

**Laurea in Informatica
A.A. 2021-2022**

Corso "Base di Dati"

Gestioni delle Transazioni

Dott. Massimiliano de Leoni



**UNIVERSITÀ
DEGLI STUDI
DI PADOVA**

Definizione di transazione

- Transazione: parte di programma caratterizzata da
 1. Un inizio di transazione (**begin-transaction**)
 2. Un corpo di transazione (serie di insert/delete/update in SQL)
 3. Una fine di transazione (**end-transaction**) che può portare:
 - ❑ **commit work** per terminare correttamente per rendere i cambiamenti definitivi
 - ❑ **rollback work** (o **abort**) per abortire la transazione, come se non fosse mai avvenuta
- Un **sistema transazionale (OLTP)** è in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti

Una transazione: Esempio

```
start transaction;
```

La transazione inizia

```
update ContoCorrente
```

```
  set Saldo = Saldo + 10 where NumConto =  
  12202;
```

```
update ContoCorrente
```

```
  set Saldo = Saldo - 10 where NumConto =  
  42177;
```

```
commit work;
```

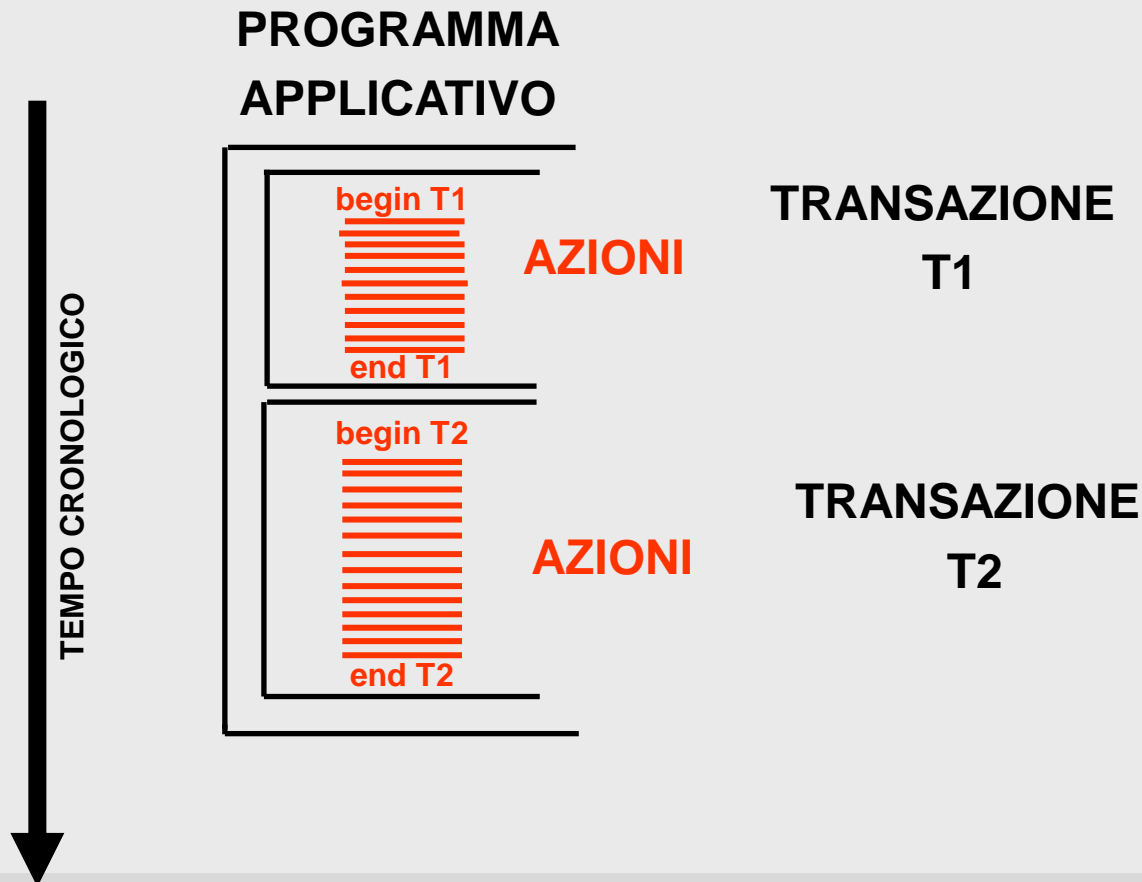
La transazione termina

**Le operazioni
vengono effettuate in
modo temporaneo**

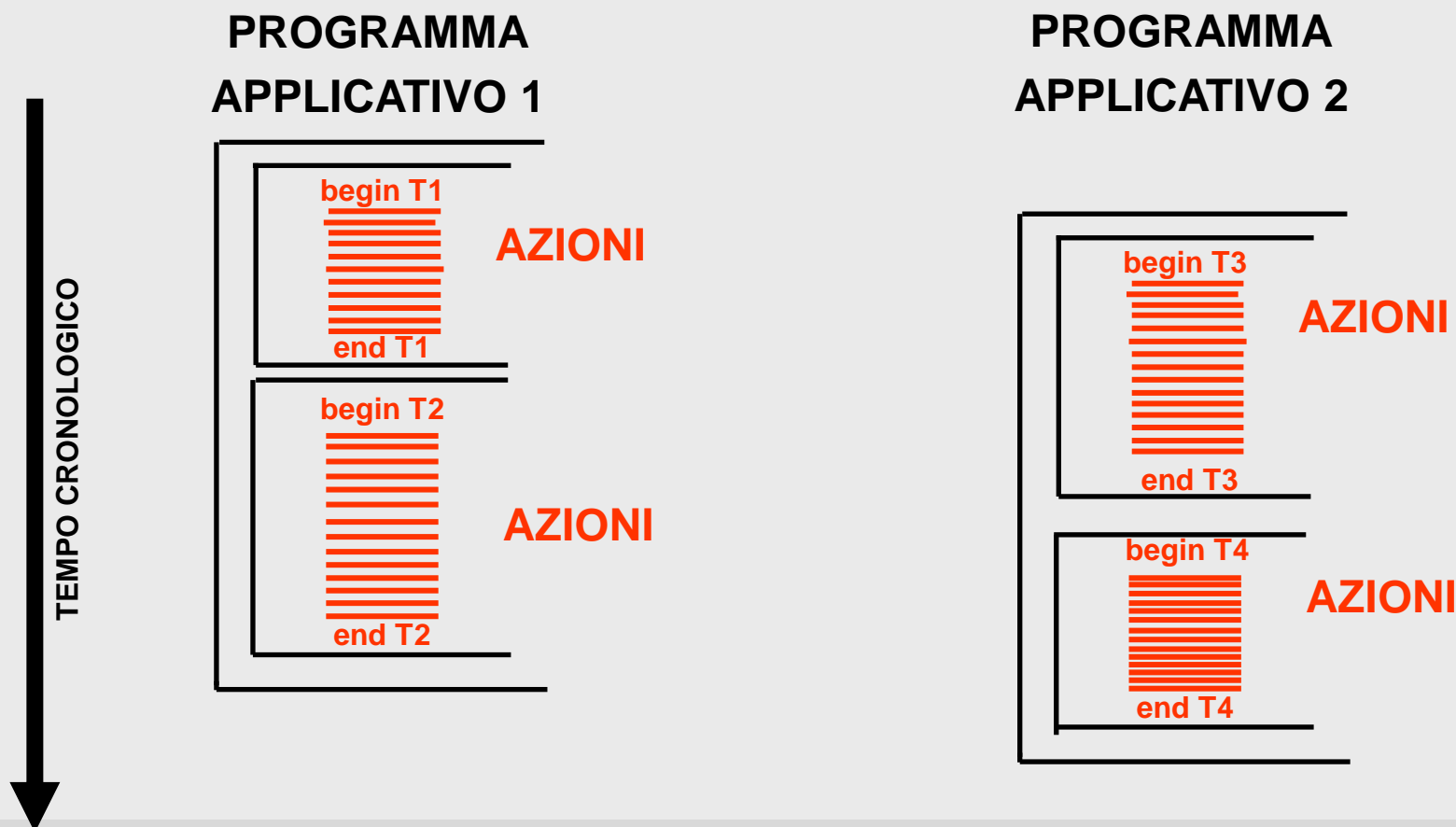
Una transazione con decisioni

```
start transaction;
  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;
```

Un'applicazione effettua tante transizioni



Diverse applicazioni effettuano transizioni in parallelo sullo stesso database



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 →
La transizione o è fatta interamente o per nulla
- 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 svolte
 - un guasto o errore dopo il commit non deve avere conseguenze; se necessario vanno ripetute (REDO) le operazioni
- Esempio (Agenzia Viaggi):
 1. Acquisto Biglietto Roma – New York
 2. Acquisto Biglietto New York – Roma
 3. Prenotazione hotel a New York
 - Se non si riescono ad acquistare entrambi i biglietti e a prenotare hotel, allora occorre fare il “rollback” di tutto!

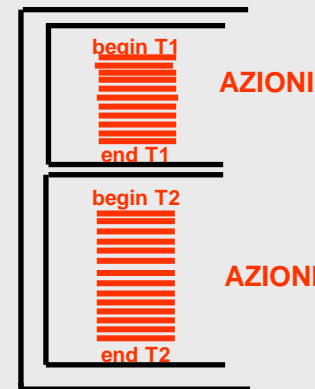
Consistenza

- La transazione rispetta i vincoli del DB: di chiave, di integrità referenziale (chiave esterna), di check, di valori, etc.
- I vincoli vanno verificato alla fine della transazione e non “durante”.
 - Possibilità di violare nel “mentre”
 - Lo stato finale deve essere corretto
- Conseguenza: se i vincoli sono violati alla fine della transazione, non c'è possibilità di «commit» (solo «rollback»)

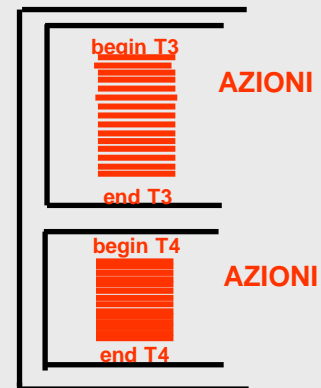
Isolamento

- 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: una transazione non espone i suoi stati intermedi
 - Si evita lo "effetto domino"

PROGRAMMA
APPLICATIVO 1



PROGRAMMA
APPLICATIVO 2

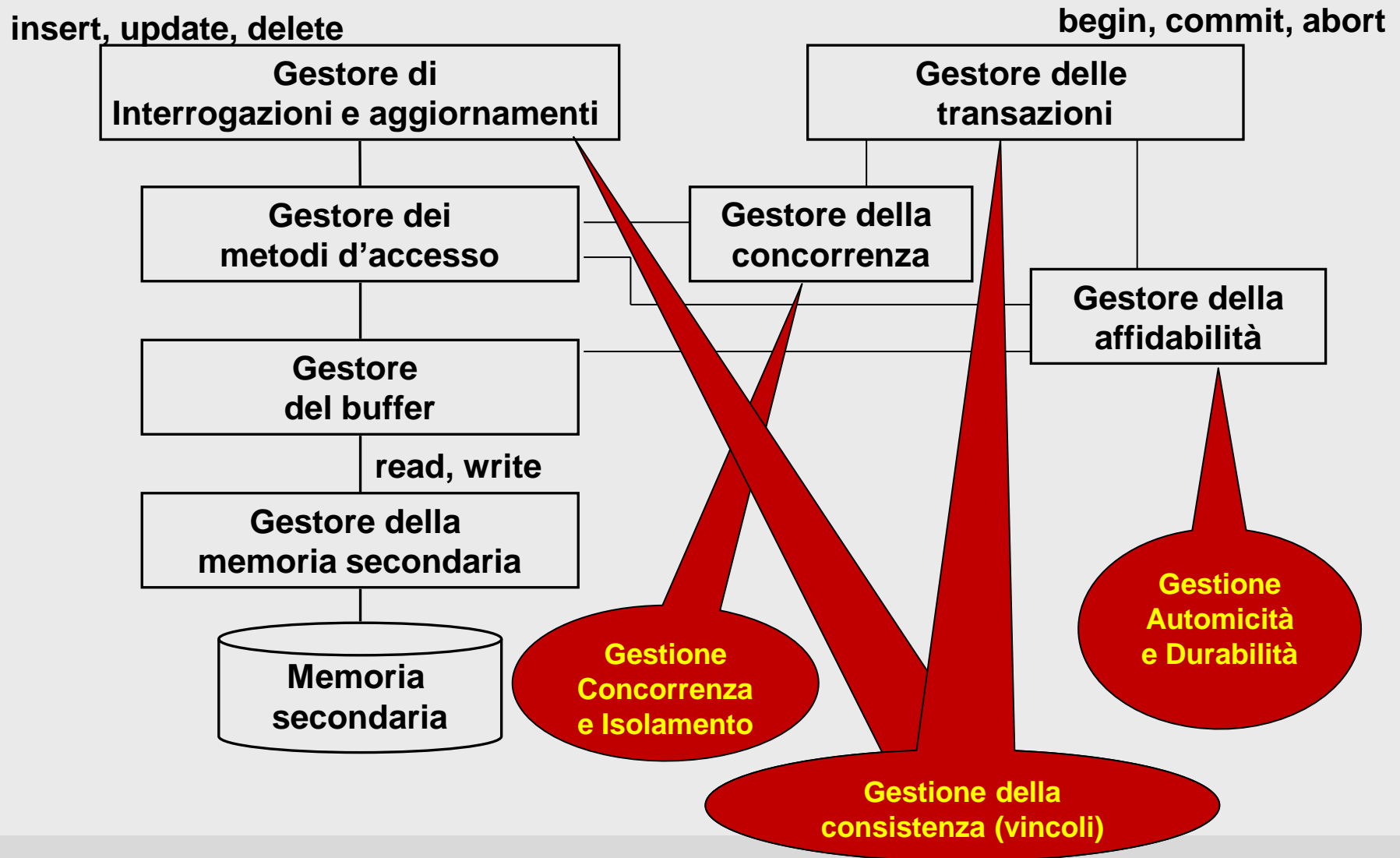


**T3 non deve vedere
i cambiamenti di T1**

Durabilità (Persistenza)

- Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"), anche in presenza di guasti:
 - **di dispositivo**: la memoria stabile (disco) si rompe
 - **di sistema**: il sistema software (applicazione e DBMS) va "in crash" ma la memoria stabile non si rompe

Architettura del DB per Interrogazioni & Transazioni



Gestore dell'affidabilità

- Assicura atomicità e durabilità
- Gestisce:
 - Esecuzione dei comandi transazionali:
 - ❑ start transaction
 - ❑ commit work
 - ❑ rollback work
 - Operazioni di ripristino (recovery) dopo i guasti
- Usa il **log**, archivio permanente delle operazioni svolte
 - Il log è memorizzato su **memoria stabile** che non può danneggiarsi
 - Memoria 100% stabile non esiste, ma “approssimabile” con ridondanza (RAID, nastri, ...)

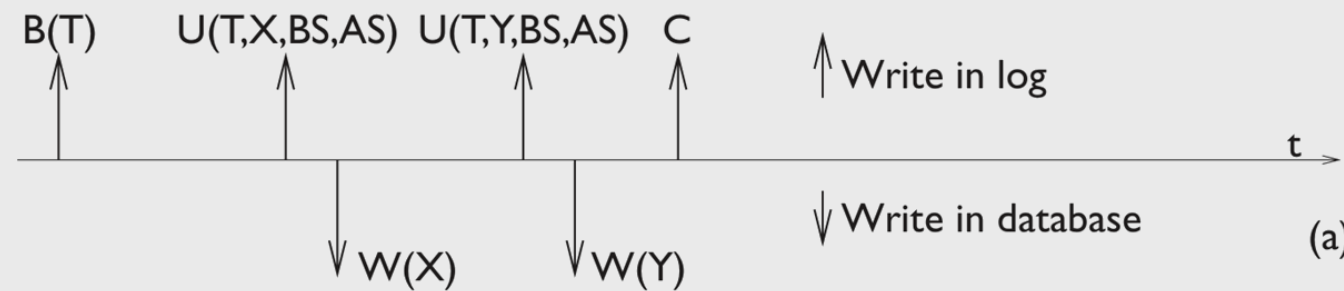
Il log

- 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*
 - ❑ **B(T)**: begin transazione T,
 - ❑ **I(T,O,AS)**: T inserisce l'oggetto O con valore AS,
 - ❑ **D(T,O,BS)**: T cancella l'oggetto O con valore BS,
 - ❑ **U(T,O,BS,AS)**: T aggiornata il valore dell'oggetto O da BS a AS
 - ❑ **C(T)**: commit transazione T
 - ❑ **A(T)**: abort transazione T
 - *record di sistema*
 - ❑ **dump**
 - ❑ **checkpoint**

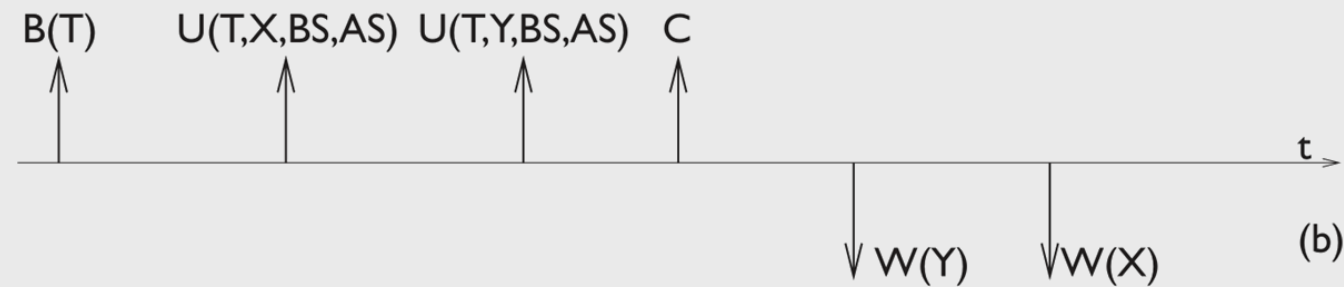
Regole fondamentali per il log

- **Write-Ahead-Log:**
 - Si scrive il log prima del database
- **Commit-Precedenza:**
 - si scrive il log prima del commit
 - consente di rifare le azioni
- Quando scriviamo nella base di dati?
 - Varie alternative

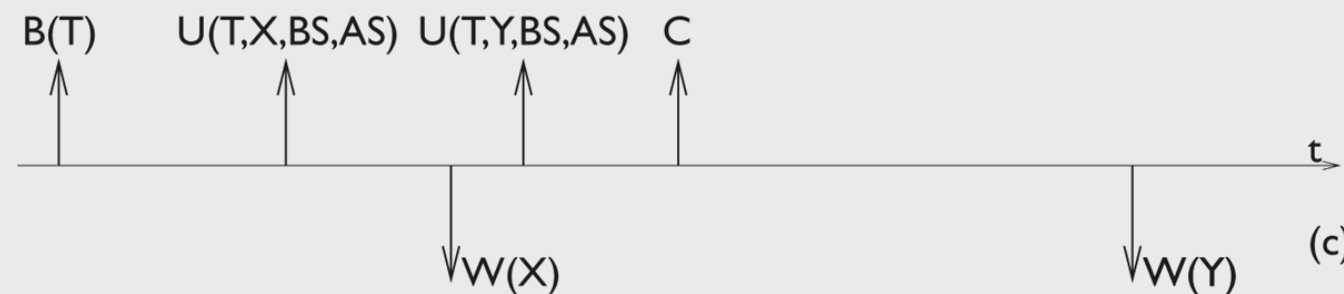
Scrittura nel log e nella base di dati



Modalità Immediata
(subito dopo log)



Modalità Differita
(dopo commit)



Modalità Mista
(quando "idle")

Modalità Immediata vs Differita vs Mista

Modalità Immediata

- Il DB contiene i valori AS (i nuovi valori aggiornati) provenienti da transazioni uncommitted
- Richiede Undo delle operazioni di transazioni uncommitted al momento del guasto

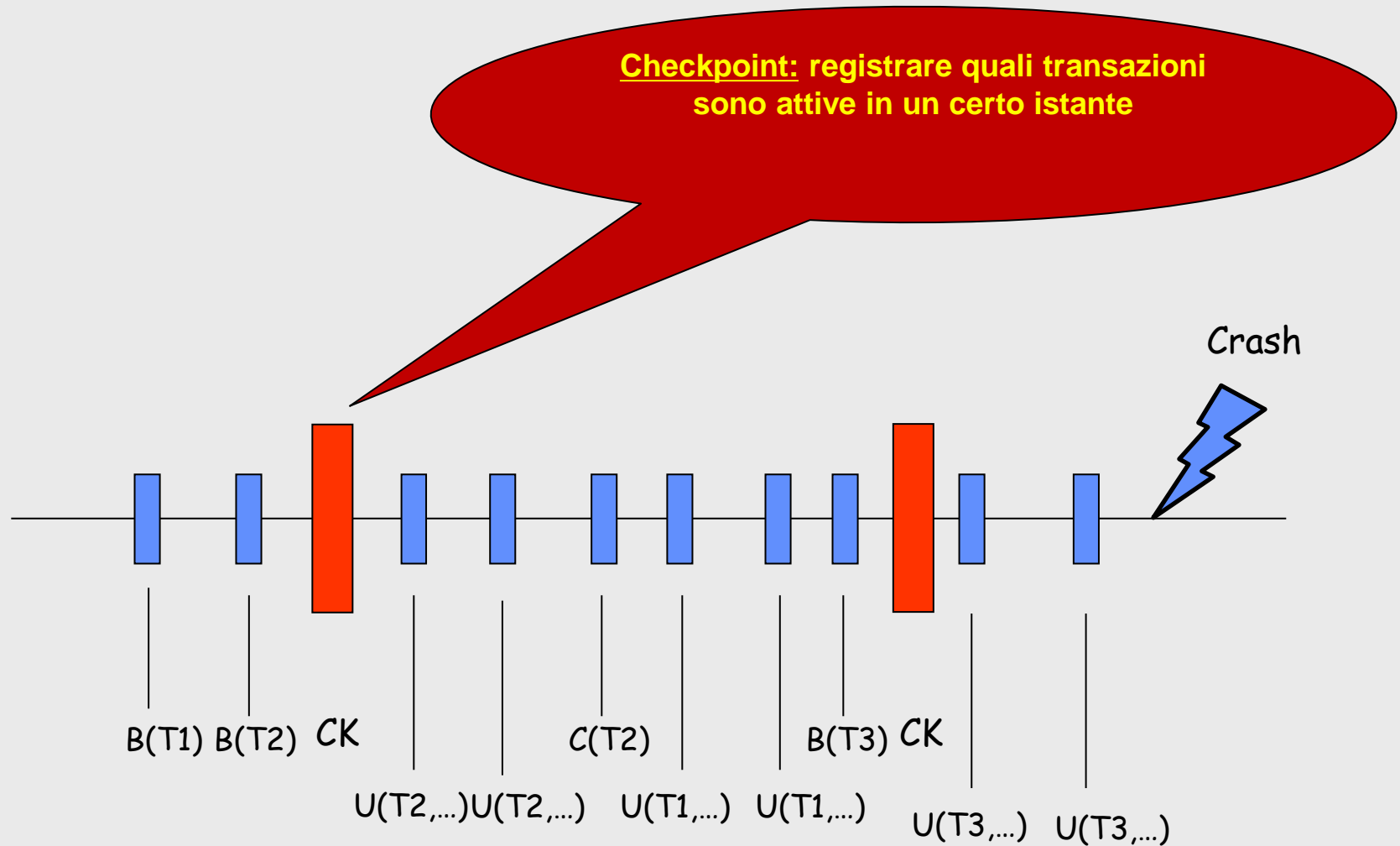
Modalità Differita

- Il DB non contiene valori AS provenienti da transazioni uncommitted
- In caso di abort, non occorre fare niente

Modalità Mista

- La scrittura può avvenire in modalità sia immediata che differita
- Consente l'ottimizzazione delle operazioni di Flush

Struttura del log (incompleta)



Checkpoint

- Operazione che serve a "fare il punto" della situazione: registrare quali transazioni attive al momento del check-point
- Varie modalità, vediamo la più semplice:
 1. si sospende l'accettazione di richieste di ogni tipo (scrittura, inserimenti, ..., commit, abort)
 2. si trasferiscono in memoria di massa (tramite *force*) tutte le pagine «sporche» relative a transazioni andate in commit
 3. si registrano sul log in modo sincrono (*force*) gli identificatori delle transazioni in corso
 4. si riprende l'accettazione delle operazioni

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)

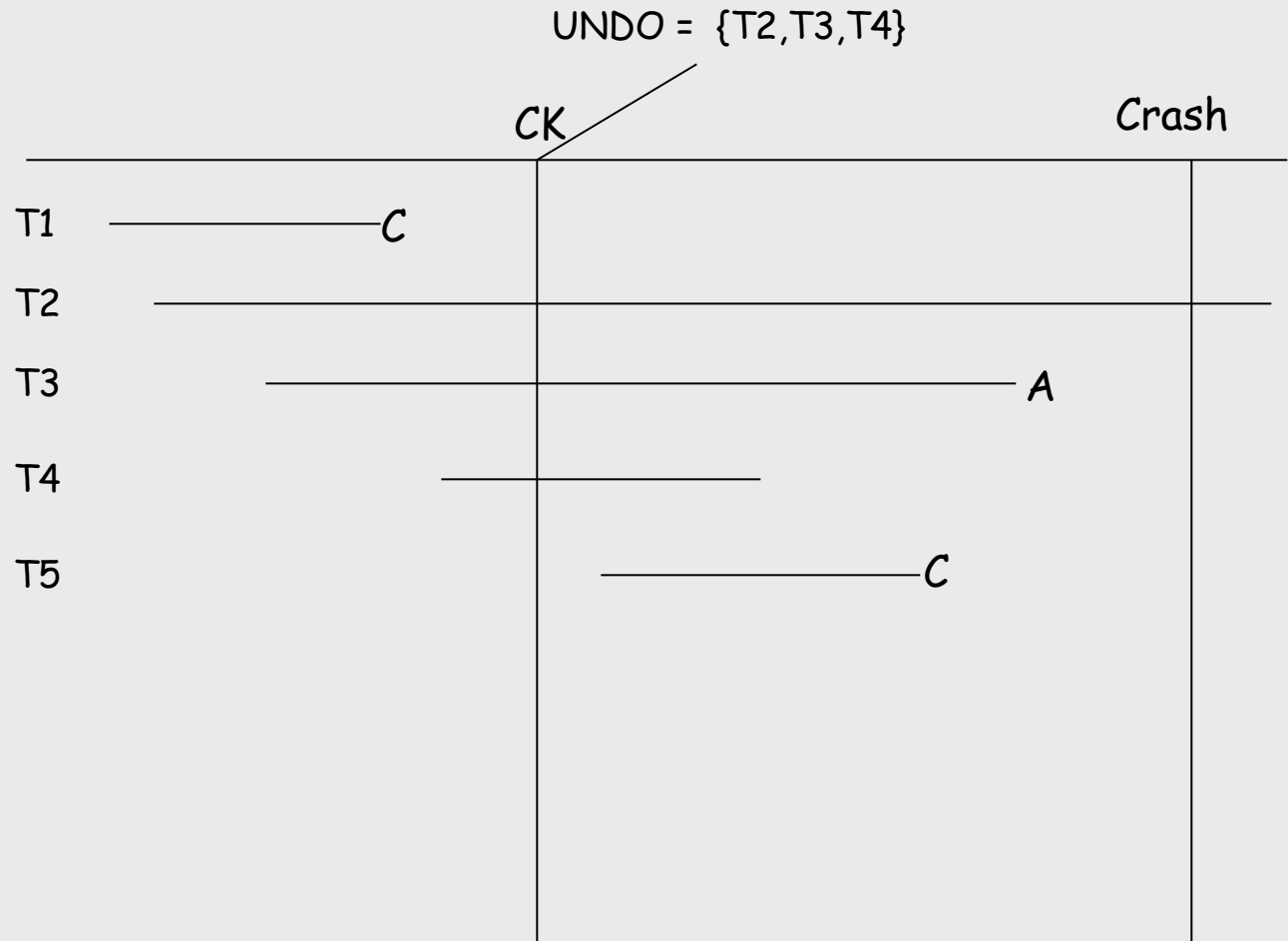
Ripresa a caldo

Quattro fasi:

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

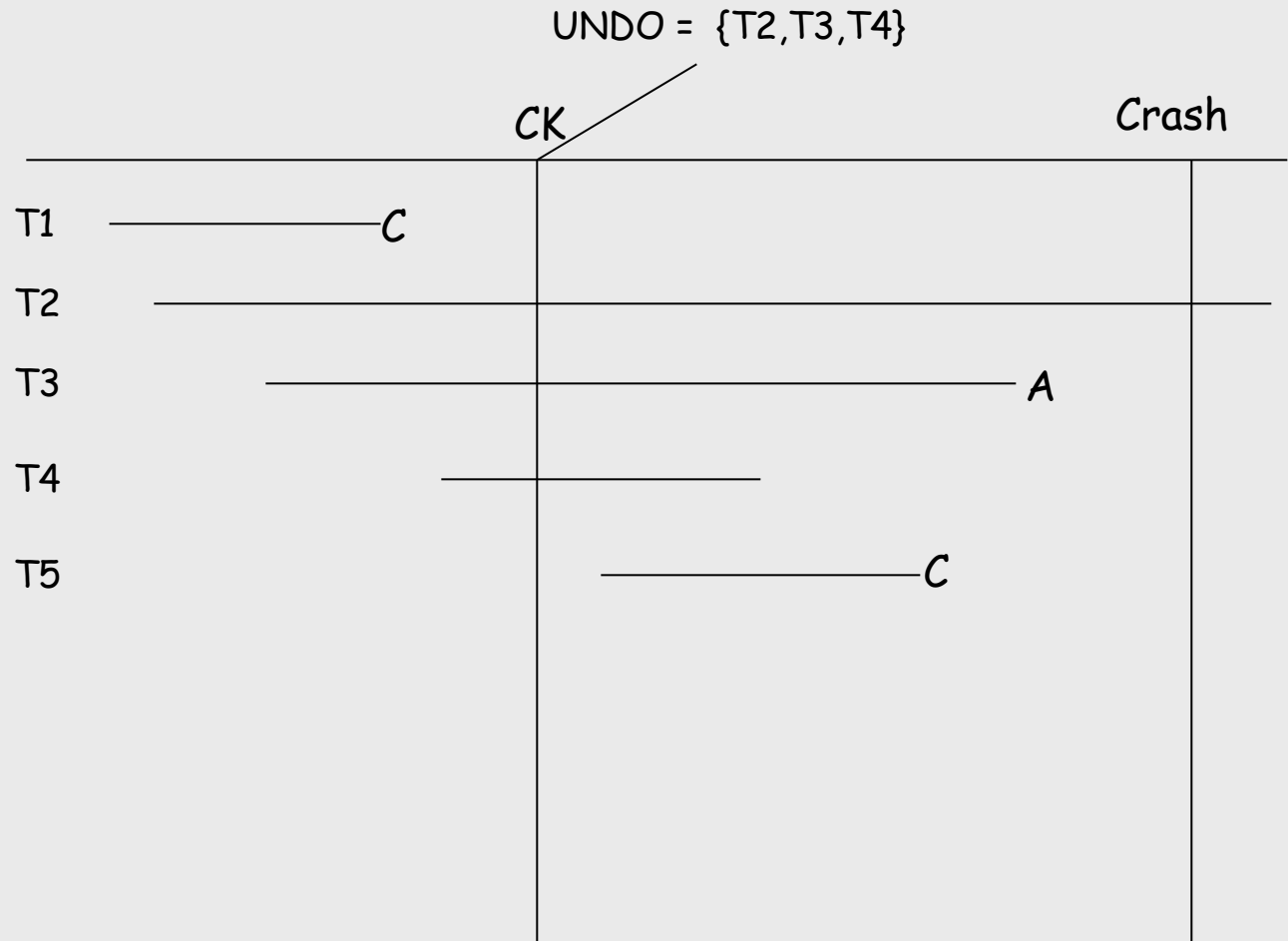
Esempio di Ripresa a Caldo

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)	0. UNDO = {T2,T3,T4}. REDO = {}	
B(T2)		
8. U(T2, O1, B1, A1)	1. C(T4) → UNDO = {T2, T3}. REDO = {T4}	
I(T1, O2, A2)	2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}	Setup
B(T3)	3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}	
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)		

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)

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

7. O2 = B3

8. O1 = B1

Undo

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

7. O2 = B3

8. O1 = B1

9. O3 = A4

10. O4 = A6

Undo

Redo



Necessità di Undo e/o Redo?

- Se il DB è scritto immediatamente dopo il log (“Modalità Immediata”), REDO non necessario
- Se il DB è scritto solo dopo un commit (“Modalità Differita”), UNDO non necessario

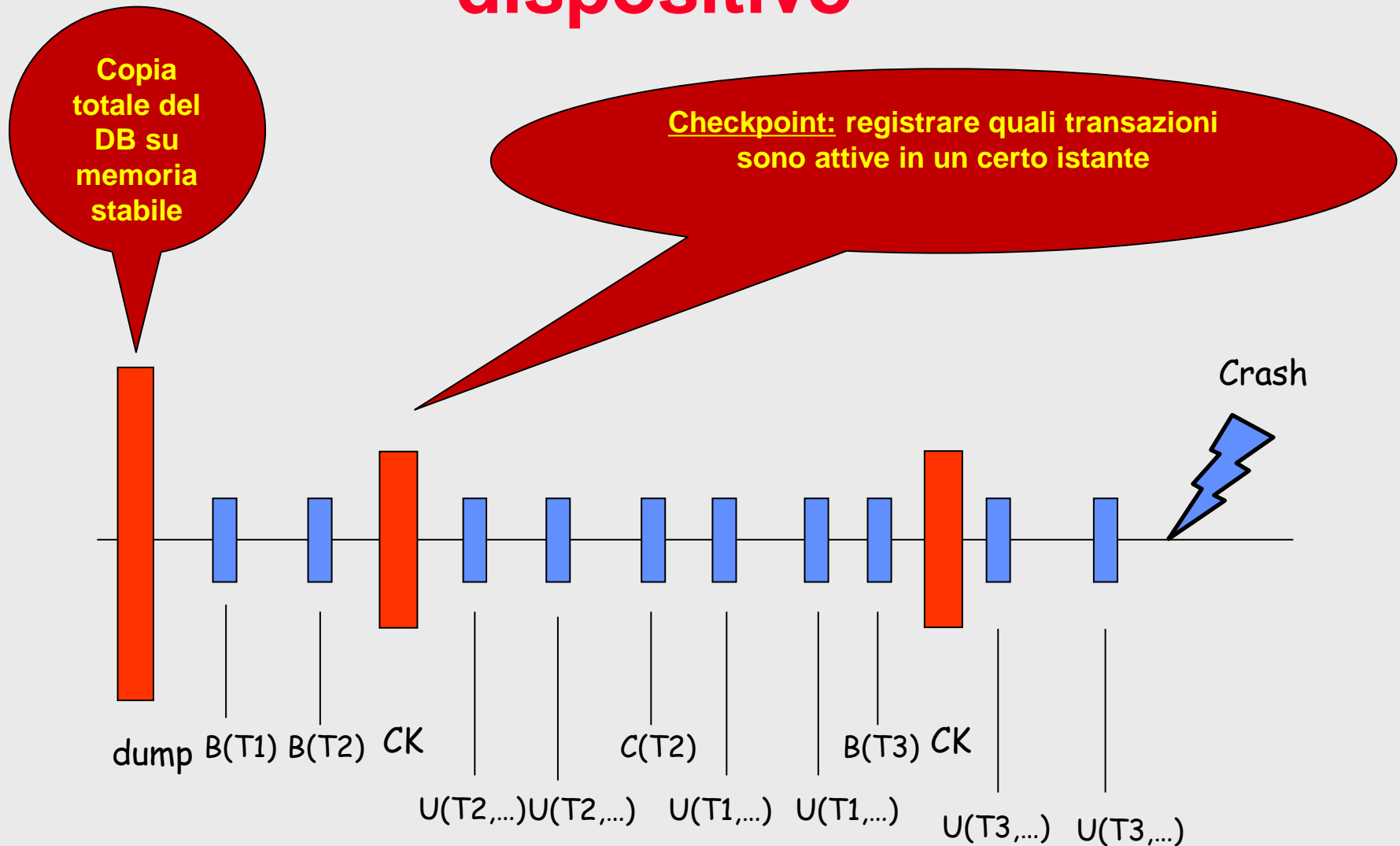
Se Modalità di scrittura nel DB è ibrida

- È possibile che viene fatto lo “undo” di operazioni il cui effetto non è già più nel database (quindi già “undo”)
→ Necessario che $\text{undo}(\text{undo}(A)) = \text{undo}(A)$
- È possibile che viene fatto il “redo” di operazioni il cui effetto è già database (quindi già “redo”)
→ Necessario che $\text{redo}(\text{redo}(A)) = \text{redo}(A)$

Guasti

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

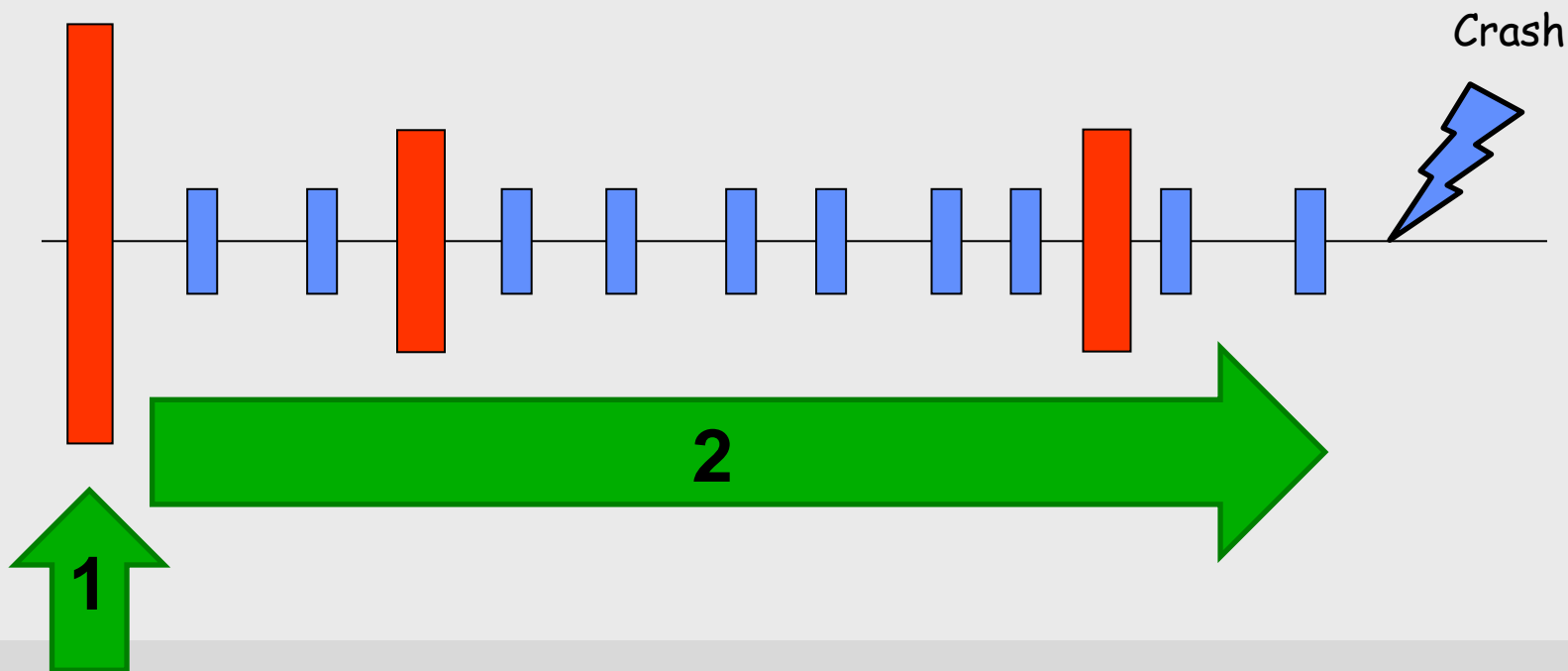
Struttura del log per guasti dispositivo



Ripresa a freddo

Da usare in caso di guasti di dispositivo

1. Si ripristinano i dati a partire dal backup
2. Si eseguono le operazioni registrate sul log fino all'istante del guasto
3. Si esegue una ripresa a caldo

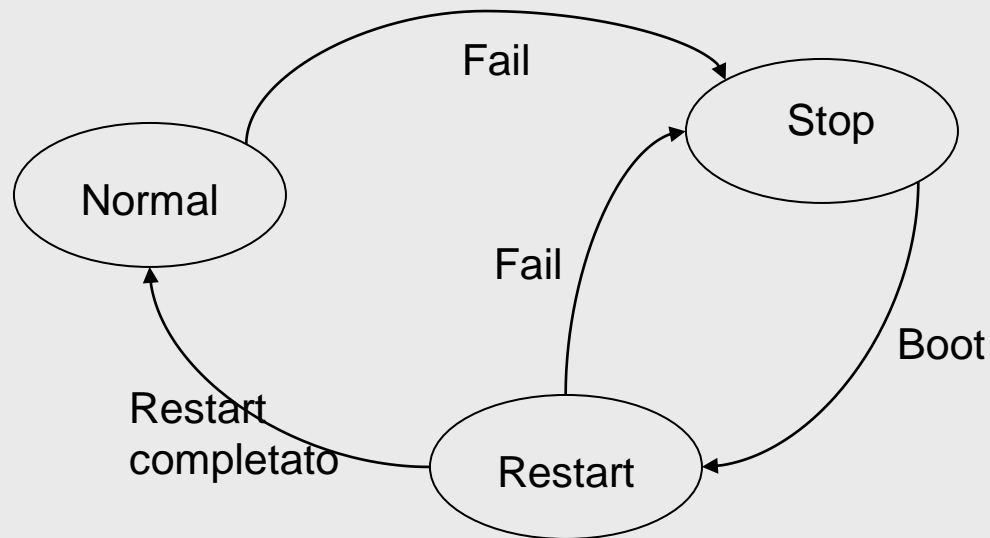


Dump

- 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, ...)

Modello "fail-stop"

- Si va in “stop”, quando c’è un problema (necessità di warm or cold restart)
- Quando in “stop”, c’è il DBMS è fatto ripartire (Boot)
- Se failure durante “boot”, di nuovo in “stop”
- Se “boot” è completo, si fa warm/cold restart



Atomicità e Persistenza versus Concorrenza

- Il problema di Atomicità e Persistenza è garantito da:
 - Commit/Abort
 - Warn/Cold Restart
- **Occorre ora pensare alla Concorrenza!**
- La concorrenza è fondamentale:
 - Decine o centinaia di transazioni al secondo
 - Le transazioni non possono essere seriali

**PROBLEMI
E
GESTIONE
DELLA CONCORRENZA**

Modello & Problema di Controllo di concorrenza

- **Modello:** operazioni di input-output su oggetti astratti x, y, z
 - Transazione $t1 : r(x), x = x + 1, w(x)$
 - Transazione $t2 : r(x), y = x + 1, w(y)$
- **Problema:** anomalie causate dall'esecuzione concorrente, che quindi va governata

Perdita di aggiornamento

- Due transazioni identiche con $x=2$ prima delle seguenti transazioni:
 - $t_1 : r(x), x = x + 1, w(x)$
 - $t_2 : r(x), x = x + 1, w(x)$
- Dopo un'esecuzione seriale?
 $x=4$
- Ma $x=3$, dopo la seguente esecuzione concorrente:

t_1
bot
 $r_1(x)$
 $x = x + 1$

$w_1(x)$
commit

t_2

bot
 $r_2(x)$
 $x = x + 1$

$w_2(x)$
commit

Lettura sporca

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

Aspetto critico: t_2 ha letto uno stato intermedio ("sporco") e lo può comunicare all'esterno

Lecture inconsistenti

- t_1 legge due volte:

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

- t_1 legge due valori diversi per x !

Aggiornamento fantasma

- Assumiamo vincolo $y + z = 1000$:

t_1
bot
 $r_1(y)$

t_2

bot
 $r_2(y)$
 $y = y - 100$
 $r_2(z)$
 $z = z + 100$
 $w_2(y)$
 $w_2(z)$
commit

$r_1(z)$
 $s = y + z$
commit

- $s = 1100$: il vincolo sembra non soddisfatto, t_1 vede un aggiornamento non coerente

Inserimento fantasma

t_1

bot

"calcola media stipendi"

"calcola media stipendi"


commit

t_2

bot

"inserisce un impiegato"

commit



La media degli stipendi
può cambiare per via del
nuovo impiegato,
comparso
improvvisamente come
uno "spettro"

Anomalie

Anomalia	Quando potenzialmente accade
Perdita di aggiornamento	Due transazioni scrivono lo stesso dato
Lettura sporca	Transazione legge un dato scritto da un'altra transazione che poi ha abortito
Letture inconsistenti	Transazione legge lo stesso dato in due momenti ma la seconda volta legge un dato aggiornato da un'altra transazione
Aggiornamento fantasma	Un dato appare "improvvisamente" aggiornato
Inserimento fantasma	Un nuovo dato appare "improvvisamente"

Gestione della concorrenza in SQL / 1

- Le transazioni possono essere definite **read-only**
- Il livello di isolamento può essere scelto per ogni transazione: **read uncommitted, read committed, repeatable read, serializable**
- Sintassi SQL:
BEGIN TRANSACTION ISOLATION LEVEL [valore];

Anomalia	RU	RC	RR	S
Perdita di aggiornamento	X	X	X	X
Lettura sporca	-	X	X	X
Lecture inconsistenti	-	-	X	X
Aggiornamento fantasma	-	-	X	X
Inserimento fantasma	-	-	-	X

X = il livello di isolamento garantisce l'assenza dell'anomalia

Gestione della concorrenza in SQL / 2

Perchè non sempre serializable?

- Un livello che “garantisce di più” richiede più risorse e blocca le transazioni.
- Occorre definire il livello che serve in funzione delle operazioni che accadono nella transazione

Anomalia	RU	RC	RR	S
Perdita di aggiornamento	X	X	X	X
Lettura sporca	-	X	X	X
Lecture inconsistenti	-	-	X	X
Aggiornamento fantasma	-	-	X	X
Inserimento fantasma	-	-	-	X

X = il livello di isolamento garantisce l'assenza dell'anomalia

Schedule

- Sequenza di operazioni di input/output operations di transazioni concorrenti

- Esempio:

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

- Ipotesi semplificativa →
ignoriamo le transazioni che vanno in abort,
rimuovendo tutte le loro azioni dallo schedule
(**commit-proiezione**)

Controllo di concorrenza per evitare anomalie

- *Scheduler*: un sistema che accetta o rifiuta (o riordina) le operazioni richieste dalle transazioni
- *Schedule seriale*: le transazioni sono separate, una alla volta

$S_2 : r_0(x) \ r_0(y) \ w_0(x) \ r_1(y) \ r_1(x) \ w_1(y) \ r_2(x) \ r_2(y) \ r_2(z) \ w_2(z)$

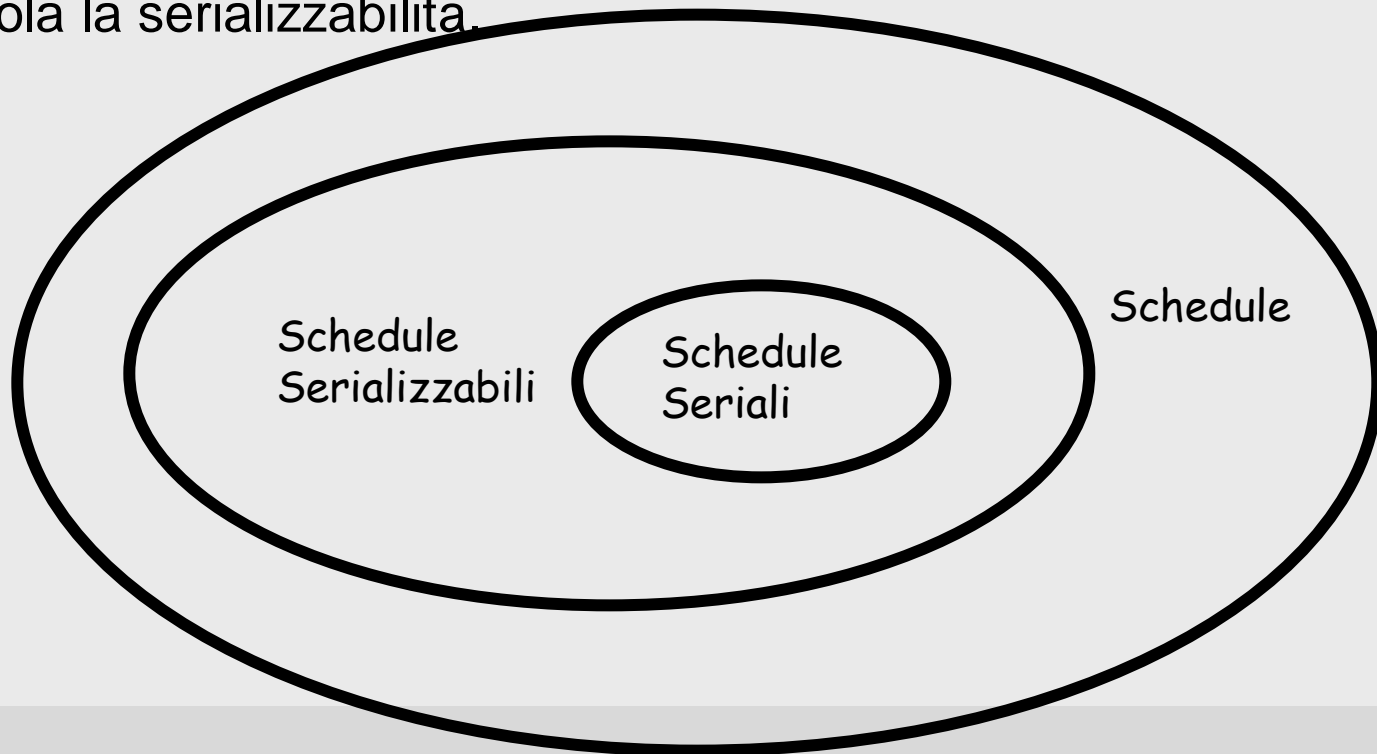
- *Schedule serializzabile*: produce lo stesso risultato di uno schedule seriale sulle stesse transazioni
- Richiede una nozione di equivalenza fra schedule

Idea base

- Lo schedule è costruito mentre si eseguono le transazioni:

$S : r_0(x) \ r_0(y) \ w_0(x) \ r_1(y) \ r_1(x) \ w_1(y) \ r_2(x) \ \dots$

- Quando una transazione fa il commit/abort, tutte le sue operazioni sono rimosse dallo schedule
- Se una operazione produce uno schedule non serializzabile, l'operazione viene messa in attesa finché la sua esecuzione non viola la serializzabilità.



View-Serializzabilità

- Definizioni preliminari:
 - $r_i(x)$ **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 $r_i(x)$ e $w_j(x)$ in S
 - $w_i(x)$ in uno schedule S è **scrittura finale** se è l'ultima scrittura dell'oggetto x in S
- Esempio: $S_3 : w_0(x) r_2(x) r_1(x) w_2(x) w_2(z)$
 - $r_2(x)$ legge da $w_0(x)$
 - $r_1(x)$ legge da $w_0(x)$
 - *Scritture finali*: $w_0(x)$, $w_2(x)$, $w_2(z)$

View-Serializzabilità

- Definizioni preliminari:
 - $r_i(x)$ **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 $r_i(x)$ e $w_j(x)$ 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
- Esempio:
$$S_3 : w_0(x) \ r_2(x) \ r_1(x) \ w_2(x) \ w_2(z) \approx_V$$
$$S_4 : w_0(x) \ r_1(x) \ r_2(x) \ w_2(x) \ w_2(z)$$

View-Serializzabilità

- Schedule **view-equivalenti** ($S_i \approx_V S_j$): hanno la stessa relazione **legge-da** e le stesse scritture finali
- Esempio:
$$S_3 : w_0(x) \ r_2(x) \ r_1(x) \ w_2(x) \ w_2(z) \approx_V$$
$$S_4 : w_0(x) \ r_1(x) \ r_2(x) \ w_2(x) \ w_2(z)$$
- Uno schedule è **view-serializzabile** (VSR) se è view-equivalente ad un qualche schedule seriale:
 - Esempio S_3 è view-serializzabile perchè view-equivalente con S_4 che è seriale.

Garanzie della View-Serializzabilità

Assenza di

- 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 sono tutte non view-serializzabili

View serializzabilità

- Complessità:
 - la verifica della view-equivalenza di due dati schedule è lineare sulla lunghezza dello schedule
 - decidere sulla view serializzabilità di uno schedule S è un problema “difficile” perchè occorre provare tutte i possibili schedule seriali, ottenuti per permutazioni dell'ordine delle transazioni.
- La “difficoltà” della verifica non lo fa utilizzabile in Pratica
- La verifica deve essere più facile.

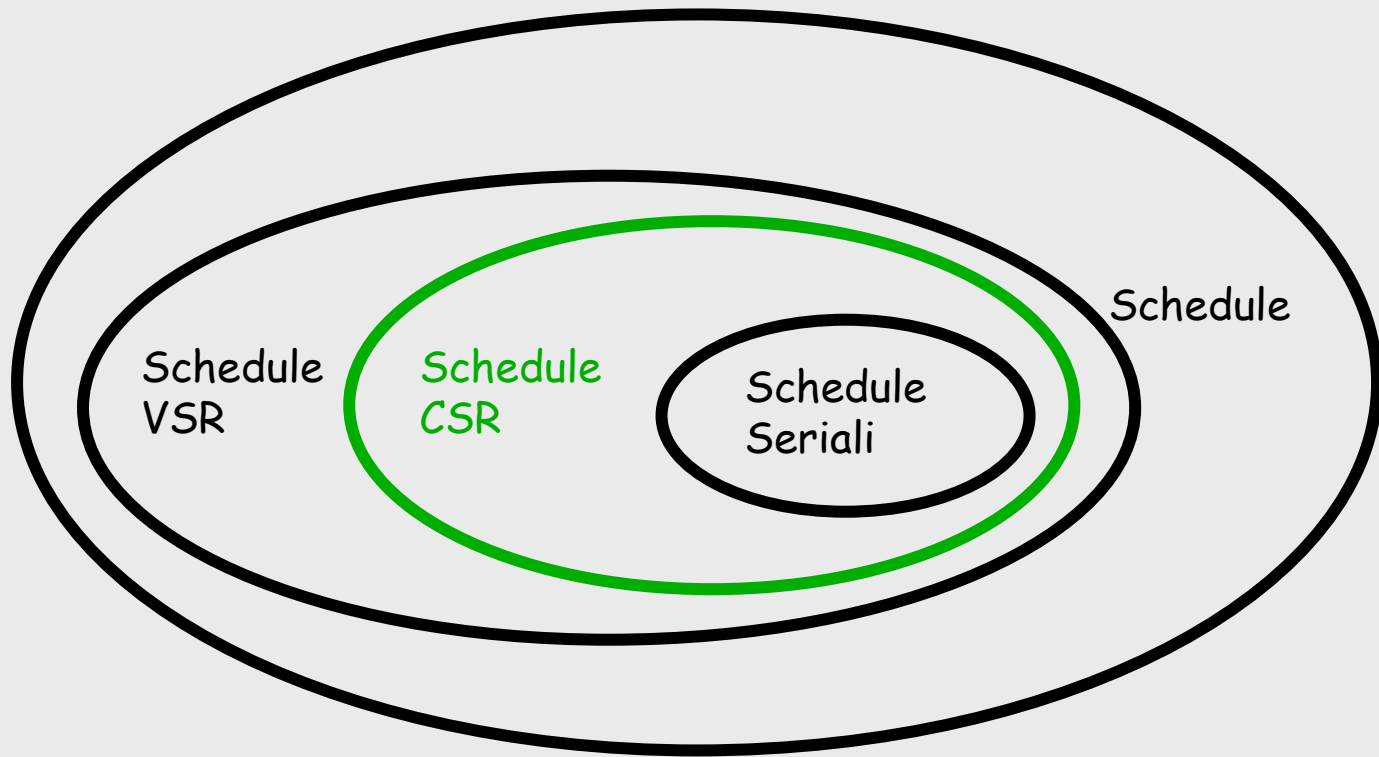
Una verifica più “facile”: Conflict-serializzabilità

- 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 view-equivalente ad un qualche schedule seriale
- L'insieme degli schedule conflict-serializzabili è indicato con **CSR**

CSR implica VSR ma VSR non implica CSR

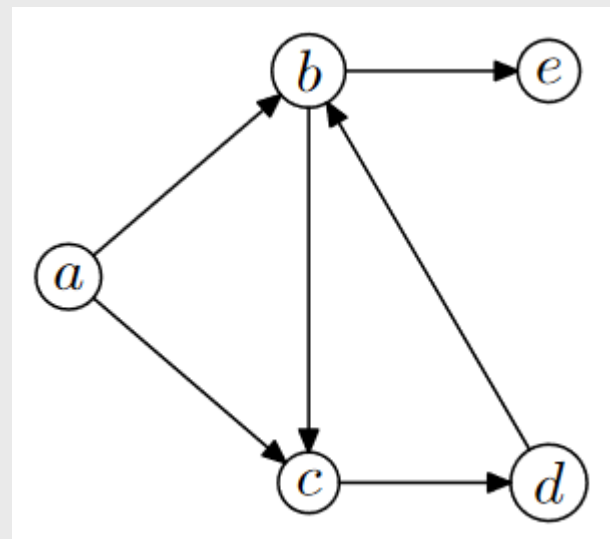
- Ogni schedule conflict-serIALIZZABILE (CSR) è view-serIALIZZABILE (VSR)
- Ci sono schedule view-serIALIZZABILI (VSR) che non sono conflict-serIALIZZABILI (CSR)
- Esempio per dimostrare che **VSR \nRightarrow CSR**
 $S_1 : r_1(x) w_2(x) w_1(x) w_3(x)$
 - VSR: View-equivalenza: $S_1 \approx_v r_1(x) w_1(x) w_2(x) w_3(x)$
 - non CSR
 - $r_1(x) w_1(x) w_2(x) w_3(x)$ inverte $w_1(x)$ e $w_2(x)$
 - $w_2(x) r_1(x) w_1(x) w_3(x)$ inverte $r_1(x)$ e $w_2(x)$
- Dimostrazione che **CSR \Rightarrow VSR** viene omessa

CSR e VSR



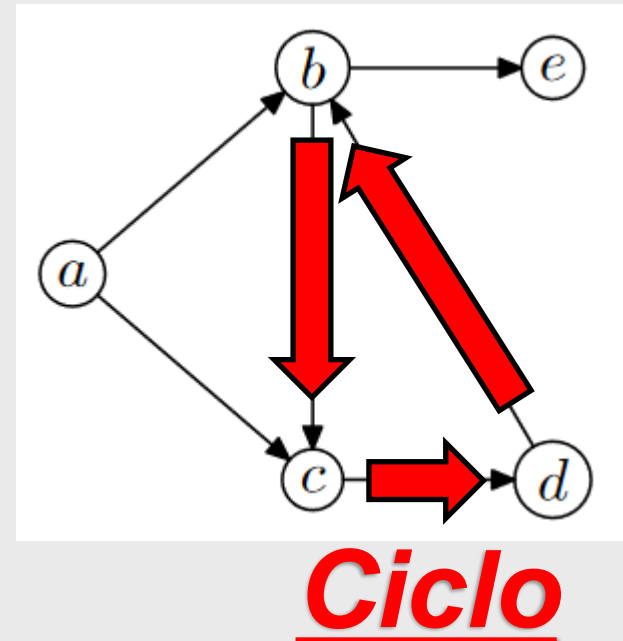
Grafo Orientato

- Grafo è una struttura formata da
 - **Nodi**: punti nel piano
 - **Archi**: frecce che congiungono i punti/nodi dell'arco



Grafo Orientato

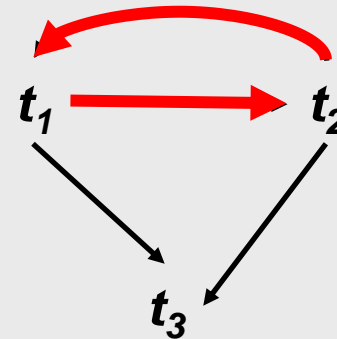
- Grafo è una struttura formata da
 - **Nodi**: punti nel piano
 - **Archi**: frecce che congiungono i punti/nodi dell'arco
- Grafo è **ciclico** se, partendo da anche un solo nodo **n**, è possibile “seguire” gli archi e tornare ad **n** percorrendo 1+ archi



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

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

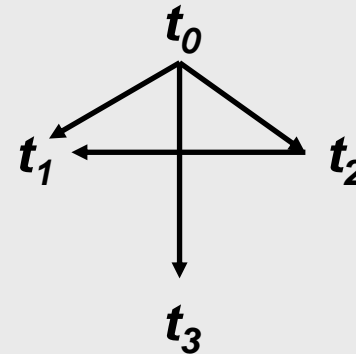


Ciclo \Rightarrow non CSR

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

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



Nessun Ciclo \Rightarrow CSR

Intrattabilità del problema

- Sistema con 100 transazioni al secondo
- Ogni transazione:
 - dura 5 secondi,
 - accede a 10 oggetti (pagine): 2 per secondo.
- Quindi, in ogni secondo ci sono 500 transazioni
- Ogni secondo occorre costruire un grafo con 500 nodi e potenzialmente fino a 5000 archi
- Non trattabile in pratica se non con basi di dati “poco usate”

Lock

- Principio:
 - Tutte le letture per una risorsa x sono precedute da $r_lock(x)$ (lock condiviso) e seguite da $unlock$
 - Tutte le scritture per una risorsa x sono precedute da $w_lock(x)$ (lock esclusivo) e seguite da $unlock$
- Quando una transazione prima legge e poi scrive una risorsa x , può:
 - richiedere subito $w_lock(x)$
 - chiedere prima $r_lock(x)$ e poi $w_lock(x)$ (*lock escalation*)

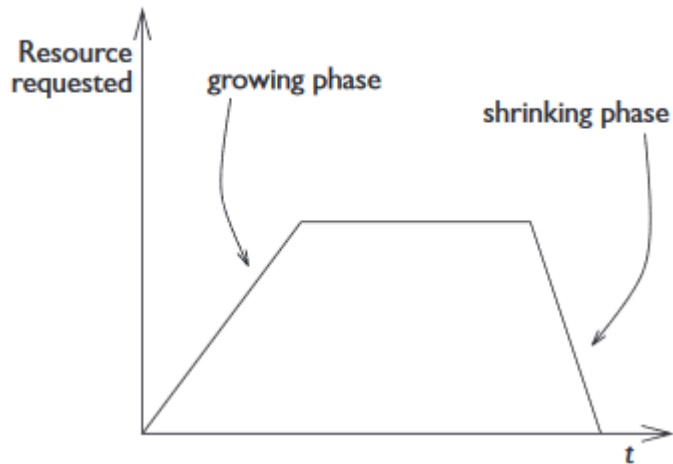
Gestione dei lock

- Basata sulla tavola dei conflitti. Per ogni risorsa :
 - Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata quando il contatore scende a zero
 - Un valore booleano tiene conto se c'è un *w_lock*

	<i>libera</i>	<i>r_locked</i>	<i>w_locked</i>
<i>r_lock(x)</i>	OK / r_locked conta(x)++	OK / r_locked conta(x)++	NO / w_locked
<i>w_lock(x)</i>	OK / w_locked	NO / r_locked	NO / w_locked
<i>unlock(x)</i>	error	OK / if (--conta(x)=0) libera else r_locked	OK / not w_locked

- Se la risorsa non è concessa, la transazione richiedente è posta in attesa (eventualmente in coda), fino a quando la risorsa non diventa disponibile

Locking a due fasi (2PL)



- Usato da quasi tutti i sistemi
- Garantisce "a priori" CSR sulla base di:
 - *Fase crescente*: si acquisiscono i lock necessari
 - *Fase decrescente*: si rilasciano i lock pian piano
- Locking a due fasi:
una transazione, dopo aver rilasciato un lock, non può acquisirne altri

Ritorniamo al problema di “Aggiornamento fantasma”

- Assumiamo vincolo $y + z = 1000$:

t_1
bot
 $r_1(y)$

t_2

bot
 $r_2(y)$
 $y = y - 100$
 $r_2(z)$
 $z = z + 100$
 $w_2(y)$
 $w_2(z)$
commit

$r_1(z)$
 $s = y + z$
commit

- $s = 1100$: il vincolo sembra non soddisfatto, t_1 vede un aggiornamento non coerente

Esempio:

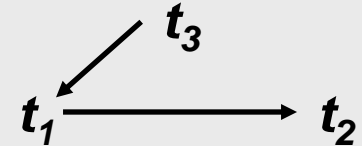
Locking a due fasi per aggiornamento fantasma

t_1	t_2	x	y	z
bot		free	free	free
$r_lock_1(x)$		1:read		
$r_1(x)$				
	bot			
	$w_lock_2(y)$		2:write	
	$r_2(y)$			
$r_lock_1(y)$			1:wait	
	$y = y - 100$			
	$w_lock_2(z)$			2:write
	$r_2(z)$			
	$z = z + 100$			
	$w_2(y)$			
	$w_2(z)$			
	commit			
	$unlock_2(y)$		1:read	
$r_1(y)$				
$r_lock_1(z)$				1:wait
	$unlock_2(z)$			1:read
$r_1(z)$				
	eot			
$s = x + y + z$				
commit				
$unlock_1(x)$		free		
$unlock_1(y)$			free	
$unlock_1(z)$				free
eot				

Relazione tra 2PL e CSR

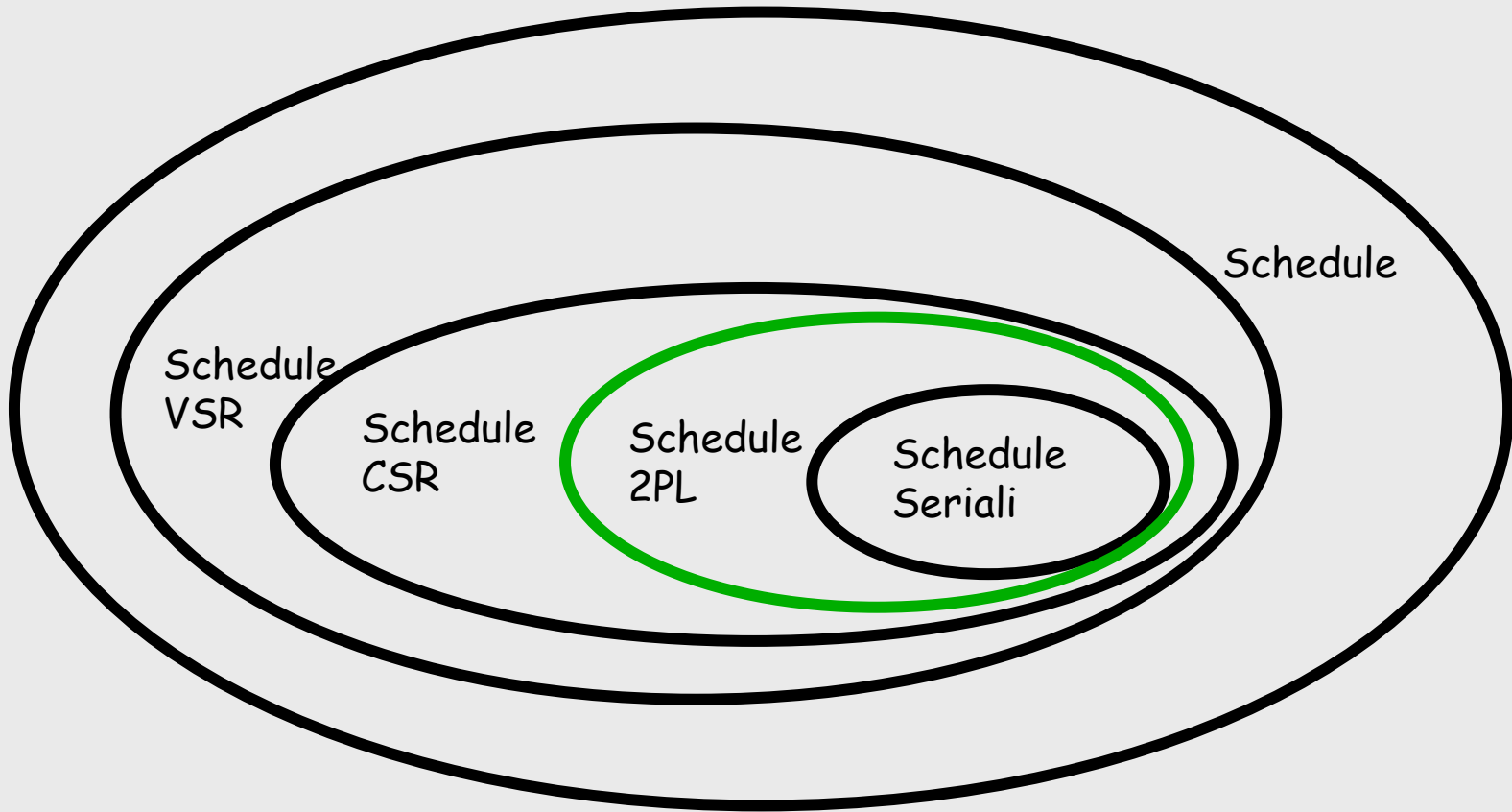
- Consideriamo:

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

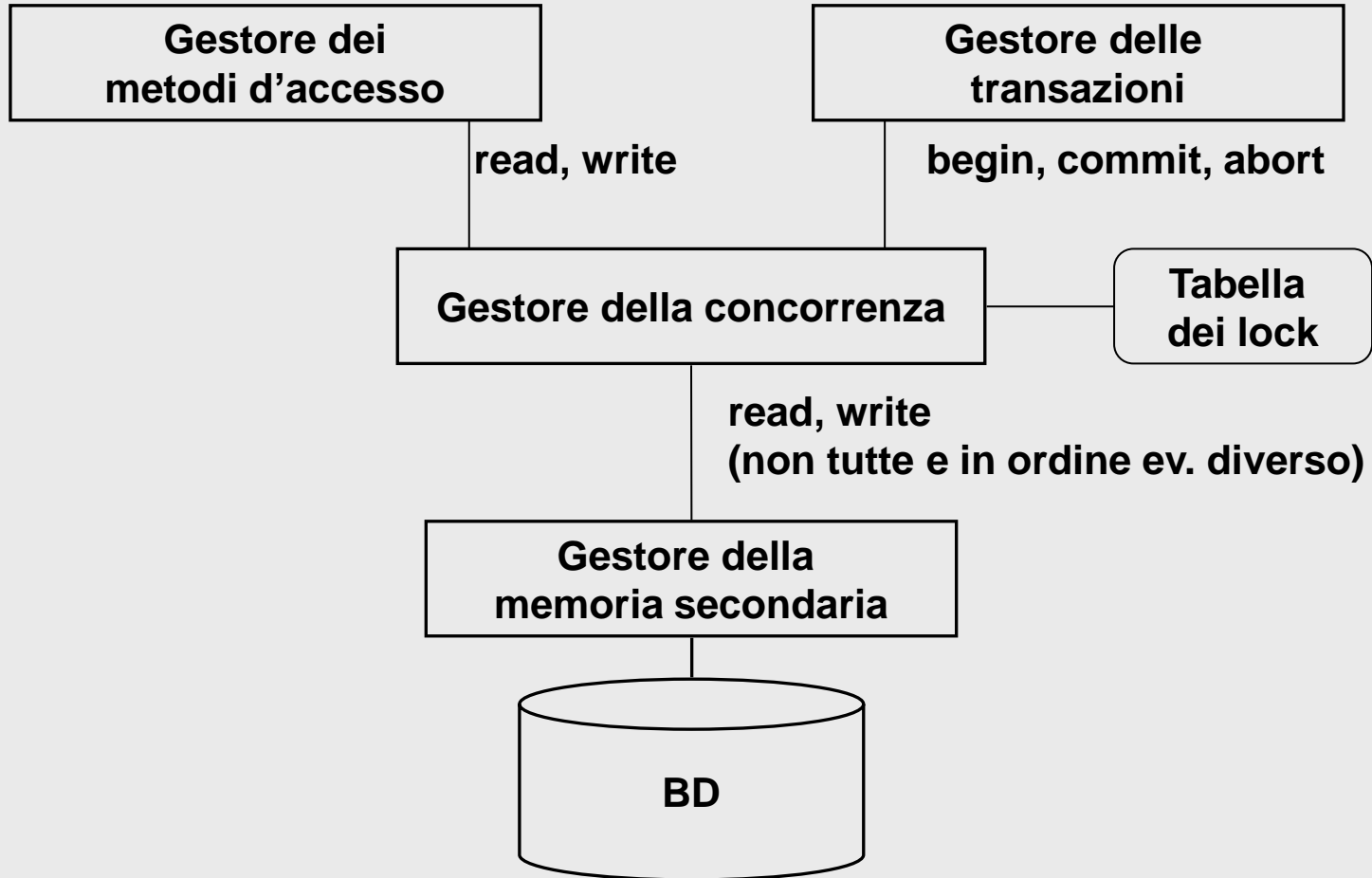


- S è CSR
- S non è 2PL, infatti:
 - Affinchè $r_2(x)$, è necessario $unlock_1(x)$:
 $w_lock_1(x) r_1(x) w_1(x) unlock_1(x) r_2(x) w_2(x) r_3(y) w_1(y)$
 - Ma c'è anche $w_1(y)$ quindi $w_lock_1(y)$ deve precedere $unlock_1(x)$:
 $\square w_lock_1(x) r_1(x) w_1(x) w_lock_1(y) unlock_1(x) r_2(x) w_2(x) r_3(y) w_1(y)$
 - Ma se $r_3(y)$ prima ci deve essere $unlock_2(y)$, che non è compatibile con $w_1(y)$ che segue
- Quindi $CSR \not\Rightarrow 2PL$
- Tuttavia $2PL \Rightarrow CSR$ (Dimostrazione Omessa)

CSR, VSR e 2PL



Per concludere (ignorando buffer e affidabilità)



Riferimenti

- Capitolo 12 fino a pagina 473, escludendo a partire dalla frase *“Qualche osservazione in più è necessaria per quando riguarda le anomalie di lettura sporca...”* (a cavallo tra pagina 473 e 474).
- Sono anche escluse le seguenti parti:
 - Dimostrazione che CSR se e solo se grafo conflitto aciclico: i due paragrafi nell'elenco puntato a pagina 469
 - Dimostrazione $2PL \Rightarrow CSR$: il primo paragrafo a pagina 472, cioè il paragrafo che inizia con “Dimostriamo, sia pure informalmente”