Transazioni - Concorrenza e ripristino

Transazioni - Concetti di base

- Per mantenere le informazioni consistenti è necessario controllare opportunamente le sequenze di accessi e aggiornamenti ai dati.
- Tali sequenze sono dette transazioni.
- Ogni transazione è eseguita o completamente (cioè effettua il commit), oppure per nulla (cioè effettua l'abort) se si verifica un qualche errore (hardware o software) durante l'esecuzione.
- Necessità di garantire che le transazioni eseguite concorrentemente si comportino come se fossero state eseguite in sequenza.
- Necessità di tecniche per ripristinare uno stato corretto della base di dati a fronte di malfunzionamenti di sistema.

Transazioni - Concetti di base

- Gli utenti interagiscono con la base di dati attraverso programmi applicativi ai quali viene dato il nome di transazioni.
- Il termine transazione viene utilizzato con diversi significati:
  1. la richiesta o il messaggio di input che avvia l'operazione desiderata dall'utente;
  2. tutti gli effetti dell'esecuzione dell'operazione che l'utente ha richiesto;
  3. tutti i programmi che eseguono l'operazione richiesta dall'utente.
- Tale diversità dipende dal fatto che l'uso di transazioni coinvolge diverse tipologie di utenti.

Transazioni - Concetti di base

- Nel seguito, si adotterà la seconda definizione:

  Una transazione è una sequenza di operazioni di lettura e scrittura; essa costituisce l'effetto dell'esecuzione di programmi che effettuano le funzioni desiderate dagli utenti.
- L'insieme di operazioni che costituiscono una transazione deve soddisfare alcune proprietà, note come ACIDity property (da Atomicità, Consistenza, Isolamento, Durabilità).

Transazioni - Atomicità

- è detta anche proprietà *tutto-o-niente*
- tutte le operazioni di una transazione devono essere trattate come una singola unità: o vengono eseguite tutte, oppure non ne viene eseguita alcuna
- l'atomicità delle transazioni è assicurata dal sottosistema di ripristino (recovery)

Transazioni - Consistenza

- una transazione deve agire sulla base di dati in modo corretto
- se viene eseguita su una base di dati in assenza di altre transazioni, la transazione trasforma la base di dati da uno stato consistente (cioè che riflette lo stato reale del mondo che la base di dati deve modellare) ad un altro stato ancora consistente
- è l'esecuzione di un insieme di transazioni corrette e concorrenti deve a sua volta mantenere consistente la base di dati
- il sottosistema di controllo della concorrenza (concurrency control) sincronizza le transazioni concorrenti in modo da assicurare esecuzioni concorrenti libere da interferenze

Transazioni - Isolamento

- ogni transazione deve sempre osservare una base di dati consistente, cioè, non può leggere risultati intermedi di altre transazioni
- la proprietà di isolamento è assicurata dal sottosistema di controllo della concorrenza che isola gli effetti di una transazione fino alla terminazione della stessa

Transazioni - Durabilità (Persistenza)

- i risultati di una transazione terminata con successo devono essere resi permanenti nella base di dati nonostante possibili malfunzionamenti del sistema
- la persistenza è assicurata dal sottosistema di ripristino
- tale sottosistema può inoltre fornire misure addizionali, quali back-up su supporti diversi e journaling delle transazioni, per garantire la durabilità anche a fronte di guasti ai dispositivi di memorizzazione

Transazioni - Esempio

- Si consideri una transazione bancaria di addebito sul conto corrente del signor Rossi
- è atomica se effettua entrambe le procedure di prelevamento ed aggiornamento del conto corrente
- è consistente se l'importo prelevato è lo stesso addebitato sul conto
- è isolata se il programma transazione non vede gli effetti di altri programmi che leggono e scrivono sul conto corrente del signor Rossi che non hanno ancora effettuato il commit
- è duratura (persistente) se, terminata la transazione, il saldo del conto riflette il prelievo effettuato

Transazioni - Proprietà ACID

- Le proprietà ACID vengono assicurate utilizzando due insiemi distinti di algoritmi o protocolli, che assicurano:
- 1. **atomicità dell'esecuzione** permette di mantenere la consistenza globale della base di dati e quindi assicurare la proprietà di consistenza delle transazioni (anche concorrenti)

  protocolli di controllo della concorrenza
- 2. **atomicità del fallimento** assicura l'atomicità, l'isolamento e la persistenza

  protocolli di ripristino

Transazioni - Transazioni flat

- tipo di transazione più semplice
- usate nei DBMS disponibili in commercio
- tecniche di implementazione e limitazioni sono ben note
- un solo livello di controllo a cui appartengono tutte le istruzioni eseguite
- transazioni semplici, di breve durata, senza alcuna struttura gerarchica

Transazioni - Transazioni flat

- Tutte le istruzioni eseguite devono essere contenute tra le istruzioni *BeginWork* e *CommitWork*
  - l'istruzione *BeginWork* dichiara l'inizio di una transazione
  - l'istruzione *CommitWork* è invocata per indicare che il sistema ha raggiunto un nuovo stato consistente
- la transazione può terminare la propria esecuzione con successo (commit) e rendere definitivi i cambiamenti prodotti sulla base di dati dalle istruzioni eseguite tra *BeginWork* e *CommitWork*, oppure sarà disfatta (cioè i suoi effetti saranno annullati) e tutti gli aggiornamenti eseguiti andranno persi (abort)
- in questo caso, si dice che viene eseguito ilrollback della transazione

Transazioni - Transazioni flat

- una transazione T i è un insieme di operazioni sui dati su cui è definito un ordinamento parziale < i , che specifica l'ordine con cui le operazioni vengono eseguite
- una transazione può, in modo esclusivo, effettuare abort oppure commit
- l'istruzione di abort o di commit deve essere l'ultima istruzione effettuata dalla transazione
- fissato un dato x, le operazioni di scrittura e lettura su tale dato sono ordinabili tramite la relazione di ordinamento < i [cioè una transazione che vuole leggere (scrivere) un dato x deve attendere che qualsiasi altra operazione di scrittura (lettura) termini, prima di poter eseguire la lettura (scrittura) richiesta]

Transazioni - Transazioni flat: supporto in SQL

- i vari DBMS forniscono specifiche istruzioni SQL per supportare l'uso di transazioni
- tali istruzioni sono invocate all'interno di programmi applicativi e consentono al programmatore di identificare l'inizio e la fine di una transazione
- l'istruzione

  exec sql connect:user_name identified by :user_pws (cioè l'istruzione di connessione) avvia la transazione, cioè implementa la primitiva BeginWork
- l'istruzione exec sql commit work esegue il commit di una transazione
- l'istruzione exec sql rollback esegue il rollback di una transazione

Transazioni - Transazioni flat: limitazioni

- una transazione flat non permette di effettuare il commit oppure l'abort di parti distinte di essa nè di eseguire modifiche sulla base di dati in molteplici passi, effettuando molteplici commit
- esempio di problema nell'uso di transazioni flat
- pianificazione di un viaggio Pittsburg-Finale Ligure
  - BeginWork
  - prenota il volo da Pittsburgh a Londra
  - prenota il volo da Londra a Milano, stesso giorno
  - prenota il volo da Milano a Genova, stesso giorno
- proposte di estensioni: transazioni concatenate, transazioni annidate, transazioni multilivello, ...
- trade-off potere espressivo - semplicità

Controllo della concorrenza

- Scopo: garantire l'integrità della base di dati in presenza di accessi concorrenti da parte di più utenti
- necessità di sincronizzare le transazioni eseguite concorrentemente
- Esempio di problema:
  - base di dati dei conti dei clienti di una banca
  - sig. Rossi titolare di due conti
  - un conto corrente (intestato anche alla sig.ra Rossi), saldo Lit. 100000
  - un libretto di risparmio, saldo Lit. 1000000
  - con la transazione T1 il sig. Rossi trasferisce Lit. 150000 dal libretto di risparmio al conto corrente
  - con la transazione T2 la sig.ra Rossi deposita Lit. 500000 sul conto corrente

Controllo della concorrenza

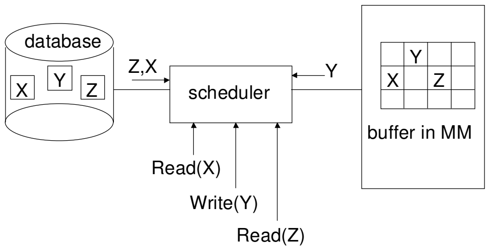
T1	T2
Read lr;	
lr = lr - 150000;	
	Read c/c;
Write lr;	
	c/c = c/c + 500000;
Read c/c;	
	Write c/c;
	Commit T2
c/c = c/c + 150000;	
Write c/c;	
Commit T1	

- la somma depositata da T2 è persa (lost update)
- non si ha l'effetto voluto sulla base di dati

Controllo della concorrenza - architettura di riferimento

- Transazioni come sequenze di operazioni di input/output (Read/Write) su oggetti astratti x, y, z
- ogni operazione di input/output legge blocchi di memoria secondaria in pagine di buffer o scrive pagine di buffer in blocchi di memoria secondaria
- per semplicità si assume mapping uno-a-uno tra blocchi e pagine

Controllo della concorrenza - architettura di riferimento



Controllo della concorrenza - anomalie dovute a esecuzione concorrente

- **Anomalia 1: perdita di aggiornamenti**
- consideriamo due transazioni identiche
  - T1: r(x), x = x+1, w(x)
  - T2: r(x), x = x +1, w(x)
- se inizialmente x = 2, dopo esecuzione seriale x = 4
- consideriamo esecuzione concorrente

T1	T2
r(x)	
x = x + 1	
	r(x)
	x = x + 1
w(x)	
commit	
	w(x)
	commit

- uno degli aggiornamenti è perso, il valore finale è x = 3

Controllo della concorrenza - anomalie dovute a esecuzione concorrente

- **Anomalia 2: letture sporche**
- consideriamo le stesse transazioni e la seguente esecuzione (si noti che T1 fallisce)

T1	T2
r(x)	
x = x + 1	
w(x)	
	r(x)
	x = x + 1
abort	
	w(x)
	commit

- il problema è che T2 legge uno stato intermedio (dirty read)

Controllo della concorrenza - anomalie dovute a esecuzione concorrente

- **Anomalia 3: letture inconsistenti**
- T1 legge due volte il dato x

T1	T2
r(x)	
	r(x)
	x = x + 1
	w(x)
	commit
r(x)	
commit	

- T1 legge valori diversi per x

Controllo della concorrenza - anomalie dovute a esecuzione concorrente

- **Anomalia 4: aggiornamenti fantasma**
- si assuma il vincolo di integrità x + y + z = 1000

T1	T2
r(x)	
	r(y)
r(y)	
	y = y - 100
	r(z)
	z = z + 100
	w(y)
	w(z)
	commit
r(z)	
s = x + y + z	
commit	

- alla fine s vale 1100: il vincolo di integrità è violato
- T1 vede un ghost update

Controllo della concorrenza - serializzabilità

- l'esecuzione concorrente di più transazioni genera un'alternanza di computazioni da parte delle varie transazioni, detta interleaving
- la sequenza di operazioni generata viene definita schedule
- l'interleaving tra le transazioni T1 e T2 nell'esempio produce uno stato della base di dati scorretto; si sarebbe ottenuto uno stato corretto se ciascuna transazione fosse stata eseguita da sola o se le due transazioni fossero state eseguite l'una dopo l'altra, consecutivamente
- uno schedule seriale è uno schedule in cui le transazioni vengono eseguite in sequenza

Controllo della concorrenza - serializzabilità

- Esempio di schedule seriale - schedule 1

T1	T2
Read lr;	
lr = lr - 150000;	
Write lr;	
Read c/c;	
c/c = c/c + 150000;	
Write c/c;	
Commit T1	
	Read c/c;
	c/c = c/c + 500000;
	Write c/c;
	Commit T2

- date n transazioni esistono n! schedule seriali

Controllo della concorrenza - serializzabilità

- Obiettivo: rifiutare gli schedule che causano le anomalie
- scheduler: un sistema che accetta o rifiuta le operazioni richieste da una transazione
- schedule serializzabile: uno schedule che produce lo stesso risultato di uno schedule seriale delle stesse transazioni
- richiede una nozione di equivalenza tra schedule
- varie nozioni: equivalenza rispetto ai conflitti, rispetto alle viste
- nota: uno scheduler permette di identificare classi più o meno vaste di schedule accettabili al costo di testare l'equivalenza

Controllo della concorrenza - serializzabilità

Serializzabilità rispetto ai conflitti (conflict-serializability)

- due azioni di due transazioni diverse sono in conflitto se:
  - coinvolgono lo stesso dato e
  - almeno una di esse è una write
- due azioni non in conflitto possono essere eseguite in qualsiasi ordine
- due schedule S1 e S2 sono equivalenti rispetto ai conflitti se possono essere trasformati uno nell'altro modificando l'ordine di azioni non in conflitto
- uno schedule è serializzabile rispetto ai conflitti se è equivalente (rispetto ai conflitti) ad uno schedule seriale

Controllo della concorrenza - serializzabilità

Serializzabilità rispetto ai conflitti (conflict-serializability)

- Schedule 2

T1	T2
Read lr;	
	Read c/c;
lr = lr - 150000;	
	c/c = c/c + 500000;
Write lr;	
	Write c/c;
	Commit T2 .
Read c/c;	
c/c = c/c + 150000;	
Write c/c;	
Commit T1	

- è equivalente allo schedule seriale T2 , T1

Controllo della concorrenza - serializzabilità

Serializzabilità rispetto ai conflitti (conflict-serializability)

- Schedule 3

T1	T2
Read lr;	
lr = lr - 150000;	
	Read lr;
	lr = lr - 100000;
	Write lr;
	Read c/c;
Write lr;	
Read c/c;	
c/c = c/c + 150000;	
Write c/c;	
Commit T1	
	c/c = c/c + 100000;
	Write c/c;
	Commit T2

- non è serializzabile

Controllo della concorrenza - serializzabilità

Serializzabilità rispetto ai conflitti (conflict-serializability)

- le transazioni T1 e T2 trasferiscono rispettivamente 150000 e 100000 dal libretto di risparmio al conto corrente
- se lo schedule fosse corretto, la somma esistente sul conto corrente sommata alla somma esistente sul libretto di risparmio sarebbe invariata
- si supponga che prima di iniziare l'esecuzione, lr = 400000 e c/c= 200000
- al termine dell'esecuzione, lr = 250000 mentre c/c=300000
- la somma delle due quantità non si è mantenuta costante
- vi sono dei lost update: il decremento di lr di T2 e l'incremento di c/c di T1

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto ai conflitti

- Dato uno schedule S a cui partecipano le transazioni T1, …, Tn , si costruisce un grafo diretto, detto grafo di precedenza G_S = (V, E), definito come segue:
- 1. L'insieme dei vertici V consiste in tutte le transazioni che partecipano allo schedule S
- 2. L'insieme degli archi E consiste in tutti gli archi Ti → Tj , i, j = 1, …, n, tali che Ti e Tj verificano una delle due seguenti condizioni:
  - (a) Ti esegue write(Q) prima che Tj esegua read(Q)
  - (b) Ti esegue read(Q) prima che Tj esegua write(Q)
  - (c) Ti esegue write(Q) prima che Tj esegua write(Q)

Controllo della concorrenza - serializzabilità

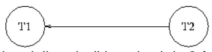
Test di serializzabilità rispetto ai conflitti

- Se esiste un arco da Ti a Tj allora nello schedule seriale S_0 , equivalente a S, Ti deve essere eseguito prima di T j
- lo schedule S è serializzabile sse il grafo G_S è aciclico
- l'ordine di serializzabilità si ottiene con un ordinamento topologico:
  - sia L una lista vuota
  - 1. sia Ti un nodo in G_S senza archi entranti (esiste sicuramente perchè il grafo è aciclico)
  - 2. si appende Ti alla lista L
  - 3. si rimuove Ti dal grafo e si rimuovono tutti gli archi uscenti da Ti
  - si rieseguono i passi 1, 2, 3 fino a che il grafo è vuoto

Controllo della concorrenza - serializzabilità

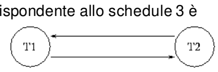
Test di serializzabilità rispetto ai conflitti

- il costo dell'algoritmo è nell'ordine di n^2 , dove n è il numero di nodi (e cioè il numero di transazioni) del grafo
- esempi:
  - il grafo corrispondente allo schedule 2 è



il grafo non ha cicli, quindi lo schedule 2 è serializzabile

  - il grafo corrispondente allo schedule 3 è



il grafo ha un ciclo, quindi lo schedule 3 non è serializzabile

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto ai conflitti

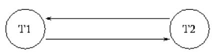
- Esempio:
- Schedule 4

T1	T2
Read lr;	
	Write lr;
Write lr;	
Commit T1	
	Commit T2

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto ai conflitti

- lo schedule 4 non è serializzabile rispetto ai conflitti
- non è infatti equivalente nè allo schedule seriale che esegue prima T1 e poi T2 né allo schedule seriale che esegue prima T2 e poi T1
  - T1 legge lr prima che sia scritto da T2 ⇒ T1 deve precedere T2 nello schedule seriale equivalente
  - T1 esegue l'ultima operazione di scrittura ⇒ T1 deve seguireT2 nello schedule seriale equivalente



Controllo della concorrenza - serializzabilità

Serializzabilità rispetto ai conflitti

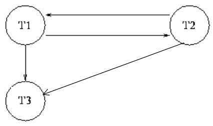
- la nozione di serializzabilità rispetto ai conflitti è molto restrittiva
- esempio: Schedule 5

T1	T2	T3
Read lr;		
	Write lr;	
Write lr;		
		Write lr;
Commit T1		
	Commit T2	
		Commit T3

Controllo della concorrenza - serializzabilità

Serializzabilità rispetto ai conflitti

- il grafo corrispondente è



- il grafo è ciclico, quindi lo schedule non è serializzabile rispetto ai conflitti
- in realtà lo schedule produce gli stessi effetti dello schedule seriale T1, T2, T3
- i valori di lr scritti da T1 e T2 non sono lettida T3 (useless writes)

Controllo della concorrenza - serializzabilità

Equivalenza rispetto alle viste (view-equivalence)

- nota: un aspetto chiave dello schedule è il fatto che viene effettuata una scrittura non preceduta da lettura del dato
- questo può accedere in transazioni reali (es. UPDATE Impiegati SET Stipendio = 1000)
- si introduce quindi la nozione di serializzabilità rispetto alle viste
- la serializzabilità rispetto ai conflitti implica la serializzabilità rispetto alle viste

Controllo della concorrenza - serializzabilità

Equivalenza rispetto alle viste (view-equivalence)

- due schedule S1 e S2 sono equivalenti rispetto alle viste sse
- sono costituiti dallo stesso insieme di transazioni
- per ogni dato Q, se in S1 la transazione Ti esegue read(Q) e il valore letto era stato scritto da una write(Q) eseguita dalla transazione Tk, lo stesso avviene in S2
- per ogni dato Q, se in S1 la transazione Ti è l'ultima ad effettuare l'operazione write(Q), lo stesso avviene in S2

Controllo della concorrenza - serializzabilità

Equivalenza rispetto alle viste (view-equivalence)

- la prima condizione assicura che lo stesso insieme di transazioni partecipi ad entrambi gli schedule
- la seconda condizione assicura che ogni transazione riceva gli stessi valori in ingresso in entrambi gli schedule (e quindi esegua la stessa computazione)
- la terza condizione, insieme alla seconda, assicura che entrambi gli schedule producano lo stesso stato finale della base di dati

Controllo della concorrenza - serializzabilità

Equivalenza rispetto alle viste (view-equivalence)

- formulazione alternativa:
- si supponga che una transazione Tb scriva i valori iniziali per ogni elemento della base di dati e una transazione Tf legga i valori finali per ogni elemento della base di dati
- per ogni azione di lettura read(A) la sorgente di tale operazione è l'azione di scrittura immediatamente precedente sul dato A

Controllo della concorrenza - serializzabilità

Equivalenza rispetto alle viste (view-equivalence)

- due schedule S1 e S2 sono equivalenti rispetto alle viste se per ogni azione di lettura in uno dei due schedule la sorgente dell'azione la stessa della corrispondente azione di scrittura nell'altro schedule
- uno schedule è serializzabile rispetto alle viste se è equivalente rispetto alle viste a uno schedule seriale

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- Idea: usare archi etichettati, dove archi con la stessa etichetta rappresentano scelte alternative $\Rightarrow$ poligrafo
- Sia {T0 , …, Tn} l'insieme di transazioni considerate in uno schedule S
- Siano Tb e Tf due transazioni dummy, tali che Tb esegue un'operazione di write(Q) per ogni dato Q riferito in S e Tf esegue un'operazione di read(Q) per ogni dato Q riferito in S
- Si costruisce un nuovo schedule S' ottenuto da S eseguendo Tb come prima operazione ed eseguendo Tf come ultima operazione

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- Il grafo di precedenza con etichette per S' si costruisce come segue:
  - 1. Si introduce un nodo per ogni transazione in S' e un arco Ti $\rightarrow^0$ Tj per ogni coppia di transazioni Ti e Tj tali che Tj legge il valore di un dato Q scritto da Ti
  - 2. Si rimuovono tutti gli archi incidenti su nodi che rappresentano transazioni useless

    [una transazione è useless se non esiste alcun cammino, nel grafo di precedenza, che connette Ti a Tf]

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- 3. Per ogni dato Q tale che:
  - Tj legge il valore di Q scritto da Ti
  - Tk esegue write(Q) e Tk $\neq$ Tb :
- a. se Ti = Tb e Tj $\neq$ Tf, si inserisce l'arco Tj $\rightarrow^0$ Tk
- b. se Ti $\neq$ Tb e Tj = Tf, si inserisce l'arco Tk $\rightarrow^0$ Ti
- c. se Ti $\neq$ Tb e Tj $\neq$ Tf, si inserisce la coppia di archi

  Tj $\rightarrow^p$ Tk e Tk $\rightarrow^p$ Ti, con p numero intero maggiore di 0 mai usato prima nell'etichettare gli archi

Controllo della concorrenza - serializzabilità

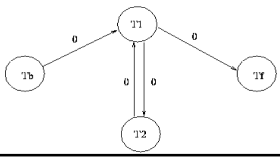
Test di serializzabilità rispetto alle viste

- caso c.: se una transazione Ti scrive un dato Q che viene letto da una transazione Tj, una transazione Tk non può essere inserita tra Ti e Tj
- quindi Tk può essere inserita nell'ordinamento seriale o prima di Ti o dopo Tj
- l'inserzione di due archi con etichette diverse da zero denota che sono possibili due scelte
- è necessario esaminare ogni possibile grafo che si ottiene scegliendo uno solo degli archi che hanno la stessa etichetta
- se almeno uno di tali grafi è aciclico, lo schedule corrispondente è serializzabile rispetto alle viste

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

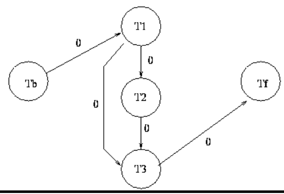
- poligrafo dello schedule 4 (poichè il grafo è ciclico e tutti gli archi hanno etichetta 0 lo schedule non è serializzabile)



Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- poligrafo dello schedule 5 (poichè il grafo non ha cicli, lo schedule è serializzabile ed è equivalente allo schedule seriale T1, T2, T3)



Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

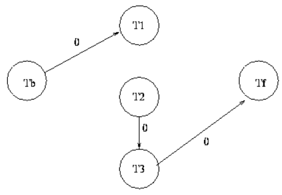
- Schedule 6

T1	T2	T3
Read lr;		
	Write lr;	
		Read lr;
Write lr;		
		Write lr;
Commit T1		
	Commit T2	
		Commit T3

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- poligrafo dello schedule 6

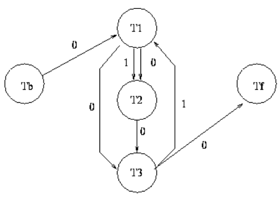


(a)

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- poligrafo dello schedule 6 (segue)

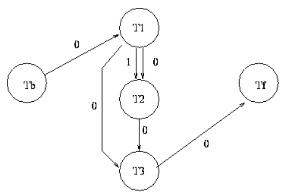


(b)

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- grafo dello schedule 6 - scelta a)

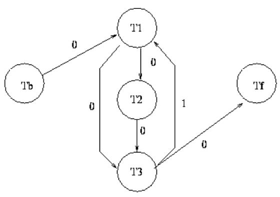


(a)

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- grafo dello schedule 6 - scelta b)



(b)

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- poichè il grafo (a) è aciclico, lo schedule 6 è serializzabile
- lo schedule seriale equivalente è T1, T2, T3

Controllo della concorrenza - serializzabilità

Test di serializzabilità rispetto alle viste

- poiché per ogni coppia di archi con etichette diverse da 0 bisogna andare a verificare l'aciclicità di due diversi grafi, e bisogna considerare tutte le possibili combinazioni, l'algoritmo è esponenziale
- se ci sono n coppie di archi con la stessa etichetta i possibili grafi sono 2^n
- decidere la serializzabilità rispetto alle viste di uno schedule generico è un problema NP-completo

Controllo della concorrenza - serializzabilità

Serializzabilità rispetto alle viste - esempi

- S1 = $w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  è equivalente allo schedule seriale T0, T1, T2
- S2 = $w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
  è equivalente allo schedule seriale T0, T1, T2
- S3 = $r_1(x)\ r_2(x)\ w_2(x)\ w_1(x)$
  non è view-serializzabile (update loss)
- S4 = $r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$
  non è view-serializzabile (inconsistent reads)
- S5 = $r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$
  non è view-serializzabile (ghost update)

Controllo della concorrenza - Protocolli: 2PLocking

- abbiamo visto due algoritmi per determinare se uno schedule è serializzabile
- tali algoritmi assumono che lo schedule sia noto
- per assicurare la serializzabilità durante l'esecuzione di più transazioni si utilizza un protocollo di controllo della concorrenza
- tra i protocolli proposti i più noti sono il protocollo two-phase locking e il protocollo timestamp ordering

Controllo della concorrenza - Protocolli: 2PLocking

- idea base: ritardare l'esecuzione di operazioni in conflitto imponendo che le transazioni pongano dei blocchi (lock) sui dati per poter effettuare operazioni di lettura e scrittura [ricordiamo che due operazioni si dicono in conflitto se sono sullo stesso dato e almeno una delle due è un'operazione di scrittura]
- una transazione può accedere ad un dato solo se ha un lock su quel dato
- le transazioni vengono quindi annotate includendo le operazioni di acquisizione e rilascio dei lock

Controllo della concorrenza - Protocolli: 2PLocking

- due tipi di lock:
  - **shared:** se una transazione T ha ottenuto un lock con modalità shared (condivisa) sul dato Q, allora T può leggere questo dato ma non può modificarlo
  - **exclusive:** se una transazione T ha ottenuto un lock con modalità exclusive (esclusiva) sul dato Q, allora T può sia leggere che modificare Q
- uno shared lock sul dato Q viene richiesto con l'istruzione Ls(Q)
- un exclusive lock viene richiesto con l'istruzione Lx(Q)
- il lock su un dato Q è rilasciato con l'istruzione Un(Q)

Controllo della concorrenza - Protocolli: 2PLocking

- matrice di compatibilità dei lock:

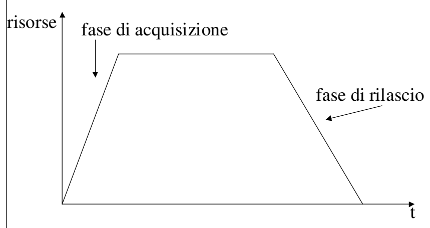
	Shared	Exclusive
Shared	vero	falso
Exclusive	falso	falso

- se Ti richiede un lock di modalità A sul dato Q su cui Tj ($Ti \neq Tj$) possiede un lock di modalità B, a Ti può essere concesso immediatamente un lock su Q se il lock di modalità A è compatibile con il lock di modalità B

Controllo della concorrenza - Protocolli: 2PLocking

- ogni transazione deve effettuare richieste di lock e unlock in due fasi distinte:
- **fase di acquisizione** in cui una transazione può ottenere dei lock, ma non può rilasciarne
- **fase di rilascio** in cui una transazione può rilasciare dei lock, ma non può ottenerne di nuovi
- inizialmente, una transazione si trova nella fase di acquisizione e acquisisce i lock necessari
- non appena la transazione rilascia un lock, inizia la fase di rilascio e non può più essere richiesto alcun lock

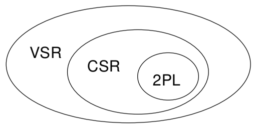
Controllo della concorrenza - Protocolli: 2PLocking



Controllo della concorrenza - Protocolli: 2PLocking

- il protocollo two-phase locking assicura la serializzabilità
- tutti gli schedule ottenibili applicando il protocollo sono serializzabili rispetto ai conflitti
- esempio di schedule serializzabile rispetto ai conflitti non ottenibile con 2PL

  $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$



Controllo della concorrenza - Protocolli: 2PLocking

- il protocollo two-phase locking assicura la serializzabilità, ma è suscettibile di deadlock
- un sistema si dice in deadlock se esiste un insieme di transazioni tali che ogni transazione dell'insieme attende un'altra transazione dell'insieme per completare la propria esecuzione
- prevenzione del deadlock:
  - locking statico [tutti i lock necessari richiesti prima dell'inizio della transazione]
  - ordinamento parziale sui dati
  - schemi wait-die e wound-wait
- rilevazione (wait-for-graph) e risoluzione del deadlock

Controllo della concorrenza - Protocolli: 2PLocking

- Esempio

T1	T2
Read a;	Read a;
Read b;	Read b;
Read c;	Commit T2
Write a;	
Commit T1	

- T1 deve acquisire un exclusive lock sul dato a ⇒ non è ammissibile alcuna esecuzione concorrente delle due transazioni
- in realtà T1 ha bisogno dell'exclusive lock sul dato a solo alla fine ⇒ potrebbe acquisire inizialmente uno shared lock sul dato a, promuovendolo poi ad exclusive lock

Controllo della concorrenza - Protocolli: 2PLocking

- per aumentare il grado di parallelismo viene consentito l'uso di conversioni:
- uno shared lock può essere promosso a exclusive lock e un exclusive lock può essere regredito ad uno shared lock
- la promozione di lock può avvenire solo durante la fase di acquisizione, la regressione solo durante la fase di rilascio
- indichiamo la promozione di un lock sul dato Q con Lx(Q) e la regressione con Ux(Q)

Controllo della concorrenza - Protocolli: 2PLocking

- Esempio (con promozione di lock)

T1	T2
Ls(a)	Ls(a)
Read a;	Read a;
Ls(b)	Ls(b)
Read b;	Read b;
Ls(c)	Un(a)
Read c;	Un(b)
Lx(a)	Commit T2
Write a;	
Un(a)	
Un(b)	
Un(c)	
Commit T1	

Controllo della concorrenza - Protocolli: 2PLocking

- per eliminare le anomalie dovute a letture sporche i lock di una transazione vengono rilasciati solo dopo aver terminato il commit o l'abort (e senza regressione)
- questo protocollo viene detto 2PL stretto (strict 2PL)
- approccio usato nei DBMS commerciali
- un problema della promozione di lock è che, nel caso in cui due transazioni effettuino entrambe prima una lettura e poi una scrittura di un dato, si può generare un deadlock
- per risolvere questo problema viene introdotto un nuovo tipo di lock [update lock]

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Altri tipi di lock

- esiste un terzo tipo di lock mode: update lock
- se una transazione ha un update lock può fare read ma non write sul dato
- un update lock può essere promosso a write lock, mentre un read lock no
- un update lock può essere acquisito anche se ci sono già lock condivisi su quel dato, ma se c'è un update lock nessun altro lock (di nessun tipo) può essere acquisito
- si ha quindi una matrice di compatibilità asimmetrica (U si comporta come S quando deve essere acquisito, come X quando già acquisito)

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Altri tipi di lock

- matrice di compatibilità dei lock con update lock

	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Altri tipi di lock

- ulteriore tipo di lock: increment lock
- molte transazioni agiscono sulla base di dati incrementando e decrementando valori
- una proprietà interessante delle operazioni di incremento è quella di commutare
- si introduce azione di incremento INC(A,c) per la sequenza (atomica) Read(A,t); t = t+c; Write(A,t);
- l'azione INC(X) è in conflitto sia con Read(X) che con Write(X), ma non con INC(X)
- increment lock permette INC ma non letture e scritture

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Altri tipi di lock

- numero arbitrario di increment lock, ma non compatibile con lock in lettura e in scrittura

	S	X	I
S	T	F	F
X	F	F	F
I	F	F	T

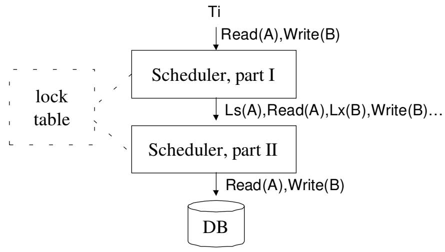
Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Altri tipi di lock

- matrice di compatibilità dei lock con update lock

	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

Controllo della concorrenza - Architettura di uno scheduler basato sui lock



Controllo della concorrenza - Architettura di uno scheduler basato sui lock

- La prima componente dello scheduler prende la sequenza di richieste generate dalle transazioni e inserisce le appropriate operazioni di lock
- la seconda componente riceve dalla prima le operazioni della transazione e le esegue
- se una transazione T viene messa in attesa, perché in attesa di un lock che non può essere concesso, ogni operazione successiva viene aggiunta alla lista delle operazioni della transazione T ancora da effettuare
- se invece T non viene messa in attesa
  - se l'azione è un accesso alla base di dati, viene trasmessa alla base di dati e eseguita

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

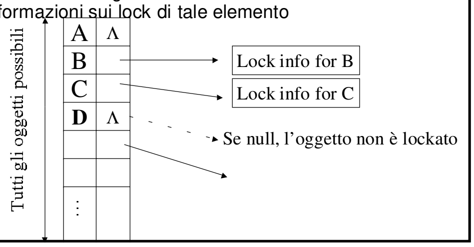
  - se l'azione è una richiesta di log, lo scheduler esamina la tabella dei lock per vedere se il lock può essere concesso
    - se sì, la tabella dei lock viene modificata per registrare il nuovo log
    - se no, si aggiunge un'entrata nella tabella dei lock per registrare che il lock è stato richiesto e la transazione viene messa in attesa
- quando una transazione T effettua il commit o l'abort la prima componente rilascia tutti i lock di T e informa la seconda componente che eventuali transazioni in attesa di tali lock possono ripartire
- quando la seconda componente viene informata che il lock sul dato X è stato rilasciato, determina la (le) transazione(i) in attesa a cui concedere il lock, e le fa ripartire

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

If null, object is unlocked

Tabella dei lock

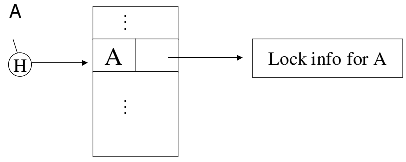
- concettualmente, la tabella dei lock è una relazione che associa a ogni elemento della base di dati le informazioni sui lock di tale elemento



Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Tabella dei lock

- in realtà, si utilizza una tabella hash, in modo che solo gli oggetti lockati compaiano nella tabella
- la dimensione quindi è proporzionale agli oggetti lockati, non a tutta la base di dati

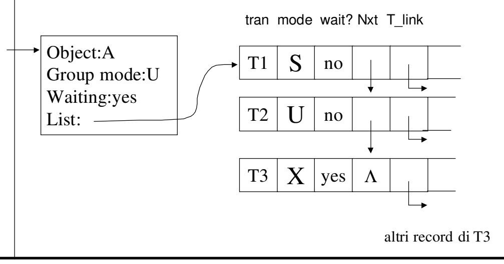


- se l'oggetto non è nella hash table, non è lockato

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Tabella dei lock

- esempio di informazioni contenute nella tabella (con lock S,X,U)



Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Tabella dei lock

- **group mode** è modo compatto di rappresentare quali tipi di lock sono stati acquisiti sul dato (es. S significa solo lock condivisi, U un update lock ed eventualmente lock condivisi, X significa un lock esclusivo)
- il **waiting bit** indica se c'è almeno una transazione in attesa di lock su quel dato
- lista che descrive le transazioni che hanno o sono in attesa di lock sul dato, per ogni transazione
  - il nome della transazione
  - il tipo di lock
  - se la transazione ha o aspetta il lock
- tutti i lock di una transazione sono poi collegati a lista (utile per commit/abort della transazione)

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

Gestione delle richieste di lock

- se T richiede un lock su A
  - se tabella non ha entrata per A, il lock viene concesso e si crea entrata corrispondente
  - altrimenti si usa entrata per stabilire (tramite il group mode) se il lock può essere concesso
    - group mode = U: nessun lock
    - group mode = X: nessun lock
    - group mode = S: lock condivisi o update
  - se la richiesta viene respinta si inserisce T nella lista con Wait? = 'yes'
  - altrimenti si inserisce T nella lista con Wait? = 'no'

Controllo della concorrenza - Architettura di uno scheduler basato sui lock

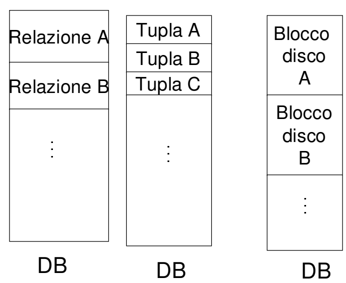
Gestione del rilascio di lock

- se T rilascia un lock su A
  - si cancella l'entrata di T dalla lista per A
  - se il tipo del lock rilasciato è lo stesso del group mode si verifica se si deve modificare il group mode (questo richiede di esaminare la lista)
  - se ci sono entrate nella lista con Waiting?= 'yes' vengono concessi uno o più dei lock richiesti, possibili approcci:
    - first-come first-served: viene concesso il lock richiesto prima
    - priorità ai lock condivisi
    - priorità alle promozioni

Controllo della concorrenza - Lock su gerarchie di elementi

- Finora abbiamo parlato di lock sul dato X/sull'elemento X
- questi possono corrispondere, nei vari sistemi, a tuple, pagine o blocchi, relazioni
- in alcune applicazioni è meglio avere granularità di lock fine, in altre grossa
- es. db di banche, relazione che tiene info sui conti correnti
  - se si locka tutta tale relazione poco spazio per computazioni concorrenti (quasi tutte le transazioni saranno depositi/prelievi, quindi serve exclusive lock)
  - è molto meglio lockare singole pagine
  - ancora più concorrenza si avrebbe lockando singole tuple, ma grosso aumento di complessità rispetto all'aumento di parallelismo
- es. db di documenti
  - operazioni soprattutto di lettura, poche modifiche
  - va benissimo lockare l'eintero documento, non servono granularità più fini (es. frase o paragrafo)

Controllo della concorrenza - Lock su gerarchie di elementi



Controllo della concorrenza - Lock su gerarchie di elementi

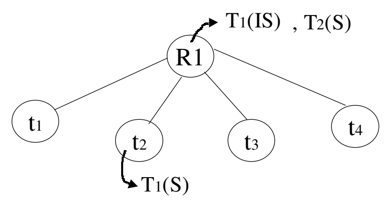
- Se si lockano oggetti grandi (es: relazioni)
  - servono pochi lock
  - basso grado di concorrenza
- se si lockano oggetti piccoli (es: tuple,attributi)
  - servono più lock
  - maggior grado di concorrenza
- In alcuni casi si possono utilizzare diverse granularità di lock
  - nell'esempio della banca, transazione che fa estratto conto ha bisogno di lockare tutte le tuple di un certo conto
- si introduce un nuovo tipo di lock chiamato warning
  - IS(X): intenzione di acquisire uno shared lock su sottolelemento
  - IX(X): intenzione di acquisire un exclusive lock su sottoelemento

Controllo della concorrenza - Lock su gerarchie di elementi

- Gerarchia di elementi della base di dati:
  - relazioni, blocchi, tuple, attributi
- per ottenere un lock S o X su un qualsiasi elemento, bisogna iniziare dalla radice della gerarchia
- se questo è l'elemento che si vuole lockare, si richiede il lock S o X
- se l'elemento che si vuole lockare è più giù nella gerarchia, si inserisce un warning in quel nodo, cioè si richiede IS o IX
- una volta ottenuto tale lock si discende (applicando stesso procedimento ad ogni nodo) verso il sottoelemento di interesse

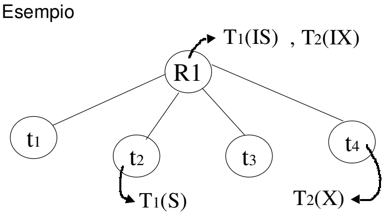
Controllo della concorrenza - Lock su gerarchie di elementi

- Esempio



Controllo della concorrenza - Lock su gerarchie di elementi

- Esempio



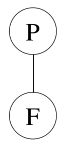
Controllo della concorrenza - Lock su gerarchie di elementi

- Matrice di compatibilità dei lock

		IS	IX	S	SIX	X	Richiedente
Detentore	IS	T	T	T	T	F	
	IX	T	T	F	F	F	
	S	T	F	T	F	F	
	SIX	T	F	F	F	F	
	X	F	F	F	F	F	

Controllo della concorrenza - Lock su gerarchie di elementi

Padre lockato in	Figlio può essere lockato in
IS	IS, S
IX	IS, S, IX, X, SIX
S	[S, IS]
SIX	X, IX, [SIX]
X	--



Controllo della concorrenza - Lock su gerarchie di elementi

Regole

(1) seguendo la matrice di compatibilità per granularità multiple

(2) per prima cosa lockare la radice dell'albero, nel modo opportuno

(3) un nodo Q può essere lockato da Ti in modo S or IS solo se parent(Q) è lockato da Ti in modo IX o IS

(4) un nodo Q può essere lockato da Ti in modo X,SIX,IX solo se parent(Q) è lockato da Ti in modo IX,SIX

(5) Ti è in due fasi (acquisizione-rilascio)

(6) Ti può rilasciare il lock sul nodo Q solo se nessuno dei figli di Q è lockato da Ti

Controllo della concorrenza - Lock su gerarchie di elementi

- Cosa succede se nuovi sottoelementi possono essere aggiunti o cancellati?
- innanzitutto bisogna modificare le regole di lock:
  - se Ti vuole cancellare A deve ottenere un lock esclusivo su A
  - se Ti vuole inserire A deve ottenere un lock esclusivo su A
- ma non funziona ancora … pb dei *phantoms:* tuple che avrebbero dovuto essere lockate, ma non esistevano ancora nel momento in cui il lock è stato acquisito

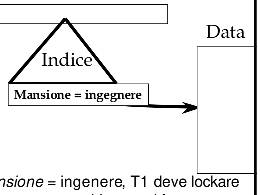
Controllo della concorrenza - Lock su gerarchie di elementi

- *esempio:*
  - T1 locka tutte le pagine contenenti record di impiegati con *mansione* = ingegnere e trova l'impiegato che guadagna di più (supponiamo *stipendio* = 3500)
  - poi T2 inserisce un nuovo impiegato, *mansione* = ingegnere, *stipendio* = 3700
  - T2 cancella anche il tecnico meglio pagato (supponiamo *stipendio* = 2800) e poi effettua il commit
  - T1 ora locka tutte le pagine contenenti tecnici e trova il meglio pagato (supponiamo stipendio = 2700)
- non esiste uno stato della base di dati rispetto a cui T1 è "corretta"
- T1 ha implicitamente assunto di aver lockato *tutti gli ingegneri*

Controllo della concorrenza - Lock su gerarchie di elementi

- L'esempio mostra che la serializzabilità rispetto ai conflitti garantisce la serializzabilità solo se l'insieme di oggetti è fissato
- Soluzione: prima di inserire sul nodo Q si deve lockare parent(Q) in modo esclusivo
- nel nostro esempio verrebbe lockata la relazione impiegati
- alternative: index locking, predicate locking (ma pb di prestazioni)

Controllo della concorrenza - Index locking



- Se c'è un indice denso su *mansione*, T1 deve lockare la pagina di indice contenente le entrate dei dati con *mansione* = ingegnere
  - se non ci sono record con *mansione* = ingenere, T1 deve lockare la pagina di indice dove questa entrata sarebbe, se ci fosse
- se non c'è un indice opportuno, T1 deve lockare tutte le pagine e lockare il file o la tabella in modo da impedire che nuove pagine vengano aggiunte
- in questo modo si assicura che non vengano aggiunti nuovi record con *mansione* = ingegnere

Controllo della concorrenza - Lock : Tree protocol

- Come si può effettuare il locking in un B+ tree?
- Come si può lockare in modo efficiente un singolo nodo foglia?
- Soluzione immediata: ignorare la struttura ad albero e semplicemente lockare le pagine mano a mano che le si attraversa, seguendo 2PL
- questo approccio funziona malissimo dal punto di vista delle prestazioni:
  - il nodo radice (e molti altri nodi a livello alto) diventano bottleneck perché ogni accesso all'albero comincia dalla radice

Controllo della concorrenza - Lock : Tree protocol

- Si può però notare che:
  - i livelli alti dell'albero servono solo per guidare le ricerche verso le pagine foglia
  - per gli inserimenti, un nodo in un cammino dalla radice a una foglia che è stata modificata deve essere lockato (ovviamente in modo esclusivo), solo se la suddivisione (split) si può propagare su fino ad esso dalla foglia modificata (e analogamente per le cancellazioni)
- queste osservazioni per,mettono di realizzare un protocollo di locking per B-tree che garantisce la serializzabilità *anche se viola il 2PL*

Controllo della concorrenza - Lock: Tree protocol

- ricerca: si parte dalla radice e si scende, ad ogni passo si locka il figlio e si rilascia il padre
- inserimento/cancellazione: si parte dalla radice e si scende, ottenendo i lock esclusivi necessari
  - una volta lockato il figlio si controlla se questo è safe: se lo è vengono rilasciati tutti i lock sui suoi antenati
- un nodo è safe se i suoi cambiamenti non si propagheranno al di fuori del nodo stesso
  - per l'inserimento: il nodo non è pieno
  - per la cancellazione: il nodo non è vuoto-a-metà

Controllo della concorrenza - Protocolli: Timestamp ordering

- assegna un timestamp ad ogni transazione e ad ogni dato è associato il timestamp dell'ultima transazione che l'ha letto o scritto
- nei protocolli di locking, l'ordine di esecuzione di transazioni in conflitto è determinato in fase di esecuzione, al tentativo di acquisire un lock incompatibile
- i protocolli timestamp ordering selezionano tale ordine in anticipo
- ad ogni transazione Ti è associata un'etichetta TS(Ti), assegnata dal sistema prima che Ti inizi la propria esecuzione
- l'assegnazione di etichette è monotona (clock di sistema o contatore logico)

Controllo della concorrenza - Protocolli: Timestamp ordering

- le etichette delle transazioni determinano l'ordine di serializzabilità:
- se TS(Ti) < TS(Tj) lo schedule prodotto è equivalente ad uno schedule seriale in cui Ti precede Tj
- ad ogni dato Q si associa:
  - W-timestamp(Q): il timestamp più alto fra tutte le transazioni che hanno eseguito write(Q) con successo
  - R-timestamp(Q): il timestamp più alto fra tutte le transazioni che hanno eseguito read(Q) con successo
- aggiornate ogni volta che si esegue write(Q) o read(Q)

Controllo della concorrenza - Protocolli: Timestamp ordering

- se Ti vuole eseguire read(Q):
  - se TS(Ti) < W-timestamp(Q), la lettura è rifiutata e viene effettuato il rollback di Ti [il valore è stato sovrascritto]
  - se TS(Ti) ≥ W-timestamp(Q), la lettura viene eseguita e R-timestamp(Q) = max(R-timestamp(Q), TS(Ti))

Controllo della concorrenza - Protocolli: Timestamp ordering

- se Ti vuole eseguire write(Q):
  - se TS(Ti) < R-timestamp(Q), la scrittura è rifiutata e viene effettuato il rollback di Ti [il valore è già stato letto]
  - se TS(Ti) < W-timestamp(Q), la scrittura viene ignorata [scrittura di un valore obsoleto]
  - altrimenti, la scrittura viene eseguita e W- timestamp(Q) = max(W-timestamp(Q), TS(Ti))

Controllo della concorrenza - Protocolli: Timestamp ordering

- se si effettua il rollback di una transazione Ti, le si assegna un nuovo timestamp e si effettua il restart
- il protocollo di timestamp ordering assicura la serializzabilità e l'assenza di deadlock
- possibile problema: rollback a cascata
- si può controllare che una transazione legga solo valori modificati da transazioni che hanno terminato con successo (bit di commit)

Controllo della concorrenza - Protocolli: Timestamp ordering

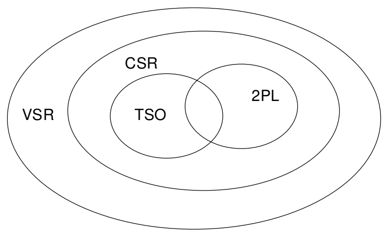
- Esempio

T1	T2
Read b;	Read b;
Read a;	b:=b-50;
Display a+b;	Write b;
Commit T1	Read a;
	a := a+50;
	Write a;
	Display a+b;
	Commit T2

Controllo della concorrenza - Protocolli: Timestamp ordering

- Esempio (segue)
- TS(T1) = 100, TS(T2) = 102, R-timestamp(a) = 80, W-timestamp(a) = 80, R-timestamp(b) = 90, W-timestamp(a) = 90
- se si considera lo schedule con Read(b) della transazione T1, poi tutte le operazioni della transazione T2, poi Read(a) della transazione T1 si ottiene un rollback
- infatti lo schedule non è serializzabile

Controllo della concorrenza - Protocolli: confronto



Controllo della concorrenza - Protocolli: confronto

- In 2PL le transazioni sono messe in attesa, in TSO sono abortite e fatte ripartire
- l'ordine di serializzazione in 2PL è imposto dalle operazioni in conflitto, mentre in TSO è imposto dai timestamp
- la necessità di aspettare il commit di una transazione (no dirty reads) porta a 2PL stretto e alla bufferizzazione delle scritture in TSO
- 2PL può causare deadlock
- poiché però il dover far ripartire le transazioni costa di più che non farle aspettare, il 2PL è più utilizzato

Controllo della concorrenza - Protocolli: Validazione

- Se i conflitti sono rari, piuttosto che prevenirli è meglio lasciare proseguire le transazioni e verificare se si sono verificati dei conflitti prima che la transazione effettui il commit
- le transazioni hanno tre fasi:
  - (1) Read
    - tutti i valori necessari sono letti dalla base di dati e scritti in una memoria temporanea (si lavora su una copia locale dei dati)
    - non viene effettuato alcun locking
  - (2) Validate
    - si controlla se lo schedule realizzato fino a questo momento è serializzabile
  - (3) Write
    - se la validazione ha successo di effettuano le scritture sulla base di dati

Controllo della concorrenza - Protocolli: Validazione

- Idea chiave:
  - la validazione deve essere atomica
  - se $T_1, T_2, T_3, \ldots$ è l'ordine di validazione allora lo schedule risultante è equivalente rispetto ai conflitti allo schedule seriale $T_1\ T_2\ T_3 \ldots$
- il sistema mantiene tre insiemi
  - <u>START</u> = transazioni che hanno iniziato l'esecuzione, ma non hanno ancora completato la fase 2 (validazione)
  - <u>VAL</u> = transazioni che hanno finito la fase 2
  - <u>FIN</u> = transazioni che hanno finito la fase 3 (e che quindi hanno completato l'esecuzione)

Controllo della concorrenza - Protocolli: Validazione

- Per ogni transazione T in START il sistema mantiene START(T) il timestamp di quando T ha iniziato la sua esecuzione
- per ogni transazione T in VAL il sistema mantiene sia START(T) che VAL(T), il timestamp di quando T ha terminato la sua validazione
- per ogni transazione T in FIN il sistema mantiene START(T), VAL(T) e FIN(T): il timestamp di quando T ha terminato la sua esecuzione
- quando FIN(T) < START(U) per ogni transazione attiva U le informazioni relative a T possono essere cancellate (garbage collection periiodica)

Controllo della concorrenza - Protocolli: Validazione

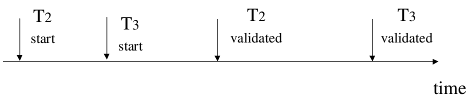
Esempio di cosa la validazione deve prevenire

(T2 potrebbe avere scritto B dopo che T3 lo ha letto)

$RS(T_2)=\{B\}$ $\quad$ $RS(T_3)=\{A,B\}$

$WS(T_2)=\{B,D\}$ $\quad$ $WS(T_3)=\{C\}$

$WS(T_2) \cap RS(T_3) \neq \phi$

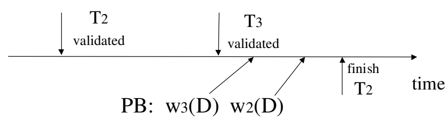


Controllo della concorrenza - Protocolli: Validazione

- Altro esempio di cosa la validazione deve prevenire

$RS(T_2)=\{A\}$ $\quad$ $RS(T_3)=\{A,B\}$

$WS(T_2)=\{D,E\}$ $\quad$ $WS(T_3)=\{C,D\}$



Controllo della concorrenza - Protocolli: Validazione

- Per validare una transazione T

  (1) si deve confrontare RS(T) con WS(U) e queste devono avere intersezione vuota per ogni transazione U che non ha terminato prima che T iniziasse, cioè t.c. FIN(U) > START(T)

  (2) si deve confrontare WS(T) con WS(U) e queste devono avere intersezione vuota per ogni transazione U che non ha terminato prima che T validasse, cioè t.c. FIN(U) > VAL(T)

Controllo della concorrenza - Protocolli: Validazione

- Algoritmo di validazione

  (1) quando T_j comincia la fase 1:

  ignore(T_j) $\leftarrow$ FIN

  (2) alla validazione di T_j:

  if check (T_j) then

  [VAL $\leftarrow$ VAL U {T_j};

  effettua la fase di write;

  FIN $\leftarrow$ FIN U {T_j}]

Controllo della concorrenza - Protocolli: Validazione

Check (T_j):

for $T_i \in$ VAL - IGNORE (T_j) do

if [$WS(T_i) \cap RS(T_j) \neq \varnothing$ OR

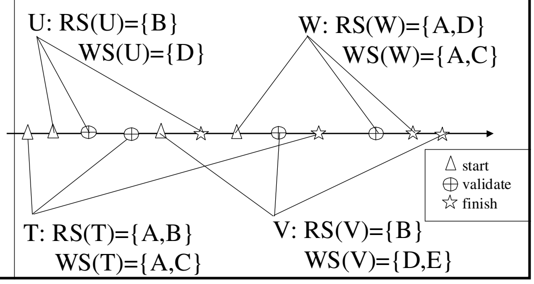
($T_i \notin$ FIN AND $WS(T_i) \cap WS(T_j) \neq \varnothing$)]

THEN RETURN false;

RETURN true;

Controllo della concorrenza - Protocolli: Validazione

Esempio



Controllo della concorrenza - Protocolli: Validazione

Esempio

- validazione di U: nessun'altra transazione è stata ancora validata, quindi non c'è niente da controllare
- validazione di T: quando T valida, U non ha ancora finito - bisogna quindi confrontare sia RS(T) che WS(T) con WS(U)
  - $RS(T) \cap WS(U) = \{A,B\} \cap \{D\} = \varnothing$
  - $WS(T) \cap WS(U) = \{A,C\} \cap \{D\} = \varnothing$

Controllo della concorrenza - Protocolli: Validazione

Esempio

- validazione di V: quando V valida, U è validata e finita, T è validata ma non ha ancora finito - bisogna quindi confrontare
  - sia RS(V) che WS(V) con WS(T)
    - $RS(V) \cap WS(T) = \{B\} \cap \{A,C\} = \varnothing$
    - $WS(V) \cap WS(T) = \{D,E\} \cap \{A,C\} = \varnothing$
  - RS(V) con WS(U)
    - $RS(V) \cap WS(U) = \{B\} \cap \{D\} = \varnothing$

Controllo della concorrenza - Protocolli: Validazione

Esempio

- validazione di W:
  - U è finita prima che W cominciasse, quindi non deve essere considerata
  - quando W valida, T è validata e finita, V è validata ma non ha ancora finito - bisogna quindi confrontare
  - sia RS(W) che WS(W) con WS(V)
    - $RS(W) \cap WS(V) = \{A,C\} \cap \{D,E\} = \varnothing$
    - $WS(W) \cap WS(V) = \{A,D\} \cap \{D,E\} = \{D\}$
  - RS(W) con WS(T)
    - $RS(W) \cap WS(T) = \{A,D\} \cap \{A,C\} = \{A\}$
- W non viene quindi validata e ne viene effettuato il rollback

Controllo della concorrenza - Protocolli: confronto

- Con la validazione l'ordine di serializzazione è imposto dall'ordine di validazione
- a differenza di 2PL e come TSO, la validazione non causa deadlock ma può causare la riesecuzione di transazioni
- poiché però il dover far ripartire le transazioni costa di più che non farle aspettare, il 2PL è più utilizzato, anche se l'approccio basato su validazione (noto anche come optimistic concurrency control) è utilizzato in alcuni contesti in cui:
  - i conflitti sono rari
  - le risorse del sistema sono molte
  - ci sono vincoli di real-time

Livelli di isolamento

- In SQL ogni transazione ha un **access mode** e un **isolation level**
- l'access mode può essere
  - READ ONLY la transazione esegue solo interrogazioni (acquisisce solo lock condivisi)
  - READ WRITE la transazione esegue interrogazioni e aggiornamenti
- il livello di isolamento specifica quanto una transazione è esposta agli effetti di altre transazioni in esecuzione contemporaneamente
- si può ottenere un livello di concorrenza più alto esponendo maggiormente la transazione a cambiamenti effettuati da altre transazioni che non hanno ancora effettuato il commit

Livelli di isolamento

- Il più alto livello di isolamento è SERIALIZABLE
- questo livello garantisce che
  - T legga solo modifiche effettuate da transazioni che hanno effettuato il commit
  - nessun valore letto o scritto da T sia modificato prima che T abbia terminato
  - se T legge un insieme di valori basandosi su una qualche condizione di ricerca, questo insieme non venga modificato da altre transazioni fino a che T non ha terminato
- in termini di lock, una transazione SERIALIZABLE ottiene i lock prima di leggere/modificare oggetti (anche sugli insiemi di oggetti che non devono essere modificati durante la sua esecuzione) e li tiene fino alla fine, in accordo al 2PL stretto

Livelli di isolamento

- Il livello REPEATABLE READS garantisce che T legga solo modifiche effettuate da transazioni che hanno effettuato il commit e che nessun valore letto o scritto da T sia modificato da altre transazioni prima che T abbia terminato
- utilizza lo steso protocollo di locking di SERIALIZABLE, tranne che non utilizza index locking o lock su collezioni

Livelli di isolamento

- Il livello READ COMMITTED assicura che T legga solo modifiche effettuate da transazioni che hanno effettuato il commit e che nessun valore scritto da T sia modificato da altre transazioni prima che T abbia terminato
- un valore letto da T può però essere modificato da altre transazioni prima che T abbia terminato
- i lock esclusivi vengono mantenuti fino al commit, quelli condivisi vengono rilasciati immediatamente

Livelli di isolamento

- Il livello UNCOMMITTED READS permette ad una transazione di vedere modifiche effettuate da transazioni concorrenti, anche se non hanno ancora effettuato il commit
- non vengono acquisiti lock condivisi prima di effettuare le letture
- una transazione a livello UNCOMMITTED READS deve per forza essere READ ONLY (quindi non richiede alcun lock)

Livelli di isolamento

Livello di isolamento	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	possib	possib	possib
Read Committed	No	possib	possib
Repeatable Reads	No	No	possib
Serializable	No	No	No

Phantom problem = possibilità che una transazione ritrovi un una collezione di tuple due volte e veda due risultati differenti, anche se non modifica nessuna di queste tuple

Livelli di isolamento

- Il livello SERIALIZABLE è il più sicuro ed è quello preferibile per la maggioranza delle transazioni
- alcune transazioni possono però essere eseguite a livelli di isolamento più bassi in modo da migliorare le prestazioni del sistema
  - ad es. una query statitistica che ritrova lo stipendio medio degli impiegati può essere eseguita al livello READ COMMITTED, o anche READ UNCOMMITTED
- access mode e livello di isolamento possono essere impostati con il comando SET TRANSACTION
  - es. SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
- i valori di default sono SERIALIZABLE e READ WRITE

Gestione del ripristino

- Tre principali tipi di malfunzionamenti:
  - malfunzionamenti del disco: le informazioni residenti su disco vengono perse (rottura della testina, errori durante il trasferimento dei dati)
  - malfunzionamenti di alimentazione: le informazioni memorizzate in memoria centrale e nei registri vengono perse
  - errori nel software: si possono generare risultati scorretti e il sistema potrebbe essere in uno stato inconsistente (errori logici ed errori di sistema)
- il sottosistema di recovery (ripristino) deve identificare i malfunzionamenti e ripristinare la base di dati allo stato (consistente) precedente il malfunzionamento

Gestione del ripristino

Classificazione delle memorie

- **memoria volatile** le informazioni contenute vengono perse in caso di cadute di sistema

  esempi: memoria principale e cache
- **memoria non volatile** le informazioni contenute sopravvivono a cadute di sistema, possono però essere perse a causa di altri malfunzionamenti

  esempi: disco e nastri magnetici
- **memoria stabile** le informazioni contenute non possono essere perse (astrazione teorica)

  se ne implementano approssimazioni, duplicando le informazioni in diverse memorie non volatili con probabilità di fallimento indipendenti

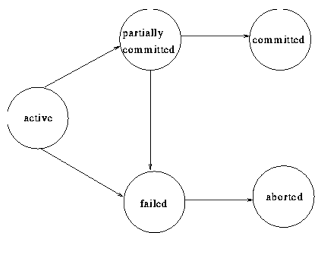
Gestione del ripristino

Modello astratto per l'esecuzione di transazioni

- una transazione è sempre in uno dei seguenti stati:
- **active:** lo stato iniziale
- **partially committed:** lo stato raggiunto dopo che è stata eseguita l'ultima istruzione
- **failed:** lo stato raggiunto dopo aver determinato che l'esecuzione non può procedere normalmente
- **aborted:** lo stato raggiunto dopo che la transazione ha subito un rollback e la base di dati è stata ripristinata allo stato precedente l'inizio della transazione
- **committed:** dopo il completamento con successo

Gestione del ripristino

Modello astratto per l'esecuzione di transazioni



Gestione del ripristino

Modello astratto per l'esecuzione di transazioni

- è importante che una transazione non effettui scritture esterne osservabili (cioè scritture che non possono essere "cancellate", ad es. su terminale o stampante) prima di entrare nello stato di commit
- dopo il rollback di una transazione, il sistema ha due possibilità:
  - **rieseguire la transazione** ha senso solo se la transazione è stata abortita a seguito di errori software o hardware non dipendenti dalla logica interna della transazione
  - **eliminare la transazione** se si verificano degli errori interni che possono essere corretti solo riscrivendo il programma applicativo

Gestione del ripristino

Esempio

- T transazione che trasferisce Lit. 100000 dal conto A al conto B, con A = Lit. 1000000, B = Lit. 15000000
- dopo la modifica di A e prima della modifica di B, si verifica una caduta di sistema e i contenuti della memoria vengono persi
  - si riesegue T ⇒ stato (inconsistente) in cui A = Lit. 800000 e B = Lit. 15.100000
  - non si riesegue T ⇒ stato corrente (inconsistente) in cui A = Lit. 900000 e B = Lit. 15000000
- in entrambi i casi lo stato risultante è inconsistente, il problema è causato dal fatto di avere modificato la base di dati prima di avere la certezza che la transazione avrebbe terminato con successo

Gestione del ripristino Recovery con log

- durante l'esecuzione di una transazione tutte le operazioni di scrittura sono registrate in un particolare file gestito dal sistema, detto **file di log**
- concettualmente, il log può essere pensato come un file sequenziale, nell'implementazione effettiva possono essere usati più file fisici
- ad ogni record inserito nel log viene attribuito un identicatore unico (LSN, log sequence number o numero di sequenza di log) che in genere è l'indirizzo logico del record

Gestione del ripristino Recovery con log

- la versione non volatile del log è memorizzata su memoria stabile
- i record di log sono memorizzati in un primo momento solo nei buffer dei file dilog nella memoria volatile
- prima di cominciare il commit, tutti i record di log fino ad un certo punto, identificato da un LSN, vengono scritti su memoria stabile (il log viene forzato fino a quel LSN)
- tramite il log, il sistema può gestire qualsiasi malfunzionamento che non implichi la perdita di informazioni contenute in memoria non volatile

Gestione del ripristino Recovery con log

Log incrementale con modifiche differite

- tutte le operazioni di scrittura eseguite da una transazione vengono differite fino a che la transazione non entra nello stato partially committed
- tutte le modifiche sono registrate nel file di log
- durante l'esecuzione di una transazione Ti:
- prima che Ti cominci la propria esecuzione viene scritto nel log il record <Ti start>
- ogni operazione di scrittura write[x] produce un record di log della forma: <Ti; x; nuovo valore>
- quando Ti entra in stato partially committed, viene scritto nel log il record <Ti commit>

Gestione del ripristino Recovery con log

Log incrementale con modifiche differite

- tutti i record di log sono scritti su memoria stabile, a questo punto, la transazione entra nello stato committed
- lo schema di recovery usa la procedura di recovery redo(Ti), che assegna a tutti i dati aggiornati dalla transazione Ti i nuovi valori
- l'operazione redo deve essere idempotente, cioè, più esecuzioni in sequenza di tale operazione devono essere equivalenti ad un'esecuzione singola (questo assicura un comportamento corretto anche in presenza di malfunzionamenti durante il recovery)
- a seguito di un malfunzionamento, una transazione Ti viene rieseguita se il log contiene entrambi i record <Ti start> e <Ti commit>

Gestione del ripristino Recovery con log

Log incrementale con modifiche differite - **Esempio**

T1	T2
Read lr	Read c/c2
lr = lr - 150000	c/c2 = c/c2 - 200000
Write lr	Write c/c2
Read c/c1	Commit T2
c/c1 = c/c1 + 150000	
Write c/c1	
Commit T1	

- viene eseguita prima T1 e poi T2
- inizialmente lr = 500000 c/c1 = 600000 c/c2 = 800000

Gestione del ripristino
Recovery con log

Log incrementale con modifiche differite - Esempio

- contenuto del log al termine dell'esecuzione delle due transazioni:
  - < T1, start >
  - < T1, lr,350000 >
  - < T1, c/c1, 750000 >
  - < T1, commit >
  - < T2, start >
  - < T2, c/c2, 600000 >
  - < T2, commit >

Gestione del ripristino
Recovery con log

Log incrementale con modifiche differite - Esempio

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write c/c1", lo stato del log al momento del crash è il seguente:
  - < T1, start >
  - < T1, lr, 350000 >
  - < T1, c/c1, 750000 >

  ⇒ non si esegue alcuna operazione di redo

Gestione del ripristino
Recovery con log

Log incrementale con modifiche differite - Esempio

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write c/c2", lo stato del log al momento del crash è il seguente:
  - < T1, start >
  - < T1, lr, 350000 >
  - < T1, c/c1, 750000 >
  - < T1, commit >
  - < T2, start >
  - < T2, c/c2, 600000 >

  ⇒ viene effettuato redo(T1) e lo stato della base di dati diventa: lr = 350000 c/c1 = 750000 c/c2 = 800000

Gestione del ripristino
Recovery con log

Log incrementale con modifiche differite - Esempio

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Commit T2 ", lo stato del log al momento del crash è il seguente:
  - < T1, start >
  - < T1, lr, 350000 >
  - < T1, c/c1, 750000 >
  - < T1, commit >
  - < T2, start >
  - < T2, c/c2, 600000 >
  - < T2, commit >

  ⇒ vengono effettuate le operazioni redo(T1) e redo(T2) e lo stato della base di dati diventa: lr = 350000 c/c1 = 750000 c/c2 = 600000

Gestione del ripristino
Recovery con log

Log incrementale con modifiche immediate

- gli aggiornamenti sono effettuati sulla base di dati e un log incrementale tiene traccia di tutti i cambiamenti
- tali informazioni sono usate per riportare la base di dati allo stato consistente precedente in caso di malfunzionamento
- prima che Ti cominci la propria esecuzione viene scritto nel log il record <Ti start>
- ogni operazione di scrittura write[x] è preceduta dalla scrittura un record di log della forma:

  < Ti; x; vecchio valore, nuovo valore >
- quando T i entra in stato partially committed, viene scritto nel log il record <Ti commit>

Gestione del ripristino
Recovery con log

Log incrementale con modifiche immediate

- non si può aggiornare effettivamente la base di dati prima che il corrispondente record di log sia scritto in memoria stabile
- due procedure di ripristino:
  - undo(Ti): ripristina i valori di tutti i dati aggiornati da Ti ai vecchi valori
  - redo(Ti): pone il valore di tutti i dati aggiornati da Ti ai nuovi valori
- entrambe devono essere idempotenti

Gestione del ripristino Recovery con log

Log incrementale con modifiche immediate

- a seguito di un malfunzionamento:
- se per una transazione Ti il log contiene entrambi i record <Ti start> e <Ti commit>, Ti viene rieseguita, cioè viene effettuato redo(Ti)
- se per una transazione Ti il log contiene il record <Ti start> ma non il record <Ti commit>, Ti viene disfatta, cioè viene effettuato undo(Ti)

Gestione del ripristino Recovery con log

Log incrementale con modifiche immediate - Esempio

- contenuto del log al termine dell'esecuzione delle due transazioni:
  - < T1, start >
  - < T1, lr,350000 >
  - < T1, c/c1, 750000 >
  - < T1, commit >
  - < T2, start >
  - < T2, c/c2, 600000 >
  - < T2, commit >

Gestione del ripristino Recovery con log

Log incrementale con modifiche immediate - Esempio

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write c/c1", lo stato del log al momento del crash è il seguente:
  - < T1, start >
  - < T1, lr, 500000, 350000 >
  - < T1, c/c1, 600000, 750000 >
  - ⇒ si esegue l'azione di undo(T1), il nuovo stato dei dati è: lr = 500000 c/c1 = 600000 c/c2 = 800000

Gestione del ripristino Recovery con log

Log incrementale con modifiche immediate - Esempio

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write c/c2", lo stato del log al momento del crash è il seguente:
  - < T1, start >
  - < T1, lr, 500000, 350000 >
  - < T1, c/c1, 600000, 750000 >
  - < T1, commit >
  - < T2, start >
  - < T2, c/c2, 800000, 600000 >
  - ⇒ viene effettuato redo(T1) e undo(T2), lo stato della base di dati diventa: lr = 350000 c/c1 = 750000 c/c2 = 800000

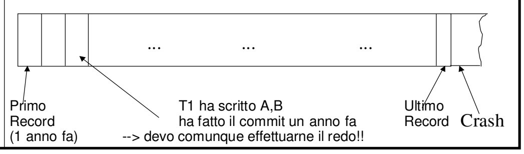
Gestione del ripristino Recovery con log

Log incrementale con modifiche immediate - Esempio

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Commit T2", lo stato del log al momento del crash è il seguente:
  - < T1, start >
  - < T1, lr, 500000, 350000 >
  - < T1, c/c1, 600000, 750000 >
  - < T1 , commit >
  - < T2 , start >
  - < T2 , c/c2, 800000, 600000 >
  - < T2 , commit >
  - ⇒ vengono effettuate le operazioni redo(T1) e redo(T2) e lo stato della base di dati diventa: lr = 350000 c/c1 = 750000 c/c2 = 600000

Gestione del ripristino Recovery con log

- Il meccanismo di log incrementale con modifiche immediate è più utilizzato perché quello con modifiche ritardate ha lo svantaggio di dover mantenere necessariamente tutti i blocchi modificati da transazioni che non hanno effettuato il commit nel buffer
- Altro problema:



Gestione del ripristino Recovery con log

Checkpoint

- meccanismo introdotto per ridurre i tempi di recovery
- il sistema periodicamente:
  - forza tutti i record di log residenti in memoria principale su memoria stabile
  - forza tutti i blocchi di buffer su disco
  - forza il record <checkpoint T1, …, Tn> su memoria stabile, dove T1, …, Tn sono le transazioni in esecuzione al momento del checkpointing
- in questo modo gli effetti delle transazioni che hanno effettuato il commit prima del checkpoint sono memorizzati in modo permanente nella base di dati

Gestione del ripristino Recovery con log

Dump

- un dump è una copia completa della base di dati, normalmente creata quando il sistema non è operativo
- la copia è memorizzata in memoria stabile, tipicamente su nastro, ed è chiamata backup
- un record dump nel log segnala la presenza di un backup effettuato in un certo momento ed identifica il file o il device su cui è stato effettuato il dump

Gestione del ripristino Recovery con log

Tipi di guasti

- guasti di sistema: bug software, per esempio del sistema operativo, cadute di tensione
  - si perde il contenuto della memoria principale
  - non si perde il contenuto della memoria secondaria
- guasti ai dispositivi di memorizzazione, per esempio rottura della testina
  - si perde il contenuto della memoria secondaria
  - non si perde il contenuto della memoria stabile (log)
- protocolli di restart:
  - ripresa a caldo: utilizzata per guasti di sistema
  - ripresa a freddo: utilizzata per guasti ai dispositivi di memorizzazione

Gestione del ripristino Recovery con log

Ripresa

- obiettivo del processo di restart è classificare le transazioni come
  - completate (quelle le cui azioni sono già state registrate in memoria stabile)
  - committed ma eventualmente non completate (le cui azioni devono essere rifatte)
  - non committed (le cui azioni devono essere disfatte)

Gestione del ripristino Recovery con log

Ripresa a caldo

- si scorre il log all'indietro fino a trovare il record checkpoint più recente
- si costruiscono gli insiemi UNDO e REDO delle transazioni da disfare e rifare
  - UNDO è inizializzato con le transazioni attive al checkpoint, REDO con l'inseme vuoto
  - si percorre il log in avanti aggiungendo ogni transazione per cui si trova strat in UNDO e spostandola in REDO quando se ne trova il commit
- si torna indietro nel log fino alla prima azione della più vecchia transazione nei due insiemi [NB: può essere prima del checkpoint]
- andando avanti nel log si effettua undo(T) per ogni T in UNDO e redo(T) per ogni T in REDO

Gestione del ripristino Recovery con log

Ripresa a freddo

- tre fasi:
- si accede al dump e si copiano le pagine danneggiate, si trova quindi il più recente record dump nel log
- si scorre il log in avanti, applicando le azioni corrispondenti alle parti danneggiate della base di dati, riportandosi nello stato della base di dati precedente il guasto
- si effettua una ripresa a caldo