GESTIONE DELLE TRANSAZIONI

Vincenzo Calabrò

Transazione (1)

Transazione

- unità elementare di lavoro svolta da una applicazione, alla quale si associano particolari caratteristiche di correttezza, robustezza, e isolamento

Sistema transazionale

- sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni da parte di più applicazioni concorrenti

Transazione (2)

Sintatticamente:

- transazione è una procedura racchiusa da due comandi
  - begin transaction (`bot`)
  - end transaction (`eot`)

- all'interno di una transazione, uno dei seguenti due comandi è eseguito esattamente una volta
  - commit work (`commit`): per terminare con successo
  - rollback work (`abort`): per fare un undo della procedura

Transazione (3)

Una transazione è detta ben formata se

- inizia con `begin transaction`
  - o equivalente a seconda del linguaggio
- finisce con `end transaction`
- contiene uno ed uno solo dei comandi `commit` o `rollback`
- non avvengono operazioni di accesso (lettura/scrittura) alla base di dati successive all'esecuzione del comando di `commit` o `rollback`

Transazione: esempi (1)

```
begin transaction
      x := x-10
      y := y+10
      z := z-y
      if z < 50
            then commit work
            else rollback
end transaction
```

Transazione: esempi (2)

```
begin transaction
      update ContoCorrente
      set Saldo = Saldo - 100
      where NumConto = '123456'
      update ContoCorrente
      set Saldo = Saldo + 100
      where NumConto = '123457'
      commit work
end transaction
```

Transazioni: proprietà

Le transazioni devono soddisfare le proprietà ACIDe

- Atomicità
- Consistenza
- Isolamento
- Durabilità

Atomicità

Una transazione è una unità atomica di lavoro

- non può lasciare la base di dati in uno stato intermedio
  - un guasto o un errore prima del commit causano l'UNDO del lavoro fatto fino a quel punto
  - un guasto o un errore dopo il commit possono richiedere il REDO del lavoro fatto precedentemente, se la sua permanenza sulla base di dati non è garantita

Possibili comportamenti di una transazione:

- commit: comportamento normale (99.9%)
- rollback richiesto dalla applicazione: suicidio
- rollback richiesto dal sistema: omicidio

Consistenza

L’esecuzione di una transazione non deve violare i vincoli di integrità definiti sulla base di dati

- il controllo sul mantenimento dell’integrità può essere:
  - immediato: durante la transazione (l’operazione che causa la violazione è rifiutata)
  - differito: alla fine della transazione (se dei vincoli sono violati, l’intera transazione è rifiutata)

Isolamento

L’esecuzione di una transazione deve essere indipendente da quella di tutte le altre transazioni concorrenti

- l’esecuzione concorrente di un insieme di transazioni deve portare allo stesso risultato di una esecuzione sequenziale arbitraria delle stesse transazioni

Durabilità (persistenza)

Gli effetti di una transazione che ha eseguito commit non devono essere persi

- il sistema deve garantire persistenza dei dati anche in caso di malfunzionamenti e guasti

Transazioni e moduli di sistema (1)

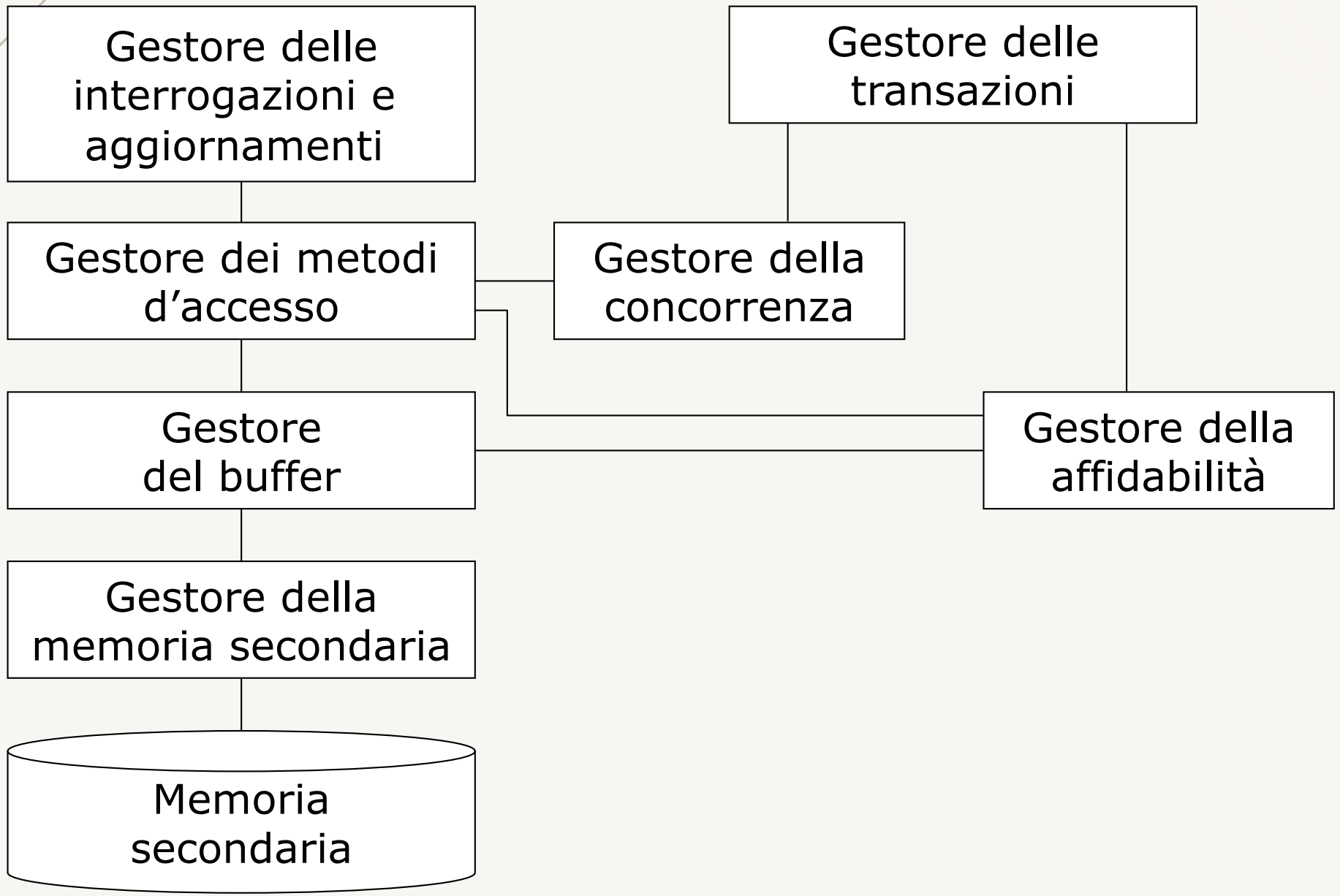
Le proprietà ACIDe sono controllate e garantite da moduli del DBMS

- atomicità: gestore dell'affidabilità
- consistenza: compilatore del DDL
  - genera le attività di verifica che sono poi effettuate al momento di esecuzione delle transazioni
- isolamento: gestore della concorrenza
- durabilità: gestore dell'affidabilità

Transazioni e moduli di sistema (2)

- Gestore delle transazioni
  - coordina le attività connesse con le transazioni attraverso l'esecuzione delle operazioni di `begin transaction`, `commit`, e `rollback`
- Gestore dell'affidabilità
  - garantisce atomicità e persistenza, interagendo con
    - gestore dei metodi di accesso (per tenere traccia delle operazioni richieste)
    - gestore del buffer (per richiedere, se necessario, scritture sui dischi)
- Gestore della concorrenza
  - garantisce isolamento filtrando ed eventualmente ripianificando gli accessi (richiesti dal gestore degli accessi)

Transazioni e moduli di sistema (3)



Gestore dell'affidabilità

Responsabile per:

- esecuzione dei comandi transazionali
  - `begin transaction` (B)
  - `commit` (C)
  - `rollback` (A, per 'Abort')
- ripristino dopo i malfunzionamenti
  - ripresa a caldo
  - ripresa a freddo

Assicura atomicità e persistenza

Memoria stabile

Memoria resistente ai guasti

- è un'astrazione
  - nessuna memoria può avere 0 probabilità di guasto
  - replicazione e protocolli di scrittura robusti possono rendere la probabilità prossima allo 0
  - organizzata in modi diversi in dipendenza della criticità dell'applicazione, es.:
    - una unità nastro
    - un nastro e un disco
    - due unità disco 'a specchio' (mirror)

Un guasto di memoria stabile è considerato catastrofico e impossibile

Log

File sequenziale che registra, in ordine cronologico, le azioni eseguite dalle varie transazioni

- scritto su memoria stabile (archivio permanente)
- gestito dal controllore dell'affidabilità
- rende possibili undo e redo
  - metafore: il filo di Arianna, le briciole di pane di Hansel e Gretel, …

Organizzazione del log (1)

File sequenziale

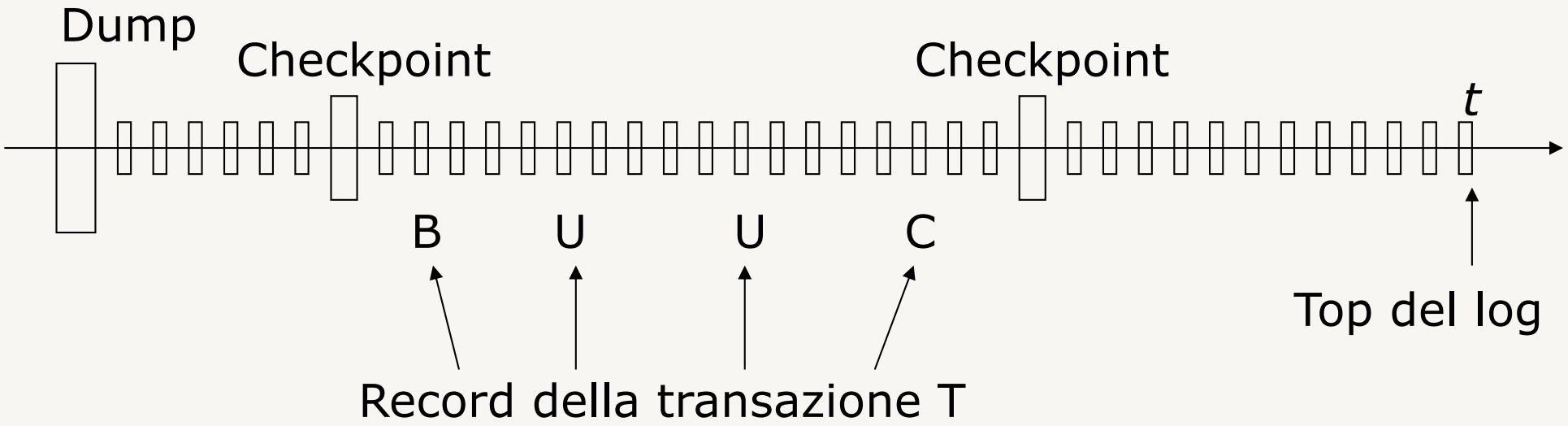
- record in ordine cronologico

Organizzazione del log (2)

Due tipi di record:

- di transazione
  - `begin`, B(T)
  - `insert`, I(T,O,AS)
  - `delete`, D(T,O,BS)
  - `update`, U(T,O,BS,AS)
  - `commit`, C(T)
  - `abort`, A(T)
- di sistema
  - `dump`, (raro)
  - `checkpoint`, (più frequente)

Organizzazione del log (3)



Organizzazione del log (4)

- I record di transazione contengono, per le operazioni (`insert`, `delete`, `update`)
  - before state (BS)
    - stato dell'oggetto prima dell'operazione
  - after state (AS)
    - stato dell'oggetto dopo l'operazione

$\Rightarrow$ è possibile fare undo e redo delle operazioni

Undo

Per disfare un'azione:

- update: `undo`(U(T,O,BS,AS))
  - copia il valore BS nell'oggetto O
- delete: `undo`(D(T,O,BS))
  - ripristina l'oggetto O con valore BS
- insert: `undo`(I(T,O,AS))
  - cancella l'oggetto O

Redo

Per rifare un'azione:

- update: `redo`(U(T,O,BS,AS))
  - copia il valore AS nell'oggetto O
- delete: `redo`(D(T,O,BS))
  - cancella l'oggetto O
- insert: `redo`(I(T,O,AS))
  - ripristina l'oggetto O con valore AS

Undo e Redo: idempotenza

Undo e redo godono della

- proprietà di idempotenza: un numero arbitrario di undo/redo della stessa azione è equivalente all'esecuzione dell'undo/redo della azione una volta sola
  - `undo(A) … undo(A) = undo(A)`
  - `redo(A) … redo(A) = redo(A)`

L'idempotenza garantisce correttezza anche in caso di ripetizione delle operazioni di undo e redo

Checkpoint

Operazione di sistema svolta dal gestore dell'affidabilità con il coordinamento del buffer manager

- registra sul log le transazioni attive
- aggiorna la memoria secondaria rispetto a tutte le transazioni completate
- è eseguita periodicamente

Checkpoint: esecuzione

- Si sospende l'accettazione di operazioni di scrittura, commit o abort da parte di transazioni
- Si trasferiscono (force) su memoria secondaria tutte le pagine sporche del buffer relative a transazioni che hanno già effettuato il commit
- Si scrive in modo sincrono (force) nel log un record di checkpoint CK(T_1,...,T_n) che contiene gli identificatori di tutte le transazioni attive
- Si riprende l'accettazione delle operazioni da parte delle transazioni

Dump

Copia completa della base di dati memorizzata su memoria stabile (backup)

- creata quando il sistema non è operativo
  - in mutua esclusione con tutte le altre transazioni
- tipicamente effettuata su nastro

Alla conclusione del dump viene scritto nel log un record di dump (dump) che segnala che si è fatto un backup ad un certo istante di tempo e che identifica la copia

Scrittura dei record di log

Deve obbedire a due regole:

- Write Ahead Log
- Commit-Precedenza

Write Ahead Log

La parte before state dei record di log deve essere scritta nel log prima di effettuare la corrispondente operazione sulla base di dati

- consente undo delle scritture effettuate da transazioni che non hanno fatto commit
- consente ripristino in caso di guasto dopo aver effettuato l'operazione sulla base di dati
  - se il log non venisse scritto prima il valore precedente verrebbe perso

Commit-Precedenza

La parte after state dei record di log deve essere scritta nel log prima di effettuare il commit

- consente redo delle scritture già decise da transazioni che hanno fatto commit, ma le cui pagine modificate non sono ancora state scritte dal buffer manager su memoria secondaria

Scrittura dei record di log

Anche se le regole fanno riferimento a before state e after state in pratica le componenti del record di log vengono scritte nello stesso momento

Versione semplificata delle regole chiede che i log siano scritti:

- prima dei corrispondenti record della base di dati
- prima di effettuare il commit

Record di commit

Scritto, in modo sincrono (force), nel record di log dalle transazioni che scelgono di terminare con successo

- guasto prima del commit
  - *undo* delle azioni effettuate e ripristino dello stato iniziale della base di dati
- guasto dopo il commit
  - *redo* delle azioni per ricostruire lo stato finale della transazione

Record di abort

Definisce la scelta di abortire (prodotta dalle transazioni o dal sistema)

- non modificando le decisioni del gestore dell'affidabilità, può essere:
  - scritto in modo asincrono nel buffer che contiene il blocco corrente del log
  - successivamente riscritto sul log in modo asincrono (flush)

Scrittura congiunta: log e base di dati (1)

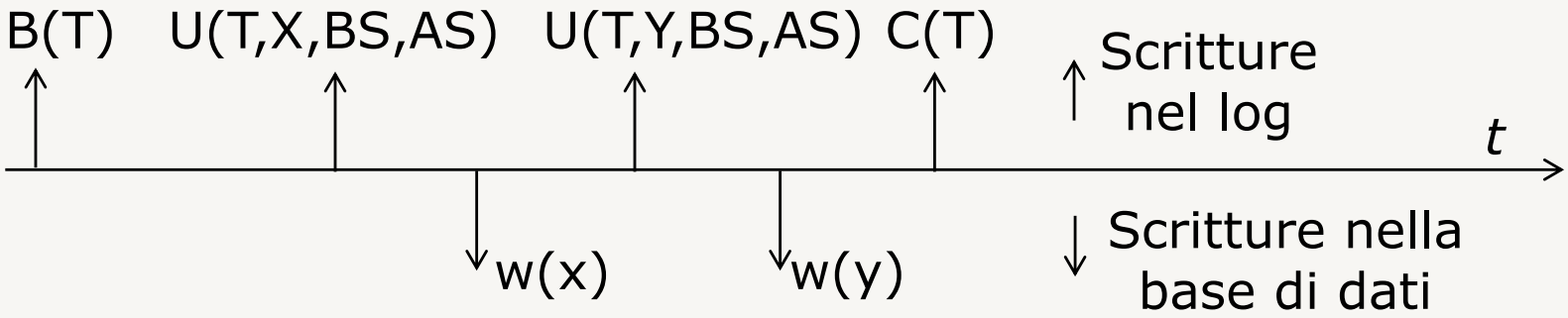
Distinguiamo tre schemi, a seconda che le modifiche sulla base di dati da parte di una transazione siano effettuate (force da buffer manager)

- prima del commit
- dopo il commit
- alcune prima e alcune dopo il commit

Scrittura congiunta: log e base di dati (2)

Base di dati modificata prima del commit

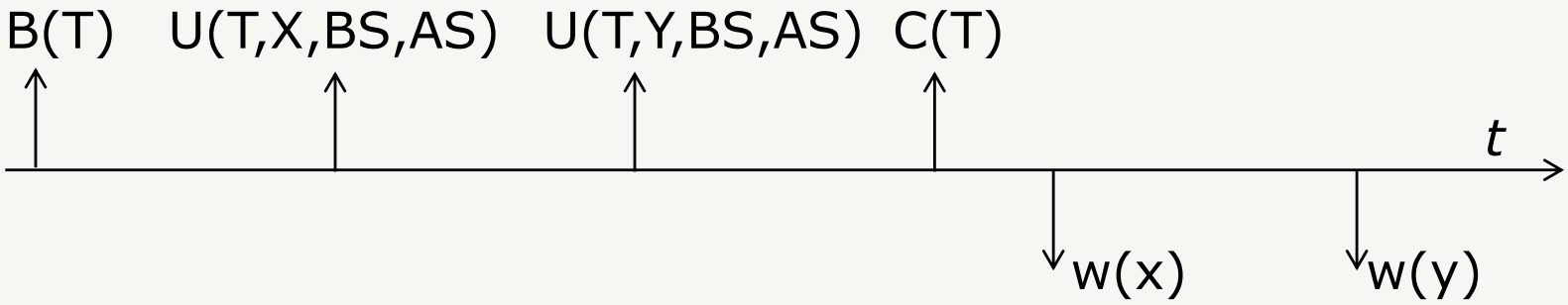
- non richiede operazioni di redo



Scrittura congiunta: log e base di dati (3)

Base di dati modificata dopo il commit

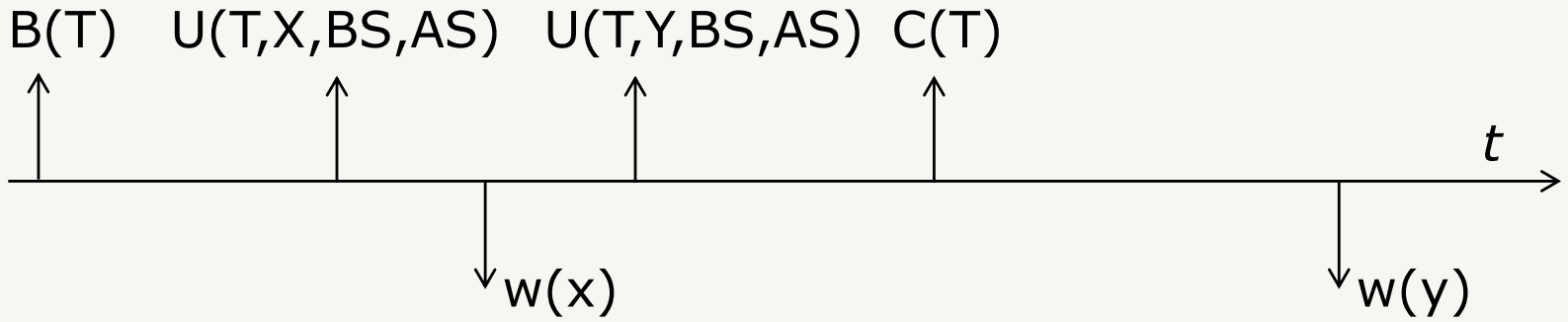
- non richiede operazioni di undo



Scrittura congiunta: log e base di dati (4)

Base di dati modificata in qualunque momento (prima e dopo) il commit

- richiede sia operazioni di redo sia operazioni di undo
- è quello più comunemente utilizzato perché consente al gestore del buffer di ottimizzare le operazioni di flush indipendentemente dal gestore dell'affidabilità



Guasti

Due classi

- guasti di sistema: bug del software (es., del sistema operativo) o interruzioni del funzionamento dei dispositivi (es., calo di tensione)

- guasti di dispositivo: guasti ai dispositivi di gestione della memoria di massa (es., strisciamento delle testine del disco)

Guasti di sistema

Bug del software (es., del sistema operativo) o interruzioni del funzionamento dei dispositivi (es., calo di tensione)

- il contenuto della memoria centrale viene perso (e così tutti i buffer)
- il contenuto della memoria secondaria non viene perso

⇒ **ripresa a caldo**

Guasti di dispositivo

Guasti ai dispositivi di gestione della memoria di massa (es., strisciamento delle testine del disco)

- il contenuto della memoria centrale viene perso (e così tutti i buffer)
- il contenuto della memoria secondaria viene perso
- il contenuto della memoria stabile non viene perso

⇒ **ripresa a freddo**

Gestione dei guasti

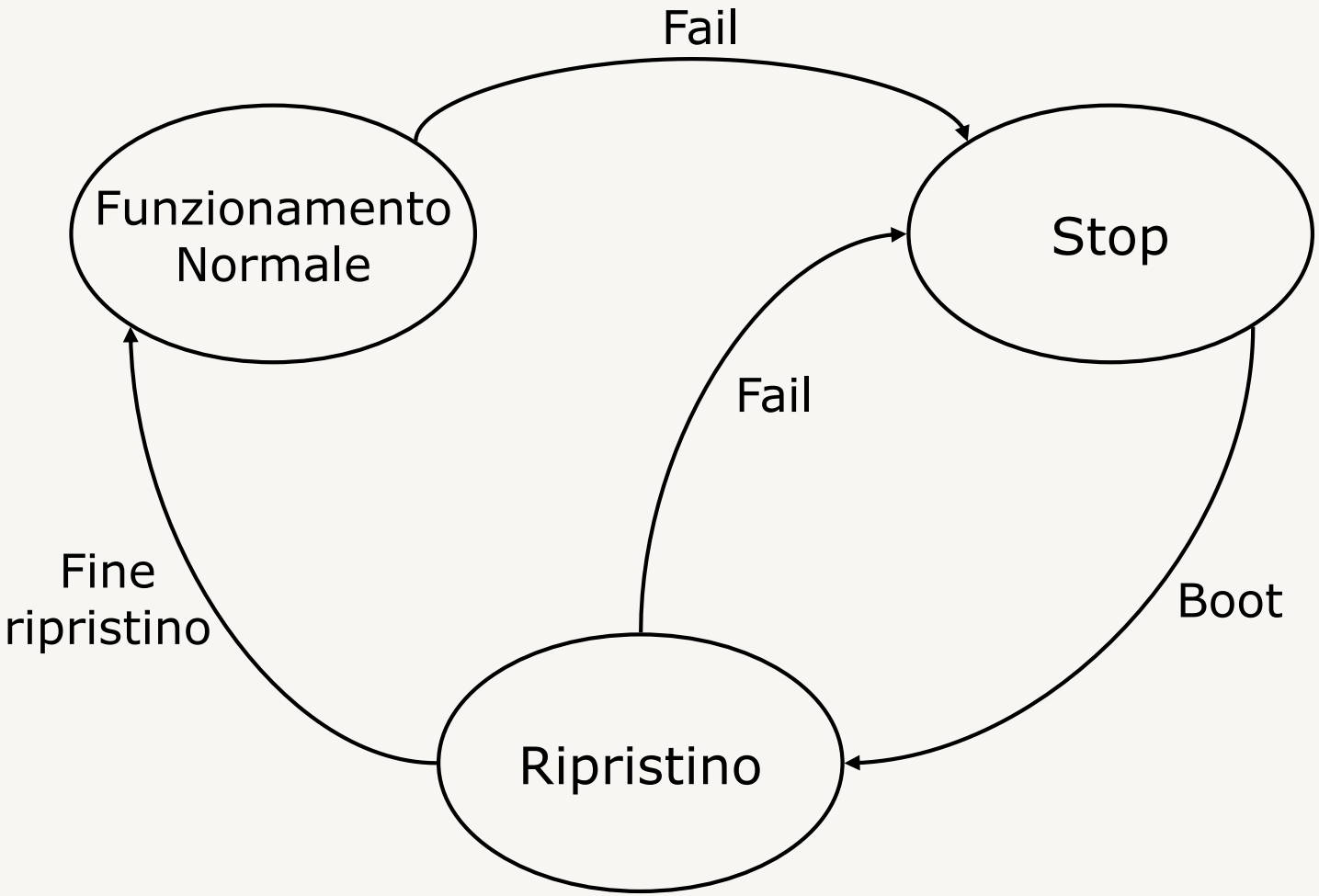
Modello fail-stop: se il sistema individua un guasto

- forza arresto completo delle transazioni
- forza corretto funzionamento del sistema operativo (boot)
- effettua ripresa
  - a caldo (per guasti di sistema)
  - a freddo (per guasti di dispositivo)

$\Rightarrow$ diventa nuovamente utilizzabile

- buffer vuoto

Modello fail-stop



Ripresa: classificazione di transazioni

Rispetto ad un guasto ci sono due classi di transazioni

- committed
  - le loro azioni devono essere rieseguite (redo)
- non committed
  - le loro azioni devono essere disfate (undo)

Ripresa a caldo (1)

Fase 1: determinare l'insieme delle transazioni da disfare (UNDO) e rifare (REDO)

- percorrere il log all'indietro fino al record di checkpoint più recente
- inizializzare UNDO e REDO
  - UNDO := transazioni nel checkpoint
  - REDO := $\emptyset$
- percorrere il log avanti:
  - per ogni B(T) $\Rightarrow$ UNDO := UNDO $\cup$ {T}
  - per ogni C(T) $\Rightarrow$ UNDO := UNDO – {T}
REDO := REDO $\cup$ {T}

Ripresa a caldo (2)

Fase 2: ripristino

- percorrere il log all'indietro
  - per ogni A delle transazioni in UNDO $\Rightarrow$ `undo`(A)

  arrivare fino alla prima azione della transazione più vecchia in UNDO $\cup$ REDO

- percorrere il log in avanti
  - per ogni A delle transazioni in REDO $\Rightarrow$ `redo`(A)

Ripresa a caldo (3)

Garantisce:

- atomicità: tutte le transazioni in corso al momento del guasto lasciano la base di dati nello stato finale o iniziale
- persistenza: tutte le pagine delle transazioni in corso sono scritte su memoria secondaria

Ripresa a caldo: esempio (1)

B(T1), B(T2), I(T2,O1,A1), B(T3), I(T3,O2,A2), D(T1,O3,B3), B(T4), U(T3,O2,B4,A4), I(T4,O4,A5), U(T4,O2,B6,A6), C(T2), CK(T1,T3,T4), C(T4), B(T5), D(T5,O4,B7), U(T1,O2,B8,A8), A(T3), C(T1), guasto

- determina UNDO e REDO

	UNDO	**REDO**
CK	T1,T3,T4	
C(T4)	T1,T3	T4
B(T5)	T1,T3,T5	T4
C(T1)	T3,T5	T4,T1

Ripresa a caldo: esempio (2)

B(T1), B(T2), I(T2,O1,A1), B(T3), I(T3,O2,A2), D(T1,O3,B3), B(T4), U(T3,O2,B4,A4), I(T4,O4,A5), U(T4,O2,B6,A6), C(T2), CK(T1,T3,T4), C(T4), B(T5), D(T5,O4,B7), U(T1,O2,B8,A8), A(T3), C(T1), guasto

- UNDO {T3,T5}

record	azione
D(T5,O4,B7)	insert O4, O4:=B7
U(T3,O2,B4,A4)	O2:=B4
I(T3,O2,A2)	delete O2

Ripresa a caldo: esempio (3)

B(T1), B(T2), I(T2,O1,A1), B(T3), I(T3,O2,A2), D(T1,O3,B3), B(T4), U(T3,O2,B4,A4), I(T4,O4,A5), U(T4,O2,B6,A6), C(T2), CK(T1,T3,T4), C(T4), B(T5), D(T5,O4,B7), U(T1,O2,B8,A8), A(T3), C(T1), guasto

- REDO {T1,T4}

record	azione
D(T1,O3,B3)	delete O3
I(T4,O4,A5)	insert O4, O4:=A5
U(T4,O2,B6,A6)	O2:=A6
U(T1,O2,B8,A8)	O2:=A8

Ripresa a freddo

Fase 1: ripristinare la base di dati come al momento del guasto

- accedere al dump e ricopiare selettivamente tutte le parti deteriorate della base di dati
- accedere al record di dump più recente registrato nel log
- percorrere il log in avanti applicando, relativamente alla parte deteriorata, sia le azioni della base di dati sia le azioni di commit/abort

Fase 2:

- eseguire ripresa a caldo

Controllo della concorrenza (1)

Permette esecuzione concorrente di più transazioni

- cruciale in sistemi informativi con alto carico
  - banche, finanziarie, sistemi di prenotazioni aeree
- consente un uso efficiente del DBMS
  - massimizzando il numero di transazioni servite
  - minimizzando i tempi di risposta
- carico applicativo misurato in transazioni per secondo (tps)
  - valori tipici: da decine a migliaia

Controllo della concorrenza (2)

Media le richieste di accesso ai dati

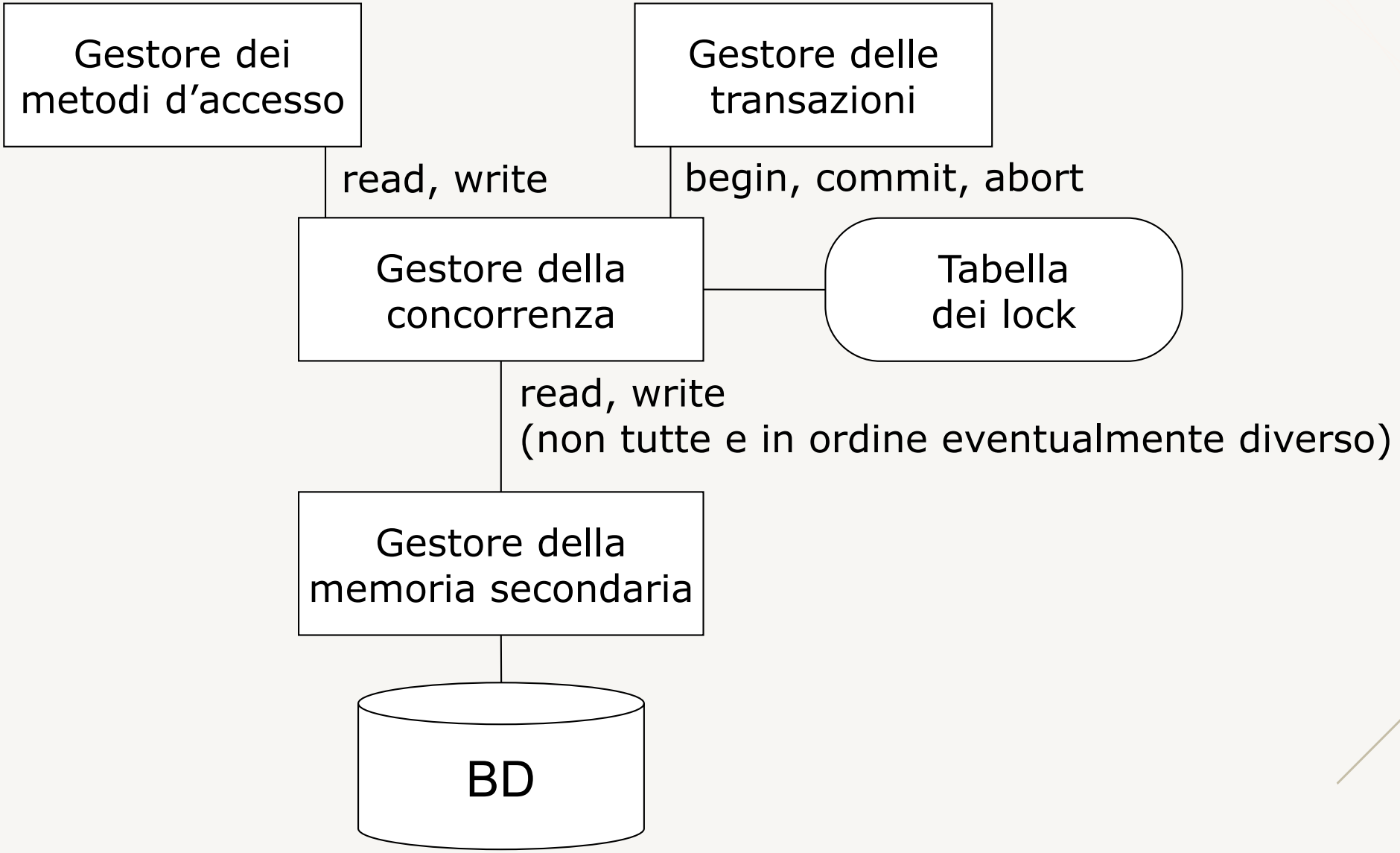
- decide se autorizzarle o meno
- stabilisce l’ordine degli accessi (scheduler)

Controllo della concorrenza (3)

Per lo studio facciamo alcune semplificazioni:

- basi di dati in termini di oggetti astratti *x*, *y*, *z* con valori numerici
- operazioni di lettura/scrittura su un dato *x*, come read/write dell'intera pagina in cui *x* è memorizzato

Controllo di concorrenza: architettura



Controllo della concorrenza (4)

L’esecuzione concorrente di più transazioni può portare problemi di correttezza e anomalie

- perdita di aggiornamenti
- lettura sporca
- letture inconsistenti
- aggiornamento fantasma
- inserimento fantasma

Perdita di aggiornamenti

Modifiche di una transazione perse perché sovrascritte da una transazione concorrente

Transazione t_1	Transazione t_2
bot	
$r_1(x)$	
x := x + 1	
	bot
	$r_2(x)$
	x := x + 1
$w_1(x)$	
commit	
	$w_2(x)$
	commit

Valore iniziale: 2
Valore finale: 3 invece di 4 (la modifica di t_1 viene persa)

Lettura sporca

Una transazione legge il risultato intermedio di un'altra transazione che poi viene abortita (e la cui modifica viene quindi annullata)

Transazione t_1	Transazione t_2
bot	
$r_1(x)$	
x := x + 1	
$w_1(x)$	
	bot
	$r_2(x)$
	x := x + 1
abort	
	$w_2(x)$
	commit

t_2 legge uno stato intermedio poi annullato

Letture inconsistenti

Una transazione legge oggetti che un'altra transazione sta modificando: alcune letture sono prima, altre dopo le modifiche

Transazione t_1	Transazione t_2
bot	
$r_1(x)$	
$w_1(y)$	
	bot
	$r_2(x)$
	x := x + 1
	$w_2(x)$
	commit
$r_1(x)$	
$w_1(z)$	
commit	

t_1 legge valori differenti per x

Aggiornamento fantasma

Una transazione osserva solo parte degli effetti di un'altra transazione (osservando uno stato dei dati che non soddisfa i vincoli di integrità)

Transazione t_1	Transazione t_2
bot	
$r_1(x)$	
	bot
	$r_2(y)$
$r_1(y)$	
	y := y - 100
	$r_2(z)$
	z := z + 100
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
s := x + y + z	
commit	

Vincolo: $x+y+z=1000$; $s=1100$ ma vincolo è soddisfatto

Inserimento fantasma

Una transazione valuta due volte un valore aggregato relativo ad un insieme di elementi di un predicato di selezione

Esempio

```
select avg(Voto)
from Esame
where Corso='Basidati'
```

Se tra una valutazione e l'altra viene inserita una nuova tupla, i due risultati possono differire

L'anomalia non può essere evitata solo facendo riferimento ai dati già presenti

Teoria del controllo della concorrenza

Formalmente una transazione

- è una sequenza di operazioni di lettura e scrittura
- ha un identificatore della transazione assegnato dal sistema
- inizia con `begin transaction` e finisce con `end transaction` (omessi)

Esempio

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

Schedule

Sequenza di operazioni di input/output
presentate da transazioni concorrenti

- tutte le operazioni di ogni transazione che ha fatto commit devono apparire nello schedule
- per ogni transazione le operazioni devono apparire nello schedule con lo stesso ordine in cui appaiono nella transazione

Per lo studio:

- consideriamo la commit-proiezione e ignoriamo le transazioni che abortiscono
  (assunzione semplificativa, non accettabile in pratica la rimuoveremo poi)

Schedule: esempio

Transazioni

- t_1 : $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$
- t_2 : $r_2(y)$ $w_2(y)$

alcuni (fra i) possibili schedule

- S_1 : $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$ $r_2(y)$ $w_2(y)$
- S_2 : $r_2(y)$ $w_2(y)$ $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$
- S_3 : $r_1(x)$ $r_2(y)$ $w_1(x)$ $r_1(y)$ $w_2(y)$ $w_1(y)$
- S_4 : $r_2(y)$ $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$ $w_2(y)$

Controllo di concorrenza

- evita schedule che causano anomalie
- gestito da un modulo che accetta o rifiuta le operazioni richieste dalle transazioni (scheduler)
- basato sull'identificazione di classi di schedule accettabili sulla base di definizioni di equivalenza
  - schedule seriale
  - schedule serializzabile

Schedule seriali e serializzabili

Schedule seriale

- per ogni transazione t_i nello schedule, tutte le operazioni di t_i sono eseguite consecutivamente
  - n transazioni $\Rightarrow$ $n!$ schedule seriali possibili

Schedule serializzabile

- schedule non seriale che produce lo stesso risultato di un qualche schedule seriale delle stesse transazioni
  - richiede nozioni di equivalenza fra schedule
  - concetti progressivi: view-equivalenza, conflict-equivalenza, two-phase locking, timestamp-based

Schedule seriali: esempio

Transazioni

- t_1 : $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$
- t_2 : $r_2(y)$ $w_2(y)$

schedule seriali:

- S_1 : $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$ $r_2(y)$ $w_2(y)$
- S_2 : $r_2(y)$ $w_2(y)$ $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$

altri fra i possibili schedule:

- S_3 : $r_1(x)$ $r_2(y)$ $w_1(x)$ $r_1(y)$ $w_2(y)$ $w_1(y)$
- S_4 : $r_2(y)$ $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$ $w_2(y)$

View-serializzabilità (1)

Definizioni preliminari

- $r_i(x)$ legge-da $w_j(x)$ in uno schedule S se:
  - $w_j(x)$ precede $r_i(x)$ in S
  - non c'è alcun $w_k(x)$ tra $w_j(x)$ e $r_i(x)$ in S
- $w_i(x)$ è una scrittura finale in uno schedule S se:
  - è l'ultima scrittura su x in S

Esempio

S: $r_1(x)$ $w_1(x)$ $w_1(y)$ $r_2(x)$ $w_2(y)$

- $r_2(x)$ legge-da $w_1(x)$
- $w_1(x)$ scrittura finale su x
- $w_2(y)$ scrittura finale su y

View-serializzabilità (2)

View-equivalenza

- due schedule S_i e S_j (con $i \neq j$) sono view-equivalenti ($S_i \equiv_v S_j$) se hanno
  - le stesse relazioni legge-da
  - le stesse scritture finali

View-serializzabilità

- uno schedule è view-serializzabile se è
  - view-equivalente a un qualche schedule seriale

Indichiamo con VSR l'insieme di schedule view-serializzabili

View-serializzabilità: esempio (1)

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

- legge-da:
  - $r_2(x) \leftarrow w_0(x)$
  - $r_1(x) \leftarrow w_0(x)$
- scritture finali:
  - x: $w_2(x)$
  - z: $w_2(z)$
- view-equivalente allo schedule seriale
  $w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$

$\Rightarrow$ **view-serializzabile**

View-serializzabilità: esempio (2)

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

- legge-da:
  - $r_1(x) \leftarrow w_0(x)$
  - $r_2(x) \leftarrow w_1(x)$
- scritture finali:
  - x: $w_1(x)$
  - z: $w_1(z)$
- view-equivalente allo schedule seriale
  $w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$

$\Rightarrow$ **view-serializzabile**

View-serializzabilità: esempio (3)

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

- legge-da:
  - $r_1(x) \leftarrow$
  - $r_2(x) \leftarrow$

- scritture finali:
  - x: $w_1(x)$

$\Rightarrow$ **non è view-serializzabile**
(perdita di update)

View-serializzabilità: esempio (4)

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

- legge-da:
  - $r_1(x) \leftarrow$
  - $r_2(x) \leftarrow$
  - $r_1(x) \leftarrow w_2(x)$

- scritture finali:
  - x: $w_2(x)$

$\Rightarrow$ **non è view-serializzabile**
(letture inconsistenti)

View-serializzabilità: esempio (5)

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

- legge-da:
  - $r_1(x) \leftarrow$
  - $r_1(y) \leftarrow$
  - $r_2(z) \leftarrow$
  - $r_2(y) \leftarrow$
  - $r_1(z) \leftarrow w_2(z)$
- scritture finali:
  - y: $w_2(y)$
  - z: $w_2(z)$

$\Rightarrow$ **non è view-serializzabile**
(aggiornamento fantasma)

View-serializzabilità: complessità

- Decidere se due schedule sono view-equivalenti: costo polinomiale

- Decidere la view-serializzabilità di uno schedule generico: problema NP-difficile
  - è necessario confrontare lo schedule con tutti i possibili schedule seriali

Nota: view-equivalenza inutilizzabile in pratica data la complessità

- è necessario definire condizioni più restrittive ma pratiche

Conflitto

Conflitto fra operazioni

- due operazioni a_i e a_j ($i \neq j$) sono in conflitto se
  - appartengono a due diverse transazioni
  - accedono allo stesso oggetto
  - almeno una è una scrittura

Due casi:

- conflitti read-write (rw o wr)
- conflitti write-write (ww)

Conflitto: esempi

t_1: $r_1(x)$ $r_1(y)$ $w_1(x)$ $w_1(y)$
t_2: $r_2(y)$ $r_2(x)$ $w_2(y)$

Conflitti:

- $w_1(x)$, $r_2(x)$
- $w_1(y)$, $r_2(y)$
- $w_1(y)$, $w_2(y)$
- $r_1(y)$, $w_2(y)$

Conflict-serializzabilità

Conflict-equivalenza

- due schedule S_i e S_j ($i \neq j$) sono conflict-equivalenti ($S_i \equiv_C S_j$) se
  - contengono le stesse operazioni
  - ogni coppia di operazioni in conflitto appare nello stesso ordine in entrambi gli schedule

Conflict-serializzabilità

- uno schedule è conflict-serializzabile se è
  - conflict-equivalente a un qualche schedule seriale

Indichiamo con CSR l'insieme di schedule conflict-serializzabili

Conflict-serializzabilità: esempio

S: $w_0(x)\ r_1(x)\ w_0(z)\ r_1(z)\ r_2(x)\ r_3(z)\ w_3(z)\ w_1(x)$

- conflitti:
  - $w_0(x)$, $r_1(x)$
  - $w_0(x)$, $r_2(x)$
  - $w_0(x)$, $w_1(x)$
  - $w_0(z)$, $r_1(z)$
  - $w_0(z)$, $r_3(z)$
  - $w_0(z)$, $w_3(z)$
  - $r_1(z)$, $w_3(z)$
  - $r_2(x)$, $w_1(x)$

Conflict-equivalente allo schedule seriale

$w_0(x)\ w_0(z)\ r_2(x)\ r_1(x)\ r_1(z)\ w_1(x)\ r_3(z)\ w_3(z)$

Conflict-serializzabilità: grafo dei conflitti

Può essere valutata costruendo il grafo dei conflitti:

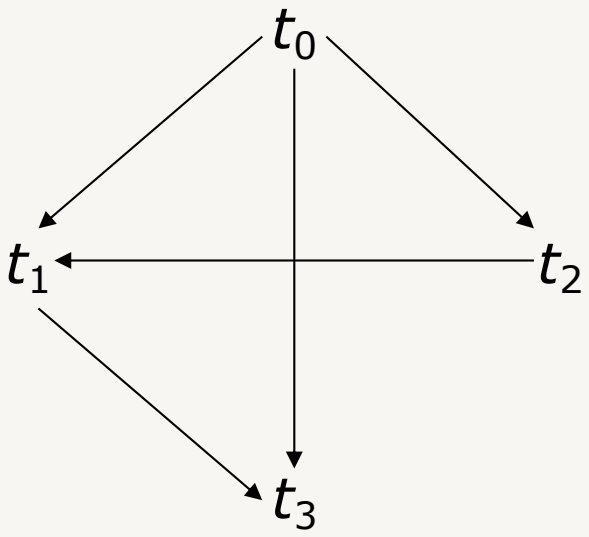
- ogni transazione è rappresentata da un nodo
- per ogni coppia di operazioni a_i e a_j in conflitto tali che a_i precede a_j nello schedule, viene definito un arco da t_i (transazione di a_i) a t_j (transazione di a_j)

Uno schedule è conflict-serializzabile se e solo se il grafo è aciclico: è conflict-equivalente a tutti gli ordinamenti topologici del grafo

Grafo dei conflitti: esempio

S: $w_0(x)\ r_1(x)\ w_0(z)\ r_1(z)\ r_2(x)\ r_3(z)\ w_3(z)\ w_1(x)$

- conflitti:
  - $w_0(x)$, $r_1(x)$
  - $w_0(x)$, $r_2(x)$
  - $w_0(x)$, $w_1(x)$
  - $w_0(z)$, $r_1(z)$
  - $w_0(z)$, $r_3(z)$
  - $w_0(z)$, $w_3(z)$
  - $r_1(z)$, $w_3(z)$
  - $r_2(x)$, $w_1(x)$

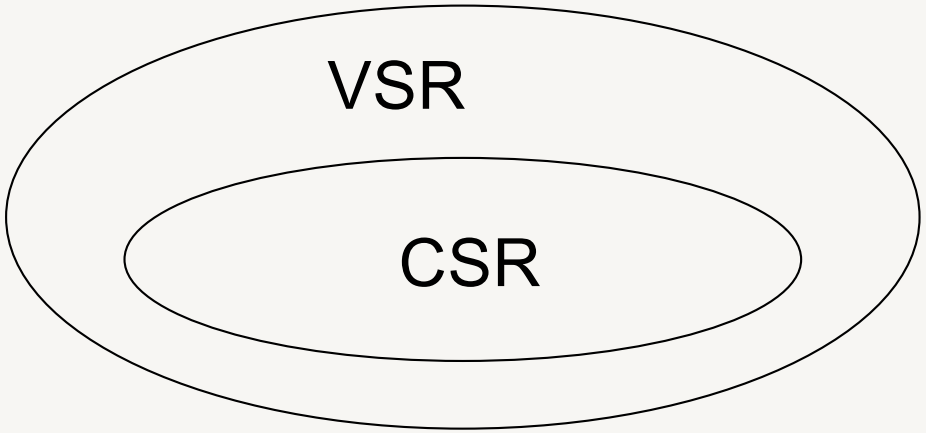


Conflict-equivalente allo schedule seriale

$w_0(x)\ w_0(z)\ r_2(x)\ r_1(x)\ r_1(z)\ w_1(x)\ r_3(z)\ w_3(z)$

CSR vs VSR

CSR $\subseteq$ VSR



- $S \in$ CSR $\Rightarrow S \in$ VSR
- $S \notin$ VSR $\Rightarrow S \notin$ CSR

- $S \notin$ CSR $\Rightarrow$ non sappiamo dire nulla se VSR
- $S \in$ VSR $\Rightarrow$ non sappiamo dire nulla se CSR

CSR vs VSR: esempio

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

$t_2 : w_2(x)$ blind write

$t_3 : w_3(x)$ blind write

$S : r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$

- S è view-serializzabile
  - view-equivalente a t_1, t_2, t_3
- S non è conflict-serializzabile

Conflict-serializzabilità: complessità

- decidere se uno schedule è conflict-serializzabile ha un costo lineare nella dimensione del grafo
- comunque troppo costosa in pratica, specialmente nel caso di dati distribuiti
- meccanismi adottati in pratica usano condizioni ancora più restrittive ma efficienti

Controllo della concorrenza

I sistemi utilizzati in pratica non valutano la serializzabilità a posteriori ma operano in modo da garantirla

Possibili tecniche:

- locking a due fasi
- timestamp
  - monoversione
  - multiversione

Lock

Variabile che descrive lo stato di un dato rispetto alle operazioni che possono essere eseguite sul dato

- lock due stati (binario)
  - unlocked (0)
  - locked (1)
- lock a tre stati:
  - unlocked
  - r_locked (in lettura, condiviso)
  - w_locked (in scrittura, esclusivo)

Locking: due stati

Regole

- per accedere ad un dato una transazione deve chiedere un lock sul dato
- la transazione rilascia il lock (unlock) quando il dato non le serve più
- una transazione può accedere ad un dato solo se ha ottenuto il lock relativo

Una transazione che segue queste regole è detta ben formata rispetto al locking

Locking: tre stati

Regole

- per accedere in lettura ad un dato una transazione deve chiedere un r_lock (condiviso)
- per accedere in scrittura ad un dato una transazione deve chiedere un w_lock (esclusivo)
- la transazione rilascia il lock (unlock) quando il dato non le serve più
- una transazione può accedere ad un dato solo se ha ottenuto il lock relativo

Una transazione che segue queste regole è detta ben formata rispetto al locking

Gestione dei lock

Modulo che funziona da gestore dei lock

- riceve le richieste di lock
- le concede o meno a seconda delle tabelle dei conflitti
  - se la richiesta non è concessa, la transazione richiedente è posta in attesa fino a che il lock le viene concesso
- modifica opportunamente le tabelle di lock (stato e eventuale contatore)

Tabella dei conflitti: lock a due stati

	Stato	
Richiesta	**unlocked**	**locked**
lock	OK locked	no locked
unlock	errore	OK unlocked

Tabella dei conflitti: lock a tre stati

	Stato		
Richiesta	**unlocked**	**r_locked**	**w_locked**
r_lock	OK r_locked	OK (c=c+1) r_locked	no w_locked
w_lock	OK w_locked	no* r_locked	no w_locked
unlock	errore	OK (c=c-1) c=0: unlocked c>0: r_locked	OK unlocked

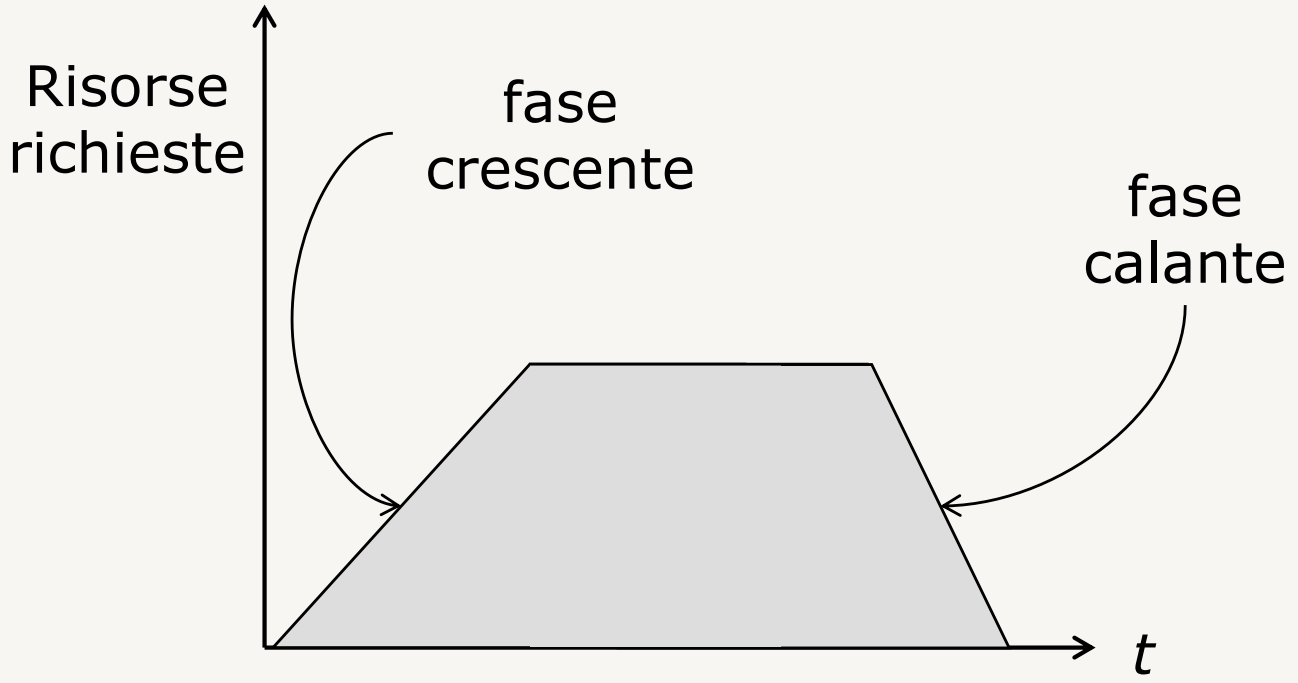
*: a meno di upgrade di un r_lock tenuto in modo esclusivo

Locking a due fasi

Una transazione, dopo aver rilasciato un lock, non può acquisirne altri

- ogni transazione ha due fasi
  - crescente (acquisce i lock)
  - decrescente (rilascia i lock)

Garantisce serializzabilità



Schedule 2PL

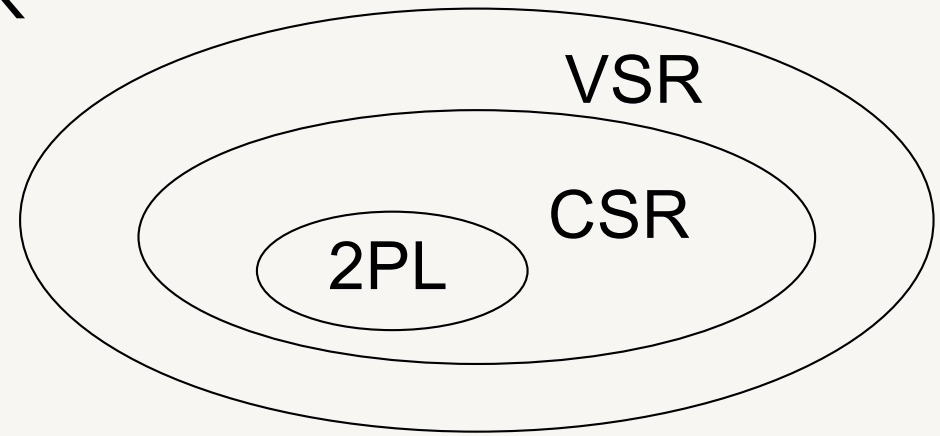
Gli schedule prodotti da uno scheduler che usa

- transazioni ben formate rispetto al locking
- gestione dei lock basata sui conflitti
- locking a due fasi

appartengono alla classe 2PL

2PL vs CSR vs VSR

$2PL \subseteq CSR \subseteq VSR$



- $S \in 2PL \Rightarrow S \in CSR$
- $S \notin CSR \Rightarrow S \notin 2PL$

- $S \notin 2PL \Rightarrow$ non sappiamo dire nulla se CSR
- $S \in CSR \Rightarrow$ non sappiamo dire nulla se 2PL

2PL vs CSR: esempio

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

- è conflict-serializzabile
  - conflict-equivalente a t_3, t_1, t_2
- non è 2PL

Locking a due fasi stretto

Lo scheduler opera senza sapere come finiranno le transazioni, quindi:

- dobbiamo rimuovere l'ipotesi di utilizzare una commit-proiezione
- aggiungiamo una restrizione al 2PL

2PL stretto (utilizzato dai DBMS commerciali)

- una transazione, dopo aver rilasciato un lock, non può acquisirne altri
- i lock di una transazione possono essere rilasciati solo dopo le operazioni di commit/abort

Variazioni del 2PL

- 2PL base
  - una transazione dopo aver rilasciato un lock non può acquisirne altri
- 2PL stretto (elimina possibilità di letture sporche)
  - 2PL base
  - una transazione può rilasciare i lock solo dopo il commit/abort
- 2PL conservativo (garantisce no deadlock)
  - 2PL base
  - una transazione deve acquisire tutti i lock prima di iniziare le operazioni

Controllo della concorrenza

I sistemi utilizzati in pratica non valutano la serializzabilità a posteriori ma operano in modo da garantirla

Possibili tecniche:

- locking a due fasi
- timestamp:
  - monoversione
  - multiversione

Timestamp

Identificatore che definisce un ordinamento totale tra gli eventi temporali del sistema

- un evento ha timestamp maggiore di quello degli eventi che lo hanno preceduto
- non ci sono due eventi con timestamp uguale

Possibili implementazioni:

- contatore
- clock di sistema

Transazioni e timestamp (1)

Ogni transazione ha associato un timestamp (ts)

- assegnatole dal sistema all'inizio della transazione
- rappresenta il tempo al quale la transazione è iniziata

Uno schedule è accettato solo se riflette l'ordine seriale delle transazioni basato sul valore del timestamp delle transazioni

Timestamp monoversione (1)

Ogni oggetto x ha associato due indicatori:

- RTM(x): max timestamp fra le transazioni che hanno letto x
- WTM(x): max timestamp fra le transazioni che hanno scritto x

Timestamp monoversione (2)

Risposta dello scheduler alle richieste

- *read*(*x*, *ts*)
  - *ts* < WTM(*x*)
    - richiesta è rifiutata, transazione abortita
  - altrimenti
    - richiesta è accettata; RTM(*x*) := max(RTM(*x*),*ts*)
- *write*(*x*, *ts*)
  - *ts* < WTM(*x*) o *ts* < RTM(*x*)
    - richiesta è rifiutata, transazione abortita
  - altrimenti
    - richiesta è accettata; WTM(*x*) := *ts*

Timestamp monoversione: esempio

Richiesta	Risposta	RTM(x)	WTM(x)
		2	2
read(x,6)	ok	6	=2
read(x,8)	ok	8	=2
read(x,9)	ok	9	=2
write(x,8)	no (t_8 è uccisa)	=9	=2
write(x,11)	ok	=9	11
read(x,10)	no (t_{10} è uccisa)	=9	=11

Controllo con timestamp

Vantaggi

- non viene richiesto alcun lock sulle risorse
  $\Rightarrow$ libero da deadlock

Svantaggi

- può forzare l'uccisione di molte transazioni
- si comporta correttamente solo con l'assunzione di commit-proiezione
  - per rimuoverla è necessario bufferizzare le scritture (scriverle su memoria secondaria solo dopo il commit)
  - letture di dati bufferizzati saranno messe in attesa del commit della transazione scrivente

Timestamp e multiversione

Ogni scrittura di un oggetto ne genera una nuova copia (versione)

- letture richieste da transazioni 'vecchie' vengono eseguite su vecchie versioni (anziché essere rifiutate e le transazioni abortite)

Per ogni oggetto x:

- esistono diverse versioni x_i ognuna con proprio RTM e WTM
- ogni scrittura eseguita genera una nuova versione
- ogni lettura è eseguita sulla versione creata dall'ultima scrittura che precede la lettura

Transazioni e timestamp con multiversione

Risposta dello scheduler alle richieste

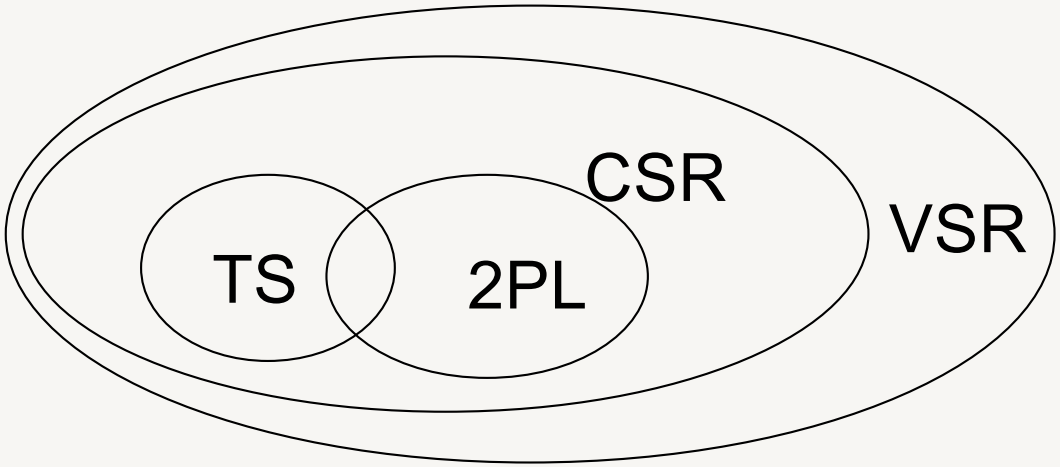
- *read*(*x*, *ts*)
  - sempre accettata
  - leggi la versione x_k tale che
    - $WTM(x_k) = max\{WTM(x_i) \mid WTM(x_i) \leq ts\}$
  - $RTM(x_k) := max(RTM(x_k), ts)$
- *write*(*x*, *ts*)
  - trova la versione x_j tale che
    - $WTM(x_j) = max\{WTM(x_i) \mid WTM(x_i) \leq ts\}$
  - $ts < RTM(x_j)$
    - richiesta è rifiutata, transazione abortita
  - altrimenti
    - richiesta è accettata; crea nuova versione x_k
    - $RTM(x_k) := ts$; $WTM(x_k) := ts$

Timestamp e multiversione: esempio

Richiesta	Risposta	x_k	RTM(x_k)	WTM(x_k)
		x_2	2	2
read(x,5)	ok	x_2	5	=2
read(x,8)	ok	x_2	8	=2
write(x,7)	no (t_7 è uccisa)			
write(x,12)	ok	x_{12}	12	12
read(x,9)	ok	x_2	9	=2
read(x,14)	ok	x_{12}	14	=12
write(x,13)	no (t_{13} è uccisa)			
write(x,10)	ok	x_{10}	10	10
read(x,11)	ok	x_{10}	11	=10

TS vs 2PL vs CSR vs VSR

TS $\subseteq$ CSR



- $S \in$ TS $\Rightarrow$ $S \in$ CSR
- $S \notin$ CSR $\Rightarrow$ $S \notin$ TS

- $S \notin$ TS $\Rightarrow$ non sappiamo dire nulla se CSR
- $S \in$ CSR $\Rightarrow$ non sappiamo dire nulla se TS

2PL vs TS

	2PL	TS
transazioni con azioni rifiutate	poste in attesa	uccise e fatte ripartire
ordine di serializzazione	imposto da conflitti	imposto dai timestamp
attesa per il commit	sì 2PL stretto	sì bufferizzazione dei write
deadlock	sì	no

È più costoso rieseguire le transazioni che metterle in attesa

$\Rightarrow$ meglio 2PL

Gestione dei lock

Il gestore dei lock è invocato da tutti i processi che intendono accedere alla base di dati

- richieste:
  - `r_lock(T,x,errcode,timeout)`
  - `w_lock(T,x,errcode,timeout)`
  - `unlock(T,x)`
- se il timeout scade, `errcode` segnala un errore e, generalmente, la transazione fa rollback e riparte

Tabelle di lock

Memorizzano informazione per il gestore dei lock

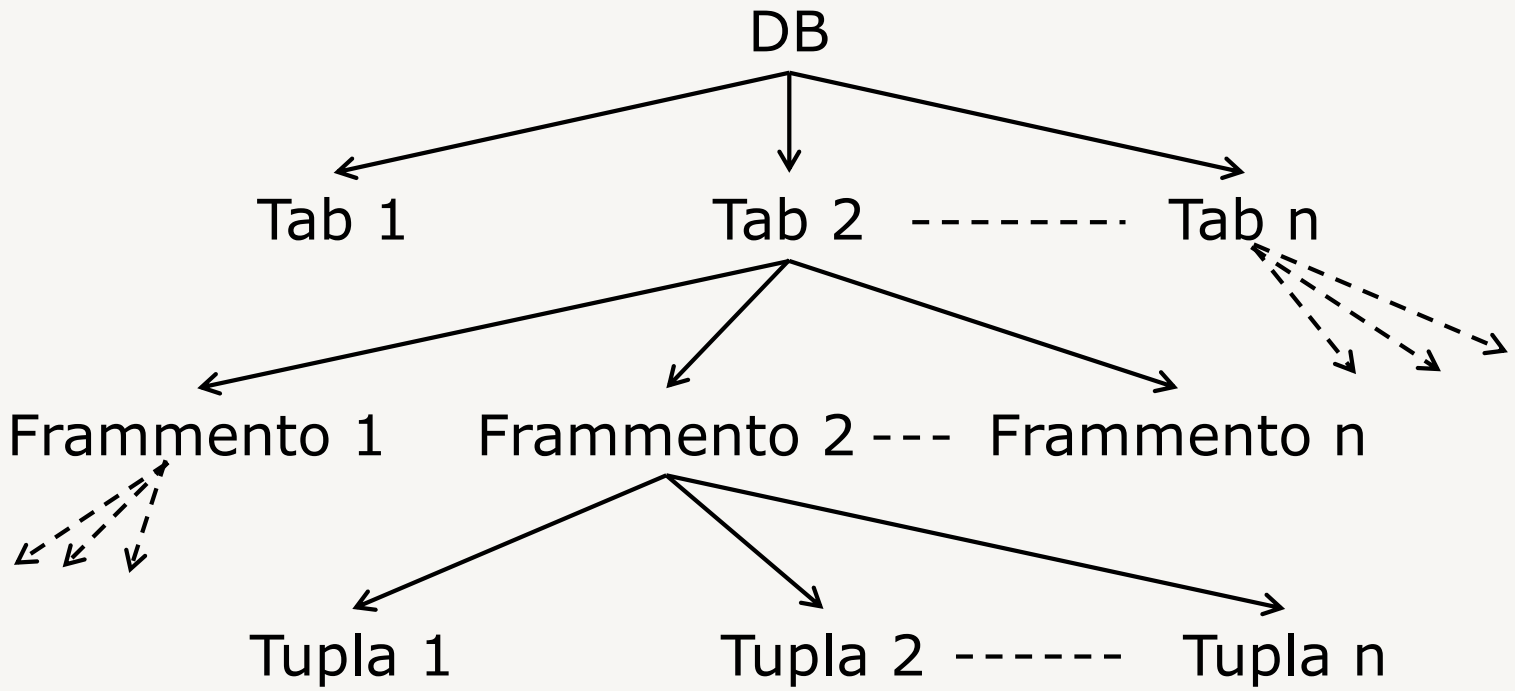
- mantengono, per ciascun oggetto:
  - due bit di stato (per rappresentare i tre stati)
  - contatore di transazioni che condividono r_lock
  - contatore che rappresenta il numero di processi in attesa

- essendo accedute di frequente sono mantenute in memoria centrale

Lock gerarchico (1)

In molti sistemi reali i lock possono essere chiesti su oggetti a granularità differente

- gerarchia di oggetti



Lock gerarchico (2)

- XL (exclusive lock)
  - corrisponde al write lock del protocollo normale
- SL (shared lock)
  - corrisponde al read lock del protocollo normale
- ISL (intentional shared lock)
  - esprime l'intenzione di bloccare in modo condiviso uno dei nodi che discende dal nodo corrente
- IXL (intentional exclusive lock)
  - esprime l'intenzione di bloccare in modo esclusivo uno dei nodi che discende dal nodo corrente
- SIXL (shared intentional-exclusive lock)
  - chiede lock condiviso sul nodo corrente e esprime l'intenzione di bloccare in modo esclusivo uno dei nodi che discende dal nodo corrente

Lock gerarchico: protocollo

- i lock si richiedono dalla radice e scendendo lungo l’albero
- i lock si rilasciano dal nodo locked e salendo lungo l’albero
- una transazione può chiedere un lock SL o ISL su un nodo, solo se ha un lock ISL o IXL sul genitore
- una transazione può chiedere un lock IXL, XL, o SIXL su un nodo, solo se ha già un lock SIXL o IXL sul genitore

Lock gerarchico: esempio

- per chiedere un lock esclusivo XL su una tupla:
  - IXL sul database
  - IXL sulla relazione
  - IXL sulla partizione in cui risiede la tupla (frammento)
  - XL per la tupla
  - i lock vengono poi rilasciati in ordine inverso

Lock gerarchico: granularità

La scelta della granularità alla quale richiedere un lock è lasciata agli sviluppatori delle applicazioni, trade-off

- troppo grande
  - molte risorse sono bloccate
  - limita il parallelismo
- troppo piccola
  - molti lock sono richiesti
  - più lavoro per il gestore dei lock
  - rischi di fallimento dopo aver acquisito molte risorse

Lock in SQL:1999 (1)

Transazioni possono essere classificate come:

- read only
  - non possono modificare il contenuto della base di dati o il suo schema
  - non possono richiedere lock esclusivi
- read write
  - default

Lock in SQL:1999 (2)

Per ogni transazione si può indicare un livello di isolamento:

- read uncommitted
- read committed
- repeatable read
- serializable

Tutti i livelli

- chiedono 2PL stretto per le scritture
  - evitano perdita di aggiornamento
- diversi approcci per le letture

Read uncommitted

Non pone alcun vincolo

- non chiede lock per leggere
- non rispetta i lock esclusivi di altre transazioni
- può presentare tutte le anomalie delle transazioni concorrenti

Read committed

Chiede lock condivisi per effettuare le letture

- esclude le letture di stati intermedi
  - non legge dati non committed
- evita letture sporche
- non garantisce serializzabilità

Repeatable read

Chiede 2PL stretto anche per la lettura

- lock a livello di tupla
- evita tutte le anomalie ad eccezione dell’inserimento fantasma

Serializable

Chiede 2PL stretto anche per la lettura e utilizza lock di predicato

- evita tutte le anomalie
- default

Deadlock (1)

Due transazioni concorrenti sono in attesa (diretta o indiretta) l'una dell'altra

- t_i aspetta che t_j rilasci un lock
- t_j aspetta che t_i rilasci un lock

Il deadlock può interessare più transazioni

- t_1 in attesa per t_2
- t_2 in attesa per t_3
- ……
- t_{n-1} in attesa per t_n
- t_n in attesa per t_1

Deadlock: esempio

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

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

- sequenza richieste:
  - $\text{r\_lock}_1(x)$
  - $\text{r\_lock}_2(y)$
  - $r_1(x)$
  - $r_2(y)$
  - $\text{w\_lock}_1(y)$
  - $\text{w\_lock}_2(x)$

$\Rightarrow$ **deadlock**

Grafo di attesa

- un nodo per ogni transazione
- un arco fra t_i e t_j se t_i è in attesa di una risorsa locked da t_j

C’è un deadlock quando c’è un ciclo nel grafo di attesa

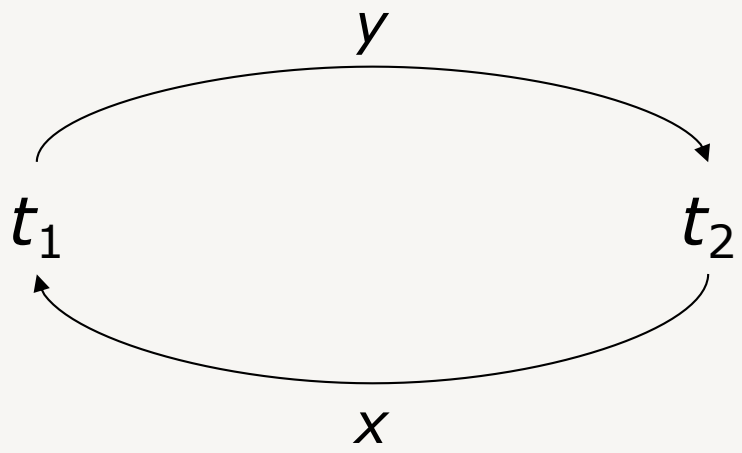
Grafo di attesa: esempio

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

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

- sequenza richieste:
  - $r\_lock_1(x)$
  - $r\_lock_2(y)$
  - $r_1(x)$
  - $r_2(y)$
  - $w\_lock_1(y)$
  - $w\_lock_2(x)$

$\Rightarrow$ **deadlock**



Deadlock: soluzioni

Tre principali tecniche

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

Timeout

Le transazioni rimangono in attesa solo per un tempo massimo prefissato (timeout)

- allo scadere del tempo se la transazione è ancora in attesa
  - viene ritornata risposta negativa
  - la transazione abortita
- scelta del timeout è questione di trade-off
  - troppo basso troppi abort non necessari
  - troppo alto porta ritardi
- è utilizzata nella maggior parte dei DBMS commerciali perché semplice

Rilevamento e risoluzione

Rileva la presenza di cicli nel grafo di attesa

- non pone vincoli al comportamento del sistema ma controlla le tabelle di lock per rilevare deadlock

- diverse scelte per:
  - quando si fa il controllo
  - quale transazione si uccide nel caso di deadlock?

Prevenzione

Evita che si verifichi un deadlock

- utilizzando tecniche preventive che assicurano non vi sarà mai mutua attesa

- uccidendo le transazioni che creerebbero cicli:
  - diverse politiche di scelta della vittima

Tecniche preventive di mutua attesa

- 2PL conservativo: allocazione dei lock all'inizio della transazione
  - può non essere sempre possibile

- ordinamento sugli oggetti: gli oggetti devono essere allocati, da ogni transazione, secondo un ordine stabilito

Uccisione di transazioni

Deadlock evitato uccidendo una delle transazioni che creerebbero cicli

- **preemptive** (**interrompenti**) uccidono la transazione che possiede la risorsa
- **non-preemptive** (**non interrompenti**) uccidono la transazione che richiede la risorsa

Uccisione di transazioni: timestamp (1)

t_i richiede lock su x

t_j ha lock su x

- non interrompente (wait-die)
  - se $ts(t_i) < ts(t_j)$: t_i aspetta
  - altrimenti: `abort`(t_i), poi rilancia t_i con lo stesso $ts(t_i)$

- interrompente (wound-wait)
  - se $ts(t_i) > ts(t_j)$: t_i aspetta
  - altrimenti: `abort`(t_j); poi rilancia t_j con lo stesso $ts(t_j)$

Uccisione di transazioni: timestamp (2)

- le transazioni uccise devono ripartire con lo stesso timestamp
  - altrimenti rischierebbero di essere sempre uccise (starvation)

- generalmente non utilizzato dai DBMS commerciali
  - la probabilità di deadlock è di molto inferiore a quella di un conflitto

Uccisione di transazioni: no timestamp

t_i richiede lock su x

t_j ha lock su x

- no waiting
  - `abort`(t_i), poi rilancia t_i
- cautious waiting
  - se t_j non è in attesa: t_i aspetta
  - altrimenti t_i è abortita
- altre scelte:
  - uccidere la transazione che ha fatto meno lavoro

Livelock e starvation

Livelock

- una transazione resta sempre in attesa di un lock che viene continuamente dato ad altri
- succede quando il gestore dei lock non gestisce bene l'allocazione

Starvation

- una transazione è continuamente uccisa
- succede se non vengono gestiti propriamente i timestamp

VINCENZO CALABRÒ

LinkedIn vincenzocalabro

www.vincenzocalabro.it