Capitolo 12

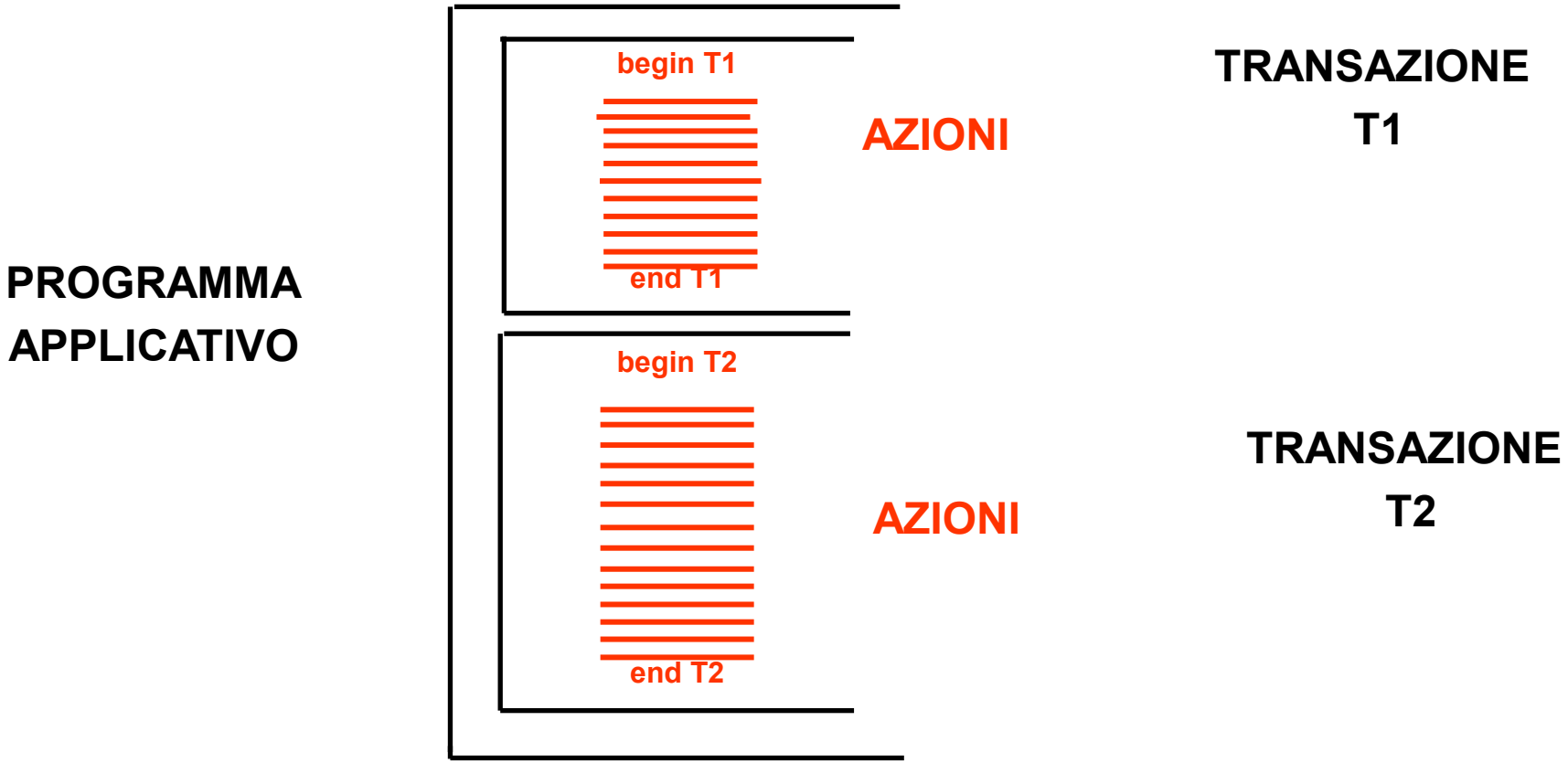
Gestione delle transazioni

Definizione di transazione

- Transazione: parte di programma caratterizzata da un inizio (**begin-transaction,** `start transaction` in SQL), 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

- Un **sistema transazionale** (**OLTP- Online Transaction Processing**) è in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti

Differenza fra applicazione e transazione



Una transazione

```
start transaction;
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;
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
  - Durata (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 svolte
  - un guasto o errore dopo il commit non deve avere conseguenze; se necessario vanno ripetute (REDO) le operaiozni
- Esito
  - Commit = caso "normale" e più frequente (99% ?)
  - Abort (o rollback)
    - ❑ richiesto dall'applicazione = suicidio
    - ❑ requested 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
  - l'esecuzione concorrente di una collezione di transazioni deve produrre un risultato che si potrebbe ottenere con una esecuzione sequenziale
- Concorrenti
- Conseguenza: una transazione non espone i suoi stati intermedi
  - Si evita l' "**effetto domino**"

Durabilità (Persistenza)

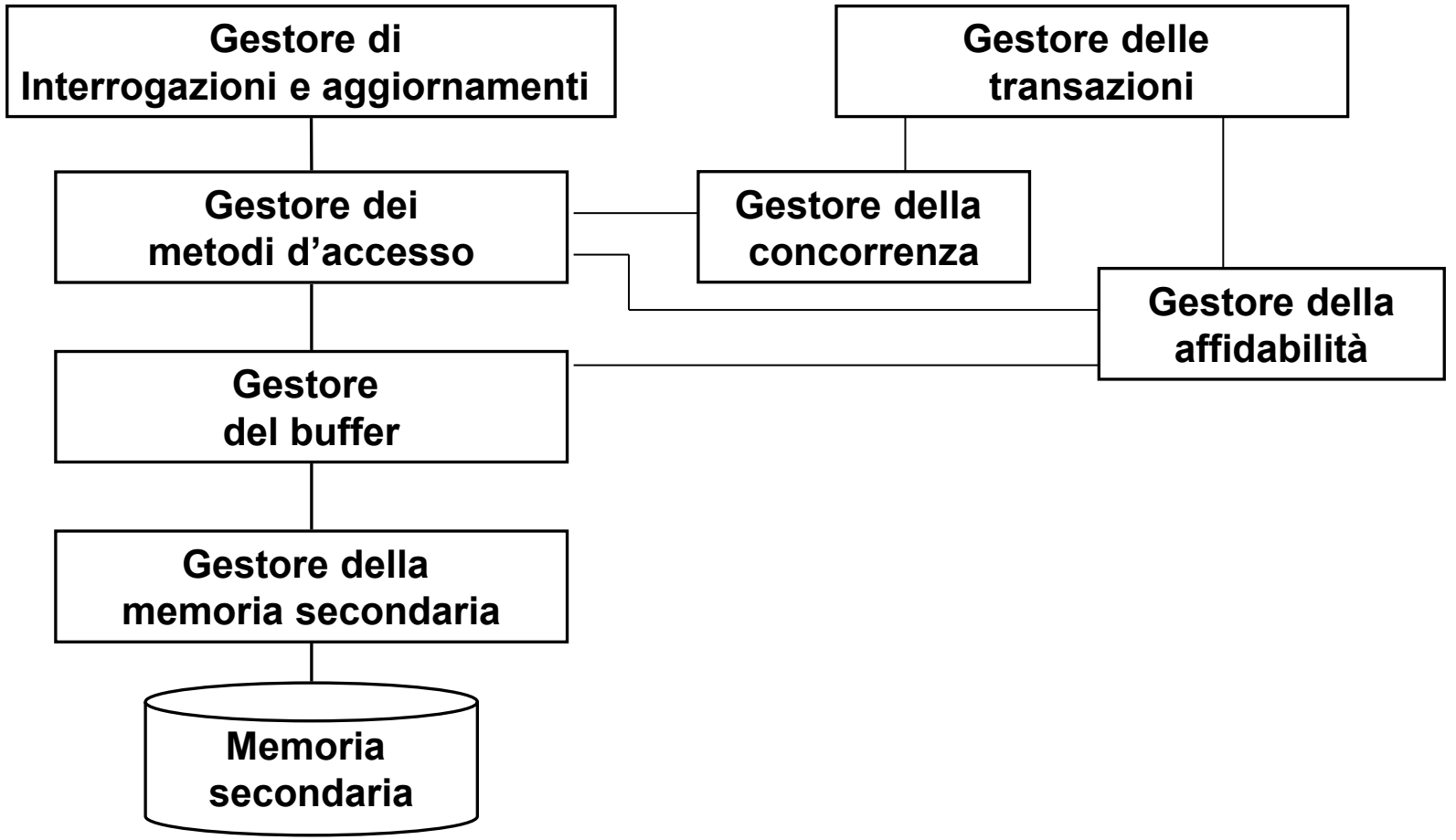
- Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"), anche in presenza di guasti
  - Commit significa **impegno**

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

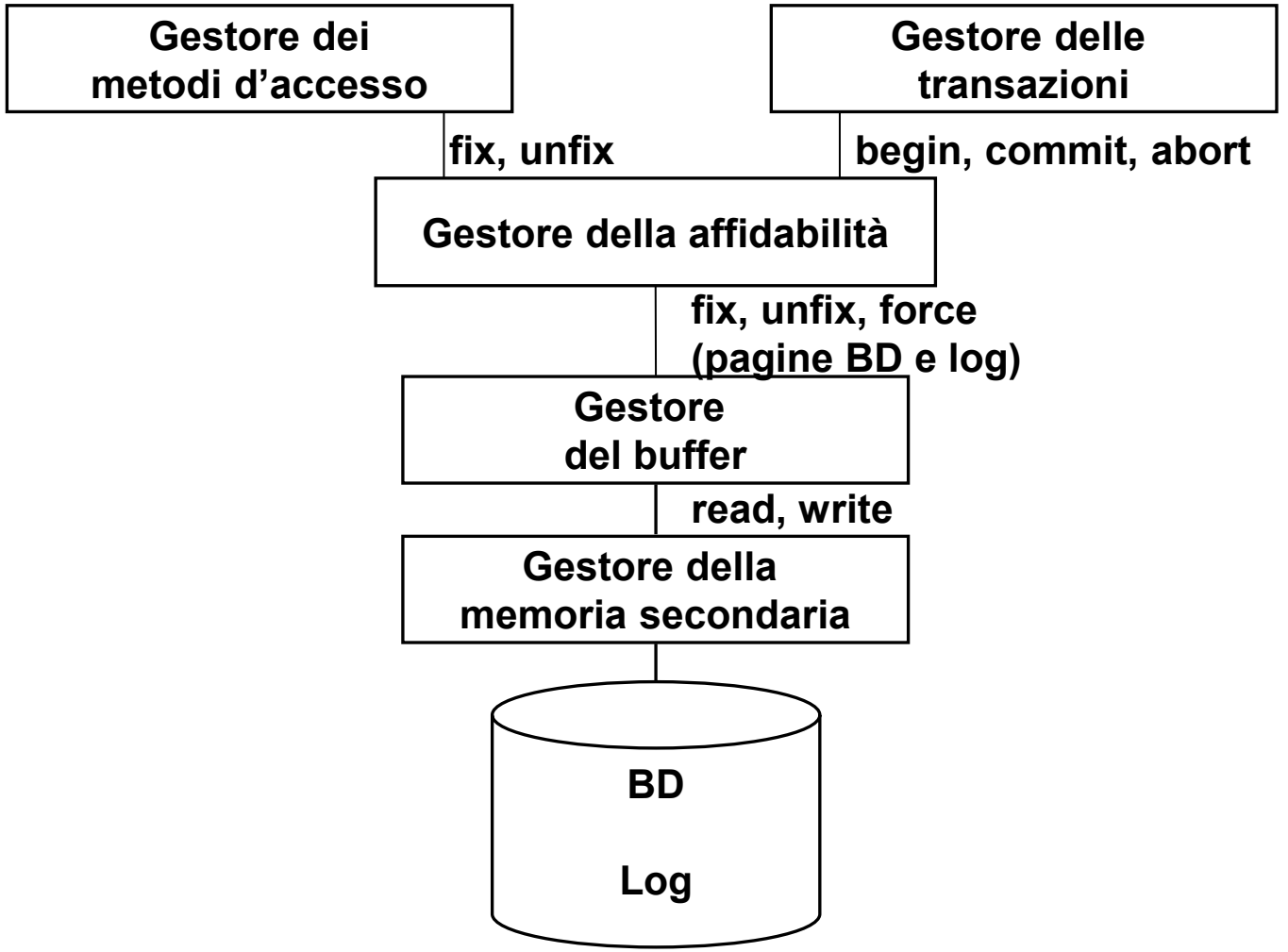


Gestore dell'affidabilità

- 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*
- Assicura atomicità e durabilità
- Usa il **log**:
  - Un archivio permanente che registra le operazioni svolte
  - Due metafore: il filo di Arianna e i sassolini e le briciole di Hansel e Gretel

Architettura del controllore dell'affidabilità



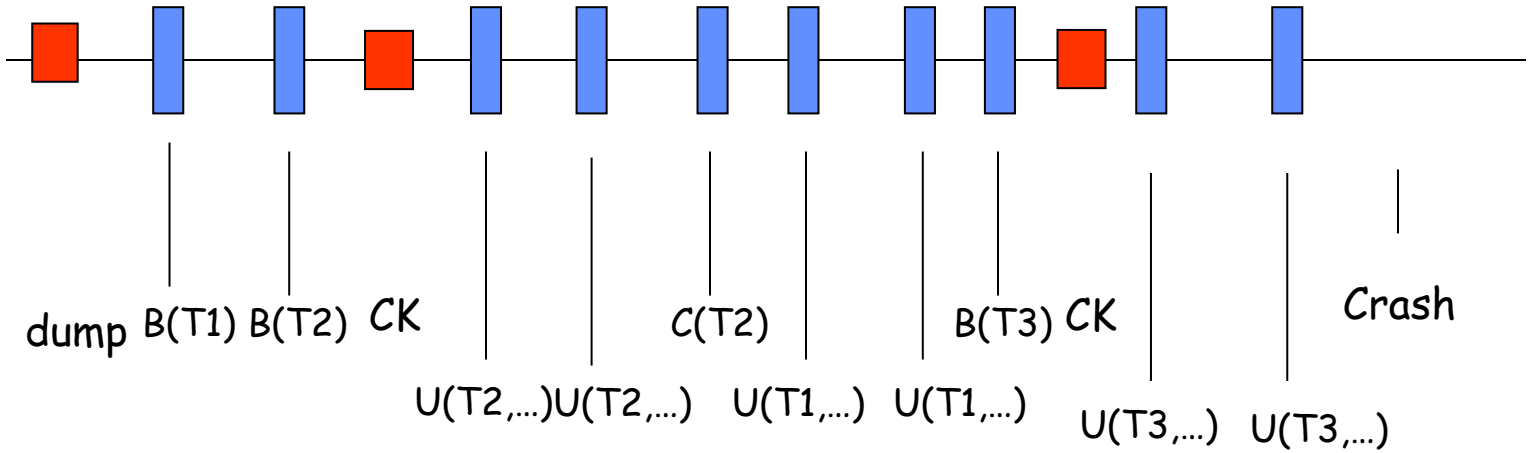
Persistenza delle memorie

- **Memoria centrale**: non è persistente
- **Memoria di massa**: è persistente ma può danneggiarsi
- **Memoria stabile**: memoria che non può danneggiarsi (è una astrazione):
  - perseguita attraverso la ridondanza:
    - dischi replicati
    - 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*
    - `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

Undo e redo

- Undo di un'azione su un oggetto *O:*
  - `update`, `delete`*:* copiare il valore del before state (*BS*) nell'oggetto *O*
  - `insert`: eliminare *O*
- Redo di una azione su un oggetto *O:*
  - `insert`, `update`: copiare il valore dell' after state (A*S*) nell'oggetto *O*
  - `delete`: reinserire *O*
- *Idempotenza* di *undo e redo:*
  - *undo*(*undo*(*A*)) = *undo*(*A*)
  - *redo*(*redo*(*A*)) = *redo*(*A*)

Checkpoint

- Operazione che serve a "fare il punto" della situazione, semplificando le successive operazioni di ripristino:
  - ha lo scopo di registrare quali transaszioni 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 25 novembre (ad esempio) non si accettano nuove richieste di "operazioni" e si concludono tutte quelle avviate prima di accettarne di nuove

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
  - 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

Dump

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

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 porta 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

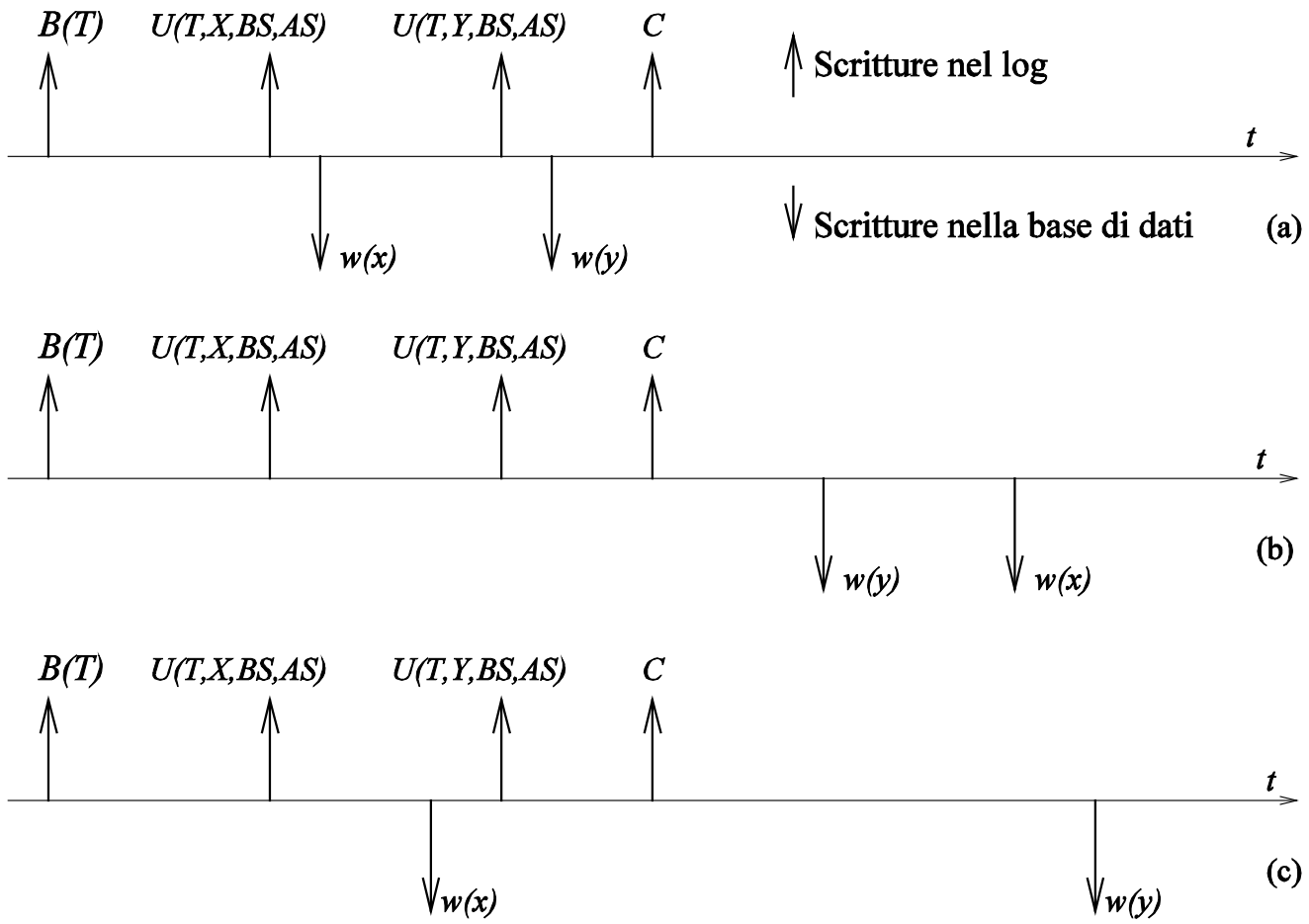
- record di `abort` possono essere scritti in modo asincrono

Regole fondamentali per il log

- **Write-Ahead-Log**:
  - si scrive Log (parte before) prima del database
    - ❑ consente di disfare le azioni
- **Commit-Precedenza**:
  - si scrive il Log (parte after) prima del commit
    - ❑ consente di rifare le azioni

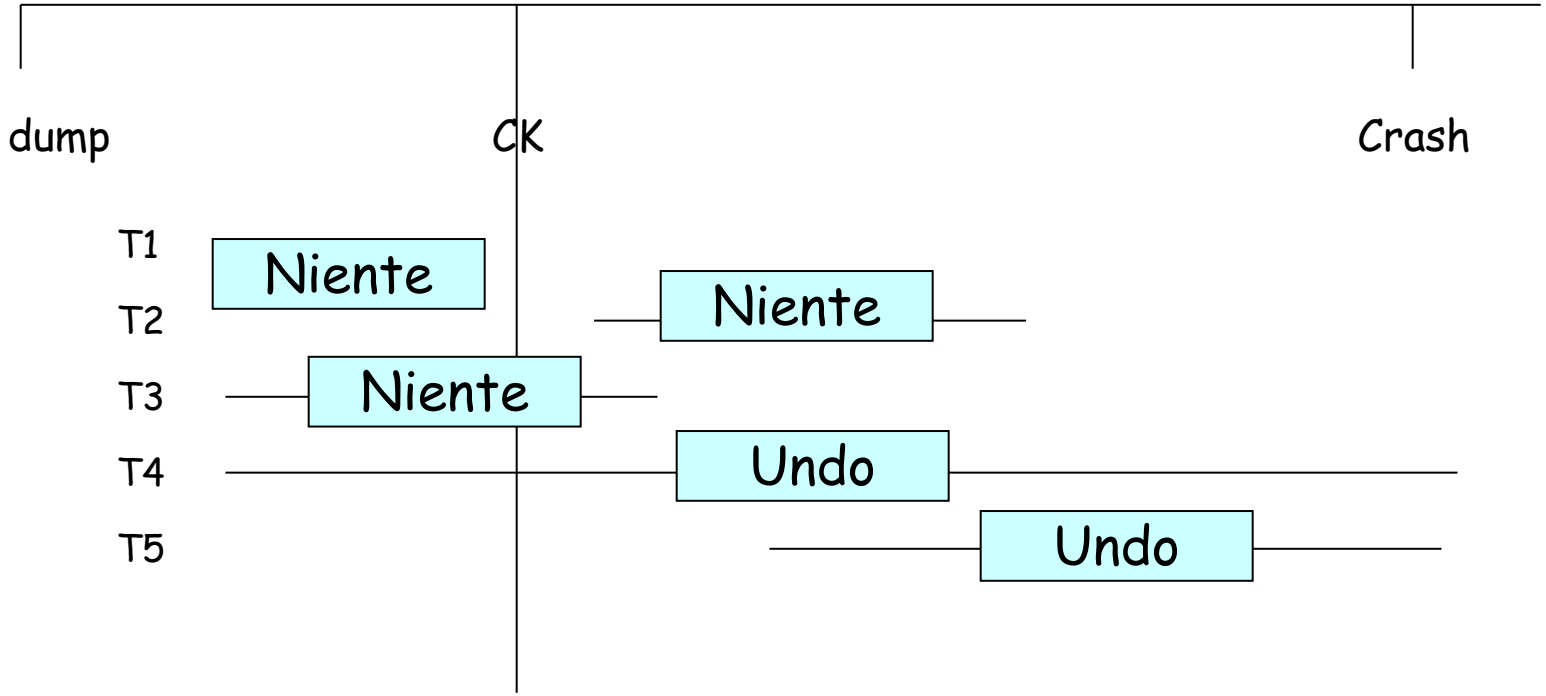
- Quando scriviamo nella base di dati?
  - Varie alternative

Scrittura nel log e nella base di dati



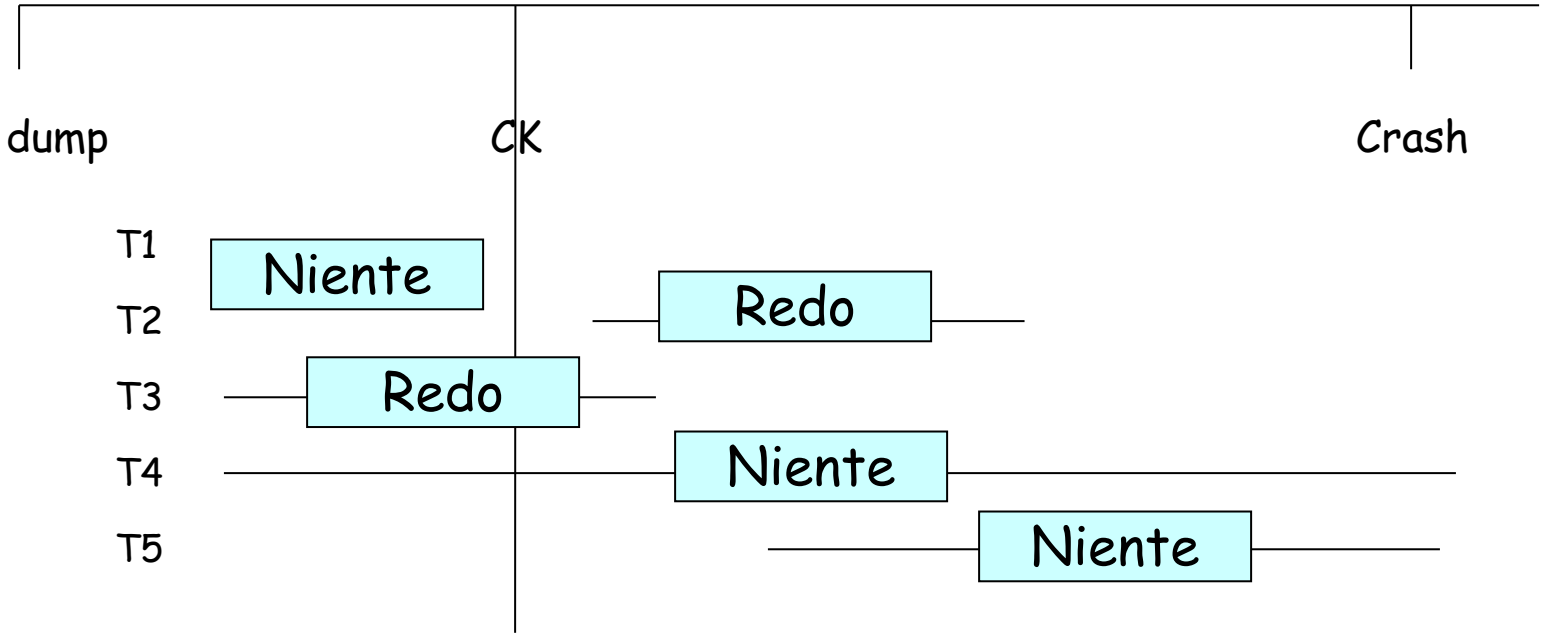
Modalità immediata

- Il DB contiene valori AS provenienti da transazioni uncommitted
- Richiede Undo delle operazioni di transazioni uncommited al momento del guasto
- Non richiede Redo



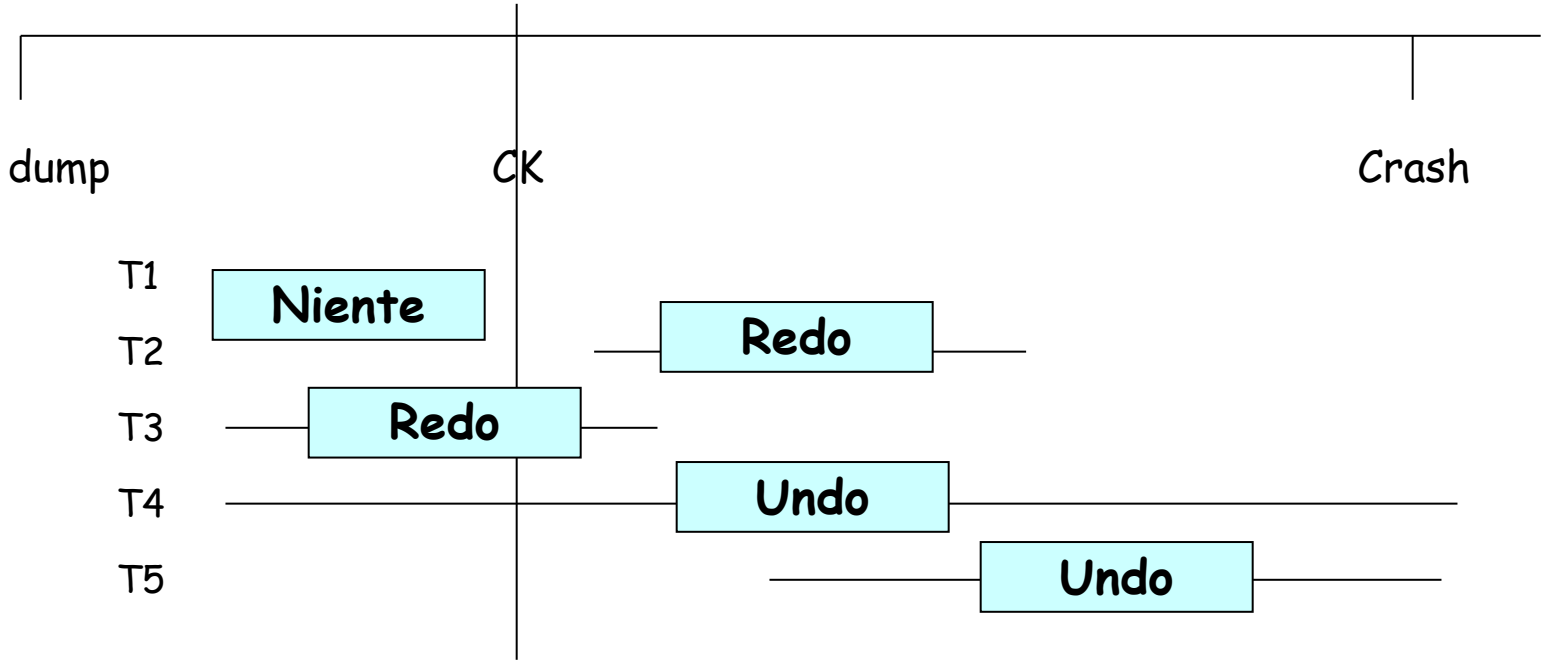
Modalità differita

- Il DB non contiene valori AS provenienti da transazioni uncommitted
- In caso di abort, non occorre fare niente
- Rende superflua la procedura di Undo. Richiede Redo



Essite una terza modalità: modalità mista

- La scrittura puo' avvenire in modalita' sia immediata che differita
- Consente l'ottimizzazione delle operazioni di flush
- Richiede sia Undo che Redo



Guasti

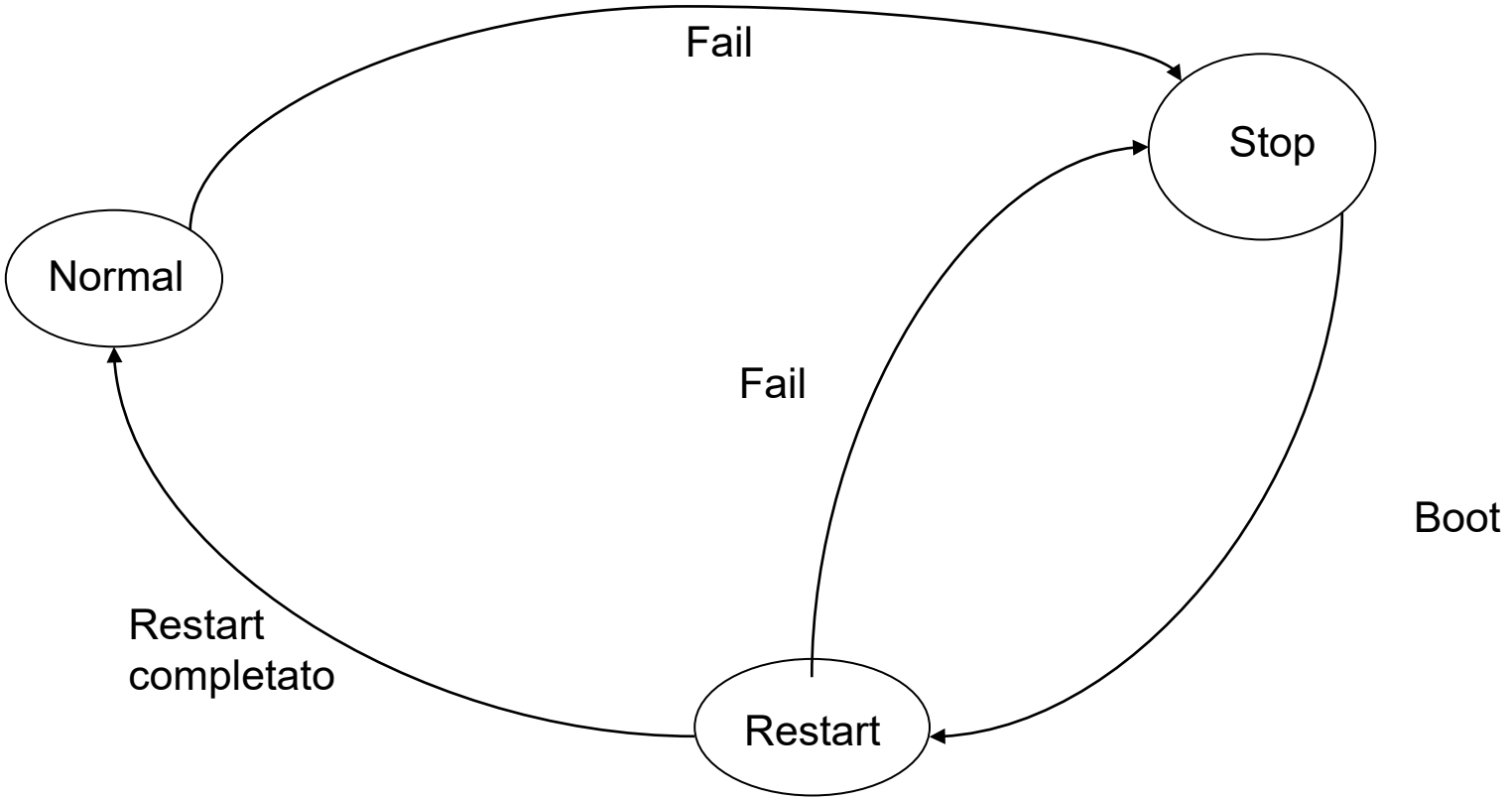
- **Guasti "soft":** errori di programma, crash di sistema, caduta di tensione
  - si perde la memoria centrale
  - non si perde la memoria secondaria

  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)

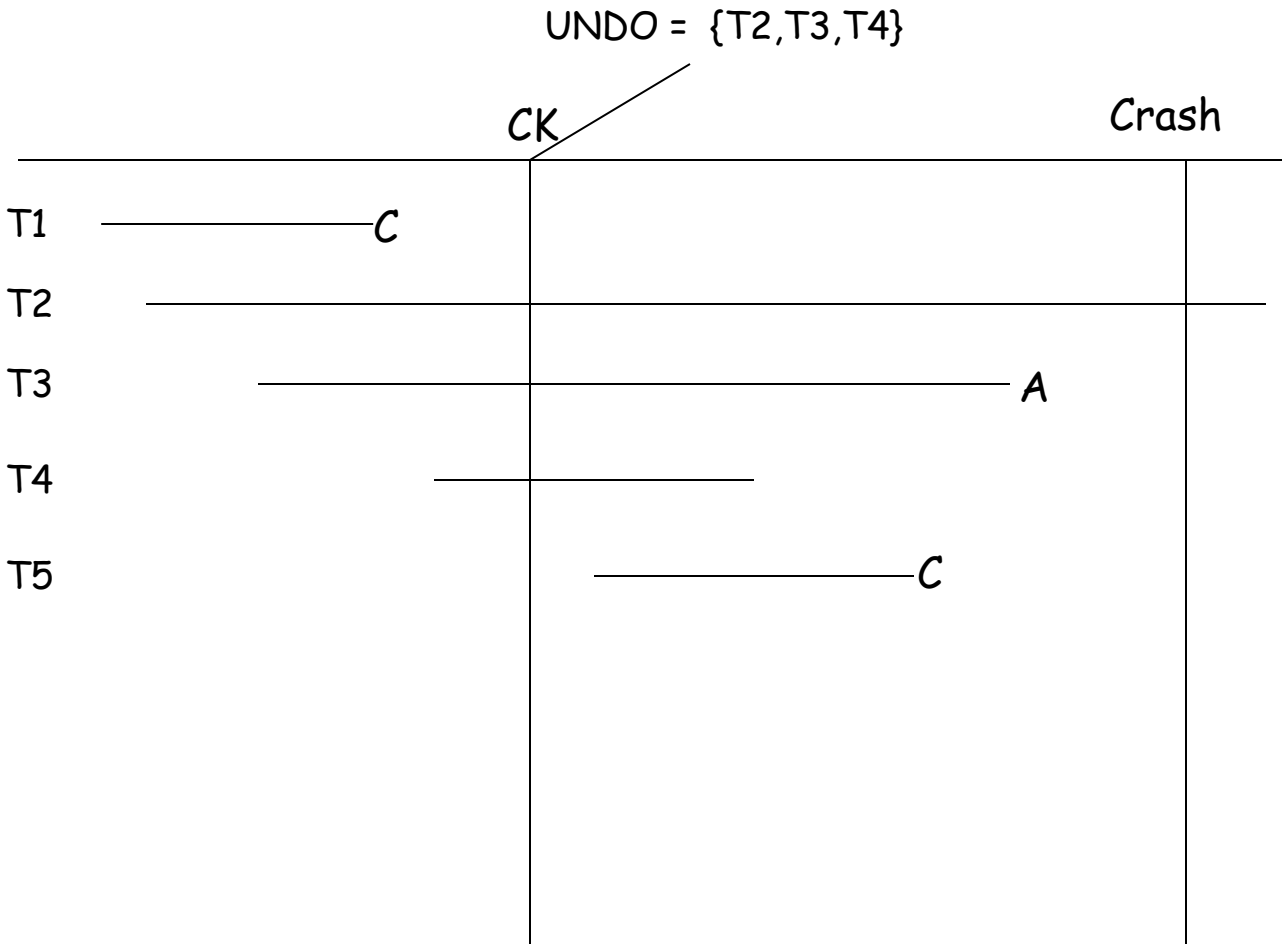
Ripresa a caldo

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

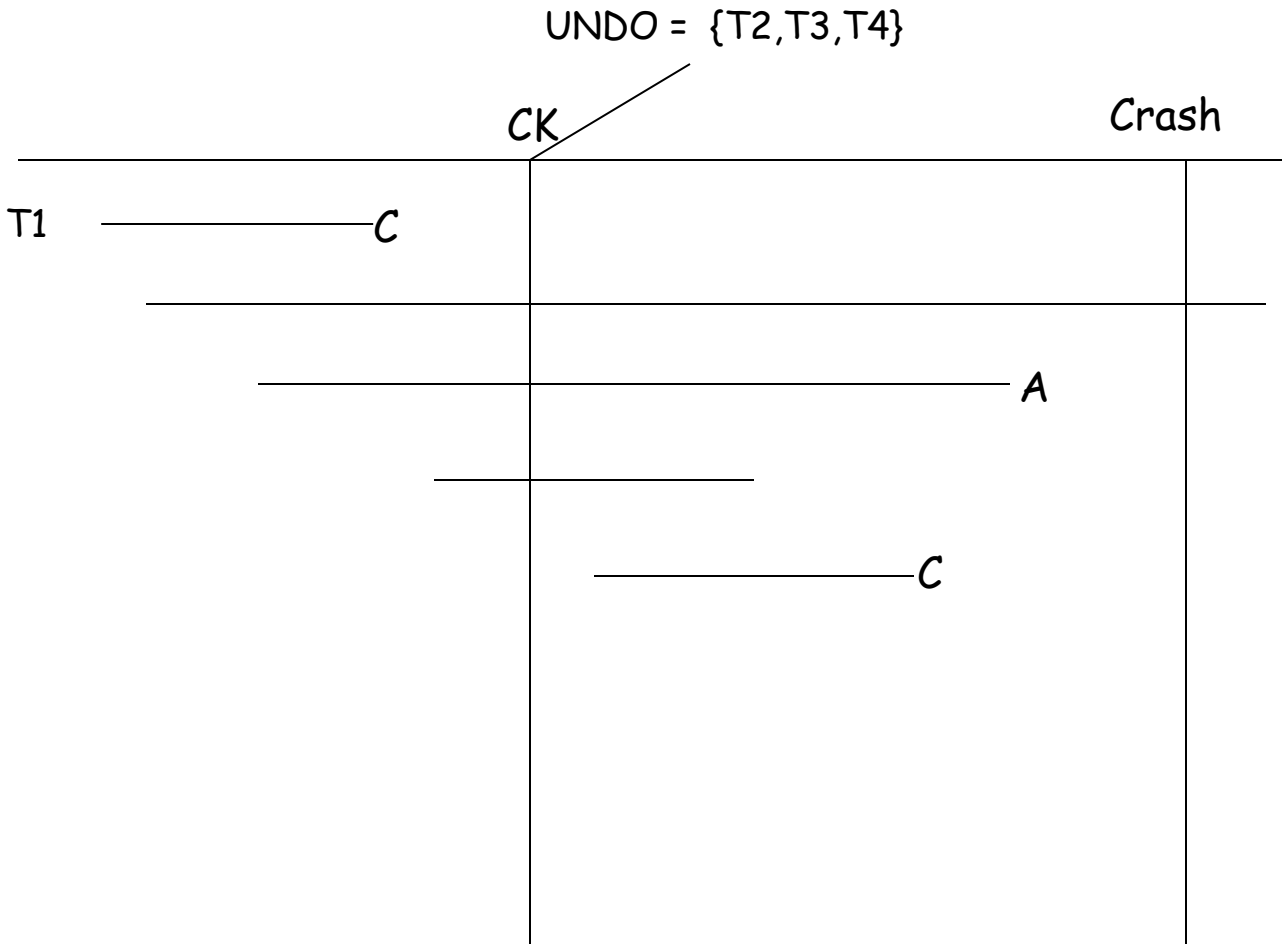
B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(t2,O6,A8)



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)

T2
T3
T4
T5



2. Costruzione degli insiemi UNDO e 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}

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

Setup

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)

3. Fase UNDO

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

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)

4. Fase REDO

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

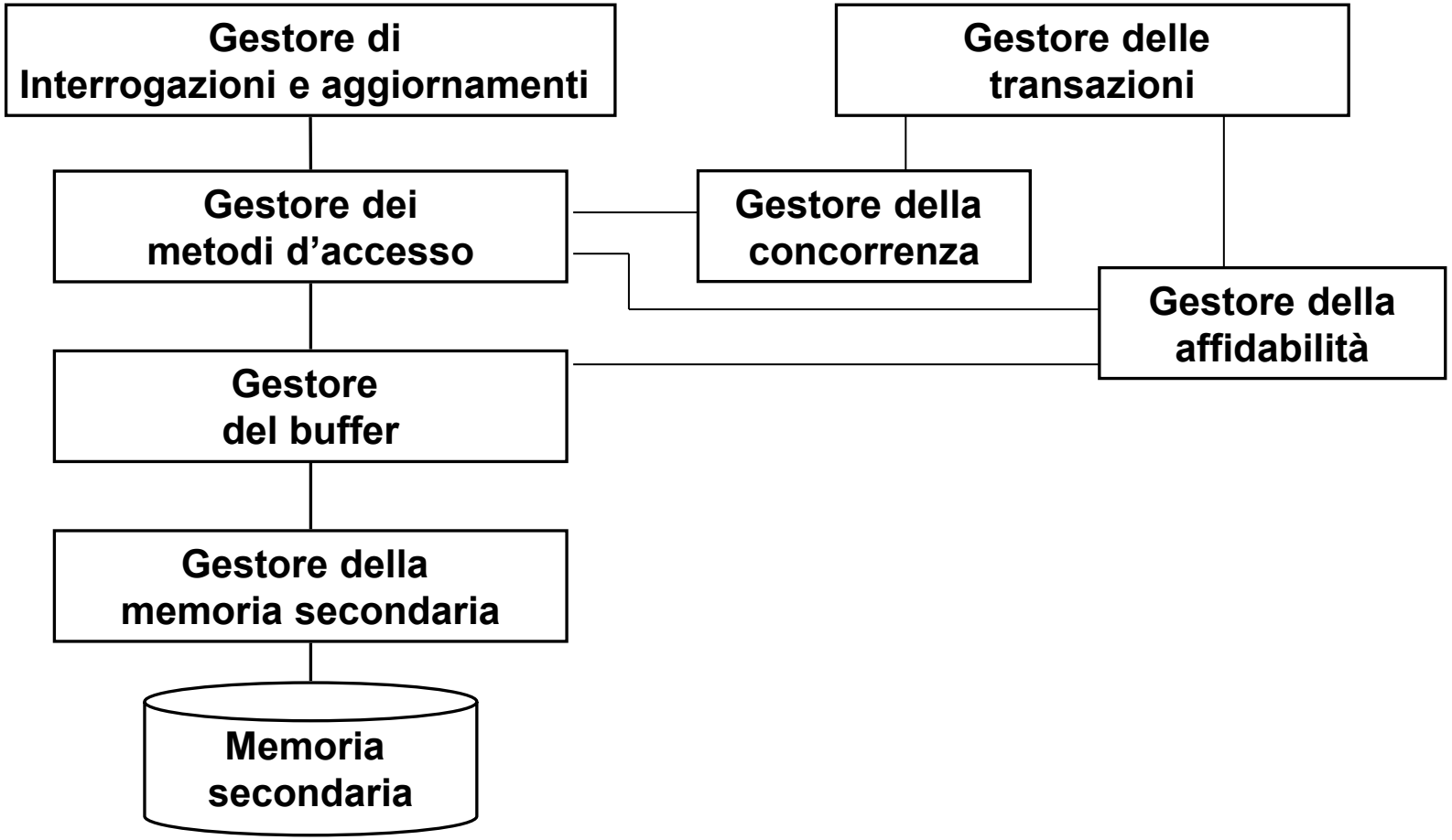
10. O4 = A6 Redo

Ripresa a freddo

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

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 (tps Transaction per Second)
- Esempi: banche, prenotazioni aeree

Modello di riferimento

- Operazioni di input-output su oggetti astratti *x*, *y*, *z*

Problema

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

Perdita di aggiornamento

- Due transazioni identiche:
  - t 1 : r(x), $x = x + 1$, w(x)
  - t 2 : r(x), $x = x + 1$, w(x)
- Inizialmente x=2; dopo un'esecuzione seriale x=4
- Un'esecuzione concorrente:

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

- Un aggiornamento viene perso: x=3

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

Letture 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

- Assumere ci sia un vincolo $y + z = 1000$;

t_1	t_2
bot	
$r_1(y)$	
	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

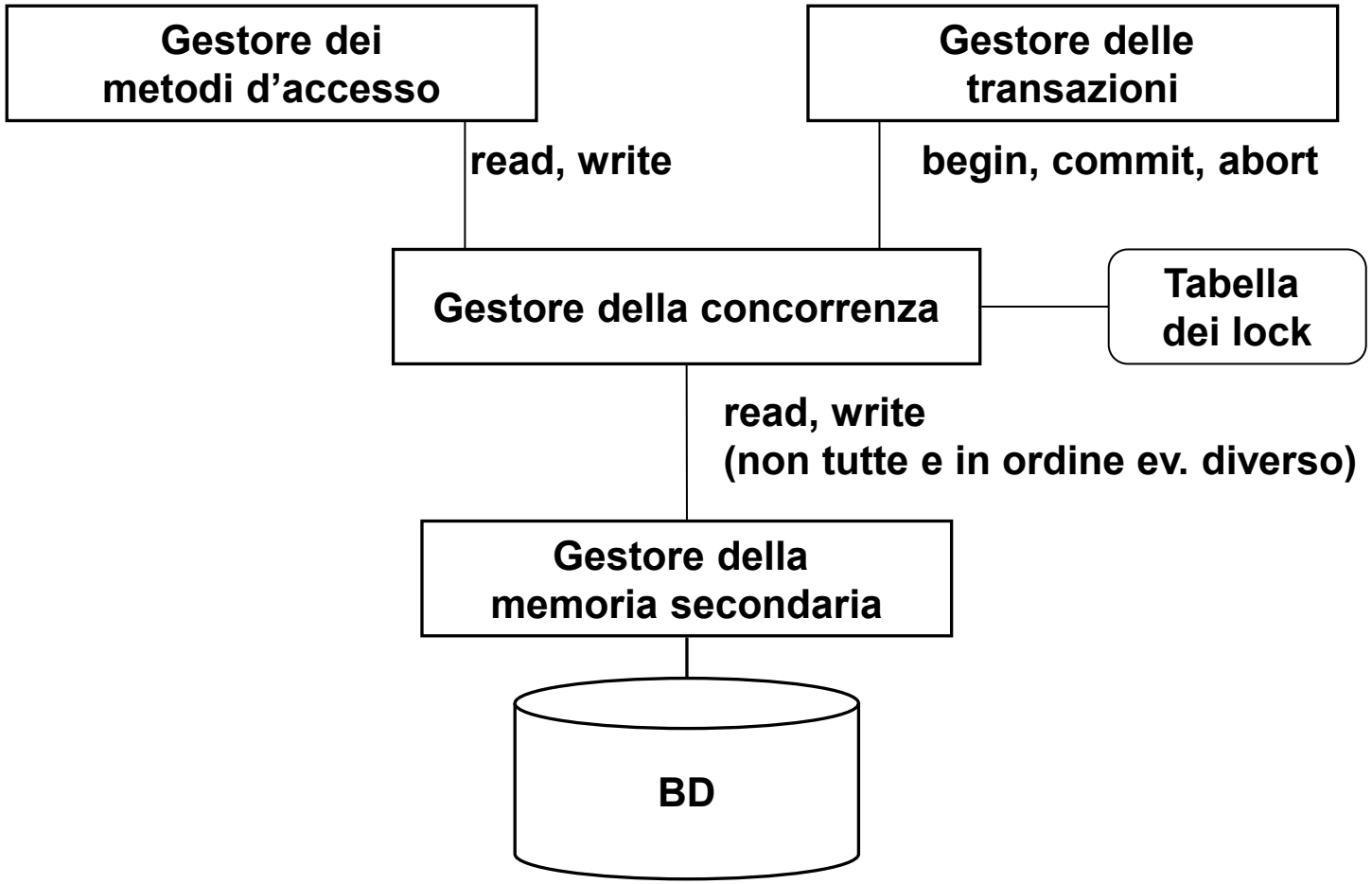
Inserimento fantasma

t_1	t_2
`bot`	
"legge gli stipendi degli impiegati del dip A e calcola la media"	
	`bot`
	"inserisce un impiegato in A"
	`commit`
"legge gli stipendi degli impiegati del dip A e calcola la media"	
`commit`	

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"

Gestore della concorrenza (ignorando buffer e affidabilità)



Schedule

- Nel controllo della concorrenza una **transazione è una sequenza di operazioni di input/output** …. Si omettoo le operazioni di manipolazione dei dati
- Esempio:

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

- Ipotesi semplificativa (che rinuoveremo in futuro, in quanto non accettabile in pratica):
  - consideriamo la **commit-proiezione** e ignoriamo le transazioni che vanno in abort, rimuovendo tutte le loro azioni dallo schedule
  - **Ipotesi teorica… Uno schedule deve decidere se accettare o rifiutare le azioni di una transazione senza conoscere il suo esito finale**.

Controllo di concorrenza

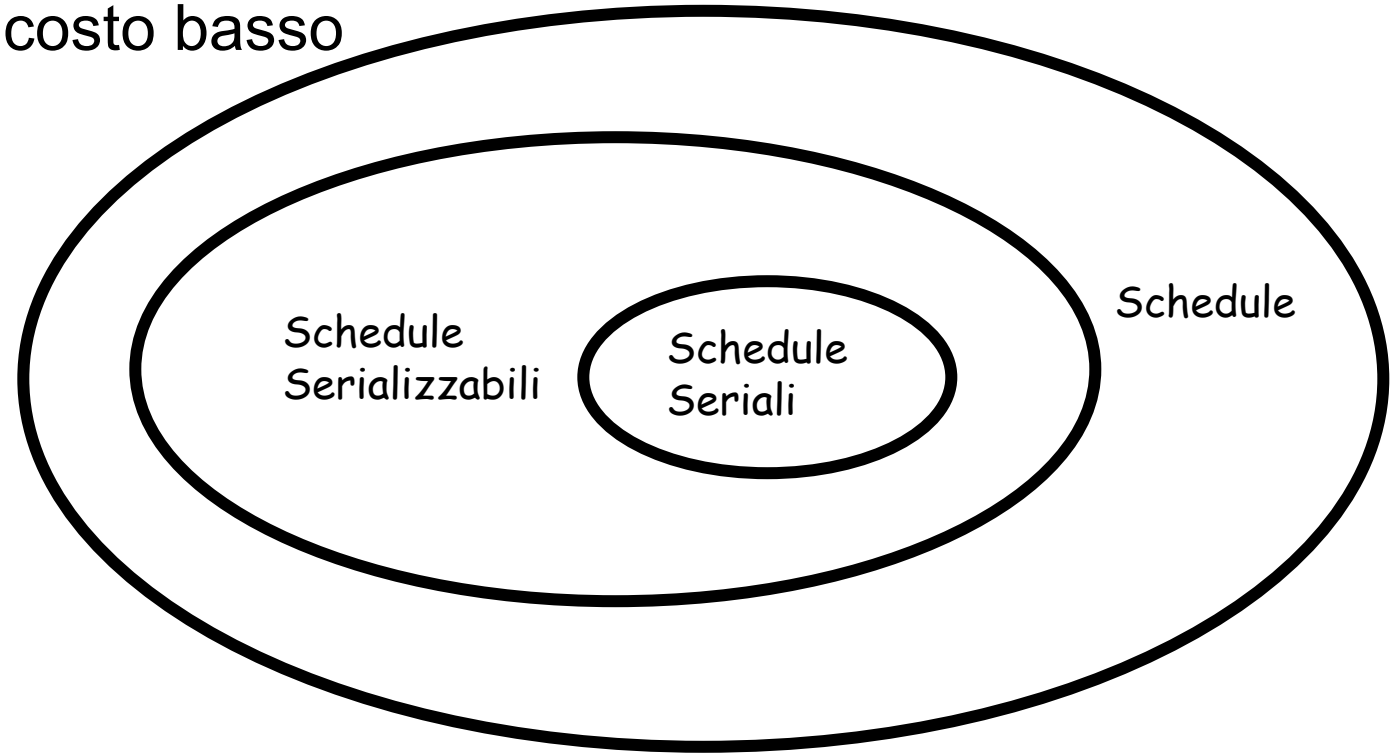
- *Obiettivo*: evitare le anomalie
- *Scheduler*: un sistema che accetta o rifiuta (o riordina) le operazioni richieste dalle transazioni (vengono rifiutate quelle che generano anomalie)
- *Schedule seriale:* le transazioni sono separate, una alla volta

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

- *Schedule serializzabile:* produce lo stesso risultato di uno schedule seriale sulle stesse transazioni
  - Richiede una nozione di equivalenza fra schedule
  - HP: si assume che ogni transazione sia corretta.

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 $r_i(x)$ 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
- 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

- $S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  $S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
  $S_5 : w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
  $S_6 : w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$
  - S_3 è view-equivalente allo schedule seriale S_4 (e quindi è view-serializzabile)
  - S_5 non è view-equivalente a S_4, ma è view-equivalente allo schedule seriale S_6, e quindi è view-serializzabile
- $S_7 : r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$ (**perdita di aggiornamento**)
  $S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$ (**letture inconsistenti**)
  $S_9 : r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$ (**aggiornamento fantasma**)
  - **S_7, S_8, S_9 non view-serializzabili**

View serializzabilità

- Complessità:
  - la verifica della view-equivalenza di due dati schedule:
    - ❑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 view-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:

- r1(x) w2(x) w1(x) w3(x)

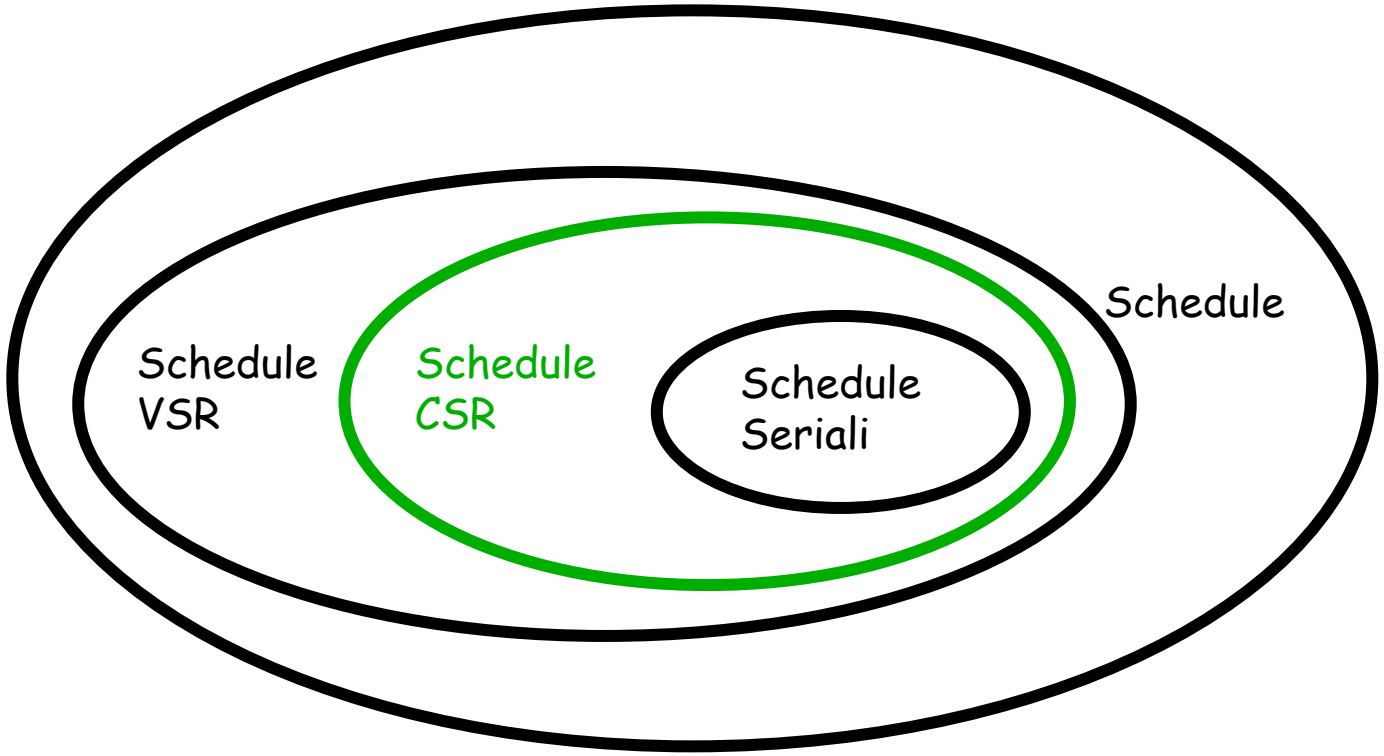
  - **view-serializzabile**: view-equivalente a r1(x) w1(x) w2(x) w3(x)
  - **non conflict-serializzabile**

- 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 $\approx_V$
- Quindi, supponiamo $S_1 \approx_C S_2$ e dimostriamo che $S_1 \approx_V S_2$
  I due schedule hanno:
  - stesse scritture finali: se così non fosse, ci sarebbero almeno due scritture in ordine diverso e poiché due scritture sono in conflitto i due schedule non sarebbero $\approx_C$
  - stessa relazione “legge-da”: se così non fosse, ci sarebbero scritture in ordine diverso o coppie lettura-scrittura 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.

Controllo della concorrenza in pratica

- Anche la conflict-serializabilità, pur più rapidamente verificabile (l'algoritmo, con opportune strutture dati richiede tempo lineare), è inutilizzabile in pratica
- La tecnica sarebbe 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
- Inoltre, la tecnica si basa sull'ipotesi di commit-proiezione
- In pratica, si utilizzano tecniche che
  - garantiscono la conflict-serializzabilità senza dover costruire il grafo
  - non richiedono l'ipotesi della commit-proiezione

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 escalation*)
- 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	OK / r_lockeded	NO / w_locked
unlock	error	OK / depends	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
- Il lock manager gestisce una tabella dei lock, per ricordare la situazione

Locking a due fasi

- Usato da quasi tutti i sistemi
- Garantisce "a priori" la conflict-serializzabilità a-priori
- Basata su due regole:
  - "proteggere" tutte le letture e scritture con lock
  - un vincolo sulle richieste e i rilasci dei lock:
    - **una transazione, dopo aver rilasciato un loc:k, non può acquisirne altri**
- **Two Phase Locking (2PL):** scheduling in cui una transazione, dopo aver rilasciato un lock non può acquisirne altri sulla stessa risorsa

2PL e CSR

- Ogni schedule 2PL è 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

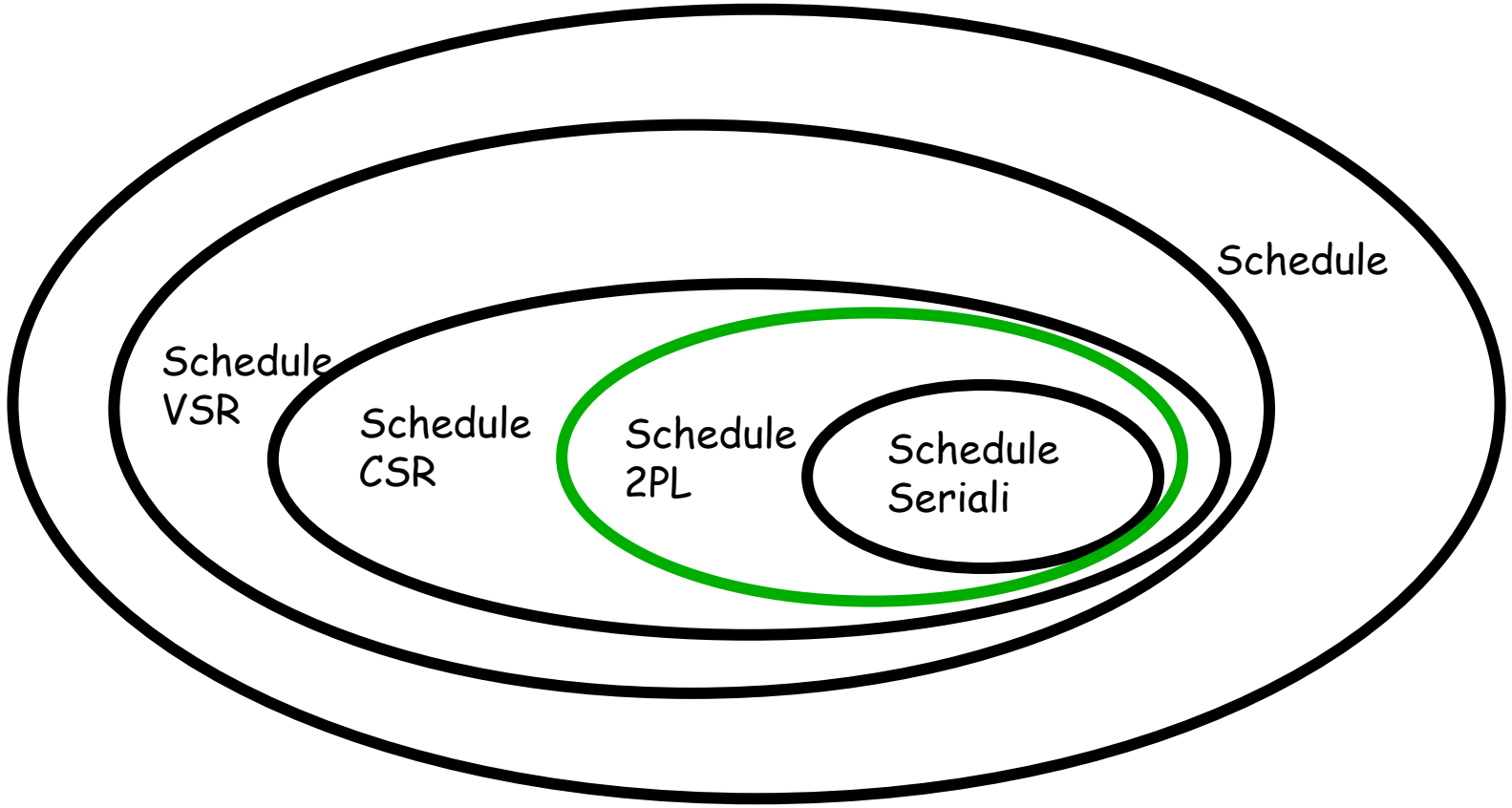
- S schedule 2PL
- Consideriamo per ciascuna transazione l'istante in cui ha tutte le risorse e sta per rilasciare la prima
- Ordiniamo le transazioni in accordo con questo valore temporale e consideriamo lo schedule seriale corrispondente
- Vogliamo dimostrare che tale schedule è equivalente ad S:
  - allo scopo, consideriamo un conflitto fra un'azione di t_i e un'azione dei 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

Locking a due fasi stretto

- Condizione aggiuntiva:
  - **I lock possono essere rilasciati solo dopo il commit o abort**

- Supera la necessità dell'ipotesi di commit-proiezione (ed elimina il rischio di letture sporche)

CSR, VSR e 2PL



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):
  - *read*(x,*ts*):
    - se *ts* < 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 *ts*
  - *write*(x,*ts*):
    - se *ts* < WTM(x) o *ts* < RTM(x) allora la richiesta è respinta e la transazione viene uccisa,
    - altrimenti, la richiesta e WTM(x) è posto uguale a *ts*
- 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
read(x,6)	ok	
read(x,8)	ok	RTM(x) = 8
read(x,9)	ok	RTM(x) = 9
write(x,8)	no, t_8 uccisa	
write(x,11)	ok	WTM(x) = 11
read(x,10)	no, t_{10} uccisa	

2PL vs TS

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

    $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_0(y)\ w_1(y)$

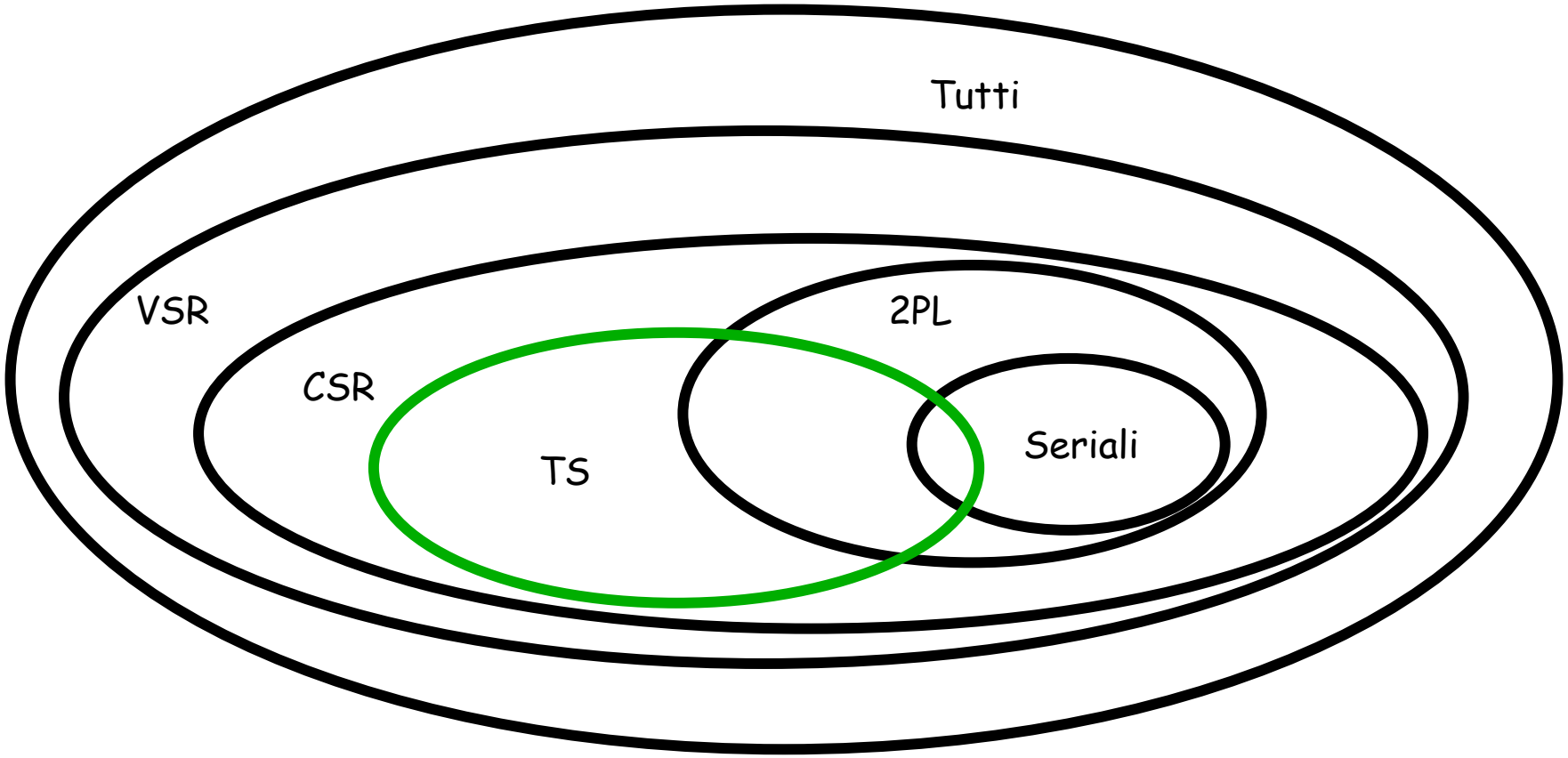
  - Schedule in 2PL ma non in TS

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

  - 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:
  - conviene il 2PL

Controllo di concorrenza multi-versione

- Idea del metodo: ogni write genera una nuova copia, I read leggono la copia “giusta”
- I write generano una nuova copia con un nuovo WTM. Istante per istante ogni oggetto x ha N >1 copie attive, con $WTM_N(x)$. C’è poi un solo RTM(x) globale
- Le vecchie copie vengono scartate quando non sono più presenti transazioni in lettura che dovrebbero leggerli.

Meccanismo

- read(x,ts): viene sempre accettata, si seleziona la copia x_k in lettura tale che: se ts > $WTM_N(x)$, allora k = N, altrimenti si prenda k tale che $WTM_k(x)$ <= ts < $WTM_{k+1}(x)$

- write(x,ts): se ts < RTM(x) la richiesta è respinta, altrimenti si crea una nuova versione (N incrementato) con $WTM_N(x)$ = ts

Esempio

Si assuma RTM(x) = 7

N=1 WTM(x_1) = 4

Richiesta	Risposta	Nuovo Valore
read(x,6)	ok	
read(x,8)	ok	RTM(x) = 8
read(x,9)	ok	RTM(x) = 9
write(x,8)	no	t8 uccisa
write(x,11)	ok	N=2, WTM(x_2) = 11
read(x,10)	ok su 1	RTM(x) = 10
read(x,12)	ok su 2	RTM(x) = 12
write(x,13)	ok	N=3, WTM(x_3) = 13

Stallo (deadlock)

- Attese incrociate: due transazioni detengono ciascuna una risorsa e aspetttano la risorsa detenuta dall'altra

- Esempio:
  - t_1: *read*(x), *write*(y)
  - t_2: *read*(y), *write*(x)
  - Schedule:

    $r\_lock_1(x)$, $r\_lock_2(y)$, $read_1(x)$, $read_2(y)$ $w\_lock_1(y)$, $w\_lock_2(x)$

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 tranzazioni 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

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

- `snapshot isolation`:
  - Un nuovo livello offerto dai sistemi, basato sulla gestione di più versioni dei dati

Lock di predicato

- Caso peggiore:
  - sull'intera relazione

- Se siamo fortunati:
  - sull'indice

Update lock

- Il deadlock più frequente avviene quando 2 transazioni concorrenti vogliono prima leggere e poi scrivere la stessa risorsa
- Per evitare questa situazione, i sistemi offrono gli update lock (UL)
- L'update lock viene acquisito da transazioni che vogliono inizialmente leggere un oggetto per poi modificarne il valore

	Stato		
Richiesta	SL	UL	XL
SL	OK	OK	No
UL	OK	No	No
XL	No	No	No