Basi di dati Vol.2

Capitolo 2

Gestione delle transazioni

22/05/2008

DEFINIZIONE DI TRANSAZIONE

- Transazione: parte di programma caratterizzata da un inizio (**begin-transaction,** **`start transaction`** in SQL, non sempre esplicitata), una fine (**end-transaction,** non esplicitata in SQL) e al cui interno deve essere eseguito una e una sola volta uno dei seguenti comandi
  - **commit work** per terminare correttamente
  - **rollback work** per abortire la transazione

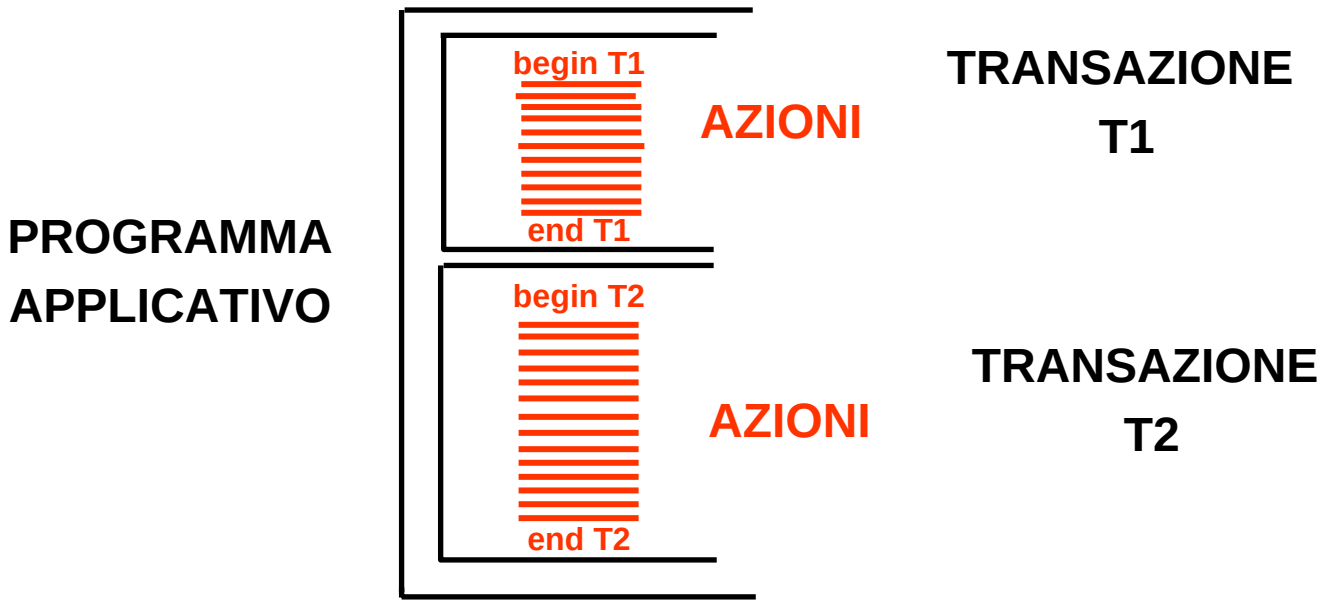
  la transazione va eseguita per intero o per niente (dettagli tra poco)

- Un **sistema transazionale** è in grado di definire ed eseguire transazioni per conto di applicazioni concorrenti

Transazioni in SQL

- Una transazione inizia al primo comando SQL dopo la "connessione" alla base di dati oppure alla conclusione di una precedente transazione (lo standard indica anche un comando start transaction, non obbligatorio, e quindi non previsto in molti sistemi)
- Conclusione di una transazione
  - commit [work]: le operazioni specificate a partire dall'inizio della transazione vengono eseguite sulla base di dati
  - rollback [work]: si rinuncia all'esecuzione delle operazioni specificate dopo l'inizio della transazione
- Molti sistemi prevedono una modalità autocommit, in cui ogni operazione forma una transazione

DIFFERENZA FRA APPLICAZIONE E TRANSAZIONE



Una transazione

```
start transaction;                                    (opzionale)
update ContoCorrente
   set Saldo = Saldo + 10 where NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo – 10 where NumConto = 42177;
commit work;
```

Una transazione con varie decisioni

```
start transaction;                                    (opzionale)
update ContoCorrente
   set Saldo = Saldo + 10 where NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo – 10 where NumConto = 42177;
select Saldo into A
   from ContoCorrente
   where NumConto = 42177;
if (A>=0)  then commit work
           else rollback work;
```

Transazioni in JDBC

- Scelta della modalità delle transazioni: un metodo definito nell'interfaccia **`Connection`**:

  `setAutoCommit(boolean autoCommit)`

- **`con.setAutoCommit(true)`**
  - (default) "autocommit": ogni operazione è una transazione

- **`con.setAutoCommit(false)`**
  - gestione delle transazioni da programma

    `con.commit()`

    `con.rollback()`
  - non c'è **`start transaction`**

Il concetto di transazione

- Una unità di elaborazione che gode delle proprietà "ACIDE"
  - Atomicità
  - Consistenza
  - Isolamento
  - Durabilità (persistenza)

Atomicità

- Una transazione è una unità atomica di elaborazione
- Non può lasciare la base di dati in uno stato intermedio
  - un guasto o un errore prima del commit debbono causare l'annullamento (UNDO) delle operazioni eventualmente svolte
  - un guasto o errore dopo il commit non deve avere conseguenze; se necessario vanno ripetute (REDO) le operazioni
- Esito
  - Commit = caso "normale" e più frequente
  - Abort (o rollback)
    - richiesto dall'applicazione = suicidio
    - richiesto dal sistema (violazione dei vincoli, concorrenza, incertezza in caso di fallimento) = omicidio

Consistenza

- La transazione rispetta i vincoli di integrità
- Conseguenza:
  - se lo stato iniziale è corretto
  - anche lo stato finale è corretto

Isolamento

- La transazione non risente degli effetti delle altre transazioni concorrenti
  - l'esecuzione concorrente di una collezione di transazioni deve produrre un risultato che si potrebbe ottenere con una esecuzione sequenziale
- Conseguenza: i risultati intermedi di una transazione non dovrebbero essere visibili
  - altrimenti si potrebbe generare un "effetto domino"

Durabilità (Persistenza)

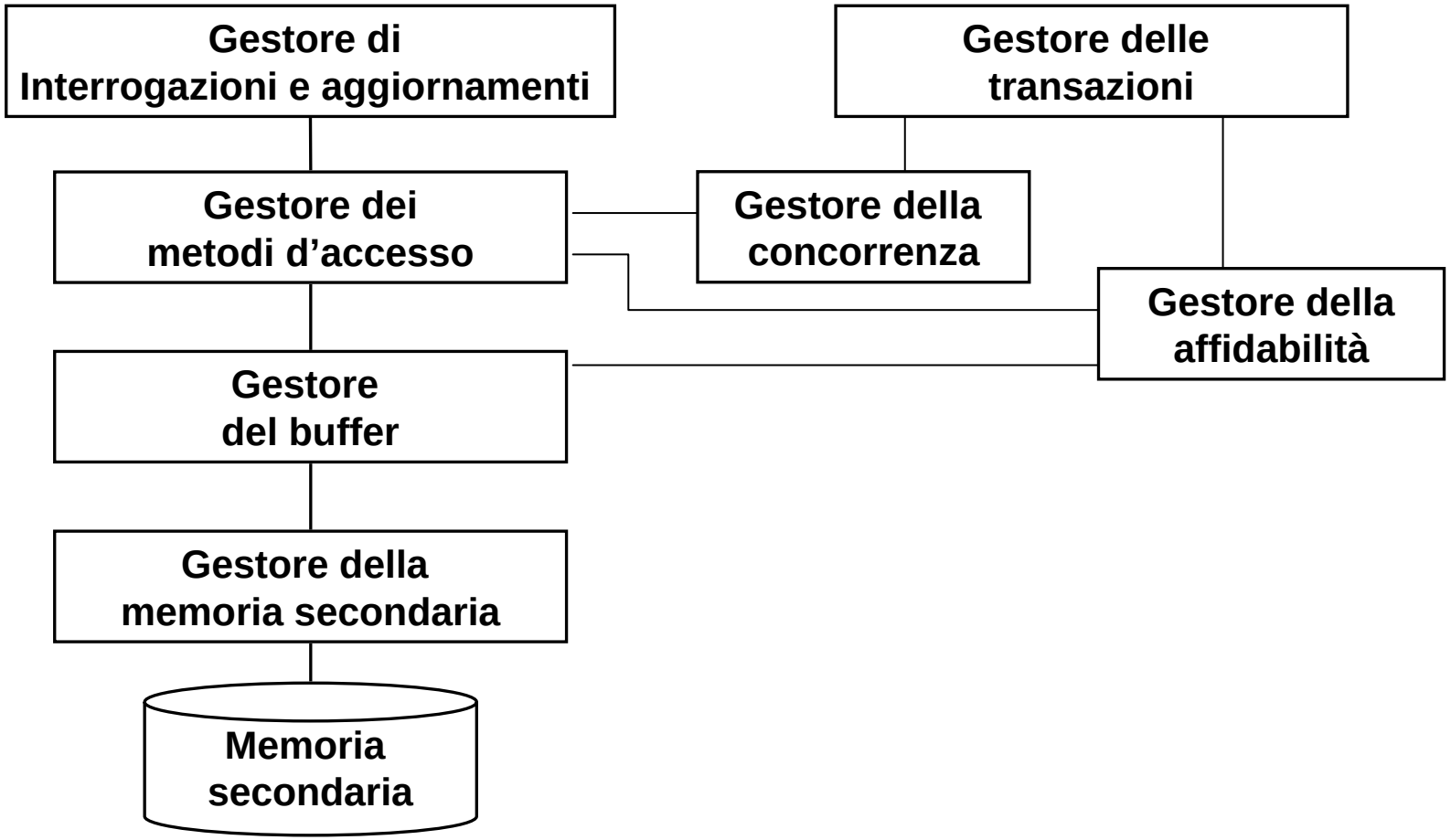
- Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"), anche in presenza di guasti
  - "commit" significa "impegno"

Transazioni e moduli di DBMS

- Atomicità e durabilità
  - Gestore dell'affidabilità (Reliability manager)
- Isolamento:
  - Gestore della concorrenza
- Consistenza:
  - Gestore dell'integrità a tempo di esecuzione (con il supporto del compilatore del DDL)

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni

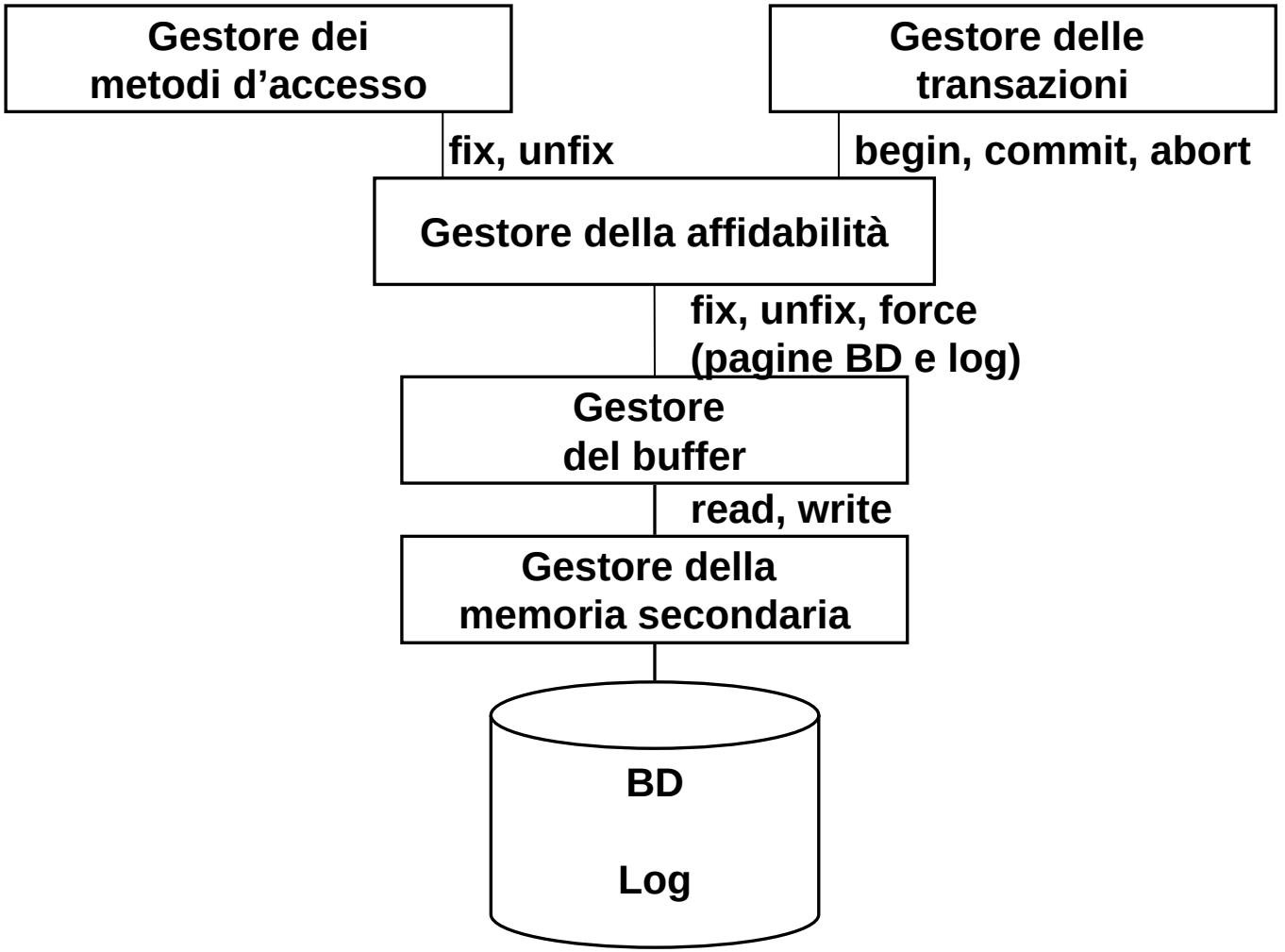


Gestore dell'affidabilità

- Assicura atomicità e durabilità
- Gestisce l'esecuzione dei comandi transazionali
  - **`start transaction`** (B, begin)
  - **`commit work`** (C)
  - **`rollback work`** (A, abort)

  e le operazioni di ripristino (recovery) dopo i guasti :
  - *warm restart* e *cold restart*
- Usa il **log**:
  - Un archivio permanente che registra le operazioni svolte
  - Due metafore:
    - il filo di Arianna
    - i sassolini e le briciole di Hansel e Gretel

Architettura del controllore dell'affidabilità



Persistenza delle memorie

- Memoria centrale: non è persistente
- Memoria di massa: è persistente ma può danneggiarsi
- Memoria stabile: memoria che non può danneggiarsi
  - astrazione “ideale,” non esiste, ma
  - perseguita attraverso la ridondanza:
    - dischi replicati
    - nastri
    - …

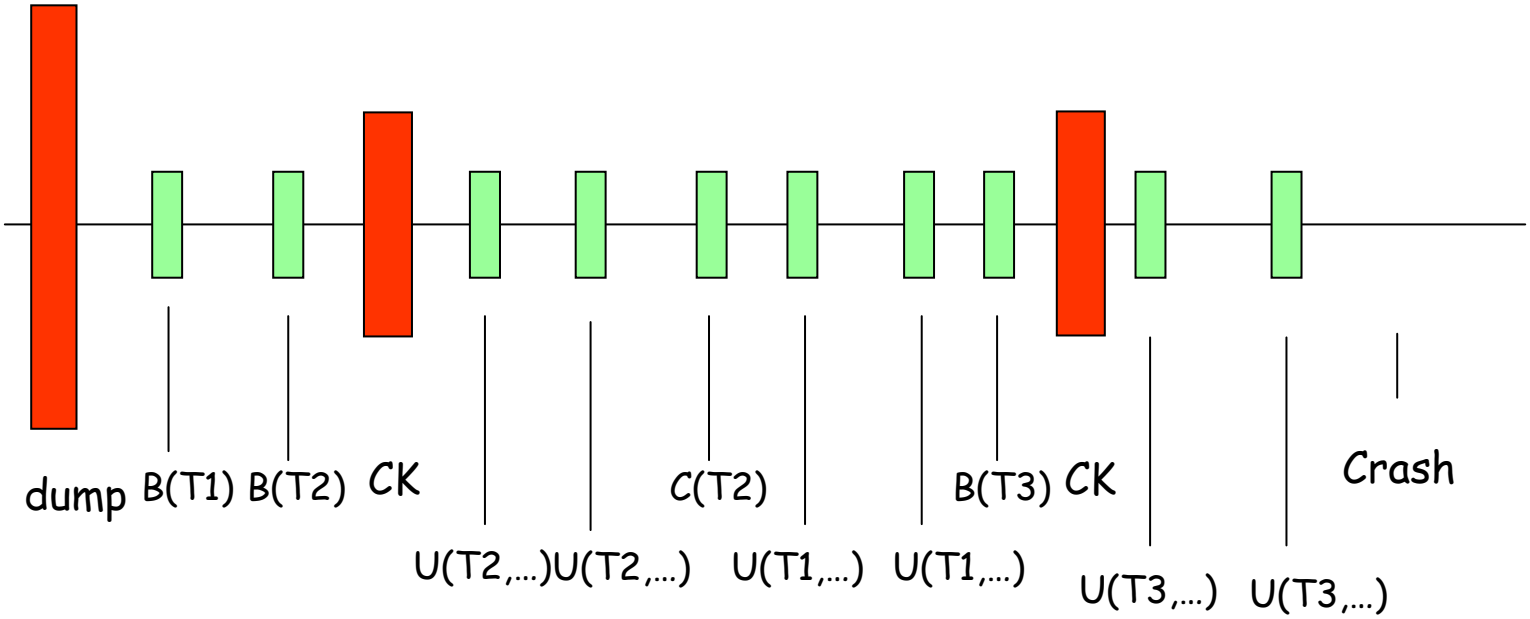
Modello di riferimento

- Una transazione è costituita da una sequenza di operazioni di input-output su oggetti astratti x, y, z (ignoriamo la semantica delle applicazioni)

Il log

- Il log è un file sequenziale gestito dal controllore dell'affidabilità, scritto in memoria stabile
- “Diario di bordo:” riporta tutte le operazioni in ordine
- Record nel log
  - operazioni delle transazioni
    - 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)
  - record di sistema
    - dump
    - checkpoint

Struttura del log



Log, checkpoint e dump: a che cosa servono?

- Il log serve “a ricostruire” le operazioni
- Checkpoint e dump servono ad evitare che la ricostruzione debba partire dall'inizio dei tempi
  - si usano con riferimento a tipi di guasti diversi
  - e anche ad attività diverse

Undo e redo

- Undo di una azione su un oggetto O:
  - update, delete: copia il valore del before state (BS) nell'oggetto O
  - insert: eliminare O (se c'è)
- Redo di una azione su un oggetto O:
  - insert, update: copia il valore dell' after state (AS) nell'oggetto O
  - delete: elimina O (se c'è)
- Idempotenza di undo e redo:
  - undo(undo(A)) = undo(A)
  - redo(redo(A)) = redo(A)

Dump

- Copia completa (“di riserva,” backup) della base di dati
  - Solitamente prodotta mentre il sistema non è operativo
  - Salvato in memoria stabile
  - Un record di dump nel log indica il momento in cui il log è stato effettuato (e dettagli pratici, file, dispositivo, …)

Checkpoint

- Operazione che serve a "fare il punto" della situazione, semplificando le successive operazioni di ripristino:
  - ha lo scopo di registrare quali transazioni sono attive in un certo istante (e dualmente, di confermare che le altre o non sono iniziate o sono finite)
- Paragone (estremo):
  - la "chiusura dei conti" di fine anno di una amministrazione:
    - dal fine novembre (ad esempio) non si accettano nuove richieste di "operazioni" e si concludono tutte quelle avviate prima di accettarne di nuove (a metà gennaio)

Checkpoint (2)

- Varie modalità, vediamo la più semplice:
  - si sospende l'accettazione di richieste di ogni tipo (scrittura, inserimenti, …, commit, abort)
  - si trasferiscono in memoria di massa (tramite **force**) tutte le pagine sporche relative a transazioni andate in commit
  - si registrano sul log in modo sincrono (**force**) gli identificatori delle transazioni in corso (“record di checkpoint”)
  - si riprende l'accettazione delle operazioni

- Così siamo sicuri che
  - per tutte le transazioni che hanno effettuato il commit, i dati sono in memoria di massa
  - le transazioni "a metà strada" sono elencate nel checkpoint

Esito di una transazione

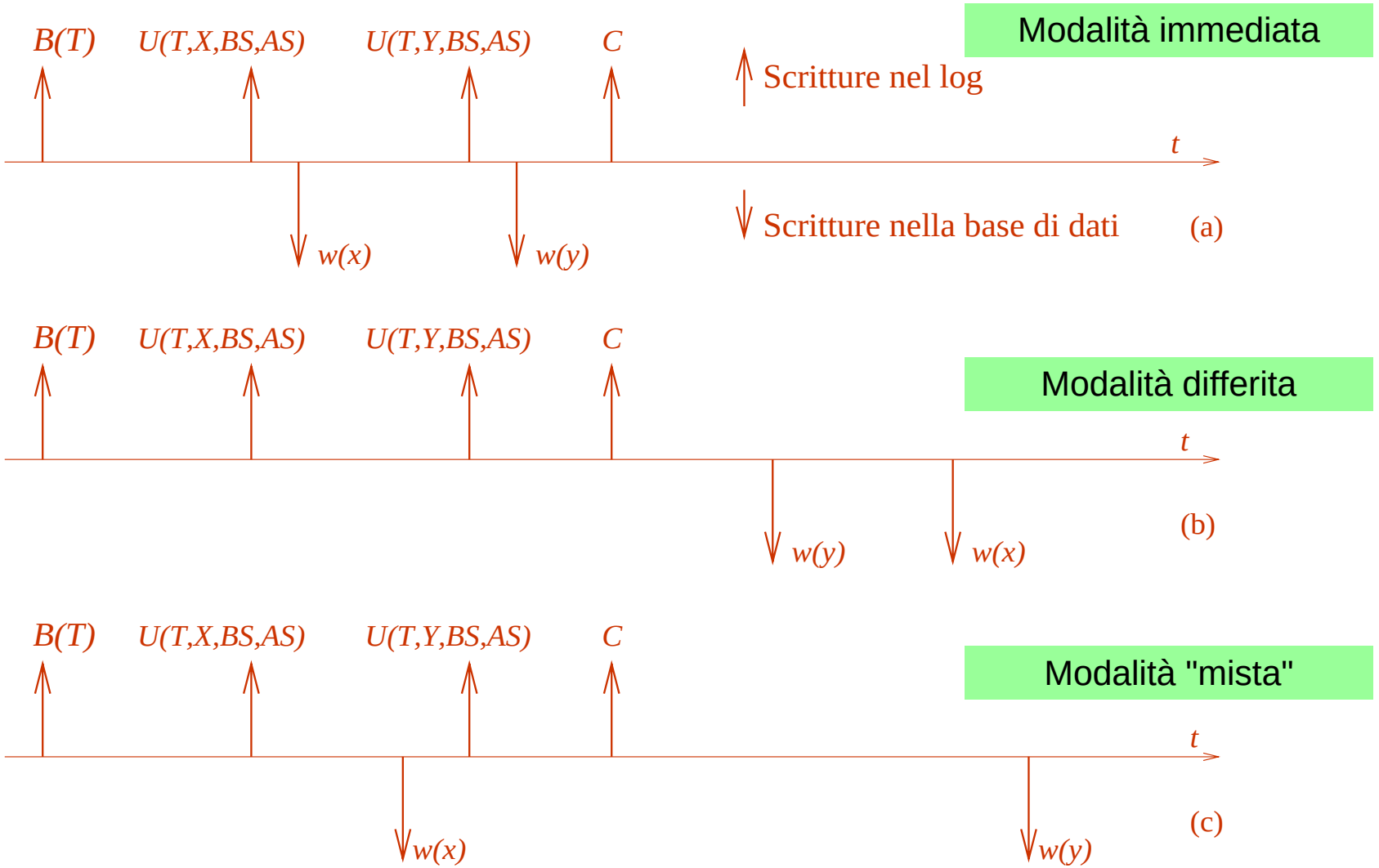
- L'esito di una transazione è determinato irrevocabilmente quando viene scritto il record di **commit** nel log in modo sincrono, con una **force**
  - una guasto prima di tale istante può portare (se necessario) ad un undo di tutte le azioni, per ricostruire lo stato originario della base di dati
  - un guasto successivo non deve avere conseguenze: lo stato finale della base di dati deve essere ricostruito, con redo (se necessario)
- record di abort possono essere scritti in modo asincrono

Regole fondamentali per il log

- **Write-Ahead-Log**:
  - si scrive sul giornale (il BS) prima che nella basi di dati
    - consente di disfare le azioni
- **Commit-Precedenza**:
  - si scrive sul giornale (l'AS) prima del commit
    - consente di rifare le azioni

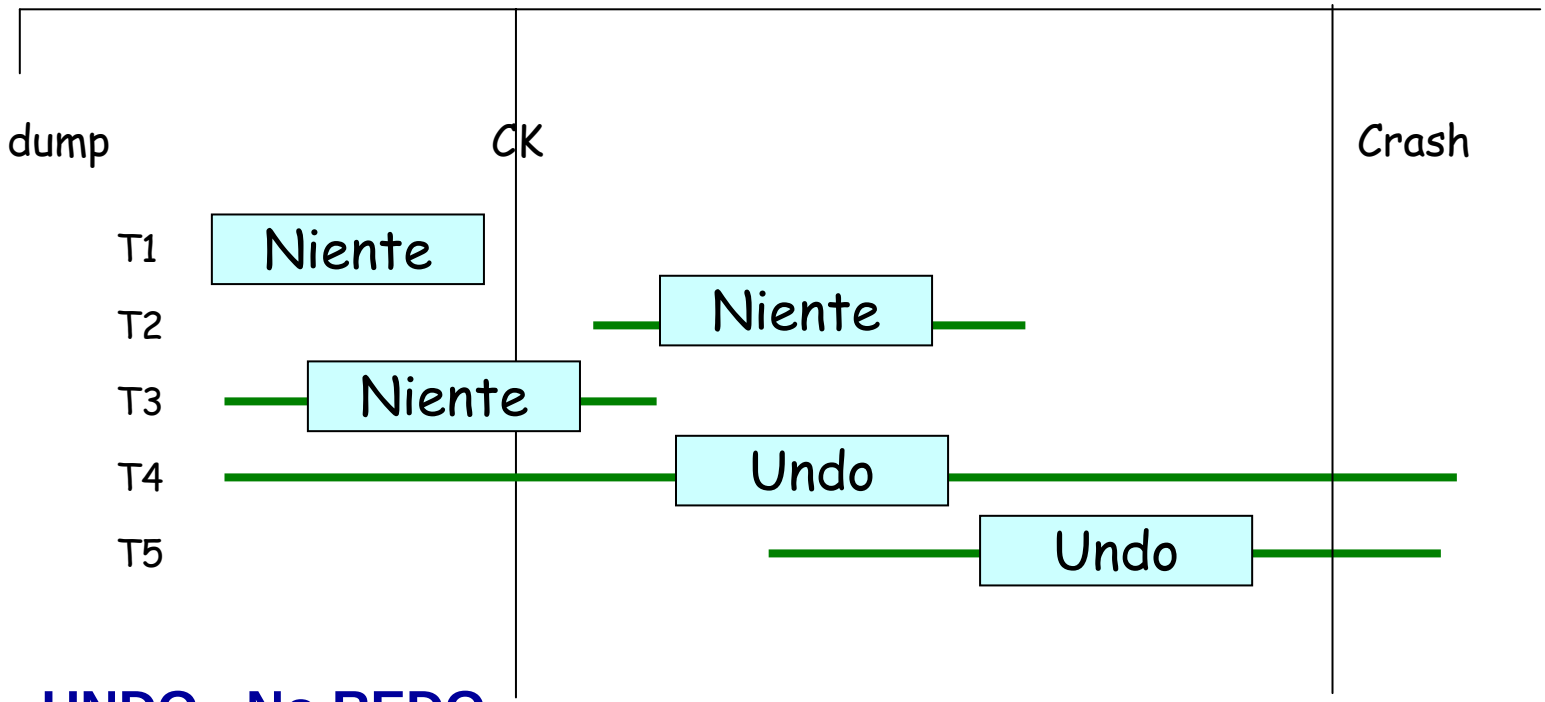
- Quando scriviamo nella base di dati?
  - Varie alternative

Scrittura nel log e nella base di dati



Modalità immediata

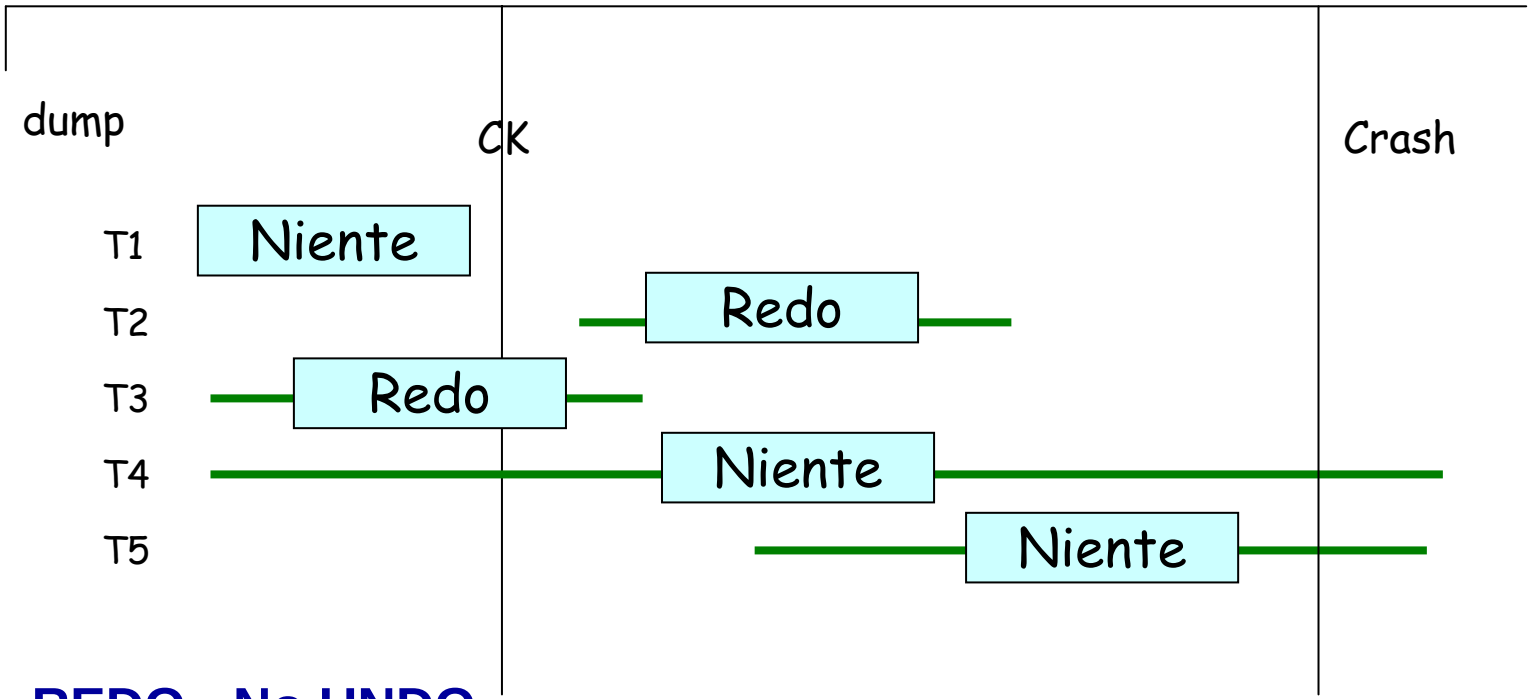
- Il DB contiene valori modificati per transazioni committed (tutte) e uncommitted



- **UNDO - No REDO**

Modalita’ differita

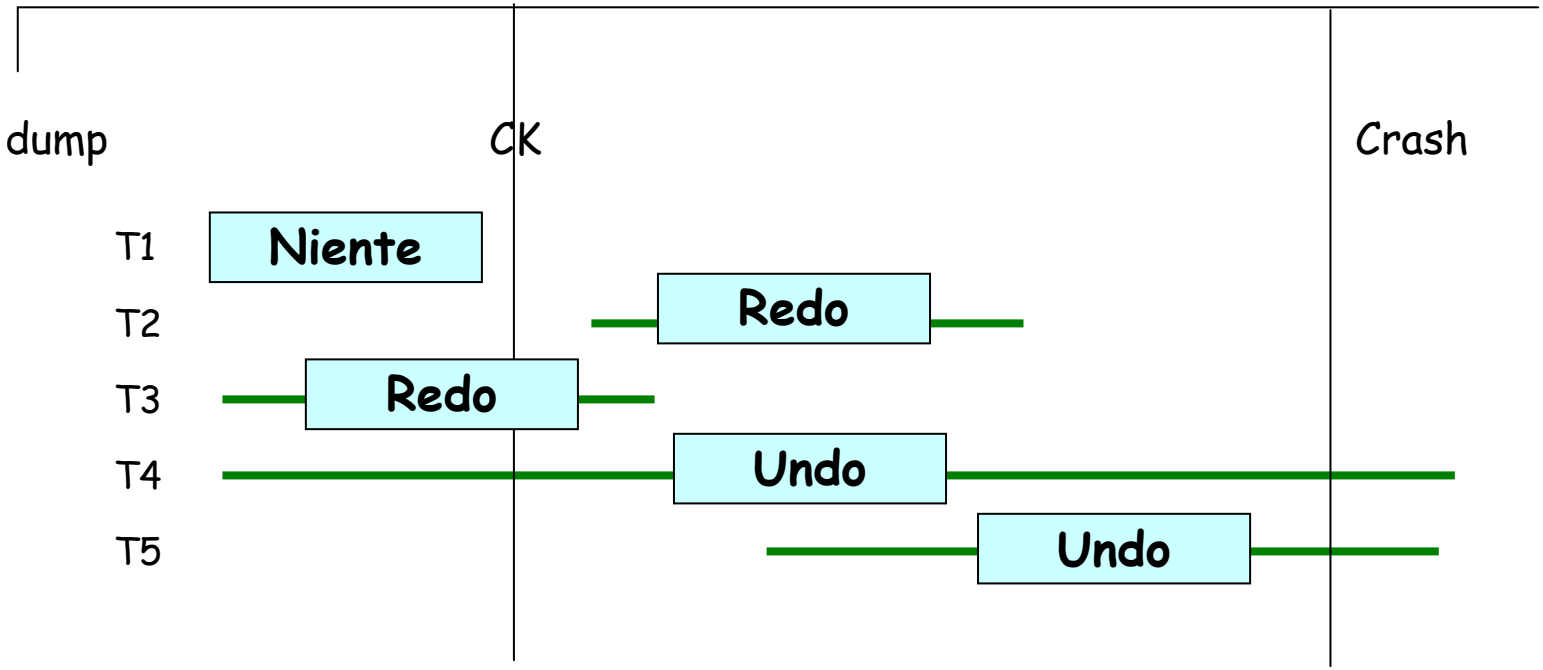
- Il DB non contiene valori modificati da transazioni uncommitted, ma non è sicuro contenga i nuovi valori per quelle committed



- **REDO - No UNDO**

Esiste una terza modalita': modalità mista

- La scrittura puo' avvenire in modalita' sia immediata che differita



- **REDO - UNDO**

Quale conviene?

- La terza, perché
  - anche se richiede operazioni diverse in caso di guasto,
  - permette al gestore del buffer di decidere quando scrivere

Ottimizzazioni

- Diversi record di log in una pagina
- Diversi record di log scritti insieme in modo sincrono subito prima del commit
- Commit di diverse transazioni in contemporanea (*group commit*)
- Parallelismo nella scrittura dei log

Guasti

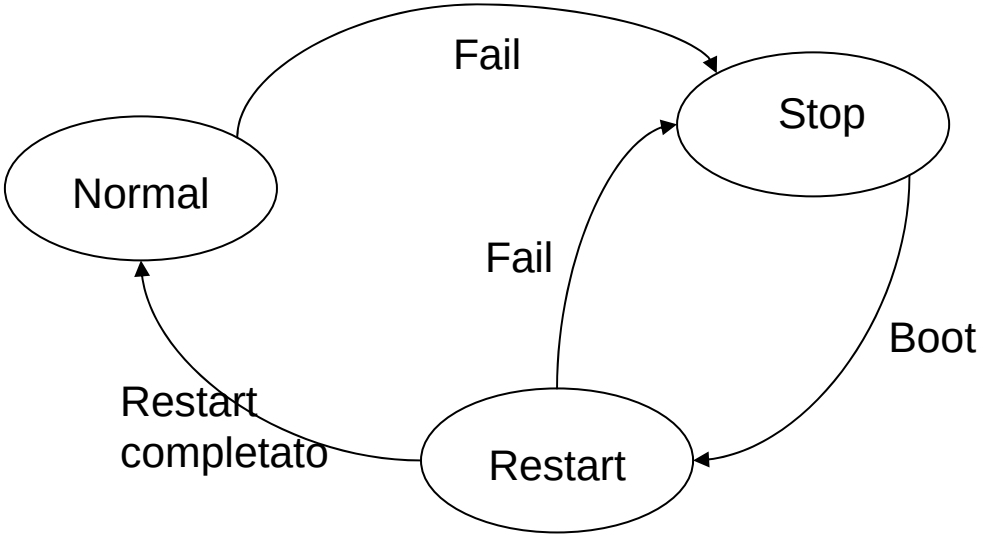
- **Guasti "soft":** errori di programma, crash di sistema, caduta di tensione
  - si perde la memoria centrale
  - non si perde la memoria secondaria

  warm restart, ripresa a caldo

- **Guasti "hard":** sui dispositivi di memoria secondaria
  - si perde anche la memoria secondaria
  - non si perde la memoria stabile (e quindi il log)

  cold restart, ripresa a freddo

Modello "fail-stop"



Processo di restart

- Obiettivo: classificare le transazioni in
  - completate (tutti i dati in memoria stabile)
  - in commit ma non necessariamente completate (può servire redo)
  - senza commit (vanno annullate, undo)

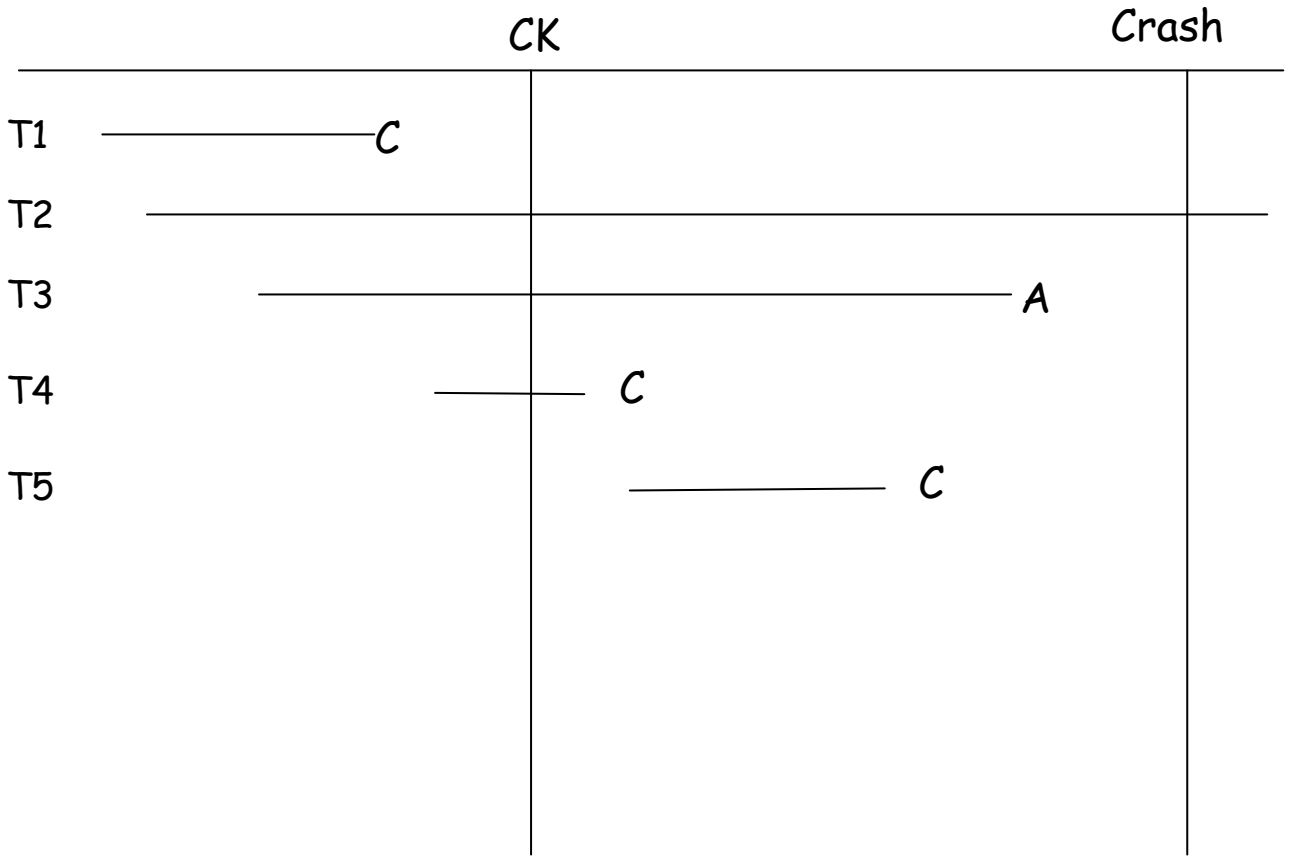
Ripresa a caldo

Quattro fasi:

- trovare l'ultimo checkpoint (ripercorrendo il log a ritroso)
- costruire gli insiemi *UNDO* (transazioni da disfare) e *REDO* (transazioni da rifare)
- ripercorrere il log all'indietro, fino alla più vecchia azione delle transazioni in *UNDO e REDO*, disfacendo tutte le azioni delle transazioni in *UNDO*
- ripercorrere il log in avanti, rifacendo tutte le azioni delle transazioni in *REDO*

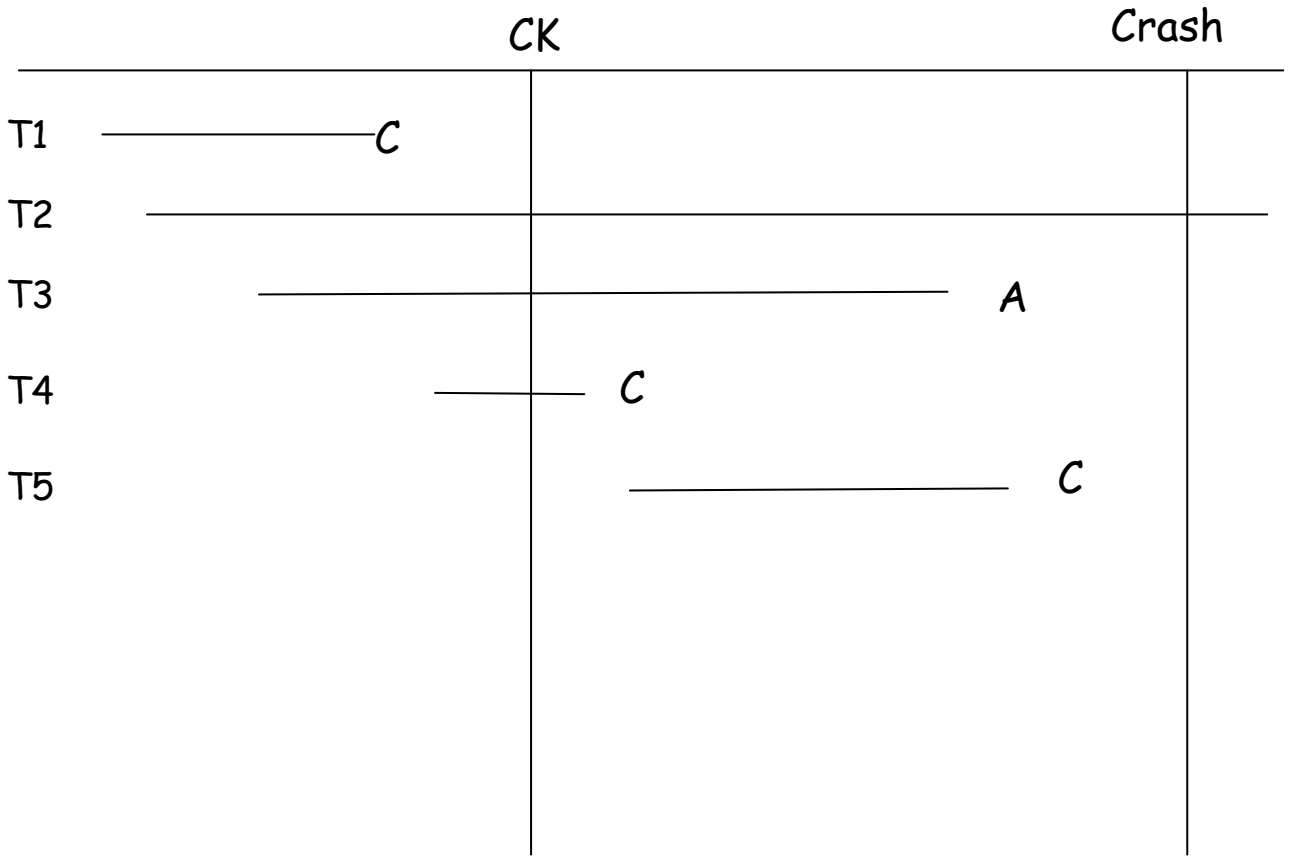
Esempio di warm restart

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(T2,O6,A8)



1. Ricerca dell'ultimo checkpoint

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(T2,O6,A8)



2. Costruzione degli insiemi UNDO e REDO

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)

1. C(T4)
2. B(T5)
   U(T3,O3,B5,A5)
   U(T5,O4,B6,A6)
   D(T3,O5,B7)
   A(T3)
3. C(T5)
   I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}
2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}
3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Setup

3. Fase UNDO

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

Setup

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

---

4. D(O6)

5. O5 =B7

6. O3 = B5

Undo

7. O2 =B3

8. O1=B1

4. Fase REDO

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
9. U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
10. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

Setup

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

---

4. D(O6)

5. O5 =B7

6. O3 = B5

Undo

7. O2 =B3

8. O1=B1

---

9. O3 = A4

Redo

10. O4 = A6

Ripresa a freddo

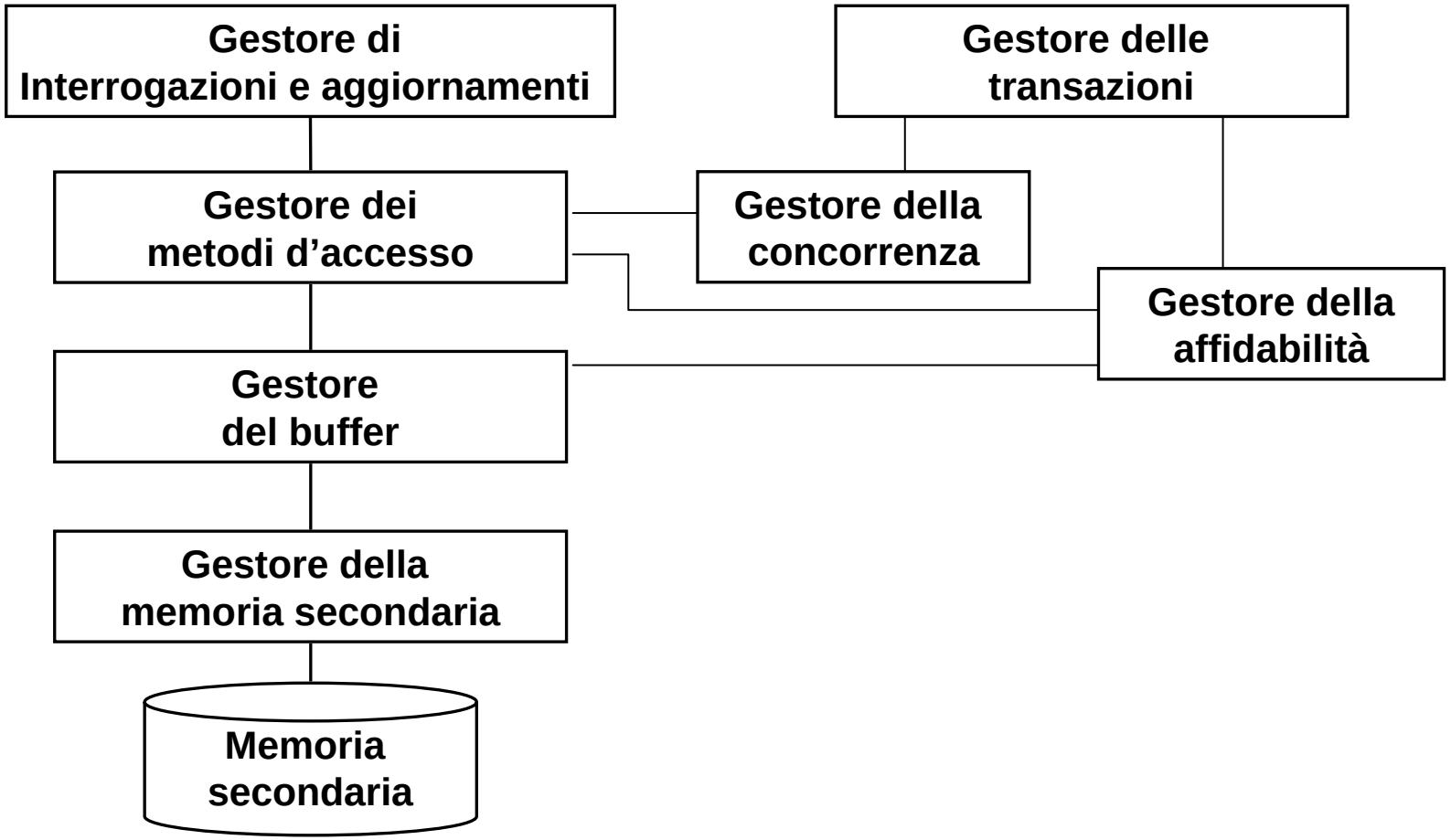
- Si ripristinano i dati a partire dal backup
- Si eseguono le operazioni registrate sul giornale fino all'istante del guasto
- Si esegue una ripresa a caldo

Backup e recovery

- Le tecniche per la gestione dell'affidabilità permettono anche
  - Ripristino di versioni salvate (dump)
    - In DB2
      - "version recovery"
  - Ripristino dello stato ad un certo istante
    - In DB2
      - "roll forward recovery"

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



Controllo di concorrenza

- La concorrenza è fondamentale: decine o centinaia di transazioni al secondo, non possono essere seriali
- Esempi: banche, prenotazioni aeree

Problema

- Anomalie causate dall'esecuzione concorrente, che quindi va governata

Perdita di aggiornamento (lost update)

- Due transazioni identiche
  - t_1 : r(x), $x = x$ -1000 , w(x)
  - t_2 : r(x), $x = x$ -1000, w(x)
- Esempio pratico: Incasso di assegni
- Inizialmente x=4000; dopo un'esecuzione seriale x=2000
- Un'esecuzione concorrente:

t_1	t_2
$r_1(x)$	
$x = x$ -1000	
	$r_2(x)$
	$x = x - 1000$
$w_1(x)$	
`commit`	
	$w_2(x)$
	`commit`

- Un aggiornamento viene perso: alla fine x=3000

Lettura sporca (dirty read)

t_1	t_2
$r_1(x)$	
$x = x + 1000000$	
$w_1(x)$	
	$r_2(x)$
`abort`	

- t_2 ha letto uno stato intermedio ("sporco") e lo può comunicare all'esterno
- Esempio: lettura del saldo su un aggiornamento sbagliato

Letture inconsistenti (inconsistent read)

- t_1 legge due volte x :

t_1	t_2
$r_1(x)$	
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	`commit`
$r_1(x)$	

- t_1 legge due valori diversi per x !
- Esempio: "quanti posti disponibili" ?

Aggiornamento fantasma (ghost update)

- Supponiamo che $y + z = 1000$;

t_1	t_2
$r_1(y)$	
	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	`commit`
$r_1(z)$	
$s = y + z$	

- $s = 1100$: ma la somma $y + z$ non è mai stata “davvero” 1100:
  - t_1 vede uno stato non esistente (o non coerente)
- Esempio: trasferimento da un conto ad un altro

Inserimento fantasma (phantom)

- Esempio: sistema di prenotazione

t_1	t_2
"conta i posti disponibili"	
	"aggiungi nuovi posti" (aereo più grande)
	`commit`
"conta i posti disponibili"	

Anomalie

- **Perdita di aggiornamento** **W-W**
- **Lettura sporca** **R-W (o W-W) con abort**
- **Letture inconsistenti** **R-W**
- **Aggiornamento fantasma** **R-W**
- **Inserimento fantasma** **R-W su dato "nuovo"**

Livelli di isolamento in SQL:1999 (e JDBC)

- Le transazioni possono essere definite **`read-only`** (non possono scrivere)
- Il livello di isolamento può essere scelto per ogni transazione
  - **`read uncommitted`** permette letture sporche, letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`read committed`** evita letture sporche ma permette letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`repeatable read`** evita tutte le anomalie esclusi gli inserimenti fantasma
  - **`serializable`** evita tutte le anomalie
- Nota:
  - la perdita di aggiornamento è sempre evitata

Anomalie

Perdita di aggiornamento	**W-W**
• `read uncommitted`	
Lettura sporca	**R-W (o W-W) con abort**
• `read committed`	
Letture inconsistenti	**R-W**
Aggiornamento fantasma	**R-W**
• `repeatable read`	
Inserimento fantasma	**R-W su dato "nuovo"**
• `serializable`	

Livelli di isolamento, perché?

- La gestione del controllo della concorrenza è costosa, se non serve, possiamo rinunciarci
- Nota bene: per le letture, non per le scritture

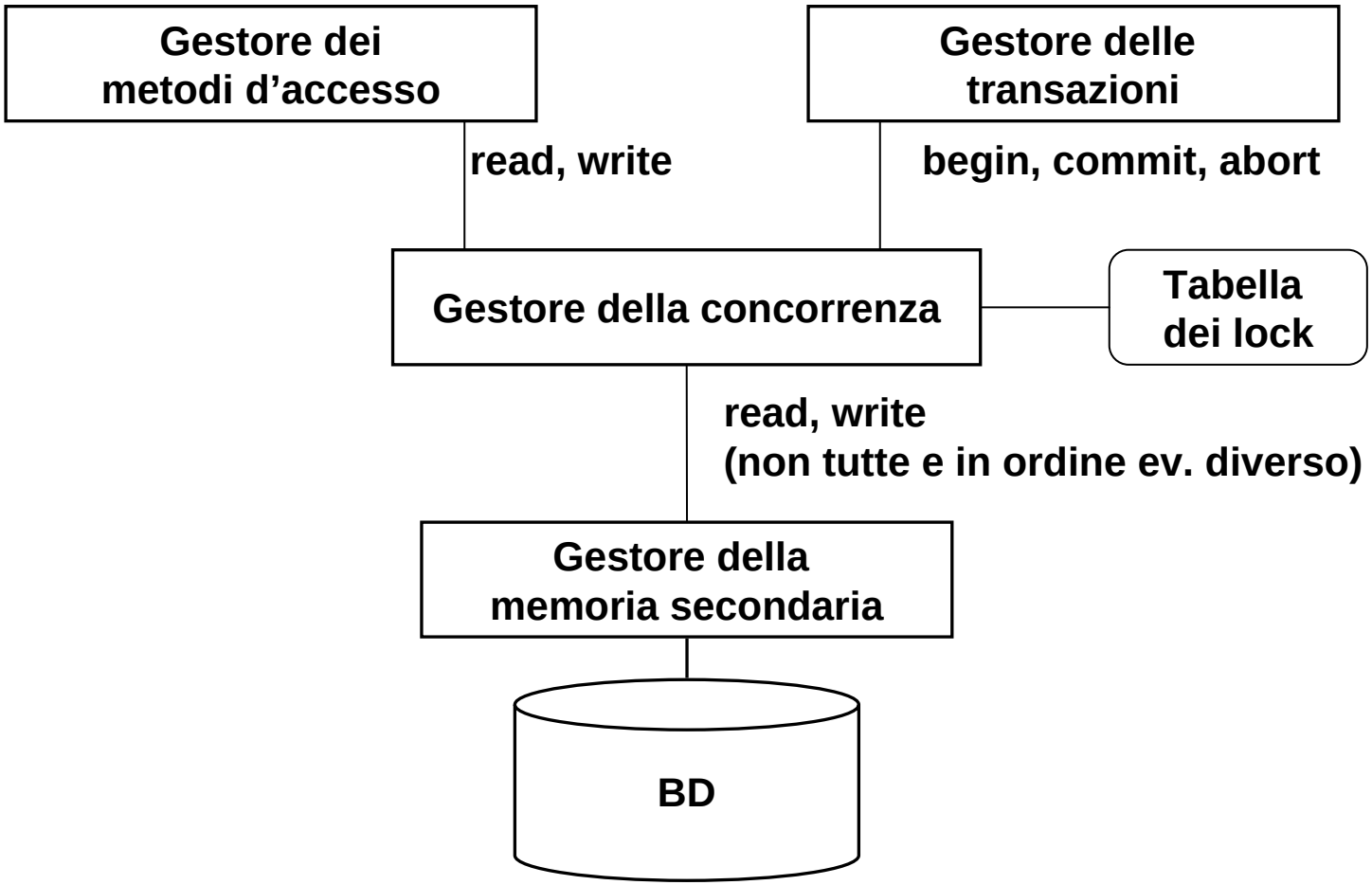
Anomalie

Perdita di aggiornamento	**W-W**
• `read uncommitted`	
Lettura sporca	**R-W (o W-W) con abort**
• `read committed`	
Letture inconsistenti	**R-W**
Aggiornamento fantasma	**R-W**
• `repeatable read`	
Inserimento fantasma	**R-W su dato "nuovo"**
• `serializable`	

Livelli di isolamento, esempi

- Vedi compito d'esame del 15/06/2005 domanda 4

Gestore della concorrenza (ignorando buffer e affidabilità)



“Teoria” del controllo di concorrenza

- Transazione: sequenza di letture e scritture (senza ulteriore semantica)
- Notazione:
  - ogni transazione ha un identificatore univoco
- Esempio:
  - $t_1 : r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$

Ipotesi semplificativa

- Ipotesi semplificativa (che rimuoveremo in futuro, in quanto non accettabile in pratica):
  - ignoriamo le transazioni che vanno in abort (e quindi non scriviamo commit e abort)

Schedule

- Sequenza di operazioni di input/output di transazioni concorrenti
- Esempio:

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

- A seguito dell'ipotesi semplificativa:
  - consideriamo la **commit-proiezione** degli schedule reali, in quanto ignoriamo le transazioni che vanno in abort, rimuovendo tutte le loro azioni dallo schedule

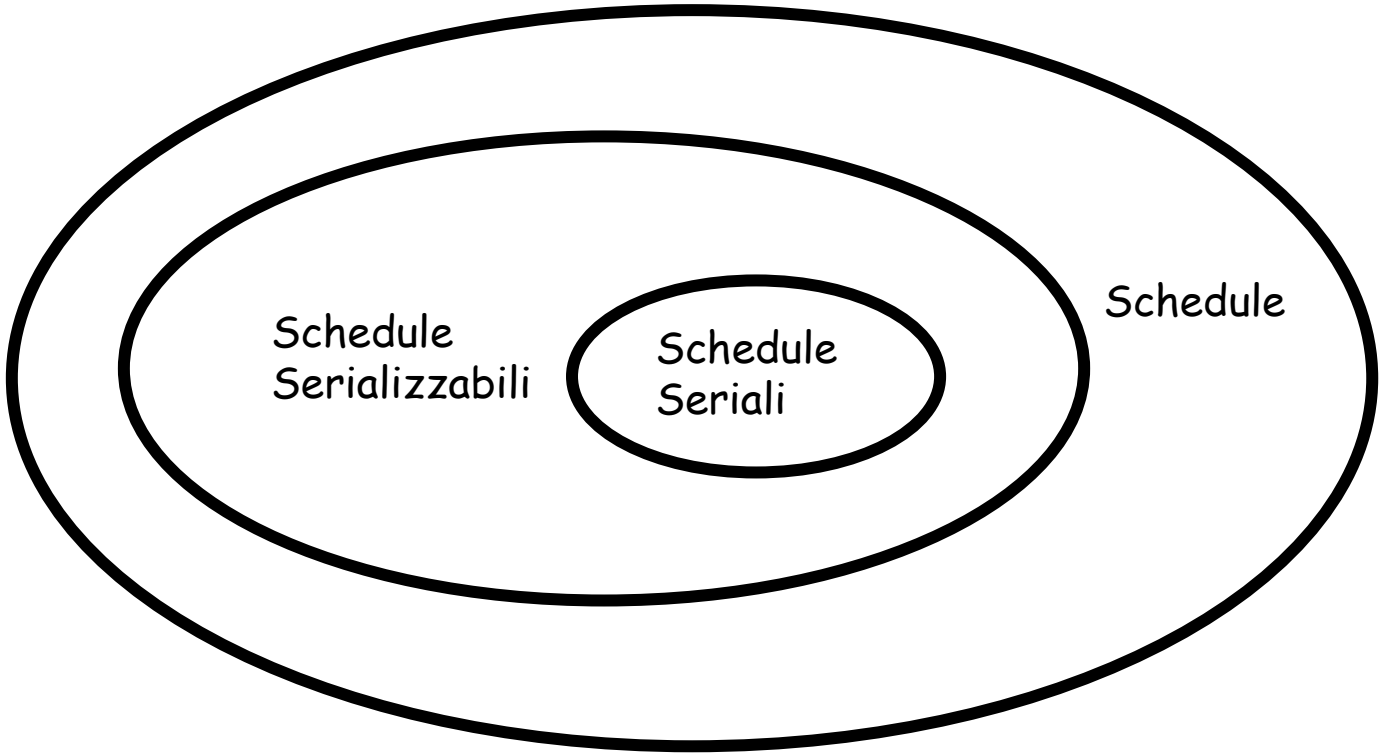
Controllo di concorrenza

- *Obiettivo*: evitare le anomalie
- *Controllore di concorrenza (Scheduler)*:
  - un sistema che accetta o rifiuta (o riordina) le operazioni richieste dalle transazioni (mantenendo l'ordine delle azioni in ciascuna transazione)
- *Schedule seriale:*
  - le transazioni sono separate, una alla volta

    $S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$
  - uno schedule seriale non presenta anomalie
- *Schedule serializzabile:*
  - produce lo “stesso risultato” di uno schedule seriale sulle stesse transazioni
    - richiede una nozione di equivalenza fra schedule

Idea base

- Individuare classi di schedule serializzabili che siano sottoclassi degli schedule possibili, siano serializzabili e la cui proprietà di serializzabilità sia verificabile a costo basso



View-Serializzabilità

- Definizioni prelilminari:
  - $r_i(x)$ ***legge-da*** $w_j(x)$ in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è $w_k(x)$ fra $w_j(x)$ e $r_i(x)$ in S
  - $w_i(x)$ in uno schedule S è ***scrittura finale*** *se* è l'ultima scrittura dell'oggetto x in S
- Schedule ***view-equivalenti*** ($S_i \approx_V S_j$): hanno la stessa relazione *legge-da* e le stesse scritture finali
  - nota: le operazioni di una transazione debbono mantenere l'ordine
- Uno schedule è ***view-serializzabile*** se è view-equivalente ad un qualche schedule seriale
- L'insieme degli schedule view-serializzabili è indicato con **VSR**

View serializzabilità: esempi

S_2 : $r_2(x)$ $w_0(x)$ $r_1(x)$ $w_2(x)$ $w_2(z)$

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

- S_2 non è view-equivalente a S_3

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

- S_3 è view-equivalente allo schedule seriale S_4 (e quindi è view-serializzabile)

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

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

- S_5 è view-equivalente allo schedule seriale S_6

View serializzabilità: esempi (2)

- perdita di aggiornamento

  $S_7 : r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$
- letture inconsistenti

  $S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$
- aggiornamento fantasma

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

- S_7, $S_{8,}$ S_9 non sono view-serializzabili

View serializzabilità

- Complessità:
  - la verifica della view-equivalenza di due schedule dati:
    - polinomiale
  - decidere sulla View serializzabilità di uno schedule:
    - problema NP-completo
- Non è utilizzabile in pratica

Conflict-serializzabilità

- Definizione preliminare:
  - Un'azione a_i è in *conflitto* con a_j ($i \neq j$), se operano sullo stesso oggetto e almeno una di esse è una scrittura. Due casi:
    - conflitto *read-write* (*rw* o *wr*)
    - conflitto *write-write* (*ww*).
- *Schedule conflict-equivalenti* ($S_i \approx_C S_j$): includono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi
- Uno schedule è *conflict-serializable* se è conflict-equivalente ad un qualche schedule seriale
- L'insieme degli schedule conflict-serializzabili è indicato con **CSR**

CSR e VSR

- Ogni schedule conflict-serializzabile è view-serializzabile, ma non necessariamente viceversa
- Controesempio per la non necessità:

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

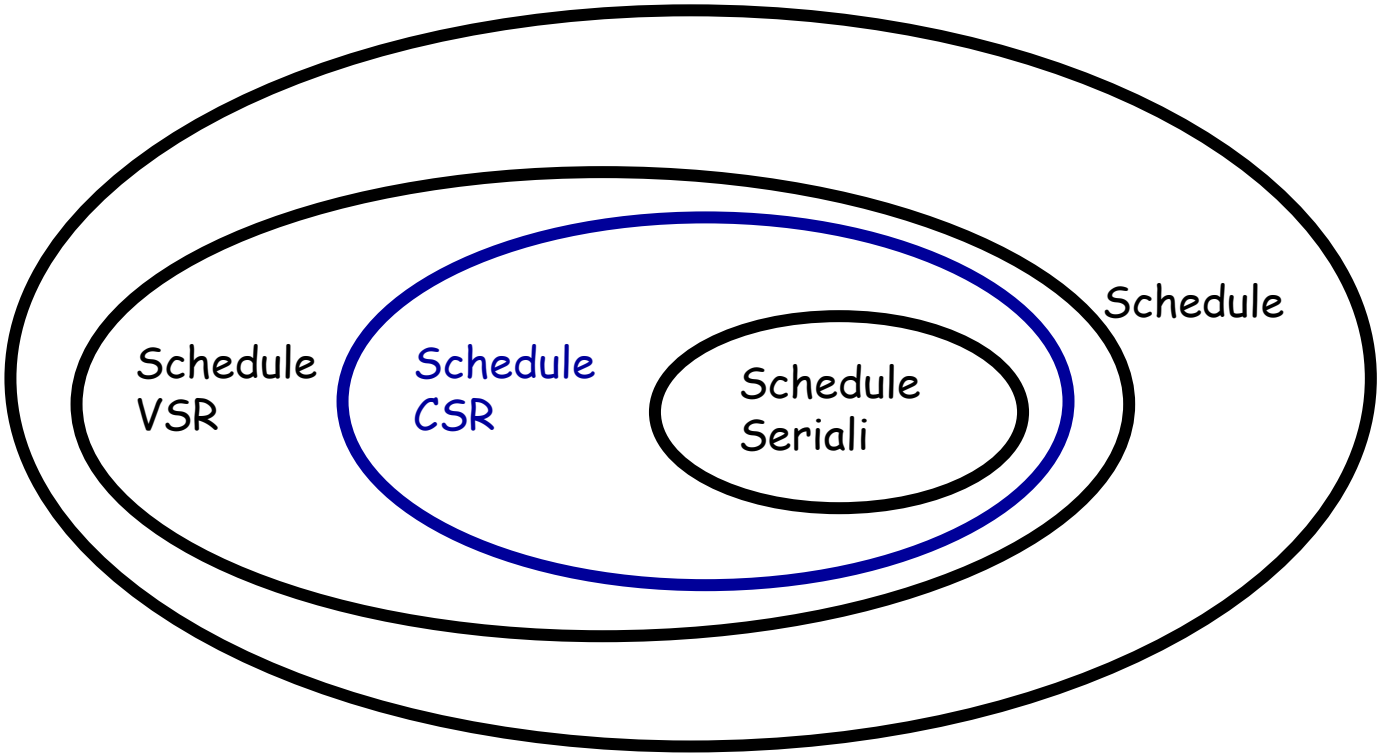
  - view-serializzabile:
    - view-equivalente a $r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$
  - non conflict-serializzabile

- Sufficienza: vediamo

CSR implica VSR

- CSR: esiste schedule seriale conflict-equivalente
- VSR: esiste schedule seriale view-equivalente
- Per dimostrare che CSR implica VSR è sufficiente dimostrare che la conflict-equivalenza $\approx_C$ implica la view-equivalenza $\approx_V$, cioè che se due schedule sono $\approx_C$ allora sono $\approx_V$
- Supponiamo $S_1 \approx_C S_2$ e dimostriamo che $S_1 \approx_V S_2$.
  - stesse scritture finali:
    - se così non fosse, ci sarebbero due scritture (su uno stesso oggetto) in ordine diverso e poiché due scritture (su uno stesso oggetto) sono in conflitto i due schedule non sarebbero $\approx_C$
  - stessa relazione “legge-da”:
    - se così non fosse, ci sarebbero scritture (su uno stesso oggetto) in ordine diverso o coppie lettura-scrittura (su …) in ordine diverso e quindi, come sopra, sarebbe violata la $\approx_C$

CSR e VSR



Verifica di conflict-serializzabilità

- Per mezzo del **grafo dei conflitti**:
  - un nodo per ogni transazione t_i
  - un arco (orientato) da t_i a t_j se c'è almeno un conflitto fra un'azione a_i e un'azione a_j tale che a_i precede a_j
- Teorema
  - **Uno schedule è in CSR se e solo se il grafo è aciclico**

CSR e aciclicità del grafo dei conflitti

- **Se uno schedule S è CSR** allora è $\approx_C$ ad uno schedule seriale. Supponiamo le transazioni nello schedule seriale ordinate secondo il TID: t_1, t_2, …, t_n . Poiché lo schedule seriale ha tutti i conflitti nello stesso ordine dello schedule S, nel grafo di S ci possono essere solo archi (i,j) con i<j e quindi **il grafo non può avere cicli,** perché un ciclo richiede almeno un arco (i,j) con i>j.
- **Se il grafo di S è aciclico**, allora esiste fra i nodi un "ordinamento topologico" (cioè una numerazione dei nodi tale che il grafo contiene solo archi (i,j) con i<j). **Lo schedule seriale** le cui transazioni sono ordinate secondo l'ordinamento topologico **è equivalente a S**, perché per tutti i conflitti (i,j) si ha sempre i<j.

Verifica di conflict-serializzabilità

- Per mezzo del **grafo dei conflitti**:
  - un nodo per ogni transazione t_i
  - un arco (orientato) da t_i a t_j se c'è almeno un conflitto fra un'azione a_i e un'azione a_j tale che a_i precede a_j
- Lemma
  - **Due schedule conflict-equivalenti hanno lo stesso grafo dei conflitti**
- Teorema
  - **Uno schedule è in CSR se e solo se il grafo è aciclico**

Grafo dei conflitti e conflict-equivalenza

Lemma **Due schedule conflict-equivalenti hanno lo stesso grafo dei conflitti**

- **$S_1 \approx_C S_2$;** dimostriamo che **S_1e S_2 hanno lo stesso grafo.**
  - Per contrapposizione: supponiamo che S_1e S_2 non abbiano lo stesso grafo e mostriamo che non sono $\approx_C$
    Se S_1e S_2 non hanno lo stesso grafo allora c'è un arco, nel grafo di uno schedule non presente nell'altro;
    supponiamo S_1 abbia tale arco; esso è relativo a due azioni in conflitto;
    ma S_1 e S_2 hanno le stesse azioni che, se sono in conflitto in S_1, allora sono in conflitto anche in S_2;
    se l'arco non compare nel grafo di S_2, allora le due azioni sono in S_2 ordine diverso e quindi non è vero che $S_1 \approx_C S_2$

CSR e aciclicità del grafo dei conflitti

Teorema **Uno schedule è in CSR se e solo se il grafo è aciclico**

- **Se S è in CSR allora il grafo di S è aciclico**

  Se S è in CSR allora è $\approx_C$ ad uno schedule seriale.

  Il grafo di uno schedule seriale è aciclico, perché ci possono essere conflitti fra azioni a_i e a_j solo se i<j (mentre un ciclo richiede almeno un arco (i,j) con i>j).

  Per il lemma, anche il grafo di S è aciclico
- **Se il grafo di S è aciclico allora S è in CSR**
- Se il grafo è aciclico, allora esiste fra i nodi un "ordinamento topologico" (cioè una numerazione dei nodi tale che il grafo contiene solo archi (i,j) con i<j).
- Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è equivalente a S, perché per tutti i conflitti (i,j) si ha sempre i<j.
- Quindi esiste uno schedule seriale equivalente ad S: S è in CSR

Controllo della concorrenza in pratica

- la conflict-serializabilità
  - è più rapidamente verificabile (con opportune strutture dati, complessità lineare),
  - ma sarebbe davvero efficiente se potessimo conoscere il grafo dall'inizio, ma così non è: uno scheduler deve operare "incrementalmente", cioè ad ogni richiesta di operazione decidere se eseguirla subito oppure fare qualcos'altro;
    non è praticabile mantenere il grafo, aggiornarlo e verificarne l'aciclicità ad ogni richiesta di operazione
  - si basa sull'ipotesi di commit-proiezione

- è inutilizzabile in pratica

Controllo della concorrenza in pratica, 2

- In pratica, si utilizzano tecniche che
  - garantiscono la conflict-serializzabilità senza dover costruire il grafo
  - non richiedono l'ipotesi della commit-proiezione

- Le tecniche utilizzate
  - 2PL (two-phase locking)
  - timestamp (implementazione "multiversion")

Lock

- Principio:
  - Tutte le letture sono precedute da *r_lock* (lock condiviso) e seguite da *unlock*
  - Tutte le scritture sono precedute da *w_lock* (lock esclusivo) e seguite da *unlock*
- Quando una transazione prima legge e poi scrive un oggetto, può:
  - richiedere subito un lock esclusivo
  - chiedere prima un lock condiviso e poi uno esclusivo (*lock upgrade*)
- Il *lock manager* riceve queste richieste dalle transazioni e le accoglie o rifiuta, sulla base della tavola dei conflitti

Gestione dei lock

- Basata sulla tavola dei conflitti

richiesta	stato della risorsa: free	r_locked	w_locked
r_lock	OK / r_locked	OK / r_locked	NO / w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	-	OK / dipende	OK / free

- Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata quando il contatore scende a zero
- Se la risorsa non è concessa, la transazione richiedente è posta in attesa (eventualmente in coda), fino a quando la risorsa non diventa disponibile (spesso con un “time-out”)
- Il lock manager gestisce una tabella dei lock, per ricordare la situazione

Locking a due fasi (2PL)

- Basata su due regole:
  - protezione con lock di tutte le letture e scritture
  - vincolo sulle richieste e i rilasci dei lock:
    - **una transazione, dopo aver rilasciato un lock, non può acquisirne altri**
- Garantisce "a priori" la conflict-serializzabilità, vediamo

2PL e CSR

- Ogni schedule 2PL e' anche conflict serializzabile, ma non necessariamente viceversa
- Controesempio per la non necessita':

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

  - Viola 2PL
  - Conflict-serializzabile

- Sufficienza: vediamo

2PL implica CSR

- S schedule 2PL
- Consideriamo per ciascuna transazione l'istante in cui ha tutti i lock sta per rilasciare il primo
- Ordiniamo le transazioni in accordo con questo valore temporale e consideriamo lo schedule seriale corrispondente
- Vogliamo dimostrare che tale schedule è equivalente ad S:
  - allo scopo, consideriamo un conflitto fra un'azione di t_i e un'azione di t_j con i<j; è possibile che compaiano in ordine invertito in S? no, perché in tal caso t_j dovrebbe aver rilasciato la risorsa in questione prima della sua acquisizione da parte di t_i

Concorrenza e fallimento di transazioni

- Rimuoviamo l'ipotesi di "commit-proiezione"
- Le transazioni possono fallire
- Conseguenze negative: rischio di:
  - rollback a cascata ("effetto domino"):
    - se T_i ha letto un dato scritto da T_k e T_k fallisce, allora anche T_i deve fallire
  - letture sporche:
    - se T_i ha letto un dato scritto da T_k e T_k fallisce, ma nel frattempo T_i è andata in commit, allora abbiamo l'anomalia

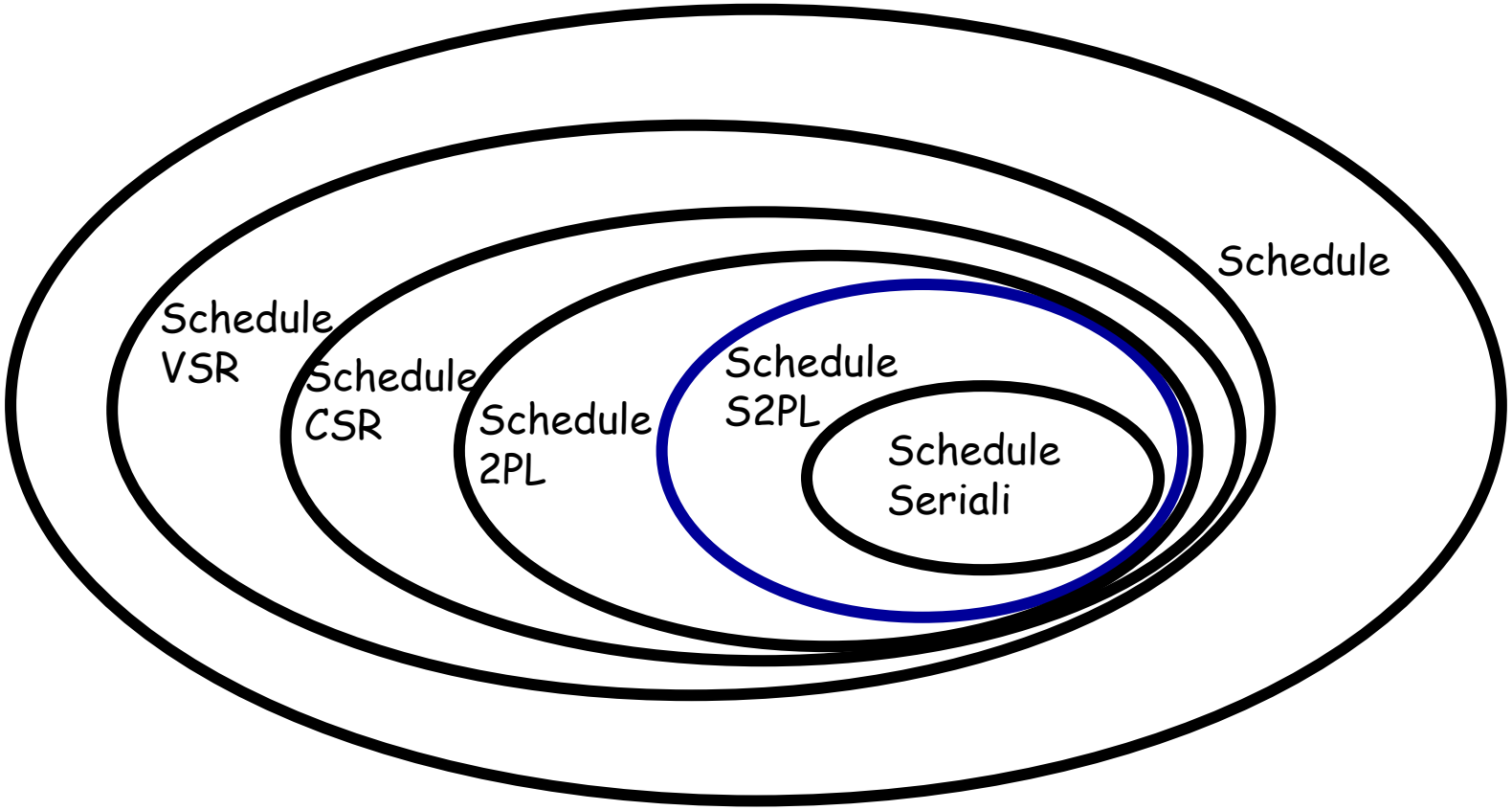
Evitiamo effetto domino e letture sporche

- letture sporche:
  - una transazione non può andare in commit finché non sono andate in commit tutte le transazioni da cui ha letto;
  - schedule che soddisfano questa condizione sono detti **recuperabili** (recoverable)
- rollback a cascata (“effetto domino”)
  - una transazione non deve poter leggere dati scritti da transazioni che non sono ancora andate in commit

Locking a due fasi stretto (S2PL)

- 2PL con una condizione aggiuntiva:
  - **i lock possono essere rilasciati solo dopo il commit o l'abort**
- Evita sia le letture sporche sia l’effetto domino

CSR, VSR e 2PL


Schedule
Schedule VSR
Schedule CSR
Schedule 2PL
Schedule S2PL
Schedule Seriali

Controllo di concorrenza basato su timestamp

- Tecnica alternativa al 2PL
- **Timestamp**:
  - identificatore che definisce un ordinamento totale sugli eventi di un sistema
- Ogni transazione ha un timestamp che rappresenta l'istante di inizio della transazione
- Uno schedule è accettato solo se riflette l'ordinamento seriale delle transazioni indotto dai timestamp

Dettagli

- Lo scheduler ha due contatori RTM(x) e WTM(x) per ogni oggetto
- Lo scheduler riceve richieste di letture e scritture (con indicato il timestamp della transazione):
  - $r_t(x)$:
    - se $t <$ WTM(x) allora la richiesta è respinta e la transazione viene uccisa;
    - altrimenti, la richiesta viene accolta e RTM(x) è posto uguale al maggiore fra RTM(x) e t
  - $w_t(x)$:
    - se $t <$ WTM(x) o $t <$ RTM(x) allora la richiesta è respinta e la transazione viene uccisa,
    - altrimenti, la richiesta viene accolta e WTM(x) è posto uguale a t
- Vengono uccise molte transazioni
- Per funzionare anche senza ipotesi di commit-proiezione, deve "bufferizzare" le scritture fino al commit (con attese)

Esempio

RTM(x) = 7

WTM(x) = 4

Richiesta	Risposta	Nuovo valore
$r_6(x)$	ok	
$r_8(x)$	ok	RTM(x) = 8
$r_9(x)$	ok	RTM(x) = 9
$w_8(x)$	no, t_8 uccisa	
$w_{11}(x)$	ok	WTM(x) = 11
$r_{10}(x)$	no, t_{10} uccisa	

2PL vs TS

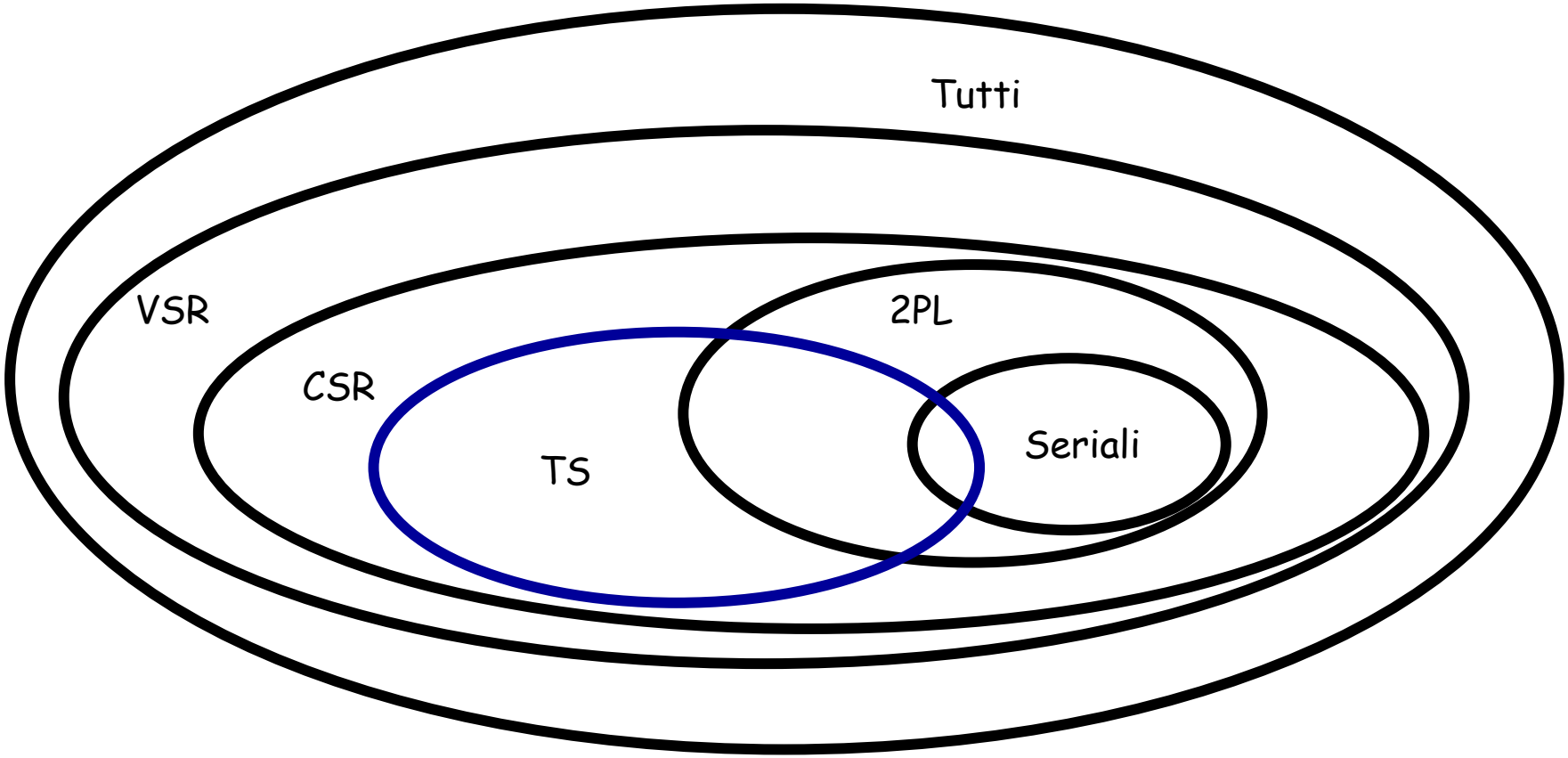
- Sono incomparabili
  - Schedule in TS ma non in 2PL

    $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_0(y)\ w_1(y)$
  - Schedule in 2PL ma non in TS

    $r_2(x)\ w_2(x) r_1(x)\ w_1(x)$
  - Schedule in TS e in 2PL

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

CSR, VSR, 2PL e TS



2PL vs TS

- In 2PL le transazioni sono poste in attesa
- In TS uccise e rilanciate
- Per rimuovere la commit proiezione, attesa per il commit in entrambi i casi
- 2PL può causare deadlock (vedremo)
- Le ripartenze sono di solito più costose delle attese:
  - molti sistemi preferiscono il 2PL (anche se si stanno diffondendo quelli con timestamp e multiversioni)

Multiversion concurrency control

- Main idea: writes generate new copies, reads make access to the correct copy
- Writes generate new copies each with a WTM. At any time, $N \geq 1$ copies of each object x are active, with $WTM_N(x)$. There is only one global $RTM(x)$
- Mechanism:
  - $r_t(x)$: is always accepted. x_k selected for reading such that: if $t > WTM_N(x)$, then k = N, otherwise k is taken such that $WTM_k(x) < t < WTM_{k+1}(x)$
  - $w_t(x)$: if $t < RTM(x)$ the request is refused, otherwise a new version of the item of data is added *(*N increased by one) with $WTM_N(x) = t$
- Old copies are discarded when there are no read transactions interested in their values

Lock management

- Interface:
  - `r_lock(T, x, errcode, timeout)`
  - `w_lock(T, x, errcode, timeout)`
  - `unlock(T, x)`

    `T`: transaction identifier
    `X`: data element
    `timeout`: max wait in queue
- If timeout expires, `errcode` signals an error, typically the transaction rolls back and restarts

Hierarchical locking

- In many real systems locks can be specified at different granularity, e.g. tables, fragments, tuples, fields. These are organized in a hierarchy (possibly a directed acyclic graph)
- 5 locking modes:
  - 2 are shared and exclusive, renamed as:
    - XL: exclusive lock
    - SL: shared lock
  - 3 are new:
    - ISL: intention shared lock
    - IXL: intention exclusive lock
    - SIXL: shared intention-exclusive lock
- The choice of lock granularity is left to application designers;
  - too coarse: many resources are blocked
  - too fine: many locks are requested

Hierarchical locking protocol

- Locks are requested from the root to descendents in a hierarchy
- Locks are released starting at the node locked and moving up the tree
- In order to request an SL or ISL on a node, a transaction must already hold an ISL or IXL lock on the parent node
- In order to request an IXL, XL, or SIXL on a node, a transaction must already hold an SIXL or IXL lock on the parent node
- The new conflict table is shown in the next slide

Conflicts in hierarchical locking

Request \ Resource state	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Stallo (deadlock)

- Attese incrociate: due transazioni detengono ciascuna una risorsa e aspetttano la risorsa detenuta dall'altra
- Esempio:
  - t_1: $r(x)$, $w(y)$
  - t_2: $r(y)$, $w(x)$
  - Schedule:

    $r\_lock_1(x)$, $r\_lock_2(y)$, $r_1(x)$, $r_2(y)$, $w\_lock_1(y)$, $w\_lock_2(x)$

Risoluzione dello stallo

- Uno stallo corrisponde ad un ciclo nel grafo delle attese (nodo=transazione, arco=attesa)
- Tre tecniche
  1. Timeout (problema: scelta dell'intervallo, con trade-off)
  2. Rilevamento dello stallo
  3. Prevenzione dello stallo
- Rilevamento: ricerca di cicli nel grafo delle attese
- Prevenzione: uccisione di transazioni "sospette" (può esagerare)

Livelli di isolamento in SQL:1999 (e JDBC)

- Le transazioni possono essere definite **`read-only`** (non possono richiedere lock esclusivi)
- Il livello di isolamento può essere scelto per ogni transazione
  - **`read uncommitted`** permette letture sporche, letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`read committed`** evita letture sporche ma permette letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`repeatable read`** evita tutte le anomalie esclusi gli inserimenti fantasma
  - **`serializable`** evita tutte le anomalie
- Nota:
  - la perdita di aggiornamento è sempre evitata

Anomalie

Perdita di aggiornamento	**W-W**
• `read uncommitted`	
Lettura sporca	**R-W (o W-W) con abort**
• `read committed`	
Letture inconsistenti	**R-W**
Aggiornamento fantasma	**R-W**
• `repeatable read`	
Inserimento fantasma	**R-W su dato "nuovo"**
• `serializable`	

Livelli di isolamento: implementazione

- Sulle scritture si ha sempre il 2PL stretto (e quindi si evita la perdita di aggiornamento)
- **`read uncommitted`**:
  - nessun lock in lettura (e non rispetta i lock altrui)
- **`read committed`**:
  - lock in lettura (e rispetta quelli altrui), ma senza 2PL
- **`repeatable read`**:
  - 2PL anche in lettura, con lock sui dati
- **`serializable`**:
  - 2PL con lock di predicato

Lock di predicato

- Caso peggiore:
  - sull'intera relazione
- Se siamo fortunati:
  - sull'indice

Casi di studio per il tuning di basi di dati

dal testo:

D. Shasha.
Database Tuning: a principled approach.
Prentice-Hall, 1992

Caso 1

- Abbiamo una base di dati con dieci relazioni su due dischi: cinque e cinque; su un disco c'è anche il log.
- Compriamo un nuovo disco; che cosa ci mettiamo?

Possibile risposta

- Il log

Caso 2

- Il tempo di risposta è variabile, in particolare ci sono rallentamenti quando vengono eseguite operazioni DDL

Possibile motivazione

- Le operazioni DDL richiedono lock in scrittura sul catalogo

Caso 3

- Transazioni così organizzate sono insoddisfacenti:
  - attribuzione di un nuovo numero di pratica
  - richiesta di informazioni interattive
  - effettuazione dell'inserimento

Possibile soluzione

- transazione troppo lunga; vanno riordinati i passi, e nella transazione ci devono essere solo il primo e il terzo

Caso 4

- Una transazione così organizzata eseguita a fine mese, di sera, è inefficiente:

  per ogni conto stampare tutte le info ...

Possibile soluzione

- essendo di sola lettura, di sera (tempo morto) potrebbe essere senza lock (Read Uncommitted)

Casi 5 e 6

- Nell'ambito di una transazione, si calcola lo stipendio medio per ciascun dipartimento. Contemporaneamente si fanno modifiche su singoli stipendi.
- Le prestazioni sono insoddisfacenti.

Possibili soluzioni

- si può eseguire in un tempo morto (senza aggiornamenti) senza lock (Read Uncommitted)
- se sono tollerate (leggere) inconsistenze, si può procedere senza lock (Read Uncommitted)
- si può fare una copia e lavorare su di essa (dati non attuali)
- se nessuna delle alternative è praticabile (non ci sono tempi morti e si vogliono dati attuali e consistenti) si può provare con Repeatable Read (non c'è rischio di "phantom")

Caso 7

- Un'applicazione prevede:
  - migliaia di inserimenti ogni ora
  - centinaia di migliaia di piccoli aggiornamenti ogni ora
- Gli inserimenti arrivano in transazioni grandi ogni mezz'ora e durano 5 minuti. In queste fasi le prestazioni sono inaccettabili (tempo di risposta 30 sec, rispetto a mezzo secondo)

Possibili soluzioni

- spezzare le transazioni con gli inserimenti

Caso 8

- Un'applicazione che prevede un'istruzione SQL all'interno di un ciclo è lenta (e usa molto tempo di CPU)

Possibile soluzione

- usare bene il cursore (facciamo fare i cicli e soprattutto i join all'SQL)

Caso 10

- Una società di servizi emette tutte le bollette a fine mese, con un programma che ha bisogno di tutta la notte, impedendo così l'esecuzione di altri programmi batch che sarebbero necessari

Possibili soluzioni

- è proprio necessario che le bollette siano emesse tutte insieme?
- se è proprio necessario, magari facciamolo durante il week-end (tempo morto più lungo)