Basi di dati
II

2- Gestione delle transazioni

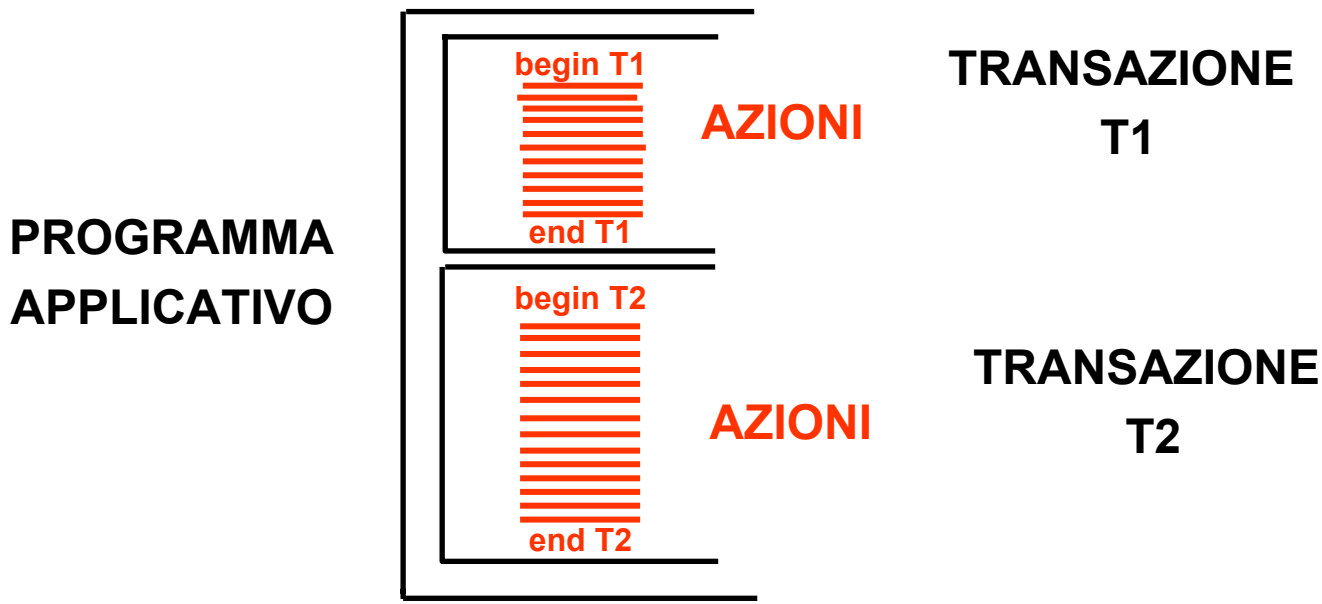
LE TRANSAZIONI

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**) è in grado di definire ed eseguire transazioni **affidabili** 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;
```

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 cioè non può lasciare al suo esito la base di dati in uno stato intermedio.
  - prima del commit un guasto o un errore ***deve*** causare l'annullamento (UNDO) delle operazioni svolte
  - dopo il commit un guasto o errore ***non deve*** avere conseguenze; se necessario vanno ripetute (REDO) le operazioni
- Esito
  - Commit = caso "normale" e più frequente (99% ?)
  - Abort (o rollback)
    - richiesto dalla transazione stessa (suicidio)
    - richiesto dal sistema (omicidio) per:
      - violazione vincoli, gestione concorrenza, incompletezza causa guasto sistema

Consistenza

- La transazione deve rispettare i vincoli di integrità: se il sistema rileva violazione interviene per annullare la transazione o per eliminare le cause della violazione.
- Vincoli di integrità ***immediati*****:** verificabili durante la transazione gestendo nel programma le condizioni anomale senza imporre l'abort.
- Vincoli di integrità ***differiti*** **:** verificabili alla conclusione della transazione dopo che l'utente ha *richiesto* il commit:
  - se non c'è violazione il commit va a buon fine
  - se c'è violazione il commit non va a buon fine e si verifica un UNDO in extremis.

Isolamento

- La transazione non deve risentire degli effetti delle altre transazioni concorrenti:
  - l'esecuzione concorrente di una insieme di transazioni deve produrre un risultato analogo a quello che si avrebbe se ciascuna transazione operasse da sola (isolamento).
  - L’esito di ciascuna transazione deve essere indipendente da tutte le altre: l’abort di una transazione non può causare l’abort di altre transazioni (effetto domino).

Durabilità (Persistenza)

- Una transazione andata in commit ***si impegna*** a mantenere in modo permanente gli effetti della sua esecuzione *anche in presenza di guasti.*

  (**Commit = impegno**)

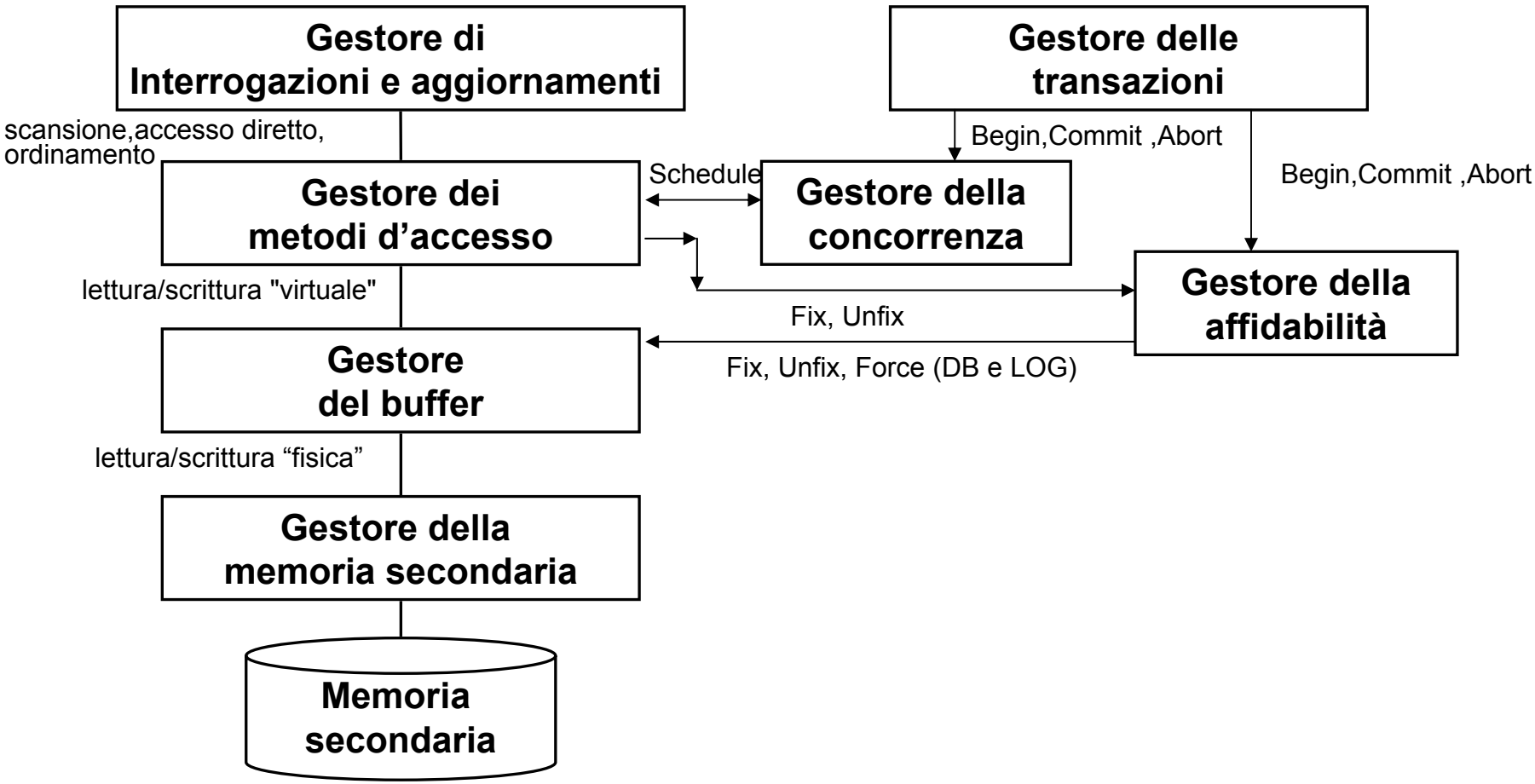
Transazioni e moduli di DBMS

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

CONTROLLO DI AFFIDABILITÀ

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni

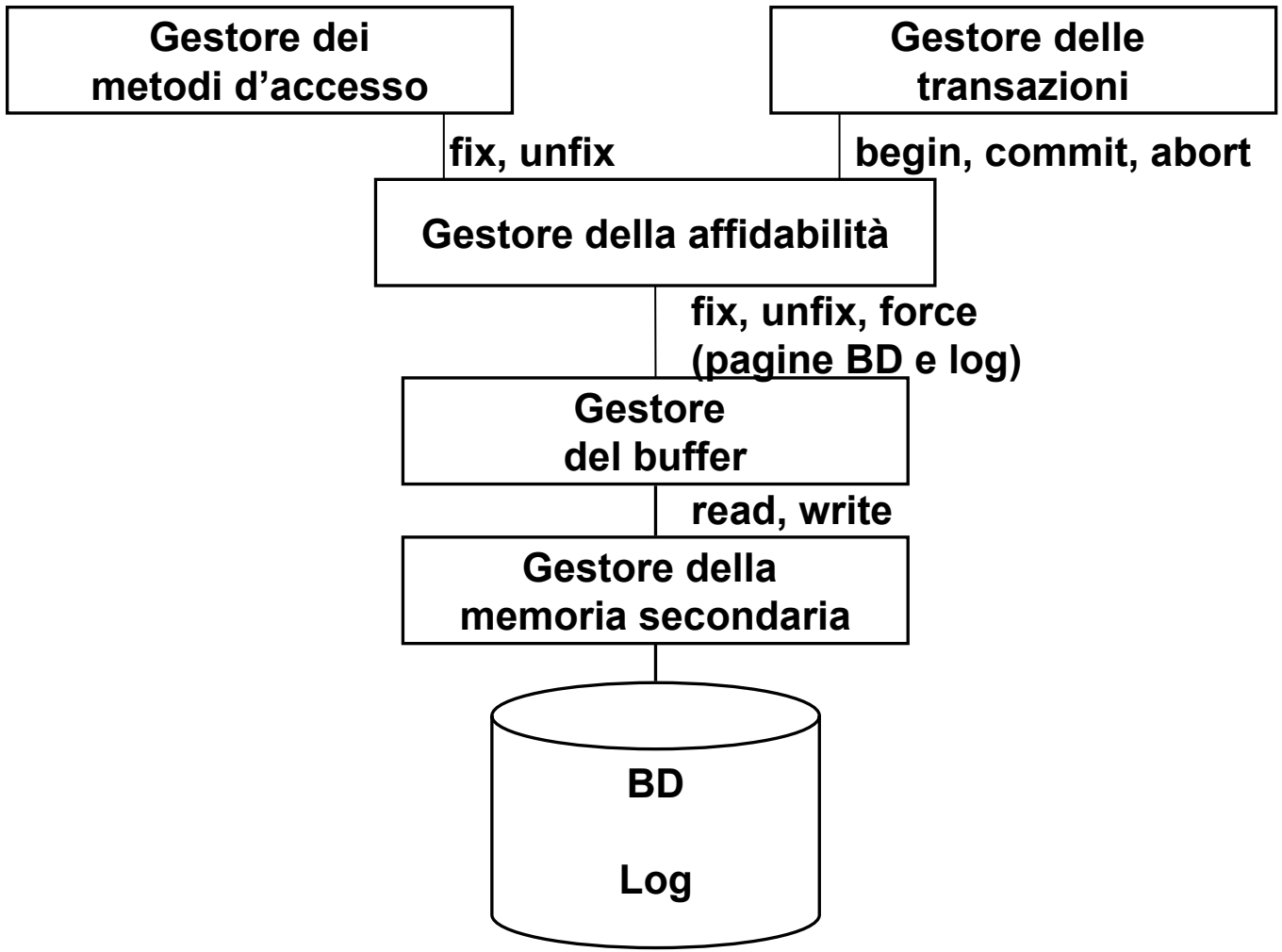


Gestore dell'affidabilità

- 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 (ripresa a caldo)*
  - *cold restart* (ripresa a freddo)
- Assicura atomicità e persistenza (durabilità)
- Usa il **log**:
  - Un archivio permanente che registra le operazioni svolte

Architettura del controllore dell'affidabilità



Persistenza delle memorie e log

- **La Memoria centrale**: non è persistente
- **La Memoria di massa**: è persistente ma può danneggiarsi
- **La Memoria stabile**: è memoria che non può danneggiarsi (è una astrazione):
  - perseguita attraverso la ridondanza:
    - dischi replicati
    - nastri
    - …
- **Il log** è un file sequenziale gestito dal controllore dell'affidabilità, scritto in **memoria stabile**

Contenuto del log

- "Diario di bordo": riporta tutte le operazioni in ordine di esecuzione
- 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`

I valori di BS e AS nelle operazioni

- Si precisa che i due parametri:

  BS (Before State) e **AS (After State)**

  descrivono rispettivamente i valori di un oggetto O **prima** e **dopo** la modifica nella base di dati

  – Nel caso di **INSERT** è definito solo il valore di AS poiché l'oggetto prima della modifica non esiste cioè:

  INSERT(T,O, —,AS)= INSERT(T,O,AS)

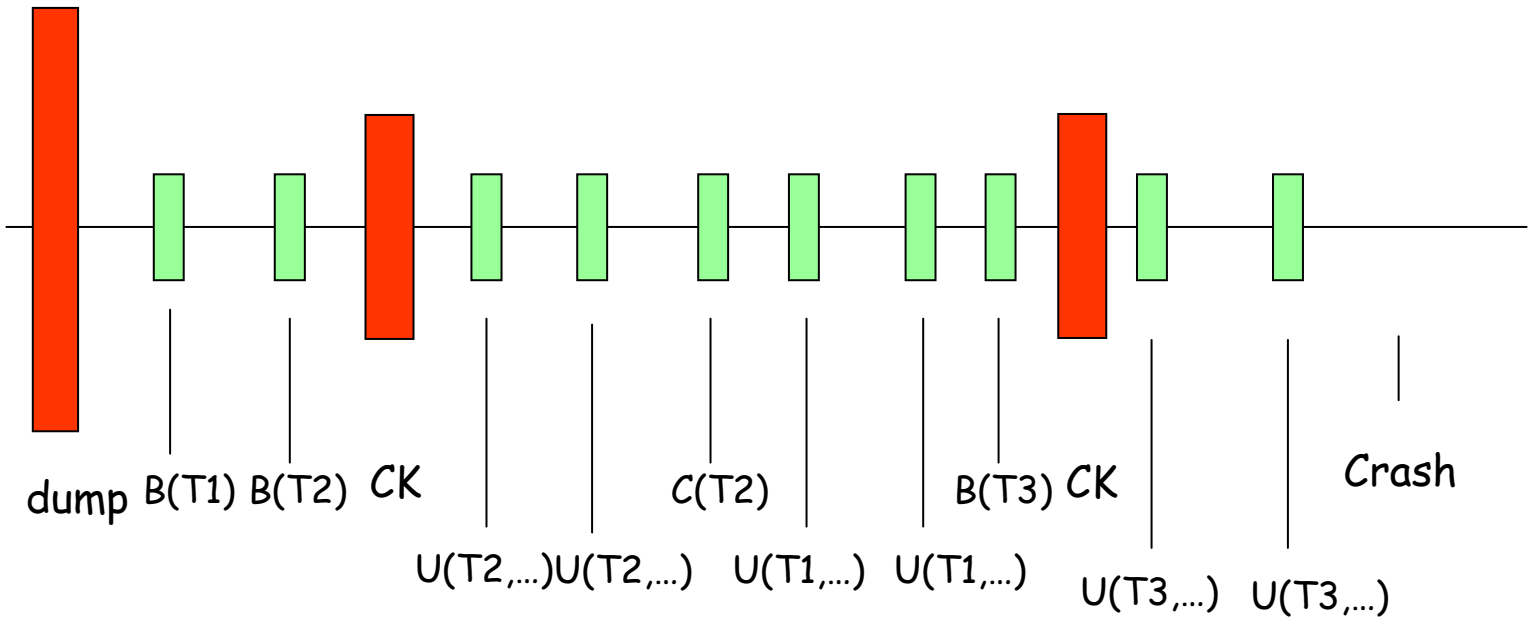
  – Nel caso di **UPDATE** sono entrambi definiti

  UPDATE(T,O,BS,AS)

  – Nel caso di **DELETE** è definito solo il valore di BS poiché l'oggetto dopo la modifica non esiste.

  DELETE(T,O,BS,—) = DELETE(T,O,BS)

Struttura del log



Log, checkpoint e dump: a che cosa servono?

- Il log serve "a ricostituire" il contenuto della base dei dati a seguito di guasti.
- I record di sistema checkpoint e dump servono ad evitare che la ricostruzione debba partire dall'inizio dei tempi:
  - si usano con riferimento a tipi di guasti diversi.

Primitive undo e redo e proprietà

I record di insert, update e delete consentono di disfare (Undo) e rifare (Redo) le rispettive azioni sulla base di dati.

- **Undo di una azione su un oggetto *O*:**
  - `update`, `delete`: copiare dal log il valore del before state (*BS*) nell'oggetto *O*
  - `insert`: *cancellare O*
- **Redo di una azione su un oggetto *O*:**
  - `insert`, `update`: copiare dal log il valore dell' after state (*AS*) nell'oggetto *O*
  - `delete`: *cancellare 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 le transazioni attive in un certo istante e di confermare che le altre transazioni 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:
  1. **si sospende l'accettazione di richieste di ogni tipo** (aggiornamenti, inserimenti, …, commit, abort);
  2. si trasferiscono in memoria di massa (tramite *force*) tutte le pagine sporche relative a transazioni andate in commit;
  3. si registra sul log in modo sincrono (con un ***force***) un record di checkpoint CK (T1,T2,…,Ti) contenente gli identificatori delle transazioni in corso;
  4. **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.

- 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**
  - un guasto prima di tale istante porta ad un **undo** di tutte le azioni sulla base di dati *se necessario*, per ricostruire lo stato originario della base di dati;
  - un guasto successivo al commit 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 poiché non modifica le decisioni del gestore dell'affidabilità.

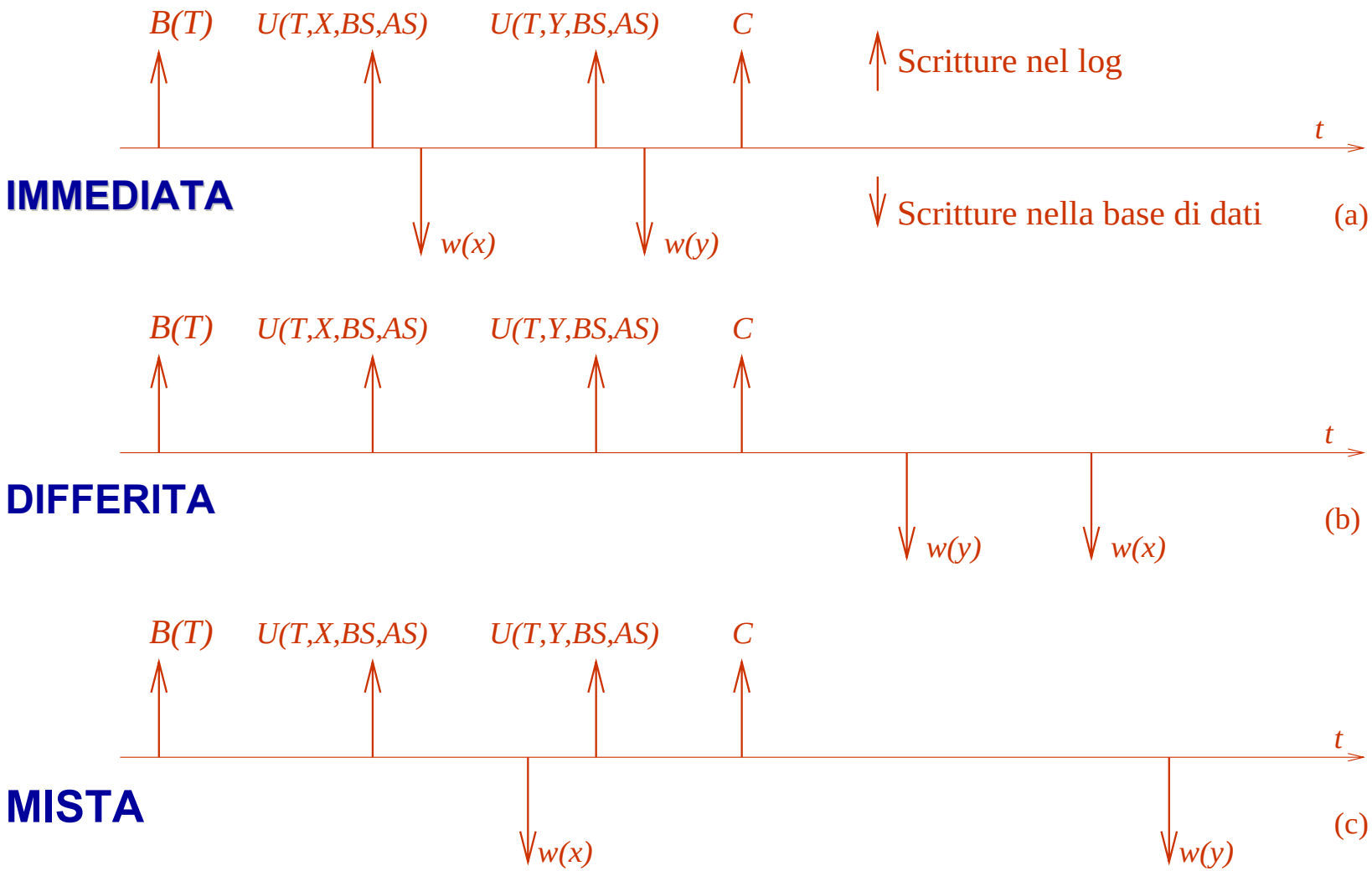
Regole fondamentali per il log

- **Write – ahead - Log**
  - **si scrive il giornale - parte before - prima di effettuare la azione sul database**
    - consente di disfare le azioni poiché per ogni aggiornamento viene reso disponibile nel log il valore prima della scrittura sul database.
- **Commit- Precedenza**:
  - **si scrive il giornale - parte after - prima del commit**
    - consente di rifare le azioni poiché se le pagine modificate non sono state ancora trascritte dal buffer manager viene reso disponibile nel log il valore in esse registrato.

Regole semplificate per il log

- REGOLA WAL SEMPLIFICATA
  - I record di log vanno scritti prima dei corrispondenti record sulla base di dati.
- REGOLA DI COMMIT-PRECEDENZA SEMPLIFICATA
  - I record di log vanno scritti prima della effettuazione della operazione di commit.
- QUANDO SI SCRIVE SULLA BASE DI DATI ?

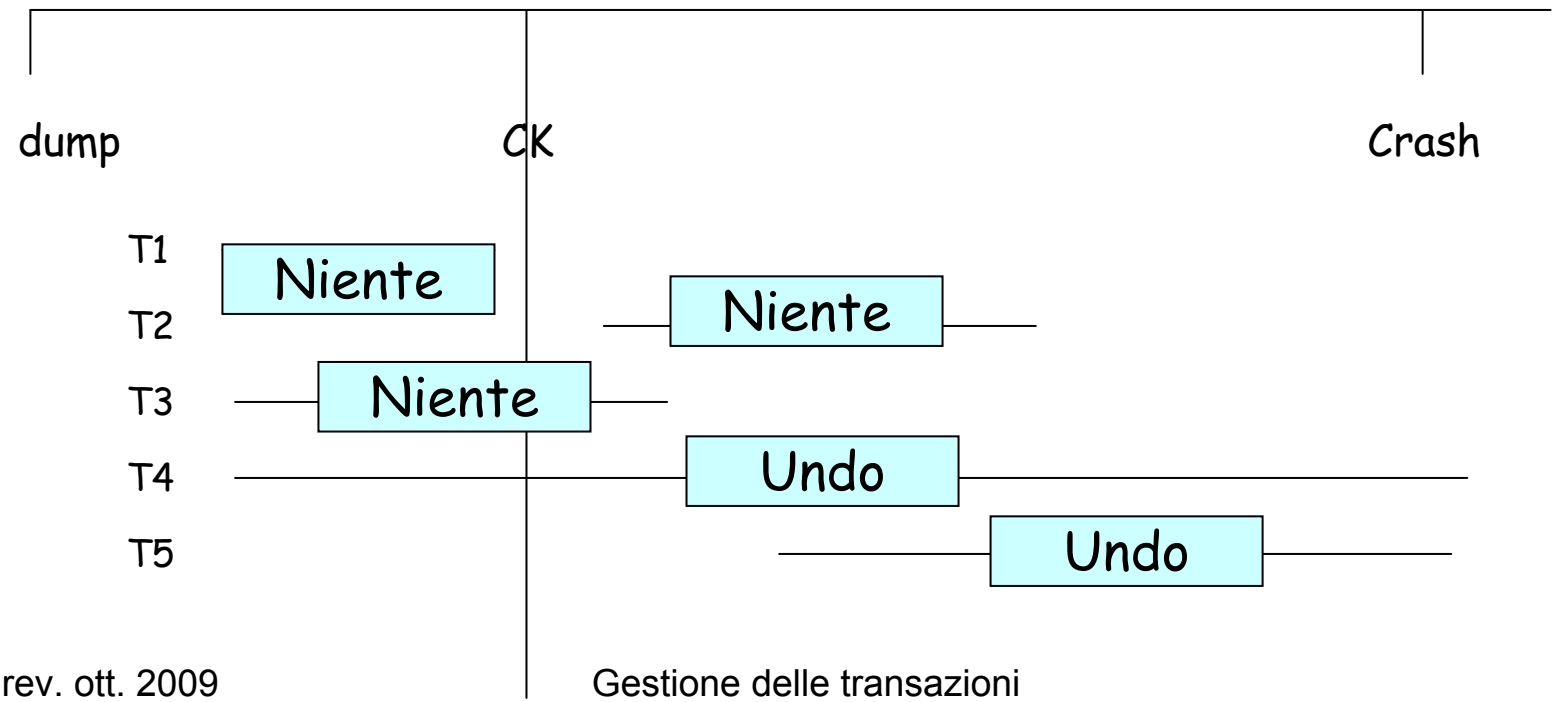
Scrittura nel log e nella base di dati



Modalità immediata

Il DB contiene valori **AS** provenienti da transazioni uncommitted

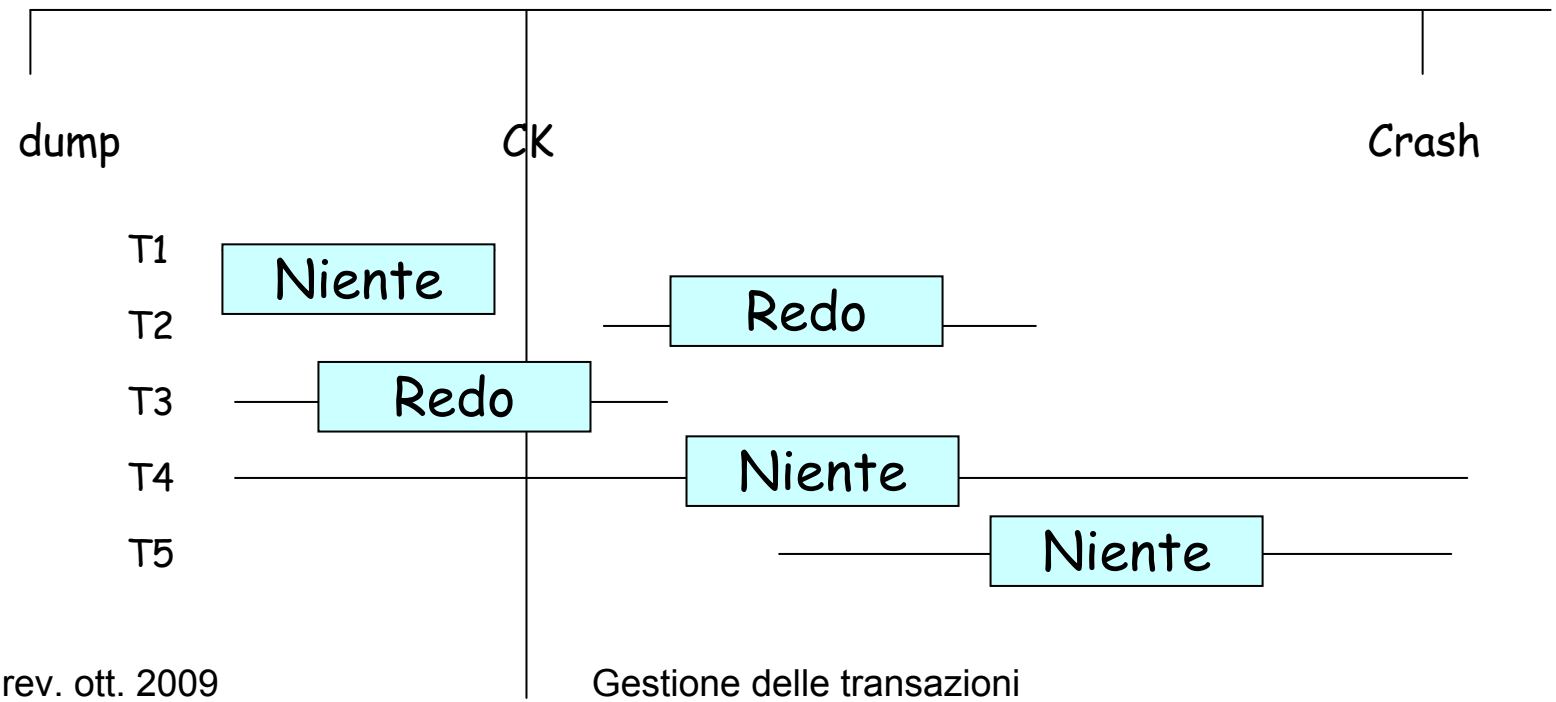
- Richiede Undo delle operazioni di transazioni uncommited al momento del guasto
- Non richiede Redo



Modalità differita

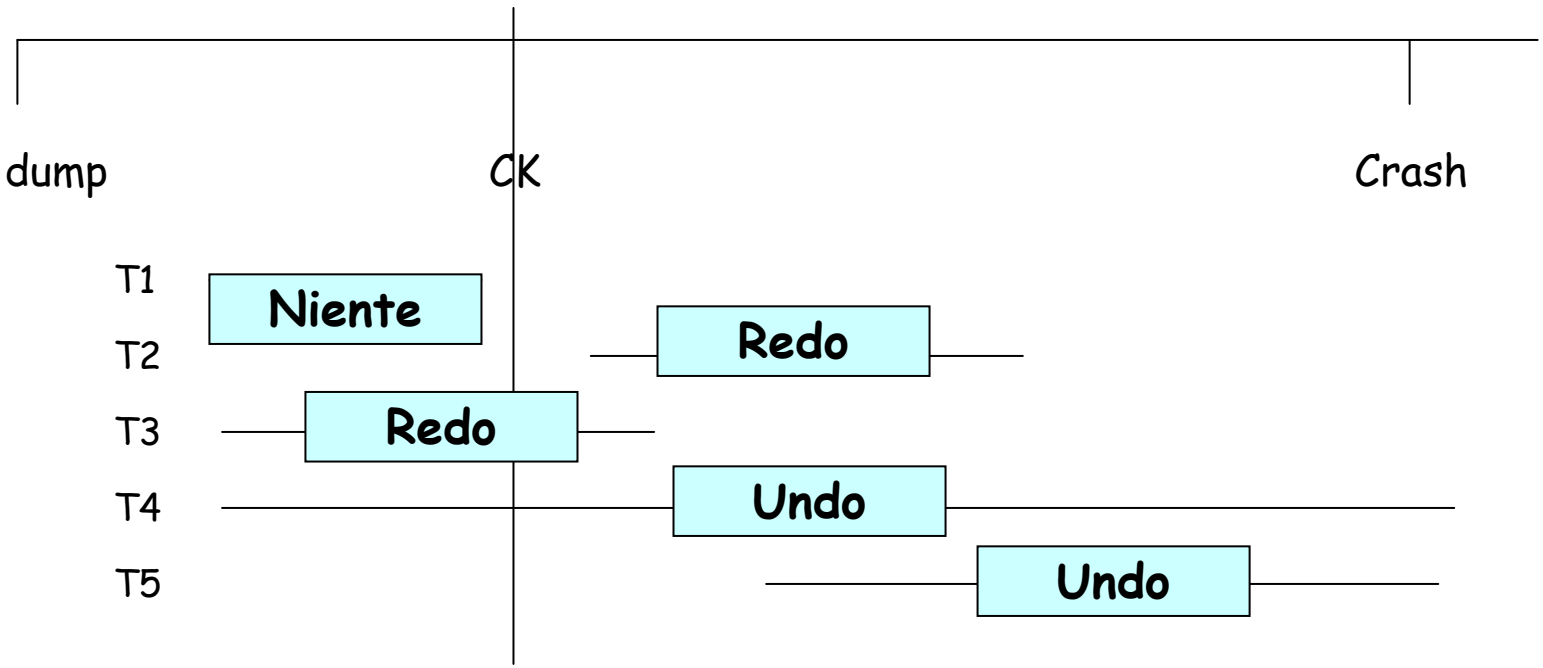
Il DB non contiene valori **AS** di transazioni uncommitted

- In caso di abort, non occorre fare niente
- Rende superflua la procedura di Undo.
- Richiede Redo delle operazioni di transazioni commited al momento del guasto



Esiste una terza modalità: modalità mista

- La scrittura può avvenire in modalità sia immediata che differita e quindi c'e incertezza su tutte le transazioni nella lista del CK.
- 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
  - non si perde la memoria stabile (e quindi il log)

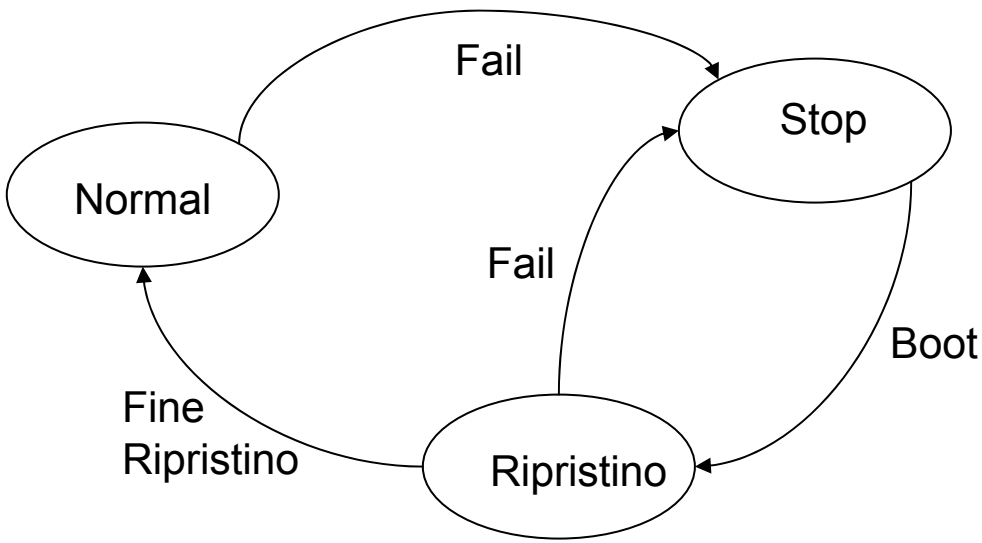
  warm restart, ripresa a caldo

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

  cold restart, ripresa a freddo

Modello "fail-stop"

- Forza arresto completo transazioni (fail→stop)
- Ripresa del corretto funzionamento del SO (Boot)
- Procedura di ripristino (warm o cold restart)



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)

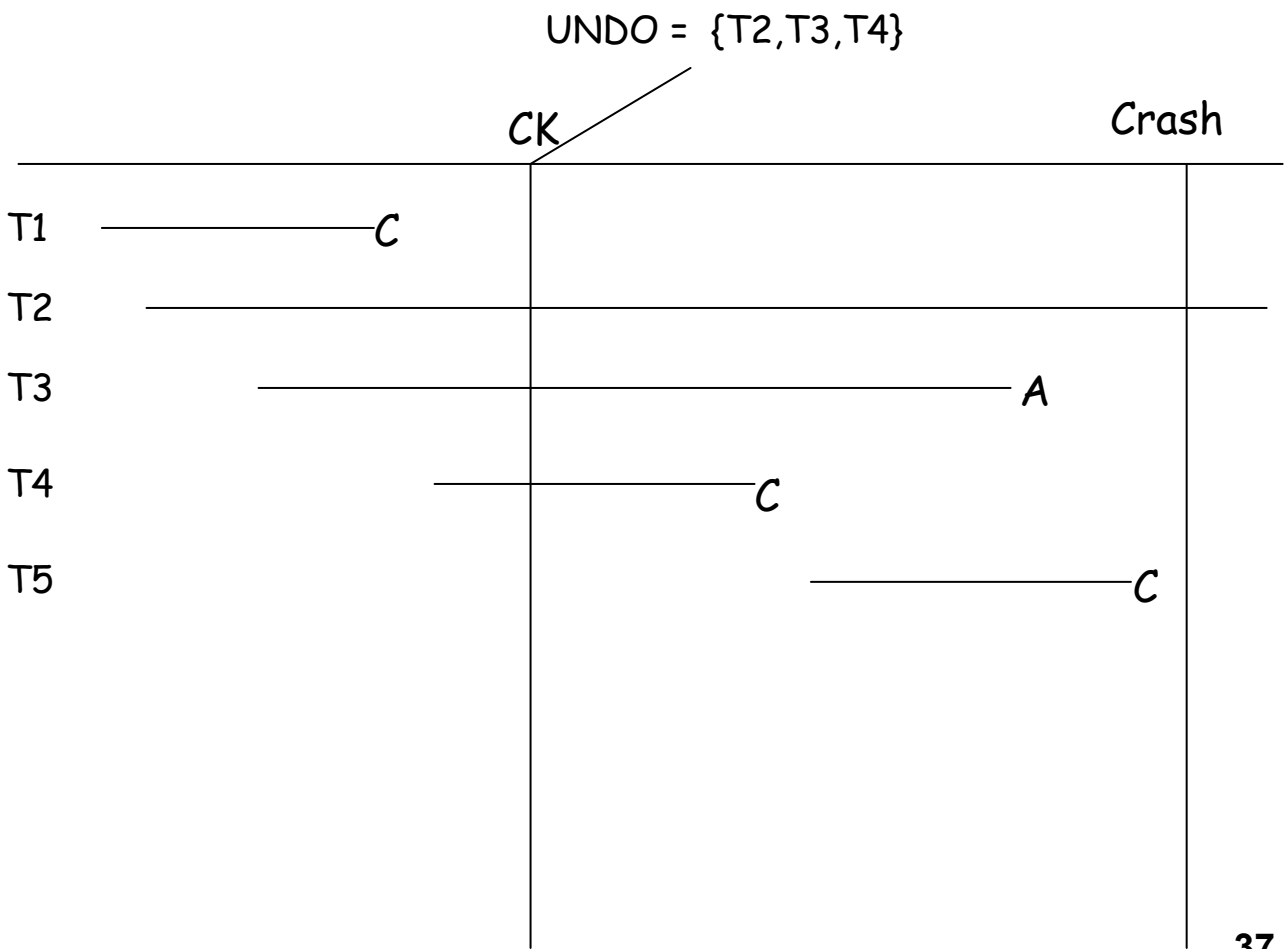
Ripresa a caldo

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*

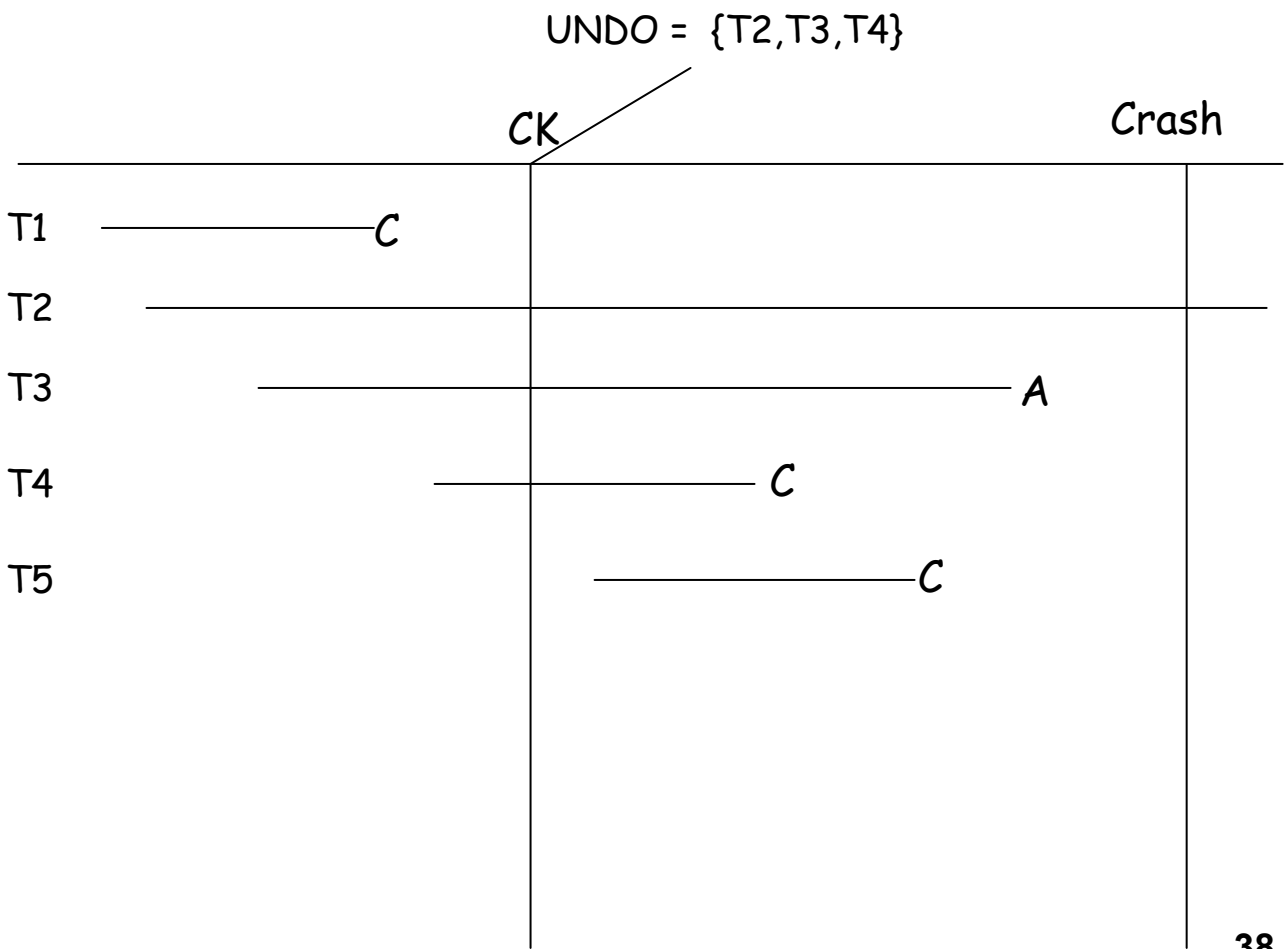
Esempio di warm restart

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)
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}

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

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

1. Si ripristinano i dati a partire dal backup e si accede al più recente record di dump del log;
2. si eseguono tutte le operazioni registrate sul log **relativamente alla parte deteriorata** riportandosi all'istante precedente il guasto;
3. si esegue una ripresa a caldo

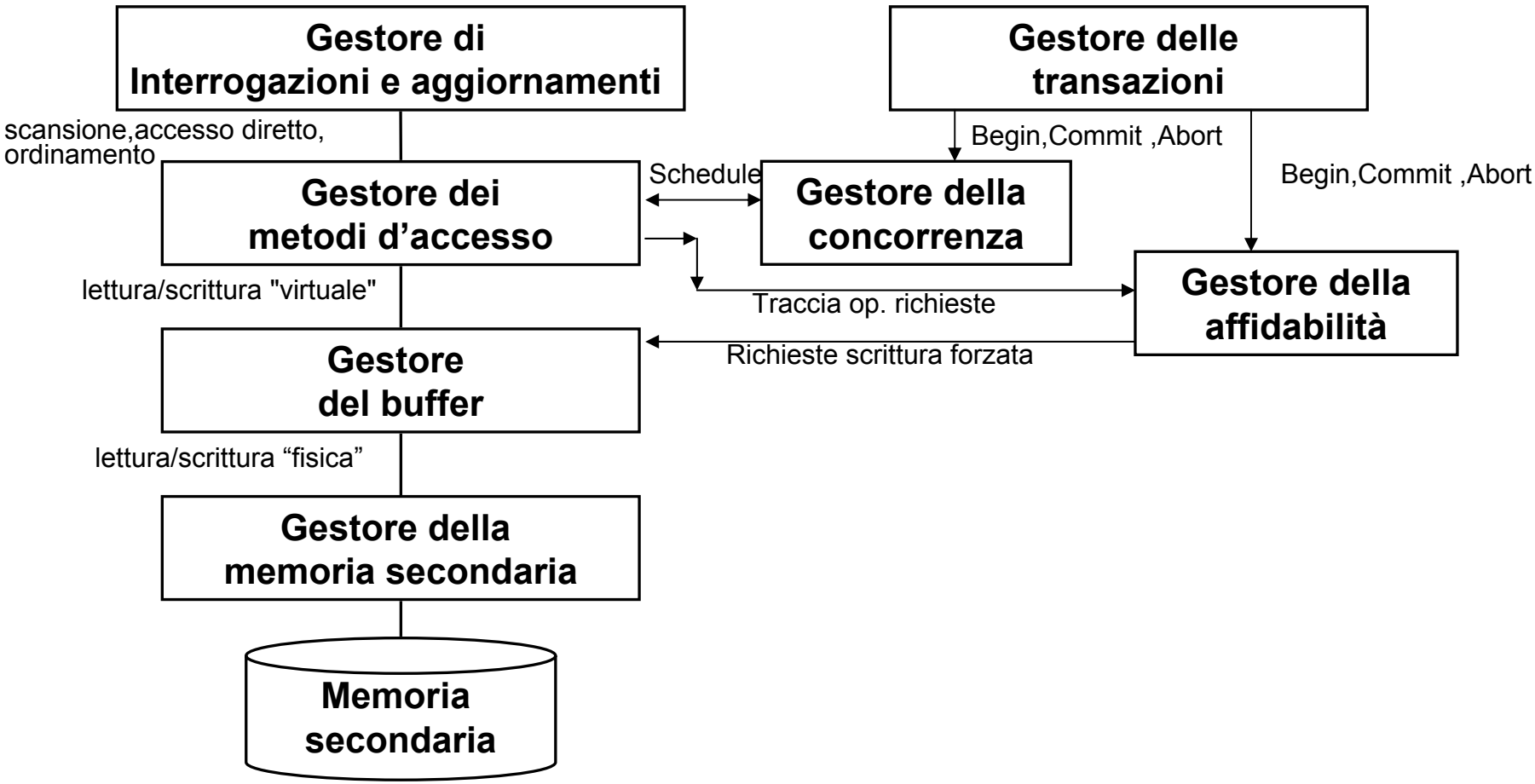
CONTROLLO DI CONCORRENZA

Controllo di concorrenza

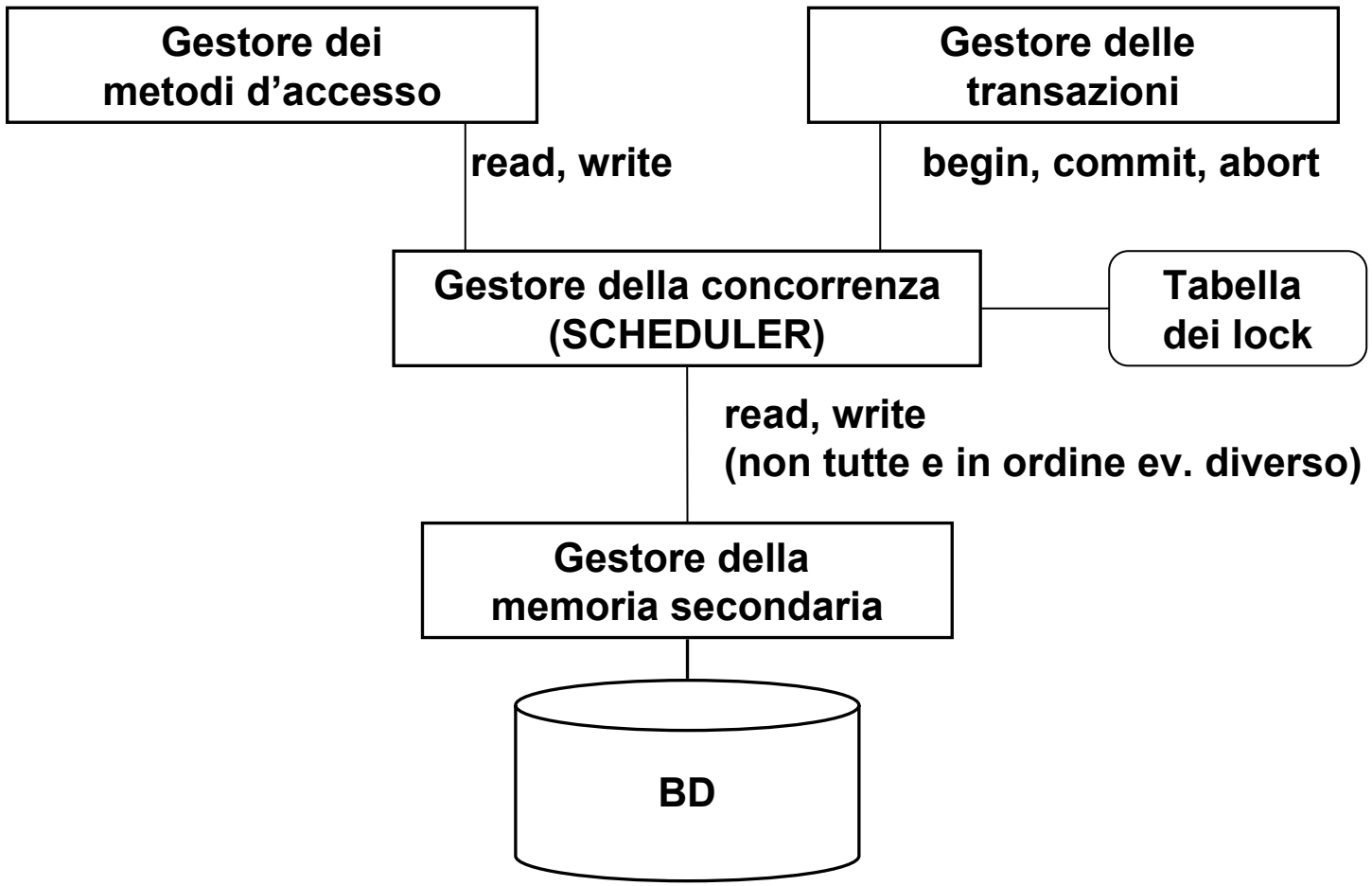
- La concorrenza è fondamentale: decine o centinaia di transazioni al secondo, non possono essere seriali
  - Esempi: banche, prenotazioni aeree
- La concorrenza consente di massimizzare il numero di transazioni servite per secondo minimizzando i tempi di risposta.

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



Gestore della concorrenza (ignorando buffer e affidabilità)



Le anomalie delle transazioni concorrenti

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

- **Modello di riferimento**
  - Operazioni di read e write su oggetti astratti *x*, *y*, *z:*
    - r(x), w(x) ; r(y), w(y); r(z),w(z)
  - Esemplificazioni di operazioni in memoria centrale su tali oggetti come se essi fossero numerici (ignorando che le operazioni richiedono l'intera pagina dove risiedono i dati).

Perdita di aggiornamento (lost update)

- 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 (dirty read)

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 (inconsistent read)

- 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 (ghost update)

- 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 (phantom)

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

Il modello di transazione

- Transazione:
  - Sequenza di azioni di lettura e scrittura
  - Viene omesso ogni riferimento alle operazioni di manipolazione in memoria da parte della transazione
- Ogni transazione è pertanto un oggetto sintattico in cui si conoscono solo le azioni di ingresso e uscita.
  - esempio di transazione:

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

- Le transazioni avvengono concorrentemente e pertanto le operazioni di lettura e scrittura vengono richieste in istanti successivi da varie transazioni ...

Schedule

- Uno **schedule** è una sequenza di operazioni di input/output presentate da transazioni concorrenti.
- Esempio:

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

(n.b. il pedice indica il numero di transazione)

- Ipotesi semplificativa (che rimuoveremo 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.

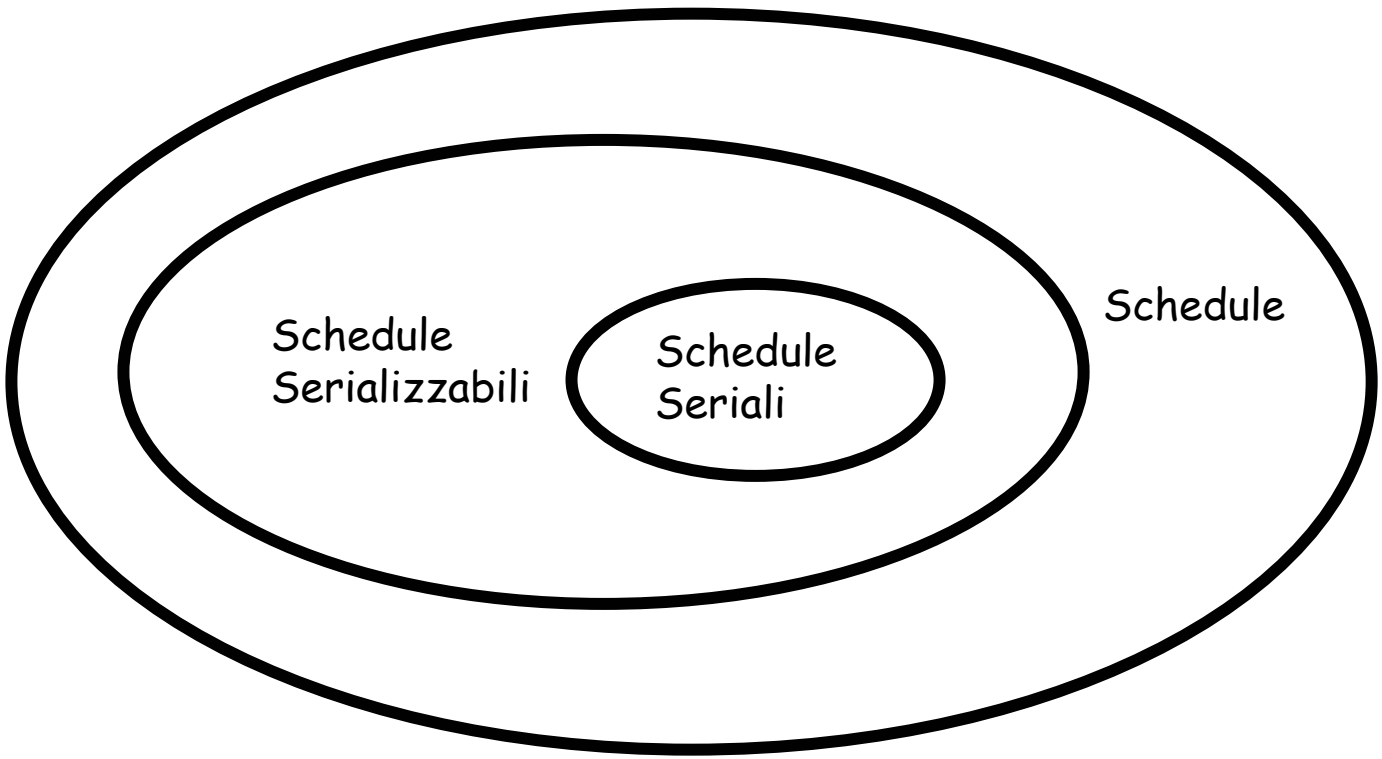
Controllo di concorrenza

- *Obiettivo*: evitare le 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

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



Conflict-serializzabilità

- Definizione preliminare:
  - Due azioni a_i ed a_j *sono* in *conflitto se:*
    - $i \neq j$
    - operano sullo stesso oggetto
    - almeno una di esse è una scrittura

      due casi: conflitto *read-write* (*rw* o *wr*)

      conflitto *write-write* (*ww*).

Uno "schedule" S_i si dice *conflict-equivalente* ad uno schedule S_j ($S_i \approx_c S_j$) se: **include le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi gli schedule**

- Uno schedule è *conflict-serializzabile* se è conflict-equivalente ad un qualche schedule seriale.
- L'insieme degli schedule conflict-serializzabili è indicato con **CSR**

La semantica della conflict - equivalenza

- Se due operazioni in conflitto sono applicate in ordine diverso nei due schedule l'effetto sulla base dati o sulle altre transazioni può essere differente:
  - r1(x), w2(x) in S1 e w2(x), r1(x) in S2
    - r1(x) **può** leggere valori diversi nei due schedule
  - w1(x), w2(x) in S1 e w2(x), w1(x) in S2
    - la successiva operazione rj(x) **potrebbe** leggere valore diverso
    - se le operazioni di scrittura fossero le ultime, il **valore** della risorsa x **potrebbe essere** diverso.

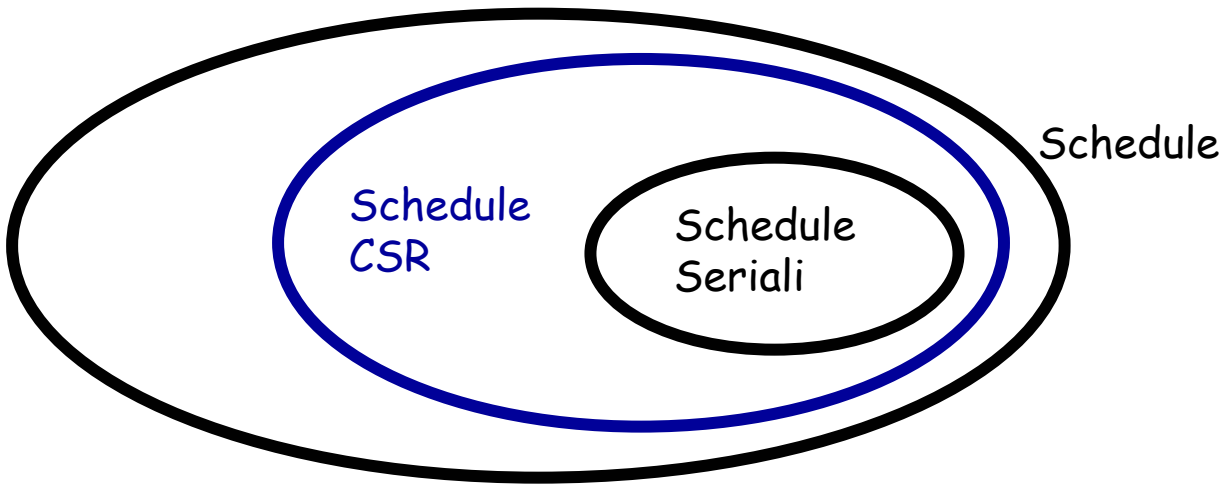
Verifica di conflict - serializzabilità

- 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 (facoltativo)
  - **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 $\approx_C$ (conflict-equivalente) 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.

Schedule CSR e “seriali”



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 stessa 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 della risorsa	**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	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
- 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 lock non può acquisirne altri

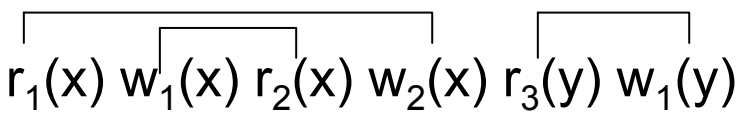
La classe 2PL

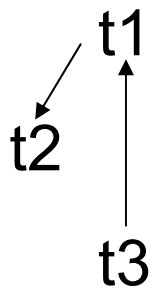
- Un sistema transazionale:
  - ben formato rispetto al locking:
    - cioè se le transazioni richiedono un lock opportuno prima di accedere alle risorse e lo rilasciano prima del termine della transazione;
  - con un gestore dei lock che rispetta le regole della tabella
  - con le transazioni che seguono il principio del lock a due fasi

  è caratterizzato dalla serializzabilità delle proprie transazioni.
- La classe 2PL contiene schedule che soddisfano queste condizioni

2PL e CSR

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





  - Viola 2PL: non è a due fasi poichè **t1** deve *cedere* un w-lock sulla risorsa x e poi richiederne un altro sulla risorsa y
  - Conflict-serializzabile rispetto alla sequenza t3 t1 t2

- Sufficienza: facoltativo

2PL implica CSR
(facoltativo)

- 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

Evitiamo effetto domino e letture sporche

Con l'ipotesi di "commit - proiezione"

- **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

Concorrenza e fallimento di transazioni

Rimuoviamo l'ipotesi di "commit - proiezione"

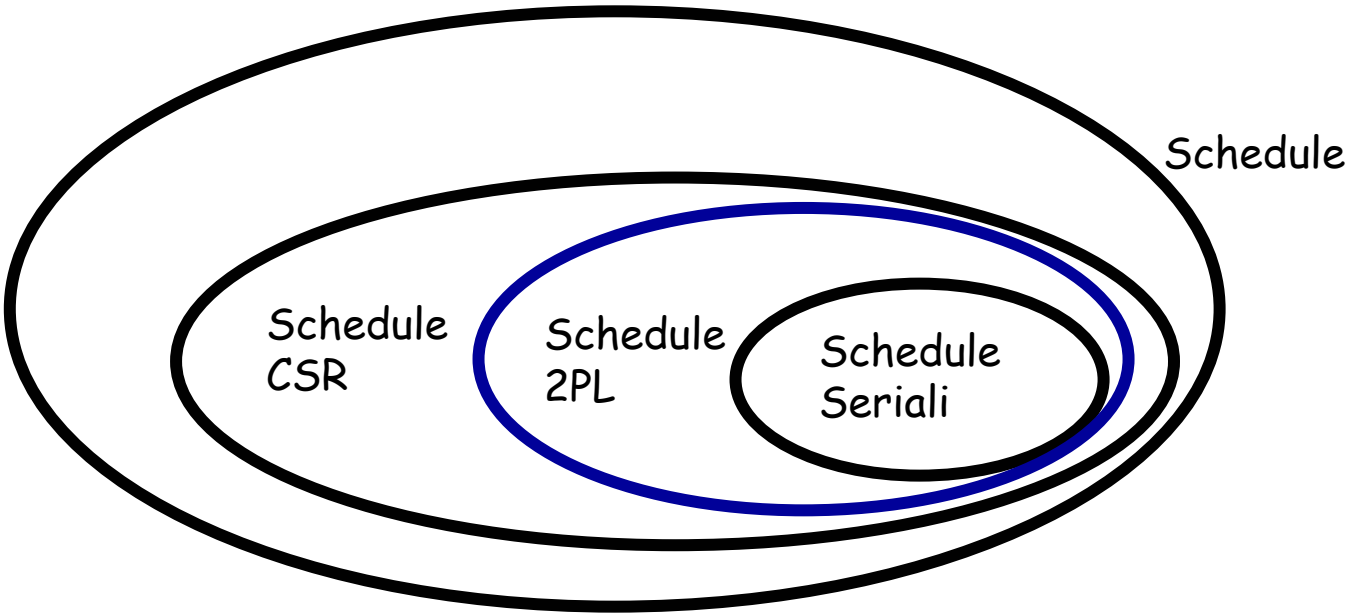
- Le transazioni possono fallire:
  - **rollback a cascata** ("effetto domino"):
    - se T_i ha letto un dato scritto da T_k e T_k fallisce, allora anche T_i deve fallire.
  - **letture sporche**:
    - se T_i ha letto un dato scritto da T_k e T_k fallisce, ma nel frattempo T_i è andata in commit, allora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

Locking a due fasi stretto

- Evita sia le letture sporche che l'effetto domino.

- 2PL con una condizione aggiuntiva:
  - **I lock possono essere rilasciati solo dopo il commit o l'abort.**

CSR 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 – sotto l'ipotesi della commit proiezione - è accettato solo se riflette l'ordinamento seriale delle transazioni indotto dai timestamp cioè:

Una transazione non può leggere un oggetto scritto da una transazione più giovane - cioè con time stamp superiore.

Una transazione non può scrivere un oggetto letto o scritto da una transazione più giovane - cioè con time stamp superiore

Algoritmo TS monoversione

- Lo scheduler ha due contatori RTM(***x***) e WTM(***x***) per ogni oggetto **x**
- Lo scheduler riceve richieste di lettura e scrittura (con indicato il timestamp ***ts*** 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 viene accolta e WTM(*x*) è posto uguale a *ts*
- Vengono uccise molte transazioni
- **Funziona sotto l' ipotesi di commit-proiezione.**

Algoritmo TS monoversione “stretto”

- Per eliminare il vincolo della commit-proiezione :
  - le scritture vengono “bafferizzate” effettuandole in memoria; trascritte solo dopo il commit della transazione scrivente.
- Inoltre:
  - le letture da parte di altre transazioni di dati bafferizzati mettono anche queste ultime in “attesa” del commit della transazione scrivente.

- Vengono in tal modo introdotti meccanismi di “wait” simili a quelli del 2PL:

Esempio TS monoversione

VALORI INIZIALI: **RTM(x) = 7** **WTM(x) = 4**

Richiesta	Risposta	NUOVI VALORI
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	

Una transazione non può leggere un oggetto scritto da una transazione più giovane – cioè con time stamp superiore.

Una transazione non può scrivere un oggetto letto o scritto da una transazione più giovane – cioè con time stamp superiore.

Algoritmo TS multiversione

- Per ogni transazione che modifica la base dati vengono mantenute diverse copie degli oggetti modificati.
- Ogni volta che una transazione modifica un oggetto x viene creata una nuova copia $WTM_i(x)$
- La copia RTM(x) rimane unica e globale
- Le copie sono rilasciate quando non vi sono più transazioni che devono leggere il loro valore.
- Le nuove regole:
  - read(x, ts)
    - la lettura è sempre accettata: ordinando le copie in funzione del timestamp, si legge dalla prima copia il cui indice è minore di ts;
  - write(x,ts)
    - Se ts< RTM(x) si rifiuta la richiesta altrimenti si aggiunge una nuova versione dell'oggetto x.

Esempio TS multiversione

VALORI INIZIALI: **RTM(x) = 7** **WTM1(x) = 4**

Richiesta	Risposta	NUOVI VALORI
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	WTM2(*x*) = 11
***read*(*x*,10)**	ok	RTM(x)=10 legge da x1

2PL vs TS

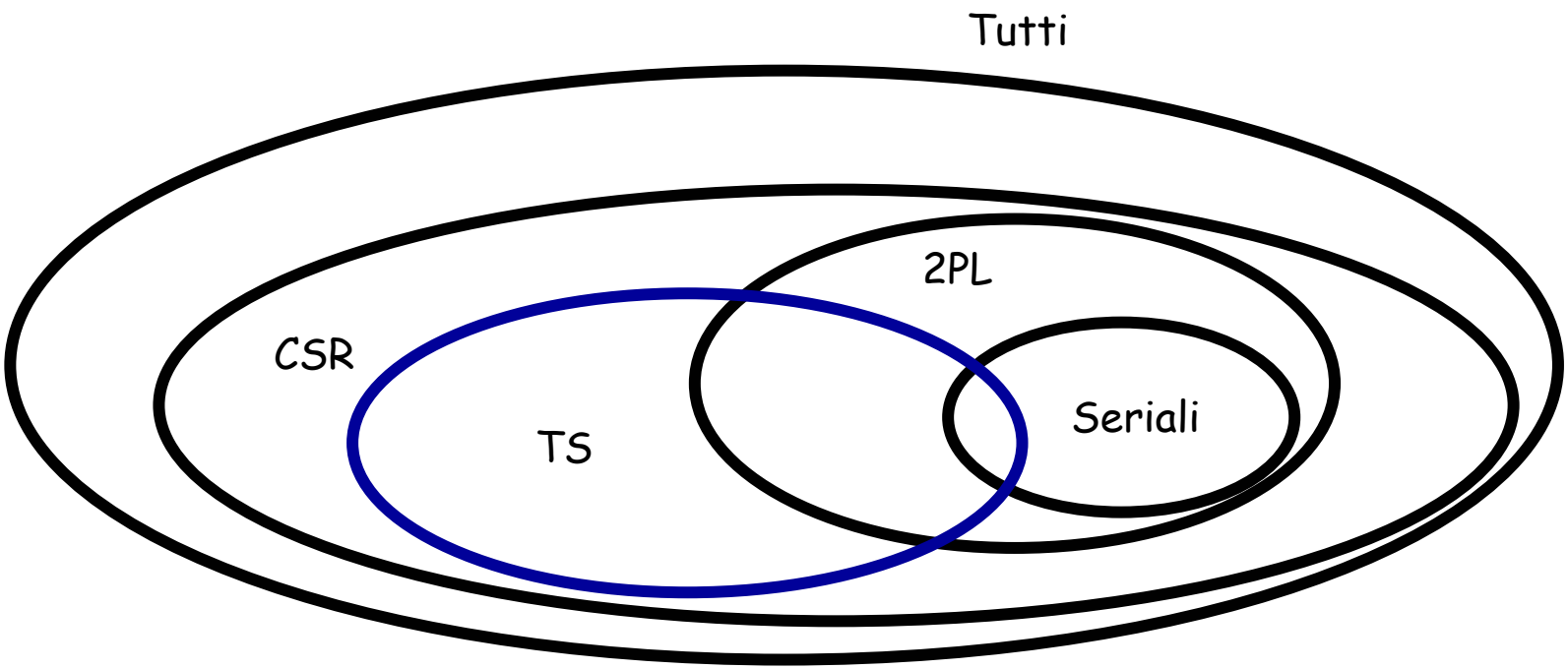
- Sono incomparabili
  - Schedule in TS ma non in 2PL

    $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_0(y)\ w_1(y)$
  - 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, 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**

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)

Time-out

- E’ la tecnica più semplice e più usata.
- Ad ogni richiesta di lock è associato un tempo massimo di attesa.
- Scaduto tale tempo, la richiesta si intende rifiutata e la transazione uccisa.

Detection

- Non si pongono vincoli alle transazioni.
- Ad intervalli prefissati, o quando succede qualcosa, il contenuto della tabella dei lock è esaminato e comparato con le richieste pendenti.
- Si costruisce un grafico delle richieste.
- Se in tale grafico esiste un ciclo, c'è un deadlock.
- Il ciclo deve essere spezzato uccidendo almeno una transazione.

Prevenction

- Tecnica di prevenzione esatta:
  - le transazione acquisiscono le risorse di cui hanno bisogno in un colpo solo;
  - se qualche risorsa non è disponibile, non viene assegnata nessuna risorsa.
- Problemi:
  - non sempre una transazione sa in partenza ciò di cui avrà bisogno;
  - si rischia di bloccare troppi oggetti inutilmente.

Prevenction

- Tecnica approssimata:
  - tutte le transazioni richiedono i lock nello stesso tempo;
  - non sempre la transazione conosce tutto quello di cui avrà bisogno;
  - non tutti i deadlock sono eliminati.
- Tecnica approssimata:
  - ad ogni transazione è associato un timestamp;
  - se un lock non è concesso, la transazione aspetta solo se essa è più giovane della transazione che detiene il lock;
  - non tutti i lock sono eliminati.

GESTIONE DELLA CONCORRENZA IN SQL

Transazioni e livelli di isolamento

- SET TRANSACTION
  [READ ONLY|READ WRITE]
  [ISOLATION LEVEL [READ UNCOMMITTED| READ COMMITTED|REPETEABLE READ|SERIALIZABLE]]
- Le transazioni possono essere a sola lettura oppure read /write (caso di default).
- Le transazioni a sola lettura non richiedono lock-esclusivi.
- L'isolamento tra le transazioni è definito tramite livelli crescenti.
- I livelli ad isolamento basso (*uncommitted, ...)* semplificano il controllo della concorrenza e vanno usati quindi quando ne ricorrono nella transazione le condizioni di funzionamento.
- **Tutti i livelli richiedono "il 2PL stretto" per le scritture.**
  - **la perdita di aggiornamento che coinvolge due transazioni che scrivono entrambe è sempre evitata**

Livelli di isolamento in SQL:1999

- per ogni transazione può essere scelto un appropriato livello di isolamento
  - **read uncommitted**
    - La transazione accetta di leggere dati modificati da una transazione che non ha ancora fatto il commit (ignora i lock esclusivi e non acquisisce lock in lettura).
    - permette letture sporche, letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **read committed**
    - La transazione accetta di leggere dati modificati da una transazione solo se questa ha fatto il commit.
      Se però essa legge due volte lo stesso dato, può trovare dati diversi.
    - **evita letture sporche** ma permette letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma

Livelli di isolamento in SQL:1999 (2)

- **repeatable read**
  - La transazione accetta di leggere dati cambiati da una transazione solo se questa ha fatto il commit. Inoltre se un dato è letto due volte, si avrà sempre lo stesso risultato.
  - evita letture sporche, letture inconsistenti, aggiornamenti fantasma ma consente inserimenti fantasma
- **serializable**
  - Si aggiunge al repeatable read la caratteristica che, se una query è fatta due volte, non vengono aggiunte righe.
  - evita tutte le anomalie

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:
  - Il blocco è sull'intera relazione
- Se siamo fortunati:
  - Il blocco è sull'indice

Uncommited Read

Transazione1

```
BEGIN TRAN
UPDATE authors
SET au_lname='Smith'

ROLLBACK TRAN
```

Transazione 2

```
SET TRANSACTION ISOLATION LEVEL
READ UNCOMMITTED

SELECT au_lname
FROM authors
WHERE au_lname='Smith'

SELECT au_lname
FROM authors
WHERE au_lname='Smith'
```

Legge dati modificati da una transazione che non ha fatto ancora commit

Commited Read

Transazione1	Transazione 2
	SET TRANSACTION ISOLATION LEVEL READ COMMITTED BEGIN TRAN
	SELECT au_lname FROM authors WHERE au_lname='Smith'
UPDATE authors SET au_lname='Smith'	
	SELECT au_lname FROM authors WHERE au_lname='Smith' COMMIT TRAN

Non legge dati modificati da una transazione che non ha ancora fatto il commit.

Repeatable Read

Transazione 1

UPDATE authors
SET au_lname='Jones'
--la query si blocca!

Transazione 2
SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ
BEGIN TRAN

SELECT au_lname
FROM authors
WHERE au_lname='Smith'

SELECT au_lname
FROM authors
WHERE au_lname='Smith'
COMMIT TRAN

Serializable (1/2)

Transazione 1

Transazione2

```
SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ
BEGIN TRAN

SELECT title FROM titles
WHERE title_id LIKE 'BU%'
```

```
INSERT titles
VALUES  ('BU2000',
'Inside SQL Server 2000',
'popular_comp', '0877', 59.95,
5000,  10, 0, null, 'Sep 10, 2000')
```

```
SELECT title FROM titles
WHERE title_id  LIKE 'BU%'
--risultato diverso!
COMMIT TRAN
```

Serializable (2/2)

Transazione 1

INSERT titles
VALUES ('BU3000',
'Itzik and His Black Belt SQL Tricks',
'popular_comp', '0877',
39.95, 10000, 12, 15000, null,
'Sep 15 2000')
--la query si blocca!

Transazione 2

SET TRANSACTION ISOLATION LEVEL
SERIALIZABLE
BEGIN TRAN

SELECT title FROM titles
WHERE title_id LIKE 'BU%'

SELECT title FROM titles
WHERE title_id LIKE 'BU%'
--risultato uguale!
COMMIT TRAN