}$
- Si aggiornano gli insiemi REDO e UNDO
 - $C(T_4)$: $UNDO = \{T_2, T_3\}$ e $REDO = \{T_4\}$
 - $B(T_5)$: $UNDO = \{T_2, T_3, T_5\}$ e $REDO = \{T_4\}$
 - $C(T_5)$: $UNDO = \{T_2, T_3\}$ e $REDO = \{T_4, T_5\}$



Un esempio

$B(T_1) B(T_2) U(T_2, O_1, B_1, A_1) I(T_1, O_2, A_2) B(T_3) C(T_1) B(T_4)$
 $U(T_3, O_2, B_3, A_3) U(T_4, O_3, B_4, A_4) CK(T_2, T_3, T_4) C(T_4) B(T_5) U(T_3, O_3, B_5, A_5)$
 $U(T_5, O_4, B_6, A_6) D(T_3, O_5, B_7) C(T_5) I(T_2, O_6, A_8) FAIL$

- UNDO
 - T_2 : Delete(O_6) T_3 : Insert($O_5=B_7$); $O_3=B_5$; $O_2 = B_3$ T_2 : $O_1 = B_1$
- REDO
 - T_4 : $O_3 = A_4$ T_5 : $O_4 = A_6$

Cold restart

- Il guasto consiste in un deterioramento di una parte del database
- I passi del cold restart sono
 - Si ripristina la parte deteriorata dall'ultimo **backup** (dump)
 - Si accede al record di dump più recente nel log
 - Si ripercorre in avanti il log ripetendo tutte le azioni per ripristinare la parte deteriorata
 - Si esegue una ripresa a caldo

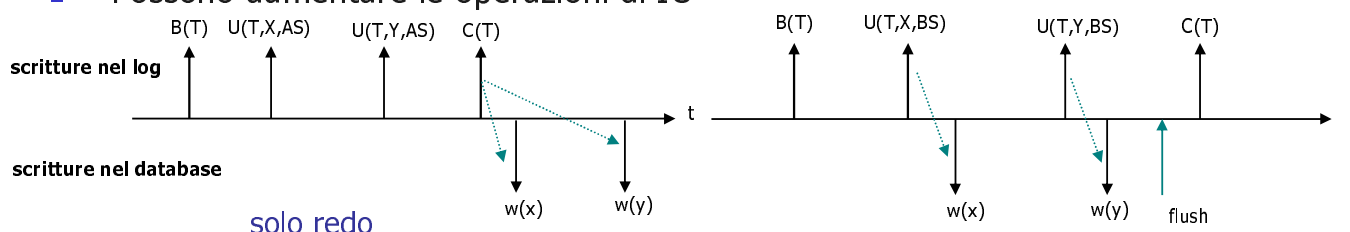
Strategie di log semplificate

solo redo

- Non si registrano le BS
- Occorre scrivere le modifiche sul disco **solo dopo la scrittura del commit** sul log
- Possono aumentare le operazioni di IO
- Può servire un buffer più grande

solo undo

- Non si registrano le AS
- Le modifiche si scrivono sul disco **solo prima della scrittura del commit** sul log
- Possono aumentare le operazioni di IO



Backup

I backup

- servono in caso di guasti con perdita dei dati
- riguardano
 - i dati: si copia il contenuto dell'archivio (dopo si può tagliare il log)
 - il log: permette di ricostruire lo stato della base di dati ad un qualsiasi istante
- sono di due tipi
 - completi:
 - si copia tutto il contenuto dell'archivio
 - il backup è più lento
 - incrementali:
 - si copia solo i cambiamenti dall'ultimo backup
 - per ripristino occorre prima ripristinare un backup completo e poi i backup incrementali successivi
 - il ripristino è più lento
- Tipicamente si alternano backup completi a backup incrementali:
(es. completo ogni settimana, incrementale ogni giorno)

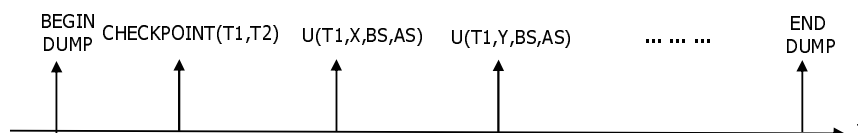
Backup on line

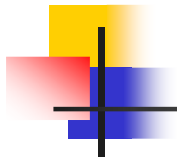
La soluzione più semplice

- disconnettere tutti gli utenti e fare il backup
- deve esistere un periodo sufficientemente lungo durante il quale il DBMS può essere fermato (ad es., la notte)

Backup con transazioni attive

- durante il backup
 - si scrive sul log il momento di inizio
 - si fa un checkpoint
 - si copiano i dati
 - si scrive sul log il momento di fine del backup
 - si copia il log

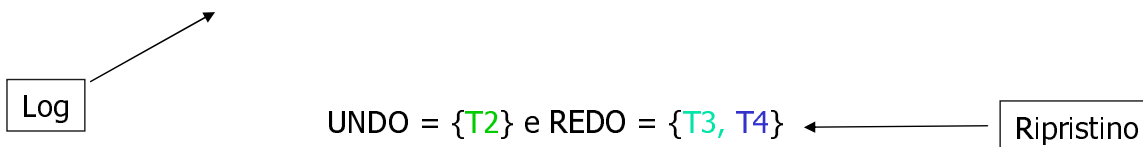




Backup on line II

- durante il ripristino
 - si ricopia dal backup i dati andati persi
 - si applica una ripartenza a caldo, come se ci fosse stato un fallimento all'istante della fine del backup
 - l'archivio si ritrova nello stato del momento in cui è finito il backup
(per metterlo nello stato del momento in cui è iniziato il backup occorre non usare i commit fra l'inizio e la fine del backup)

B(T1) B(T2) U(T2,O1,B1,A1) I(T1,O2,A2) C(T1) B(T3) U(T3,O3,B4,A4)
BEGIN_DUMP CK(T2,T3) C(T3) B(T4) U(T4,O4,B6,A6) C(T4) I(T2,O6,A8) END_DUMP



Gestione della concorrenza

Controllo di concorrenza

- Accesso al DBMS da parte di più utenti
- Il carico si misura in **tps** (trasactions per second)
 - 10-10² tps in sistemi bancari
 - 10³ tps sistemi di prenotazione dei voli, gestori delle carte di credito
- La concorrenza permette un uso efficiente del DBMS massimizzando il numero di transazioni eseguite al secondo e minimizzando i tempi di risposta
- Si considerano le **operazioni di ingresso/uscita** di basso livello
 - trasferimenti fra la memoria di massa e la memoria centrale di blocchi di dati (pagine) - letture/scritture
- Il controllore della concorrenza è uno **scheduler** determina se le richieste possono essere soddisfatte

Scarselli Franco

Sistemi per basi di dati 2005-2006

121

Architettura

gestore delle concorrenza (scheduler)

- garantisce che le transazioni concorrenti non interferiscano. Può usare una tabella dei lock

gestore dell'affidabilità

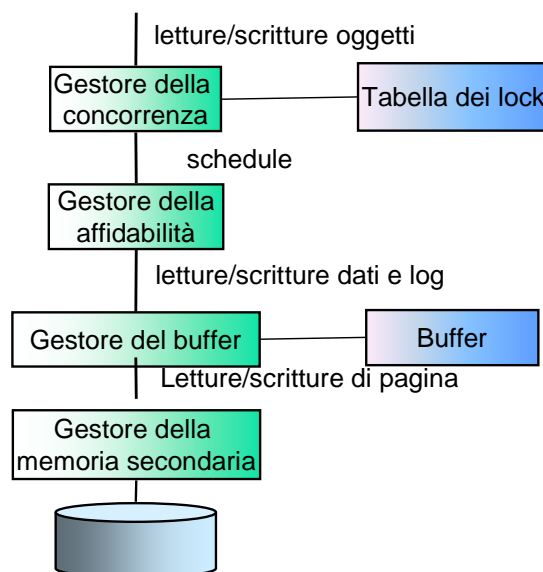
- gestisce le strategie di log e ripristino

gestore del buffer

- gestisce i trasferimenti da disco a memoria primaria

PS

Nota che tali moduli non corrispondono necessariamente a moduli di un DBMS reale



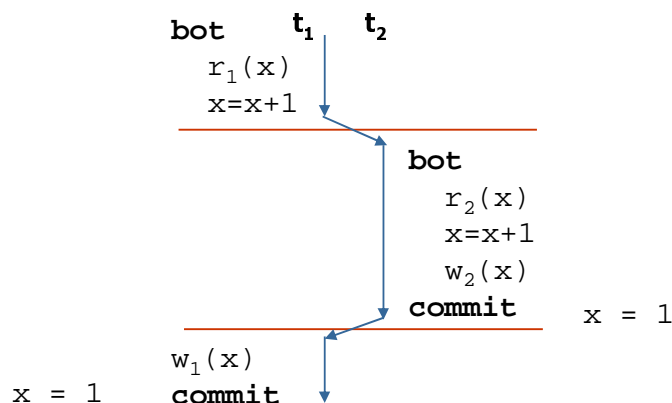
Scarselli Franco

Sistemi per basi di dati 2005-2006

122

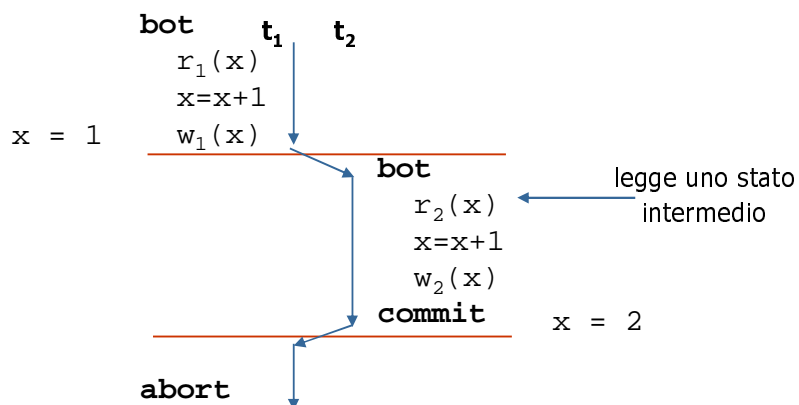
Perdita di aggiornamento

- Si considerano due transazioni che operano sullo stesso dato
 - $t_1 : r(x), x=x+1, w(x)$
 - $t_2 : r(x), x=x+1, w(x)$
- se inizialmente $x=0$, dopo l'esecuzione seriale $x=2$



Lecture sporche

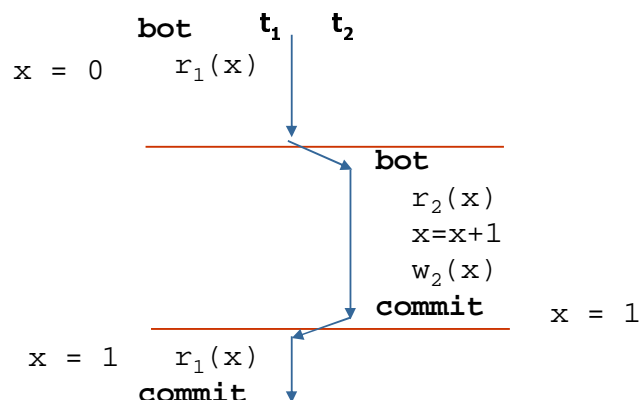
- Si considerano le due transazioni t_1 e t_2 con fallimento di t_1



- Per il fallimento di t_1 , dovrebbe essere $x=1$ alla fine
- L'abort di t_1 dovrebbe causare il fallimento di t_2 (effetto domino)

Lecture inconsistenti

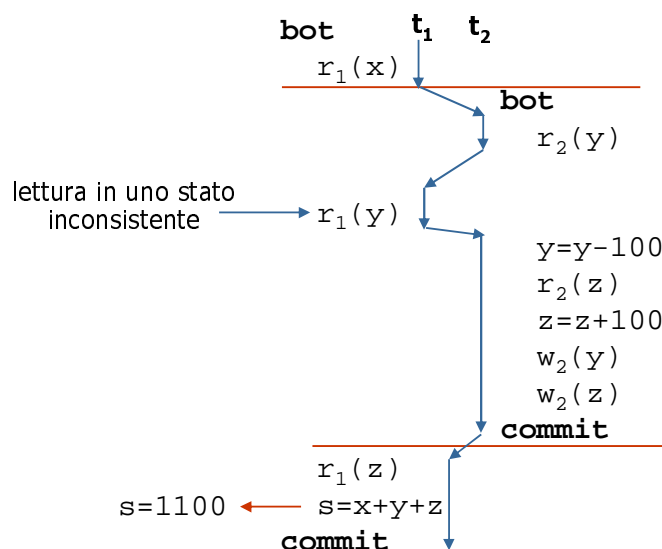
- La transazione t_1 ripete la lettura di x in istanti successivi



- All'interno della stessa transazione il valore letto non deve risentire dell'effetto di altre transazioni

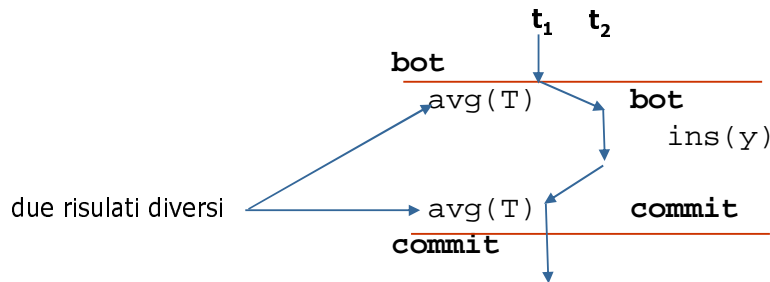
Aggiornamento fantasma

- Si suppone che esista il **vincolo di integrità** $x + y + z = 1000$



Inserimento fantasma

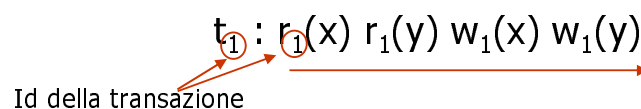
- Una transizione calcola due volte un valore aggregato $avg(T)$ mentre l'altra inserisce un nuovo dato $ins(y)$



- Esiste anche la rimozione fantasma

Teoria della concorrenza

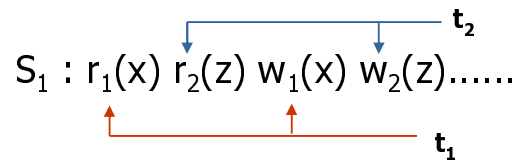
- Transazione** = sequenza di azioni di scrittura o lettura
- Ogni transazione ha un unico identificatore assegnato
- Ogni transazione è compresa fra i comandi **begin transaction** e **end transaction** (omessi)
- Un esempio è



- Il controllo di concorrenza **accetta/rifiuta esecuzioni concorrenti durante l'evoluzione delle transizioni** (senza saperne l'esito)

Schedule

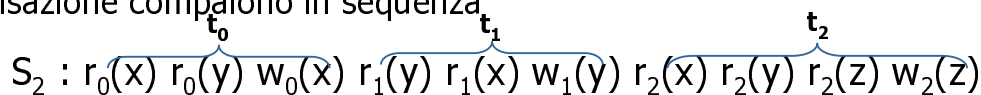
- Uno schedule è una sequenza di operazioni di lettura/scrittura corrispondente a transazioni concorrenti



- In teoria, si considerano solo transazioni che terminano con un commit (**commit-proiezioni**)
 - In pratica, il controllore della concorrenza deve occuparsi anche delle transazioni che terminano con un abort
- Obiettivo del controllo di concorrenza è di riuscire a generare solo schedule che non causano anomalie

Schedule seriali e serializzabili

- Uno **schedule seriale** è uno schedule in cui tutte le operazioni di ogni transazione compaiono in sequenza



- le transazioni godono della proprietà dell'isolamento
- Uno schedule è **serializzabile** se produce lo stesso risultato di uno schedule seriale delle stesse transazioni
 - l'utente non può distinguere fra schedule seriali e schedule serializzabili
- gli schedule serializzabili
 - si comportano come schedule seriali
 - sono più efficienti degli schedule seriali

Schedule seriali e serializzabili II

La teoria

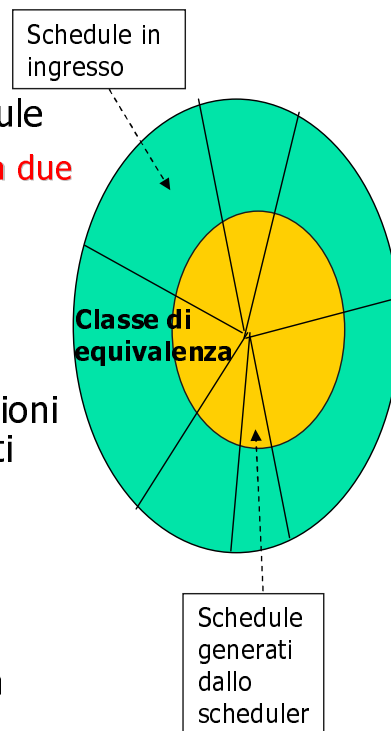
- si introduce la nozione di equivalenza di schedule
 - **view-equivalenza, conflict-equivalenza, locking a due fasi**, controllo basato su **timestamp**
- ciascuna equivalenza identifica una classe di schedule accettabili

La pratica

- uno scheduler trasforma la sequenza di transizioni in ingresso in schedule serializzabili equivalenti
 - fa in modo che le transizione serializzabili non si verifichino oppure
 - uccide le transazioni non serializzabili
- Il progetto dello scheduler deve bilanciare l'efficienza delle transazioni e l'efficienza quella dello scheduler

Scarselli Franco

Sistemi per basi di dati 2005-2006



131

View-equivalenza

Definizioni

- $r_i(x)$ **legge-da** $w_j(x)$ ($legge(r_i(x), w_j(x))$)
se $w_j(x)$ precede $r_i(x)$ e non esiste $w_k(x)$ compreso fra $w_j(x)$ e $r_i(x)$

$$..... w_j(x) r_i(x)$$
- $w_i(x)$ è una **scrittura finale** se è l'ultima scrittura di x nello schedule
- Due schedule sono **view-equivalenti** ($S_i \approx_v S_j$) se su di essi sono definite le stesse relazioni legge-da e le stesse scritture finali
- Uno schedule è **view-serializzabile** se è view-equivalente ad un generico schedule seriale
- L'insieme degli schedule view-serializzabili è detto **VSR**



Un esempio: schedule view-equivalenti

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

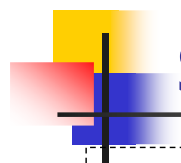
$S_4 : w_0(x) \ r_1(x) \ r_2(x) \ w_2(x) \ w_2(z)$

S_4 è uno schedule seriale; S_3 è serializzabile

$S_5 : w_0(x) \ r_1(x) \ w_1(x) \ r_2(x) \ w_1(z)$

$S_6 : w_0(x) \ r_1(x) \ w_1(x) \ w_1(z) \ r_2(x)$

S_6 è uno schedule seriale; S_5 è serializzabile



Schedule non serializzabili

- Perdita di aggiornamento

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

$r_1(x) \ w_1(x) \ r_2(x) \ w_2(x)$

$r_2(x) \ w_2(x) \ r_1(x) \ w_1(x)$

- Lettura inconsistente

$S_8 : r_1(x) \ r_2(x) \ w_2(x) \ r_1(x)$

$r_1(x) \ r_1(x) \ r_2(x) \ w_2(x)$

$r_2(x) \ w_2(x) \ r_1(x) \ r_1(x)$

- Aggiornamento fantasma

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

Schedule seriali disponibili



Complessità

- Decidere se due schedule sono view-equivalenti ha complessità **lineare**
 - E' efficiente confrontare fra loro due schedule
- Decidere se un generico schedule è view-serIALIZZABILE è un problema **NP-completo**
 - Occorre confrontare lo schedule con tutti gli schedule seriali ottenuti permutando le transazioni
 - Non si può usare questa nozione per verificare la serializzabilità



Conflitti

- L'azione a_i è in **conflitto** con l'azione a_j ($i \neq j$) se operano sullo stesso oggetto ed almeno una è un write
 - Conflitti lettura-scrittura (rw o wr)
 - Conflitti scrittura-scrittura (ww)
 - Azioni della stessa transizione
- Intuitivamente:
 - due azioni sono in conflitto se non si può cambiare l'ordine senza effetti

Esempi: si può scambiare l'ordine delle azioni senza effetti

- $r_i(x) \dots r_j(y)$ non è mai un conflitto anche se $x=y$
Le letture non modificano i valori
- $r_i(x) \dots w_j(y)$ o $w_j(y) \dots r_i(x)$ o $w_j(y) \dots w_i(x)$ non è un conflitto se $x \neq y$
La modifica su y non cambia la lettura/scrittura su x



Conflitti

Esempi: non si può scambiare l'ordine delle seguenti azioni

- $r_i(x) \dots r_i(y)$ generano un conflitto (azioni della stessa transazione)
Le azioni in una stessa transazione hanno un ordine prefissato
- $w_i(x) \dots w_j(x)$ è un conflitto
Il valore di x dopo l'esecuzione è definito da t_j . Scambiando l'ordine sarebbe invece definito da t_i
- $w_i(x) \dots r_j(x)$ è un conflitto
Il valore di x che deve essere letto è quello scritto da t_j
- $r_i(x) \dots w_j(x)$ è un conflitto
Il valore di x che deve essere letto è quello precedente alla scrittura da parte di t_j

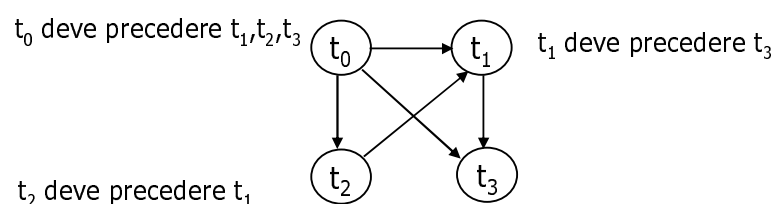
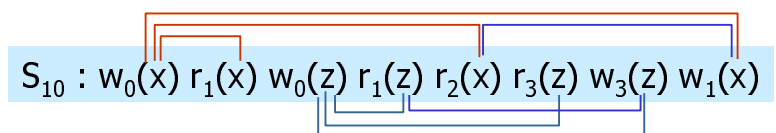


Conflict-equivalenza

- Si possono fare scambi che non cambiano i conflitti presenti
- Due schedule sono **conflict-equivalenti** ($S_i \approx_c S_j$) se
 - i due schedule presentano le stesse operazioni
 - ogni coppia di operazioni in conflitto è nello stesso ordine in entrambi gli schedule
- Uno schedule è **conflict-serializzabile** se è conflict-equivalente ad un generico schedule seriale
- Se due azioni sono in conflitto le corrispondenti transazioni nello schedule serializzato devono essere nello stesso ordine
- L'insieme degli schedule conflict-serializzabili è detto CSR
- Risulta $CSR \subset VSR$

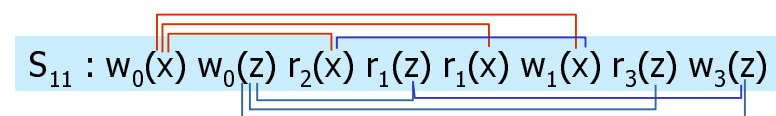
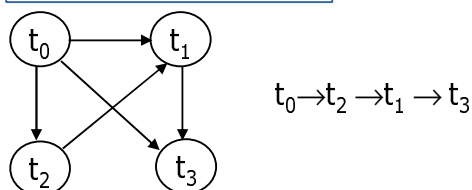
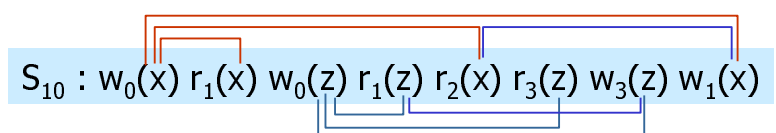
Conflict-serializzabilità

- Si costruisce il grafo dei conflitti (o di precedenza)
 - ogni nodo corrisponde ad una transazione
 - Un arco fra t_i e t_j indica che c'è almeno un conflitto fra un'azione a_i e un'azione a_j tali che a_i precede a_j



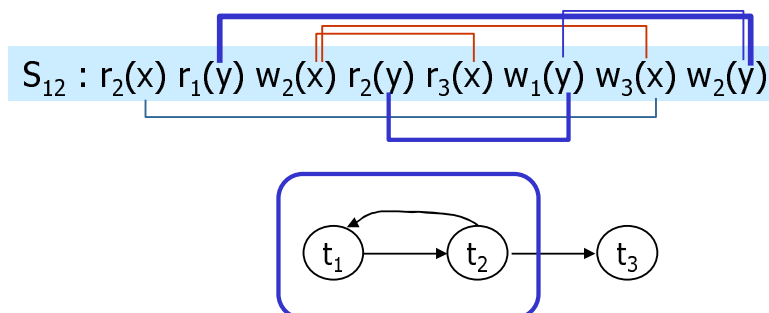
Conflict-serializzabilità

- Uno schedule è CSR se e solo se il grafo è aciclico
- La serializzazione è data da un ordinamento topologico dei nodi
- La complessità è lineare nel numero dei nodi del grafo (transazioni)



Conflict-serializzabilità

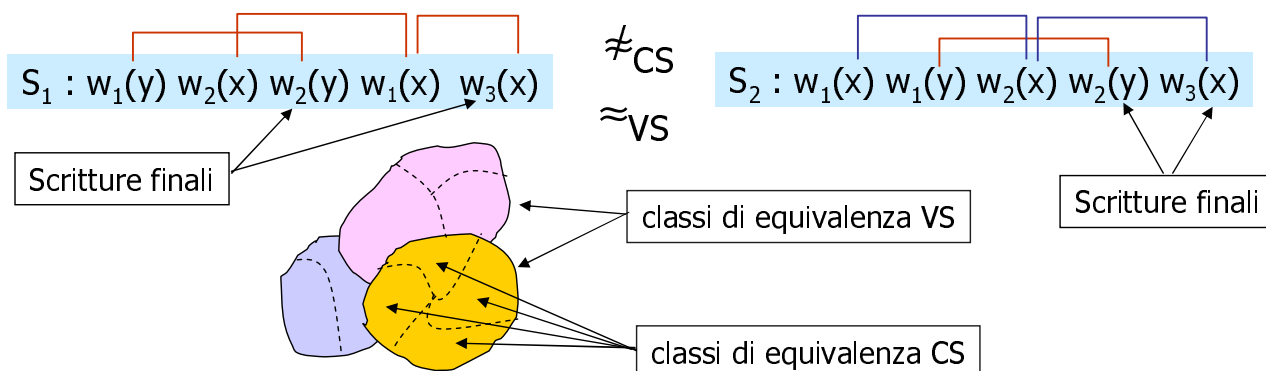
- Un esempio di schedule non conflict-serializzabile



- Non si può decidere se mettere prima t_1 o t_2
- Questo mostra perché lo schedule non è conflict-serializzabile se ci sono cicli nel grafo delle precedenze

Conflict e view serializzabilità

- Uno schedule conflict-serializzabile e' view-serializzabile ($CSR \subset VSR$)
- Non è vero il viceversa: ci sono schedule view-serializzabili che non sono conflict-serializzabili
 - le classi di equivalenza definiti dalla view-serializzabilità sono più piccole di quelle definite dall'conflict-serializzabilità



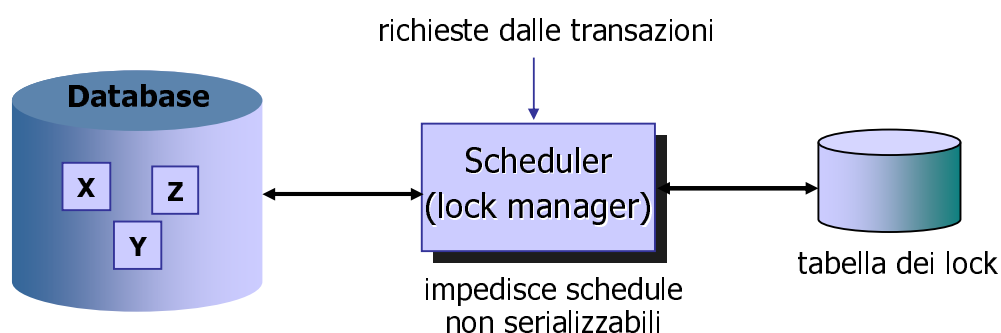
Implementazione dello scheduler

Esistono due approcci diversi all'implementazioni dello scheduler

- Approcci basati sul **controllo delle transazioni**
 - si controlla le transazioni evitando **il verificarsi delle anomalie**: lo scheduler genera solo schedule serializzabili
 - tipicamente questi approcci sono basati su locking
- Approcci **ottimistici**
 - assumono che **tutto andrà bene**, se poi si verificano anomalie, lo scheduler interviene
 - l'intervento consiste nell'uccidere la transazione colpevole dell'anomalia

Locks

- L'uso del locking è molto comune nei DBMS commerciali
- Le operazioni di lettura e scrittura sono protette dalle primitive
 - **r_lock** - read lock
 - **w_lock** - write lock
 - **unlock**



Tipi di lock

- Ogni lock è applicato ad uno specifico dato del database
- I vincoli da rispettare sono
 - Ogni lettura è preceduta da un **r_lock** e seguita da un **unlock**
Il lock è **condiviso** dato che sullo stesso dato possono essere attivi più lock di questo tipo (tipo S)
 - Ogni scrittura è preceduta da un **w_lock** e seguita da un **unlock**
Il lock è **esclusivo** perché non può coesistere con altri lock sullo stesso dato (tipo X)
- Una transazione che rispetta queste regole si dice **ben formata rispetto al locking**
- In genere le richieste di lock e unlock sono inserite in modo trasparente

Richieste di lock

- Quando la richiesta di lock è concessa, la transazione **acquisisce** la corrispondente risorsa
- Con l'esecuzione di unlock la risorsa è **rilasciata**
- Quando la richiesta di lock non viene concessa la transazione richiedente viene messa in **stato di attesa**

Richiesta	Stato della risorsa		
	libero	S	X
S	OK/S	OK/S	NO/X
X	OK/X	NO/X	NO/X
Unlock	Error	OK/dipende	OK/libero

Si: la risorsa viene concessa

No: la transazione è bloccata

Tabella di compatibilità

libero se non ci sono più richieste S

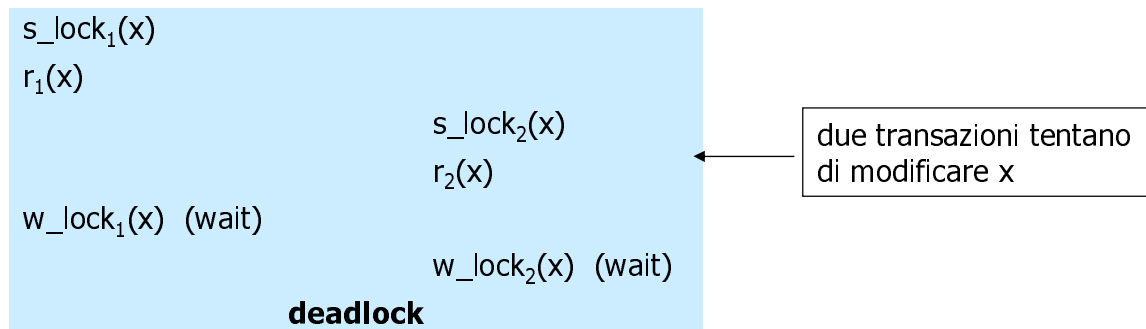
Aumentare il livello di lock

Se si vuol prima leggere e poi scrivere un oggetto

- si acquisisce prima un lock condiviso poi si incrementa a lock esclusivo

`s_lock(x) r(x) w_lock(x) w(x) unlock(x)`

- questa strategia aumenta la probabilita' di deadlock



Lock di tipo update

- $u_lock_i(x)$: lock di tipo update (tipo U)
 - fornisce il privilegio solo di lettura su x alla transazione t_i
 - può essere trasformato in un lock esclusivo (write)
 - una volta che è attivo su x non si possono aggiungere ulteriori lock attivi (le transazioni corrispondenti si mettono in wait)

Richiesta	Stato		
	S	X	U
S	OK/S	No	No
X	No	No	No
U	OK/U	No	No

Si: la risorsa viene concessa

No: la transazione è bloccata

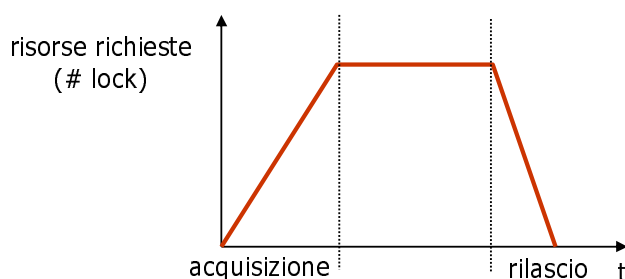
Tabella di compatibilità

Locking a due fasi

- Per garantire che le transazioni seguano uno schedule serializzabile si introduce il **two phase locking** (2PL)

Una transazione dopo aver rilasciato un lock non può acquisirne altri

- Perché 2 fasi
 - prima si acquisiscono i lock per le risorse necessarie (fase crescente)
 - poi si rilasciano i lock acquisiti (fase calante)



2PL e CSR

- Ogni schedule che rispetta il protocollo 2PL è anche serializzabile rispetto alla conflict-equivalenza **2PL \subset CSR**

- **Dimostrazione per assurdo**

Se $S \in 2PL$ e $S \notin CSR$, il grafo dei conflitti è ciclico

$$t_1 \rightarrow t_2 \rightarrow \dots, t_n \rightarrow \dots t_1$$

- esiste una risorsa **R1** su cui t_1 e t_2 operano entrambe in modo conflittuale
- t_2 può proseguire solo quando t_1 rilascia il lock sulla risorsa
- esiste una risorsa **Rn** su cui t_n e t_1 operano entrambe in modo conflittuale
- Affinché t_1 possa procedere è necessario che acquisisca il lock sulla risorsa **Rn** rilasciata da t_n
- La transazione t_1 rilascia una risorsa (**R1**) prima di acquisirne un'altra (**Rn**), ovvero lo schedule non può essere 2PL

CSR e 2PL

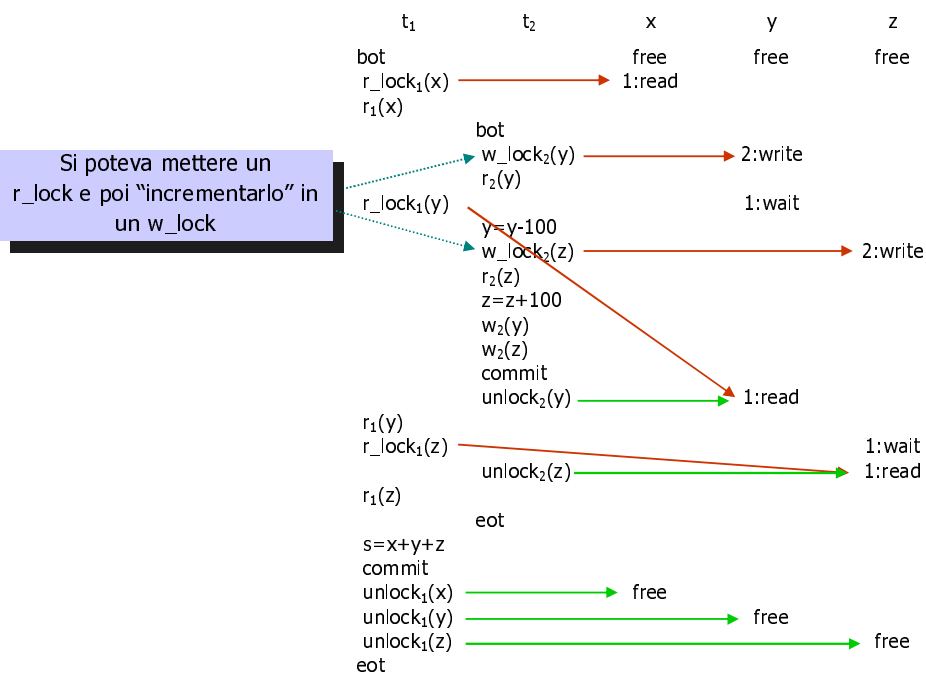
- Uno schedule che è conflict-serializzabile non è detto che sia 2PL

$S_{13} : r_1(x) \ w_1(x) \ r_2(x) \ w_2(x) \ r_3(y) \ w_1(y)$
 $t_3 \rightarrow t_1 \rightarrow t_2$

$S_{14} : r_3(y) \ r_1(x) \ w_1(x) \ w_1(y) \ r_2(x) \ w_2(x)$

- Non è 2PL perché in S_{13} la transazione t_1 deve liberare il lock su x e poi richiedere il lock su y

Modifica fantasma: soluzione





Locking a 2 fasi "stretto"

- Si rimuove l'ipotesi che tutte le transazioni vadano a buon fine (commit-proiezione)

I lock di una transazione possono essere rilasciati solo dopo aver effettuato correttamente le operazioni di commit/abort

- I lock vengono rilasciati solo al termine della transazione dopo che ogni dato è in uno stato consistente
- E' utilizzato nei DBMS commerciali
- Elimina l'anomalia della lettura sporca



Gestione dei lock

- Le operazioni di gestione dei lock hanno in genere il seguente prototipo
 - `r_lock(T,x,errcode,timeout)`
 - `w_lock(T,x,errcode,timeout)`
 - `unlock(T,x)`
- **T**: identificatore della transazione
- **x**: risorsa su cui si richiede/rilascia il lock
- **errcode**: codice che indica l'esito della richiesta (0 richiesta eseguita correttamente, diverso da 0 in caso di errore)
- **timeout**: tempo massimo che si attende per ottenere il lock sulla risorsa (se scade errcode assume il valore opportuno)

Gestione dei lock

- Se la richiesta può essere soddisfatta
 - il lock manager cambia lo stato della risorsa nelle tabelle dei lock
 - viene restituito il controllo al processo che ha effettuato la richiesta
- Se la richiesta non può essere soddisfatta
 - il processo richiedente viene inserito in una coda associata alla risorsa
 - il processo richiedente viene sospeso
 - quando la risorsa viene rilasciata si controlla se ci sono processi in attesa e nel caso si concede la risorsa al processo in testa alla coda
- Se scatta un timeout
 - la transizione fallisce e si esegue un rollback
 - si prova a richiedere nuovamente il lock

Tabelle dei lock

- Sono mantenute in memoria principale per motivi di efficienza
- Ad ogni oggetto è associato lo stato

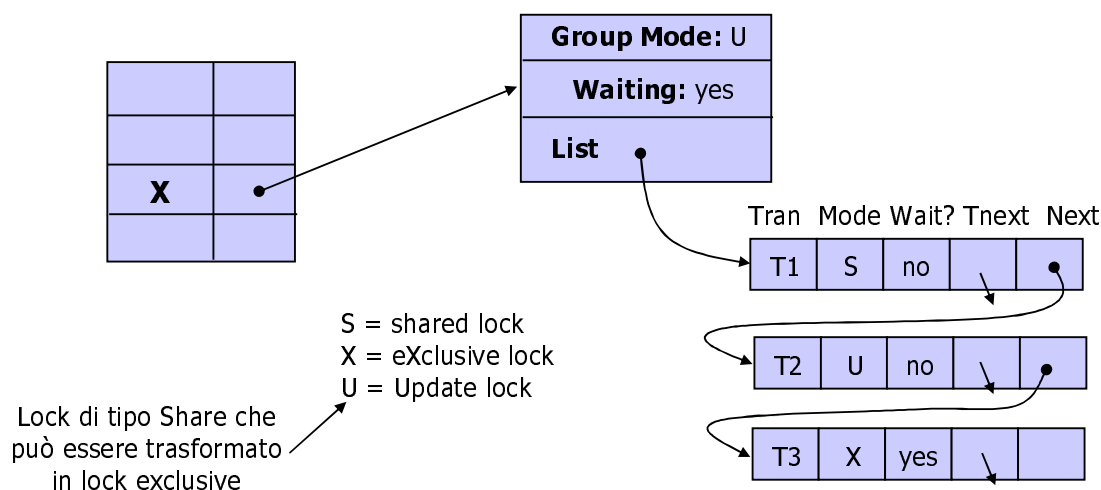
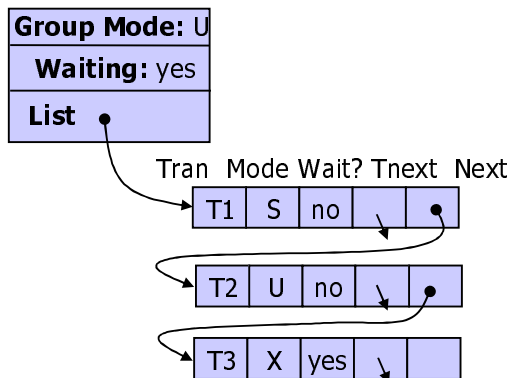
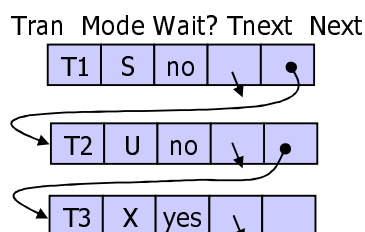


Tabelle dei lock



- Il **group mode** individua la condizione più stringente attiva sulla risorsa
 - S se sono attivi solo shared lock
 - U se c'è un update lock e forse uno o più shared lock
 - X se c'è un exclusive lock
- Il bit **Waiting** indica se c'è almeno una transazione in attesa sulla risorsa
- **List** contiene tutte le transazioni che al momento possiedono il lock o che sono in attesa di ottenerlo

Tabelle dei lock



- **Tran** è l'identificatore della transazione che detiene o attende un lock
- **Mode** indica il tipo di lock (S X U)
- **Wait?** è un flag che indica se la transazione detiene o attende un lock
- **Tnext** permette di collegare i lock relativi alla stessa transazione. Questa lista può essere usata quando la transazione esegue un commit o un abort per rilasciare tutti i lock

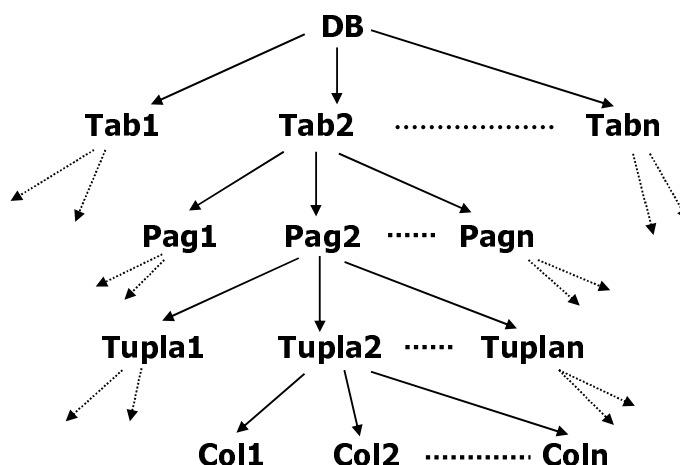
Granuralita' di locking

Quali sono gli oggetti su cui si deve porre un lock ?

- L'intera tabella
 - Per eliminare il problema dell'inserimento fantasma
 - Per un'interrogazione con operatore aggregato
- Alcuni blocchi
 - operazioni di modifica su un range (record consecutivi)
- Le tuple
 - la modifica di una tupla
- Il campo
 - La modifica di uno o pochi campi

Granuralita' di locking II

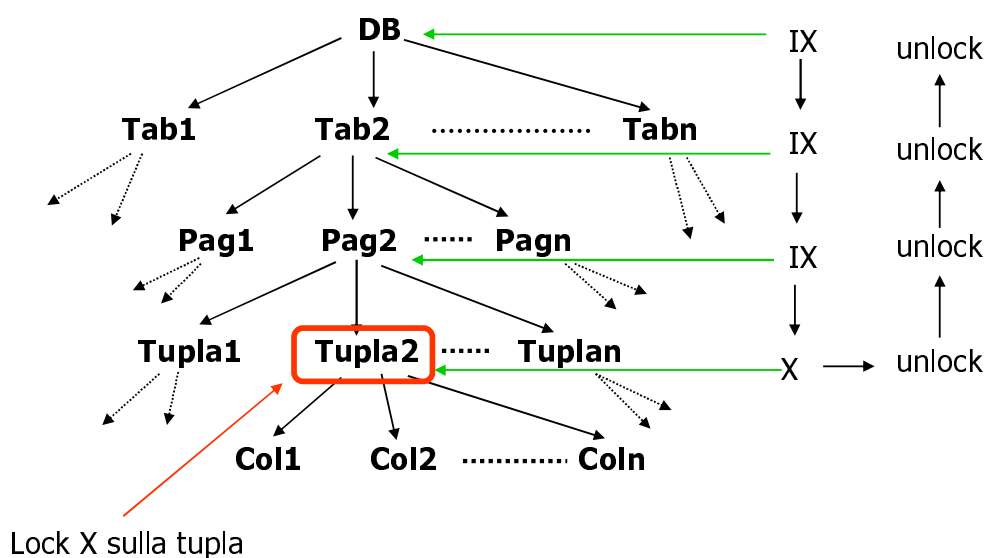
- E' possibile specificare i lock a livelli diversi (**granularità dei lock**)
 - tabelle, pagine, tuple, campi di singole tuple



Primitive per il lock gerarchico

- Si aggiungono lock specifici oltre ai lock S e X
 - **IS : Intentional Shared lock**
Esprime l'intenzione di bloccare in modo condiviso uno dei nodi discendenti del nodo corrente
 - **IX : Intentional eXclusive lock**
Esprime l'intenzione di bloccare in modo esclusivo uno dei nodi discendenti del nodo corrente
 - **SIX : Shared Intentional-eXclusive lock**
Blocca il nodo corrente ed esprime l'intenzione di bloccare in modo esclusivo uno dei nodi discendenti del nodo corrente

Lock gerarchico





Protocollo di lock gerarchico

- Si richiedono i lock a partire dalla radice scendendo lungo l'albero
- Si rilasciano i lock a partire dal nodo verso la radice
- Per richiedere un lock S o IS su un nodo si deve possedere un lock IS o IX sul padre
- Per poter richiedere un lock IX, X o SIX su un nodo si deve già possedere un lock SIX o IX sul padre
- E' definita la tabella di compatibilità per stabilire se accettare la richiesta di lock



Tabella di compatibilità

Richiesta	Stato risorsa				
	IS	IX	S	SIX	X
IS	Si	Si	Si	Si	No
IX	Si	Si	No	No	No
S	Si	No	Si	No	No
SIX	Si	No	No	No	No
X	No	No	No	No	No

Lock e B tree

Problema

- si parte dalla radice acquisendo i lock di tutti i nodi incontrati fino ai dati che vogliamo modificare/leggere
- il locking a due fasi prevede che un lock sia rilasciato solo dopo il commit
- quindi la root **rimane bloccata**
 - con gli inserimenti tutto l'albero rimane completamente bloccato (concorrenza inesistente)

Soluzione

- definire un nuovo algoritmo di lock per gli alberi che
 - rilasci i lock non appena possibile
 - garantisca comunque la serializzabilità

Lock e B tree II

Algoritmo

- **Inizialmente**, la transizione acquisisce un **lock sulla radice**
- Successivamente, può essere acquisito un lock su un qualsiasi nodo solo **se la transizione possiede un lock sul padre**
- Un lock può essere rilasciato **in qualsiasi momento**
- Un lock rilasciato **non può essere riacquisito**

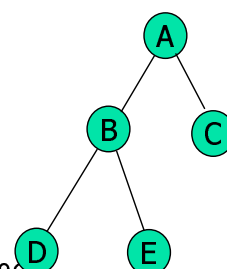
$w_lock_1(A), r_1(A)$
 $w_lock_1(C), w_1(C)$
 $unlock_1(C)$

$s_lock_2(A)$ (wait)

$w_lock_1(B), r_1(B)$
 $unlock_1(A)$

$s_lock_2(C)$
 $unlock_2(A)$
 $r_2(C)$
 $unlock_2(C)$

$w_lock_1(E)$
 $unlock_1(B)$
 $w_1(E)$
 $unlock_1(E)$



Lock e B tree III

Osservazioni

- Un lock condiviso su un nodo **n** puo' essere rilasciato
 - **non appena** si acquisisce il lock sul figlio **f**
- Un lock esclusivo su un nodo **n** puo' essere rilasciato
 - quando si acquisisce il lock sul figlio **f**
 - se **non ci sono rischi che la modifica si propaghi**
 - modifica di inserimento ed f e' pieno
 - modifica di eliminazione ed f e' al limite della minima occupazione

Serializzabilita'

- Si puo' dimostrare che usando il protocollo descritto si generano schedule conflict-serializzabili
 - Le transizioni possono essere ordinate sulla base all'ordine di acquisizione del lock sulla radice

Deadlocks

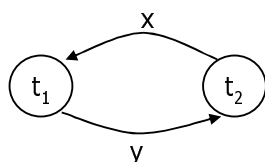
- E' possibile che si verifichi uno stallo (**deadlock**)
 - si verifica in caso di **schedule non serializzabile secondo 2PL**

$t_1 : r_1(x) w_1(y)$

$t_2 : r_2(y) w_2(x)$

- Possibile schedule con 2PL

$r_lock_1(x) r_lock_2(y) r_1(x) r_2(y) w_lock_1(y) w_lock_2(x)$



t_1 si blocca aspettando che si liberi y e bloccato da t_2

t_2 si blocca aspettando che si liberi x e bloccato da t_1



Deadlocks II

La probabilità di un deadlock

Numero transazioni: t

Numero di oggetti disponibili: n

Numero medio di oggetti bloccati per transazione: m

La probabilità che una transazione T_1 stia attendendo una risorsa bloccata da un'altra transazione T_2 è m/n

La probabilità di un deadlock causato da T_1 e T_2 (deadlock di lunghezza due) è m^2/n^2

La probabilità che esista un deadlock di lunghezza due è $O(t^2m^2/n^2)$

- Poiché normalmente $n \gg mt$, la probabilità di avere un deadlock è bassa
- Si tratta di una sottostima a causa della **località** delle transazioni
- In pratica, **i deadlock sono rari ma possibili**



Soluzioni al deadlock

Tre soluzioni

- **Timeout**
 - Le transazioni rimangono in attesa di una risorsa per un tempo prefissato
 - Una risorsa in deadlock viene comunque liberata dopo il timeout
 - Rimane difficile determinare il timeout ottimale
 - Il timeout è comunque utile (una transazione blocca una risorsa a lungo)
- **Prevenzione (deadlock prevention)**
 - Se una transazione può causare un deadlock, allora viene uccisa
 - si fa in modo da non generare mai deadlock
- **Rilevamento (deadlock detection)**
 - Si controlla periodicamente il sistema alla ricerca di situazioni di stallo e si uccide una delle transazioni coinvolte nel deadlock



Rilevamento e prevenzione attraverso grafo dei conflitti

Strategia

- Si costruisce il grafo dei conflitti
- Se è **ciclico** allora c'è un deadlock

Osservazioni

- Il grafo può essere aggiornato tutte le volte che una transazione richiede/rilascia una risorsa
 - in questo caso si tratta di **prevenzione dei deadlocks**
- Il grafo può essere aggiornato in corrispondenza dei checkpoint o a scadenze predefinite
 - in questo caso si tratta di **rilevazione dei deadlocks**
 - la transazione da uccidere può essere scelta in base ad un timestamp o scegliendo quella che ha fatto meno lavoro (accessi)



Prevenzione attraverso ordinamento

La strategia

- Si usa 2PL
- Si ordinano gli oggetti del database
 - es. in base al loro indirizzo
- Una transazione deve richiedere le risorse rispettando l'ordinamento
 - es. se una transazione richiede in sequenza R_1, R_2, \dots, R_n , deve essere verificato che $R_1 < R_2 < \dots < R_n$

Osservazioni

- In questo modo **non si verificano stalli**
- spesso non è possibile conoscere in anticipo quali risorse serviranno

T_1 ha bloccato le risorse R_1, R_2, \dots, R_n T_2 ha bloccato le risorse S_1, S_2, \dots, S_m
 T_2 è in attesa della risorsa R_i di T_1

T_1 non può essere in attesa di S_j di T_2 altrimenti $R_i \leq R_n \leq S_j \leq S_m < R_i$

Un esempio

- Si suppone che gli oggetti A,B,C,D siano ordinati alfabeticamente
- T1: r(A), w(B)
- T2: r(C), w(A)
- T3: r(B), w(C)
- T4: r(D), w(A)

l=lock
u=unlock

T1	T2	T3	T4
l(A), r(A)			
	l(A) (wait)		
		l(B), r(B)	
			l(A) (wait)
		l(C), w(C)	
		u(B), u(C)	
l(B), w(B)			
u(A), u(B)			
	l(A), l(C)		
	r(C), w(A)		
	u(C), u(A)		
			l(A), l(D)
			r(D), w(A)
			u(D), u(A)

Prevenzione attraverso timestamp

- Ad ogni transazione si assegna un **timestamp** (inizio della transazione)
- Quando una transazione T richiede una risorsa bloccata da S si confrontano i due timestamp h_t e h_s
- **Politica non interrompente**
 - Se $h_t < h_s$, T puo' attendere il rilascio della risorsa
 - Se $h_t > h_s$, T viene uccisa
- **Politica interrompente**
 - Se $h_t < h_s$, S viene forzata a rilasciare la risorsa
 - Se $h_t > h_s$, T puo' attendere il rilascio della risorsa
- Le transazioni uccise sono rilanciate con lo stesso timestamp che avevano all'inizio per evitare problemi di **starvation**

Esempi

T1: r(A), w(B); T2: r(C), w(A); T3: r(B), w(C); T4: r(D), w(A)

T1	T2	T3	T4
l(A), r(A)			
	l(A) (uccisa)		
		l(B), r(B)	
			l(A) (uccisa)
		l(C), w(C)	
		u(B), u(C)	
l(B), w(B)			
u(A), u(B)			
			l(A), l(D)
	l(A) (wait)		
			r(D), w(A)
			u(D), u(A)
	l(A), l(C)		
	r(C), w(A)		
	u(C), u(A)		

Politica non interrompente

T1	T2	T3	T4
l(A), r(A)			
	l(A) (wait)		
		l(B), r(B)	
			l(A) (wait)
l(B), w(B)		(uccisa)	
u(A), u(B)			
	l(A), l(C)		
	r(C), w(A)		
	u(C), u(A)		
			l(A), l(D)
			r(D), w(A)
			u(D), u(A)
		l(B), r(B)	
		l(C), w(C)	
		u(B), u(C)	

Politica interrompente

Prevenzione attraverso timestamp II

Perche' funziona ?

- Si supponga che esiste un ciclo $T_1, T_2, \dots, T_n, T_1$ nel grafo dei conflitti
- Con la politica non interrompente le transizioni piu' vecchie aspettano quelle piu' giovani: $h_1 < h_2 < \dots < h_n < h_1$
- Con la politica interrompente le transizioni piu' giovani aspettano quelle piu' vecchie: $h_1 > h_2 > \dots > h_n > h_1$

Vantaggi e svantaggi (assumendo che molti lock siano acquisiti all'inizio)

- Politica non interrompente
 - Si uccide le transazioni che hanno fatto **meno lavoro**
- Politica interrompente
 - Si uccide **meno transazioni**



Un paragone fra i meccanismi per risolvere i deadlock

Il grafo dei conflitti

- E' il metodo **comunemente usato** dai DBMS commerciali
- riduce al minimo i roll back
- e' dispendioso e complicato: occorre mantenere il grafo dei conflitti

Prevenzione attraverso ordinamento

- richiede di conoscere in anticipo gli oggetti da bloccare

Prevenzione attraverso timestamp

- puo' succedere che transazioni che non causeranno conflitti vengano uccise e devano essere rilanciate
- ha un ruolo piu' importante in un ambiente distribuito, dove il mantenimento del grafo dei conflitti e' piu' gravoso



Riassumendo

- La **teoria suggerisce** le proprieta' che uno schedule deve avere perche' **non possieda anomalie**: deve essere serializzabile
- La view-serializzabilita' fornisce **schedule piu' efficienti**, ma implicerebbe uno **scheduler inefficiente**
- 2PL permette di realizzare uno **scheduler efficiente**
- Con **i lock si implementa 2PL**, quindi si garantiscono schedule senza anomalie (**isolamento delle transazioni**)
- L'uso dei lock puo' introdurre deadlock (si verificano quando lo schedule e' non serializzabile secondo 2PL)
- Occorre adottare **strategie di prevenzione/riconoscimento** dei deadlock



Altri approcci al controllo della concorrenza

Il meccanismo dei lock assume

- se non si controlla le transazioni si verificano anomalie

Gli **approcci ottimistici** assumono

- tutto andrà bene, altrimenti interveniamo
- l'intervento consiste nell'uccidere la transazione colpevole dell'anomalia
- **basati su timestamp**
 - si verifica che lo schedule sia equivalente ad uno seriale usando dei timestamp per le transazioni e le operazioni di lettura e scrittura
- **basati su validazione**
 - si esaminano i timestamp prima che la transazione faccia il commit



Controllo della concorrenza basato su timestamp

Si assegnano

- per ogni transazione T un timestamp $TS(T)$
- per ogni oggetto X
 1. Il maggiore timestamp fra quelli delle transazioni che hanno letto X :
 $RT(X)$ (**tempo di lettura**)
 2. Il maggiore timestamp fra quelli delle transazioni che hanno scritto X :
 $WT(X)$ (**tempo di scrittura**)
 3. Un booleano che è vero se la transazione in (2) ha fatto commit:
 $c(X)$ (**bit di commit**)

L'idea di base

- $TS(T)$, $RT(X)$, $WT(X)$ servono a riconoscere gli schedule non serializzabili
- $c(X)$ serve a riconoscere le letture sporche

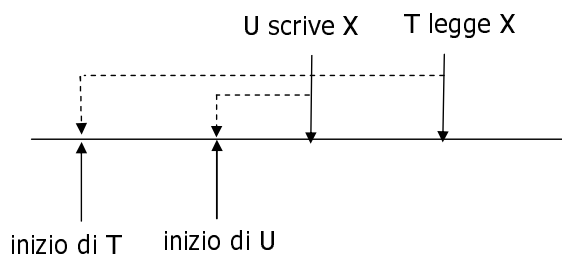


Comportamenti fisicamente non realizzabili

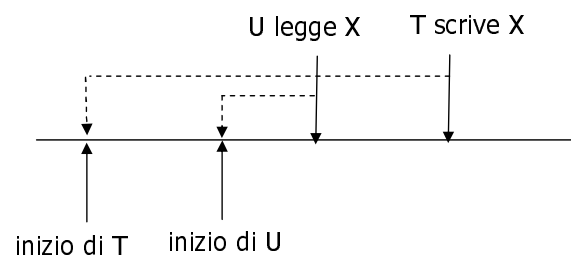
I comportamenti **fisicamente non realizzabili**

- sono gli schedule che non si comportano come uno schedule seriale
- lo schedule seriale e' quello in cui le transazioni vengono eseguite all'istante $TS(T)$

Lettura in ritardo

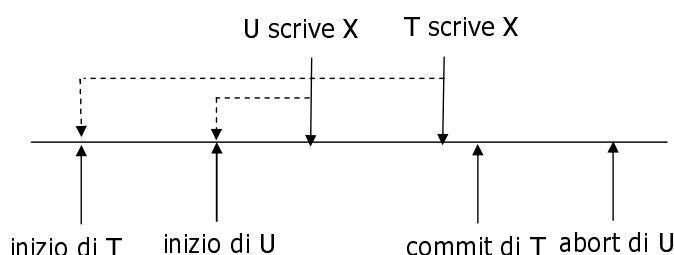
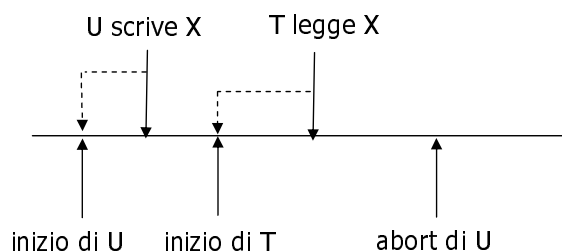


Scrittura in ritardo



Lecture sporche

Sono problemi dovuti alla presenza di ABORT in alcune transazioni



Una scrittura seguita da un'altra scrittura potrebbe essere non eseguita. Se la seconda scrittura viene abortita questo genera una anomalia.



L'algoritmo

Se lo scheduler riceve una richiesta di lettura $r_T(X)$

- se $TS(T) \geq WT(X)$ la lettura e' fisicamente realizzabile
 - se $c(X)$ e' vero, esegui la richiesta. Aggiorna $RT(X)$
 - se $c(X)$ e' falso, ritarda T fino a quando la transazione che ha scritto X abortisce o fa commit
- se $TS(T) < WT(X)$ la lettura non e' fisicamente realizzabile
 - Abortisci e rilancia T



L'algoritmo II

Se lo scheduler riceve una richiesta di scrittura $w_T(X)$

- se $TS(T) \geq RT(X)$ e $TS(T) \geq WT(X)$ la scrittura e' fisicamente realizzabile
 - esegui la scrittura
 - aggiorna $WT(X)$ e assegna $c(X) = \text{falso}$
- se $TS(T) \geq RT(X)$ e $TS(T) < WT(X)$ la scrittura e' fisicamente realizzabile ma X contiene un nuovo valore scritto da una transazione U
 - se $c(X)$ e' vero , non fare niente
 - se $c(X)$ e' falso, allora occorre uccidere la transazione
- se $TS(T) < RT(X)$ la scrittura non e' fisicamente realizzabile
 - Abortisci e rilancia T

L'algoritmo III

Se lo scheduler riceve una richiesta commit da una transazione T

- per tutti gli oggetti X modificati da T
 - assegna $c(X)=true$
 - se ci sono transazioni in attesa, attivale

Se lo scheduler riceve una richiesta abort da una transazione T

- le transazioni in attesa devono ripetere il loro tentativo di scrittura e lettura e vedere cosa succede

Un esempio

T ₁	T ₂	T ₃	A	B	C
200	150	175	RT=0 WT=0 C=true	RT=0 WT=0 C=true	RT=0 WT=0 C=true
r(B)				RT=200	
	r(A)		RT=150		
		r(C)			RT=175
w(B)				WT=200 C=false	
w(A)			WT=200 C=false		
Commit			C=true	C=true	
	W(C) -> SA				
		W(A)			
		Commit			

oggetti

Abort forzato dallo scheduler

Si va avanti senza fare niente

Confronto fra locking e controllo basato su timestamp



Pro e contro

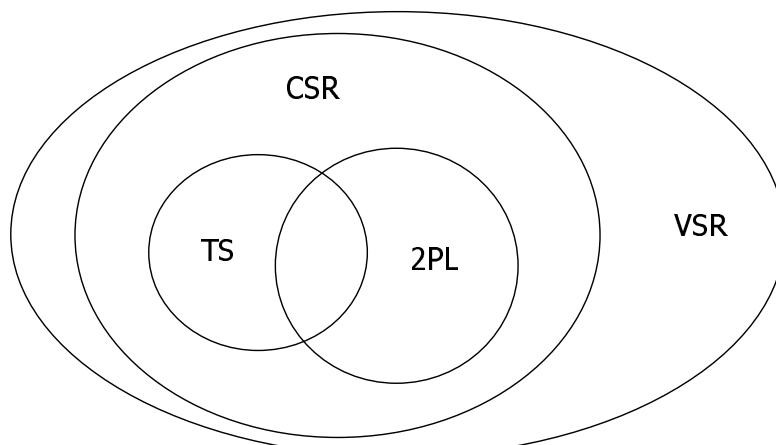
- Il controllo basato su timestamp
 - e' migliore nel caso la maggior parte delle transazioni sia in **sola lettura**
 - se ci sono molte transazioni in scrittura provoca **numerosi rollback**
- Il controllo basato su locking
 - spesso **ritarda** senza motivo l'esecuzione delle transazioni
 - e' adottato dalla **maggior parte dei DBMS** commerciali
- Una soluzione **intermedia** in sistemi commerciali
 - dividere le transazioni fra quelle sola lettura e lettura e scrittura
 - quelle in sola lettura usano il controllo basato su timestamp, le altre il controllo basato su locking

Confronto fra locking e controllo basato su timestamp II



Da un punto di vista teorico

- Il controllo basato su concorrenza genera una classe di schedule TS contenuti in CSR
- 2PL e TS hanno un'intersezione, ma non vale nessuna inclusione





Controllo di concorrenza basato su validazione

Controllo **basato su validazione**

- E' controllo di concorrenza **ottimistico**
- Lo scheduler mantiene una lista delle operazioni di ciascuna transazione
- Subito prima di procedere al commit verifica che non esistano comportamenti non realizzabili fisicamente

L'algoritmo prevede **tre fasi** per ogni transazione

- **Lettura**: Le transazioni leggono dal database e scrivono nel loro spazio di lavoro i loro risultati
- **Validazione**: Lo scheduler valida le transazioni comparando le operazioni fatte con quelle delle altre transazioni. Se la validazione fallisce la transazione viene abortita.
- **Scrittura**: la transazione scrive i dati sul database



La validazione

Per la validazione lo scheduler deve memorizzare **tre insiemi**

- **START**: l'insieme delle transazioni che sono iniziate, ma non hanno terminato la validazione
- **VAL**: l'insieme delle transazioni che sono state validate ma che non hanno concluso la scrittura
- **FIN**: l'insieme delle transazioni che hanno finito la scrittura

e **tre timestamp** per ogni transazione T

- **START(T)**: il momento in cui T e' iniziata
- **VAL(T)**: il momento in cui T e' stato validato
(**indica anche l'ordine seriale assunto**)
- **FIN(T)**: il momento in cui T ha finito

I tre valori sono necessari fino a quando esiste U per la quale $START(U) \leq FIN(T)$

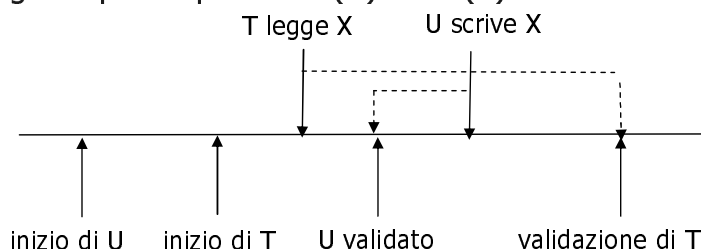
La validazione II

Per la validazione di T si controlla che

- $RS(T) \cap WS(U) = \emptyset$, per ogni U per il quale $FIN(U) > START(T)$
- $WS(T) \cap WS(U) = \emptyset$, per ogni U per il quale $FIN(U) > VAL(T)$

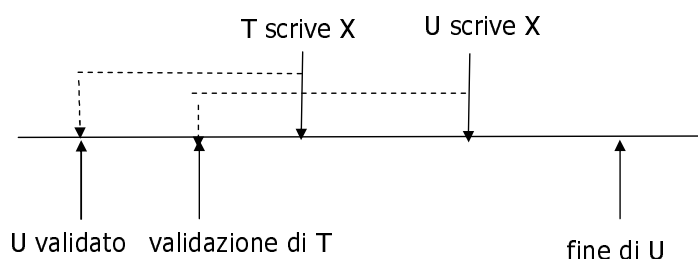
Lettura/scrittura

T ha letto il valore di X prima che U finisse di scrivere



Scrittura/scrittura

T potrebbe scrivere prima di U



Un esempio

Validazione di U

- **ok**: nessun altra transazione validata

Validazione di T

- **ok**: $WS(T) \cap WS(U) = \emptyset$, $RS(T) \cap WS(U) = \emptyset$

Validazione di V

- **ok**: $RS(V) \cap WS(U) = \emptyset$
 $WS(V) \cap WS(T) = \emptyset$, $RS(V) \cap WS(T) = \emptyset$

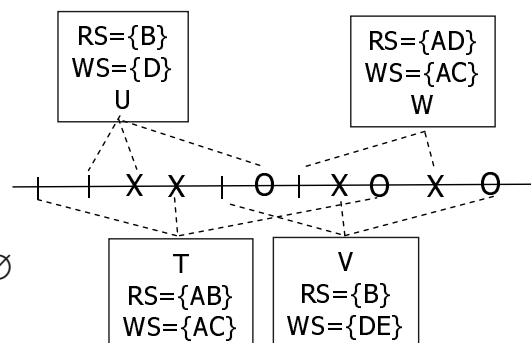
Validazione di W

- **no**: $RS(W) \cap WS(V) = \{D\}$, $WS(W) \cap WS(V) = \emptyset$
 $RS(W) \cap WS(T) = \{A\}$

I = START

X=VAL

O=FIN





Confronto fra i metodi di controllo della concorrenza

Spazio occupato dai i tre metodi è simile

- **Locking:**
 - la tabella di lock e' proporzionale al numero di richieste di lock in corso
- **Timestamp:**
 - 2 timestamp per ogni oggetto acceduto dalle transazioni e 1 timestamp per ogni transazione
 - Non è sufficiente tenere conto solo delle transazioni in corso, ma servono anche quelle recenti
- **Validazione:**
 - spazio per gli insiemi RS, WS di oggetti acceduti dalle transazioni e 3 timestamp per ogni transazione
 - pero' deve tenersi in un area locale le modifiche fatte da ogni transazione



Confronto fra i metodi di controllo della concorrenza II

Osservazioni

- Il locking ritarda le transazioni, ma evita i rollback
- Il timestamp e la validazione sono migliori in caso di bassa interferenza fra le transazioni
- quando si deve uccidere una transazione
 - il timestamp identifica il problema prima
 - la validazione aspetta il completamento delle transazioni
- nel caso di transazioni che possono causare un abort
 - la validazione permette di continuare l'esecuzione
 - il timestamp la blocca