

# Gestione delle transazioni

---

PROF. DIOMAIUTA CRESCENZO

# Transazioni

- Una transazione su una base dati è un'operazione, o una sequenza di operazioni, generata da un utente o programma applicativo che legge o aggiorna il contenuto di una base dati
  - Commit work (termina in uno stato finale previsto)
  - Rollback work (per abortire la transazione, porta il sistema ad uno stato precedente)
- Un sistema transazionale (OLTP) è un sistema in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti

Non avrò mai più nulla da eseguire

gestire come arbithro



# Transazioni: Esempio



*begin transaction*

**UPDATE PRODOTTI SET Quantita=Quantita-y WHERE Codice=x;**

**INSERT INTO CARRELLO(Cliente,Prodotto,Quantita,Data)**

**VALUES (z,x,v,'13-Gen-2015 20:31:54')**

*commit*

# Transazioni

---

Una **transazione** è una **unità logica di elaborazione** che **corrisponde ad una serie di operazioni fisiche elementari** (**lettura/scrittura**) su **una base dati**, 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 **devono** 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
- **Abort** (o **rollback**)
  - richiesto dalla transazione stessa (suicidio)
  - richiesto dal sistema (omicidio) per:
    - violazione vincoli, gestione concorrenza, incompletezza causa guasto sistema



Undo: operazioni che non dobbiamo disfare.

Se suppongo per HP che transazione viene salvata ma va via la corrente, bisogna rifarlo in automatico

# 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 evitandone 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 abort.

# 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).

# Durability (Persistenza)

---

Una transazione andata in commit deve mantenere in modo permanente gli effetti della sua esecuzione anche in presenza di guasti.

# Transazioni: Esempio con controlli sui vincoli

*begin transaction*

**SELECT Quantita-x INTO Q FROM Prodotti WHERE Codice=x;**

**IF (Q>0) THEN**

**UPDATE PRODOTTI SET Quantita=Quantita-y WHERE Codice=x;**

**INSERT INTO CARRELLO(Cliente,Prodotto,Quantita,Data) VALUES  
(z,x,v,'13-Gen-2015 20:31:54')**

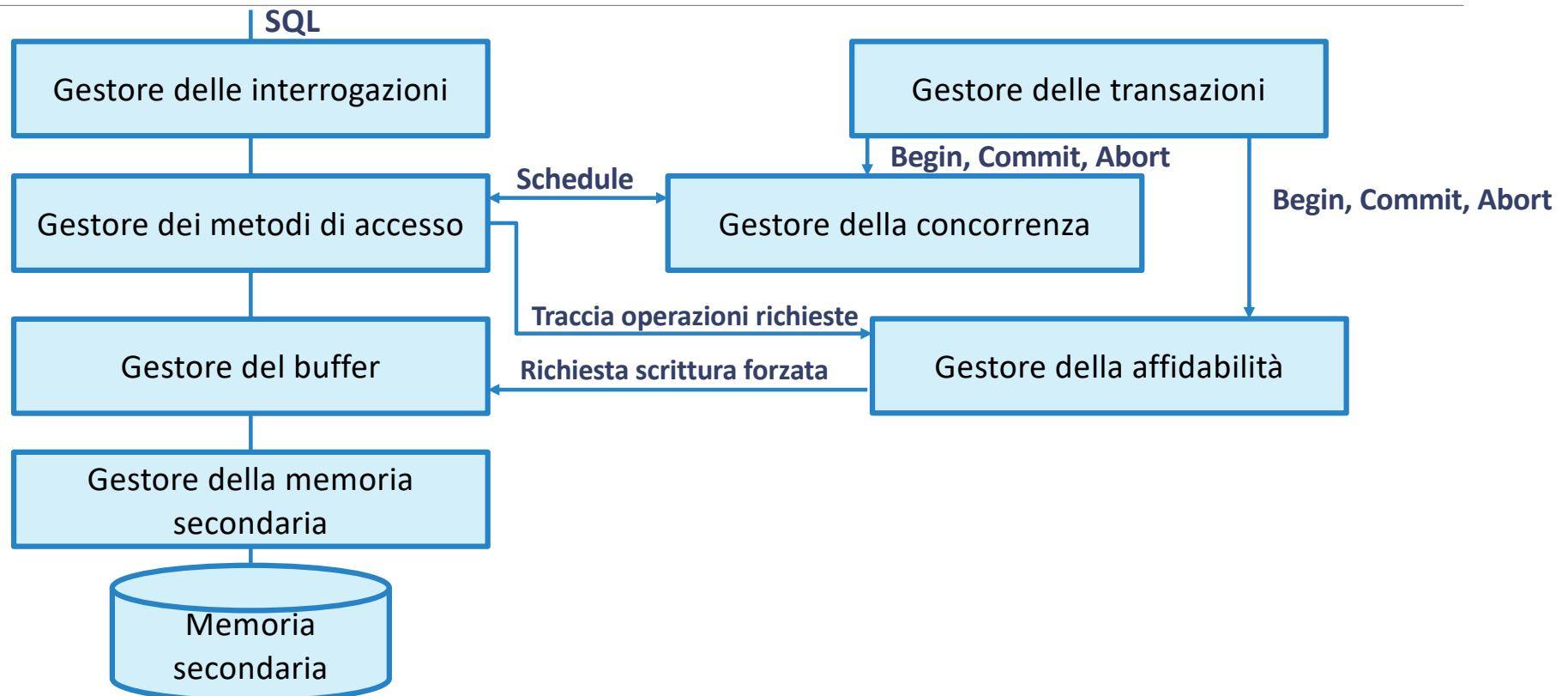
**commit**

**ELSE**

**rollback**

**END IF**

# Moduli di un DBMS per gestire interrogazioni e transazioni



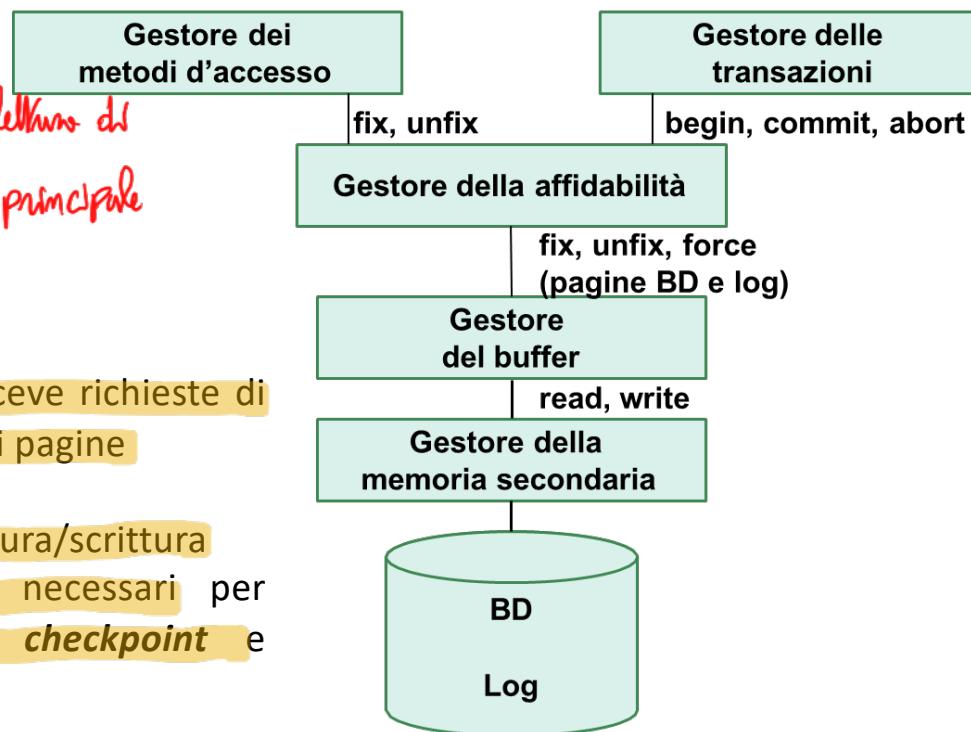
# Gestore dell'affidabilità

- Il gestore dell'affidabilità garantisce due proprietà fondamentali delle transazioni:  
***l'atomicità*** e la ***persistenza***.
- Gestisce l'esecuzione dei comandi transazionali
  - ***start transaction*** (B, begin) *B usual magli schema*
  - ***commit work*** (C)
  - ***rollback work*** (A, abort)
- Gestisce le operazioni di ripristino (recovery) dopo i guasti:
  - ***warm restart*** (ripresa a caldo) e ***cold restart*** (ripresa a freddo)  
→ *guasti più soft vs guasti più hard*
- Utilizza un ***log***:
  - Un archivio permanente che regista le operazioni svolte dal DBMS  
*↳ file sequenziale per tenere traccia delle informazioni*

# Architettura del controllore di affidabilità

Hp: ogni lettura eguale a lettura di intera pagina da memoria principale

1. Il gestore dell'affidabilità riceve richieste di accessi in lettura/scrittura di pagine
2. Le passa al buffer manager
3. Genera altre richieste di lettura/scrittura
4. Infine predisponde i dati necessari per garantire meccanismi di **checkpoint** e **dump**.



# Memoria stabile

---

- Il gestore dell'affidabilità deve disporre di **memoria stabile**, ovvero memoria resistente ai guasti (nastri, repliche RAID di dischi, ecc.)
- In generale:
  - **Memoria centrale**: non è persistente
  - **Memoria di massa**: è persistente ma può danneggiarsi
  - **Memoria stabile**: memoria che non può danneggiarsi (è una astrazione, poiché è impossibile):
    - Si ottiene attraverso la ridondanza (**mirrored**), ovvero l'informazione viene memorizzata in maniera identica su entrambi i dischi:
      - In alcune applicazioni sono dischi replicati
      - In altre applicazioni sono unità a nastro

# Organizzazione del log

---

<<Il log è un file sequenziale gestito dal controllore dell'affidabilità, scritto in memoria stabile che registra tutte le operazioni svolte dalle transazioni nel loro ordine di esecuzione>>

- "Diario di bordo": riporta tutte le operazioni in ordine temporale di esecuzione, ha un blocco corrente (detto **top**)
- Il log di un sistema di basi dati contiene due tipologie di record:
  - Record di transazione (riguardano una transazione o più transazioni)
  - Record di sistema (aggiunge dei record)

# Organizzazione del log

## **Record di transazione** che contengono:

- L'identificativo della transazione (**T**);
- Il timestamp delle varie azioni (**t<sub>s</sub>**) → BCA riguardano specifica transazione
- L'azione (**O<sub>p</sub>**) svolta da una transazione: begin (**B**), update (**U**), delete (**D**), insert (**I**), commit (**C**), abort (**A**)
- L'identificativo dell'oggetto della base di dati coinvolto (**O**); → B<sub>I</sub>/B<sub>S</sub>
- Il valore dell'oggetto prima della modifica (**B<sub>I</sub>**) da parte di una transazione (*before-image*)
- Il valore dell'oggetto dopo della modifica (**A<sub>I</sub>**) da parte di una transazione (*after-image*)

Es: cambio valore dello stipendio di impiegato, prima e dopo salviamo quello

NOTA: Per ogni azione che fa la transazione è salvato un record

# Organizzazione del log

- Record di sistema** → Record di dump: contiene una copia della base dati. Durante il dump la base dati non può essere usata
- Dump (DP)** contenente l'istante di tempo ( $t_s$ ) in cui è stato effettuato l'ultimo backup della base di dati ed alcuni dettagli pratici (es. file e dispositivi coinvolti)
- Checkpoint (CK)** contenente l'insieme delle transazioni attive ( $T$ )  
↳ indicano da dove partono nel caso ho un guasto
- È possibile schematizzare i record delle transazioni nella forma  $O_p(T, t_s, O, B_l, A_l)$  mentre i record di sistema nella forma  $DP(t_s)$  e  $CK(t_s, T)$

$$(T_1, t_1, O; B_1, A_1)$$

↑ before-s      ↗ after-s

# Organizzazione del log: Esempio

- In riferimento all'esempio precedente è riportato un esempio di file di log

```

DP, B(T1,-,,-,-), U(T1,-,qtaP,100,90), U(T1,-,qtaC,NULL,10), C(T1,-,,-,-,-),
B(T2,-,,-,-), CK(T2)*, U(T2,-,qtaP,90,70), U(T2,-,qtaC,NULL,20), C(T2,-,,-,-,-),
B(T3,-,,-,-), U(T3,-,qtaP,100,90), U(T3,-,qtaC,NULL,10), C(T3,-,,-,-,-), ...

```

↑ info temporale                                  ↑ oggetto, valore before e valore after

- Con **qtaP** il campo quantità della tabella **Prodotti** e **qtaC** campo quantità della tabella **Clienti**

\* Informazioni sulle transazioni attive da quel momento. T<sub>1</sub> è terminata. Se ho questi dati dopo ck(T<sub>2</sub>) debbo ripetere T<sub>2</sub>

# Organizzazione del log

---

- Questi record consentono di “**disfare**” e “**rifare**” le rispettive azioni sulla base di dati, attraverso operazioni specifiche
  - Undo di una azione su un oggetto O:
    - update, delete: copiare dal log il valore del before state ( $B_i$ ) nell'oggetto O
    - insert: cancellare O
  - Redo di una azione su un oggetto O:
    - insert, update: copiare dal log il valore dell' after state ( $A_i$ ) nell'oggetto O
    - delete: cancellare O

# Organizzazione del log

---

- Si precisa che i due valori  $B_I$  e  $A_I$  descrivono rispettivamente i valori di un oggetto  $O$  prima (before image) e dopo (after image) la modifica.
- Assumiamo per semplicità che  $B_I$  e  $A_I$  contengono copie complete delle pagine modificate

- Nel caso di UPDATE sono entrambi definiti

$$\text{UPDATE}(T, O, B_I, A_I)$$

- Nel caso di INSERT è definito solo il valore di  $A_I$  poiché l'oggetto prima della modifica non esiste cioè:

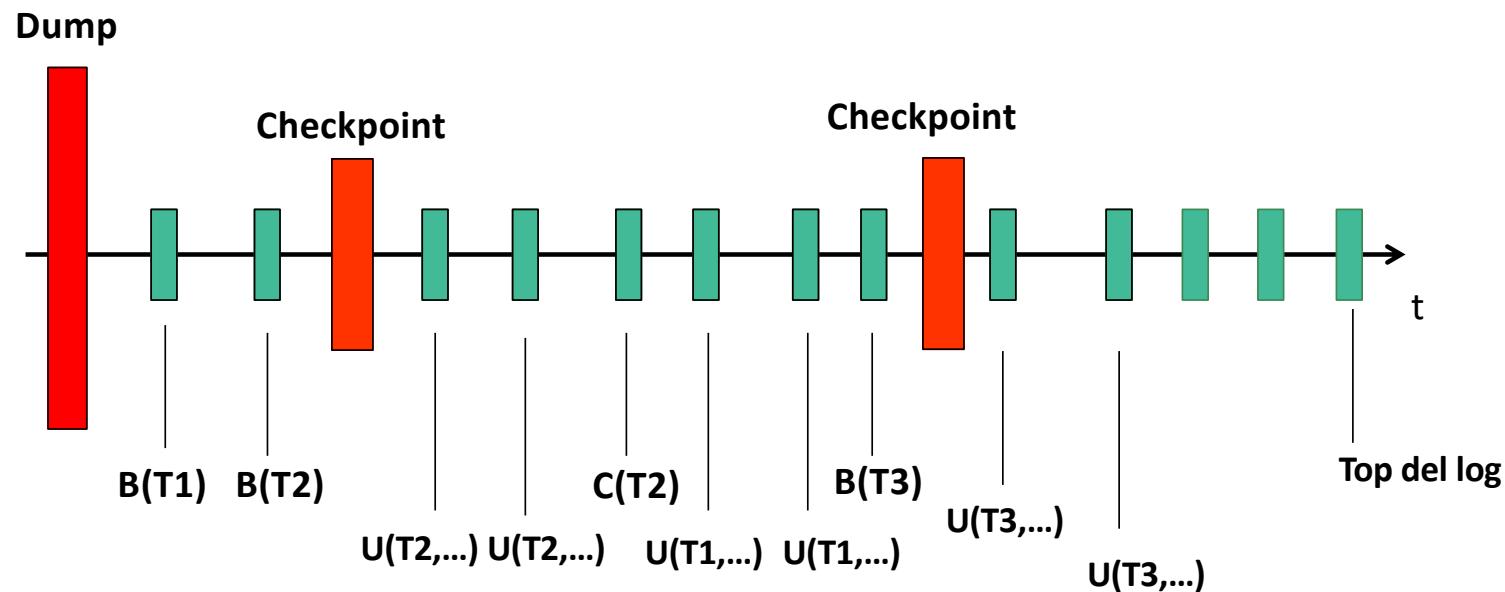
$$\text{INSERT}(T, O, -, A_I) = \text{INSERT}(T, O, A_I)$$

- Nel caso di DELETE è definito solo il valore di  $B_I$  poiché l'oggetto dopo la modifica non esiste.

$$\text{DELETE}(T, O, B_I, -) = \text{DELETE}(T, O, B_I)$$

# Struttura del log

---



# Checkpoint

---

- I record di log servono a “ricostruire” le operazioni a seguito di un guasto. Per evitare che la ricostruzione parte dall’inizio del log (in genere sono molto lunghi), sono stati introdotti i **checkpoint** e i **dump**.
  
- Un **checkpoint** è una operazione svolta periodicamente dal gestore dell’affidabilità con lo scopo di registrare le transazioni **attive** in un certo istante e confermare che le altre transazioni o non sono iniziate o sono finite

# Checkpoint

---

- Esistono diverse versioni di questa operazione, la più semplice è:
  1. Si sospende l'accettazione di operazioni di scrittura (aggiornamenti, inserimenti, ...), commit o abort, da parte di ogni transazione  
↳ quelle che modificano la base dati
  2. Si trasferiscono in memoria di massa (tramite **force**) tutte le pagine del buffer modificate, relative a transazioni andate in commit.  
↳ con **setfink 1**.
  3. Si scrive in modo sincrono (**force**) nel log un record di checkpoint **CK(T1,T2,...,Ti)** contenente gli identificatori delle transazioni attive (in corso)
  4. Si riprende l'accettazione delle operazioni sospese
- Questo meccanismo garantisce che:
  - Gli effetti delle transazioni che hanno effettuato il commit i dati sono in memoria di massa (registrati permanentemente)
  - Nel checkpoint sono elencate le transazioni "a metà strada" (non si conosce ancora l'esito finale commit o abort)

# Dump

---

- È una copia completa e consistente (**backup**) della base di dati
  - Normalmente effettuata in mutua esclusione con le altre transazioni quando il sistema non è operativo (Es.: di notte o nel fine settimana)
  - La copia è memorizzata in **memoria stabile**.  
*→ non parallelamente secondaria, ma quella con effetto immediato.*
  - Al termine del dump viene scritto nel log un record di **DUMP(...)** che indica il momento in cui il dump è stato effettuato (e dettagli pratici, file, dispositivo, ...)

# Esecuzione delle transazioni e scrittura nel log

---

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

# Regole per il log

---

## □ Regola WAL (Write-ahead-Log – scrivi il log per primo):

- Impone che la parte **before image** dei record di un log venga scritta nel log prima di effettuare la corrispondente azione sul database
  - consente di **disfare** le azioni poiché per ogni aggiornamento viene reso disponibile nel log il valore prima della scrittura sul database **UNDO PRIVILEGE**

## □ Regola Commit-Precedenza:

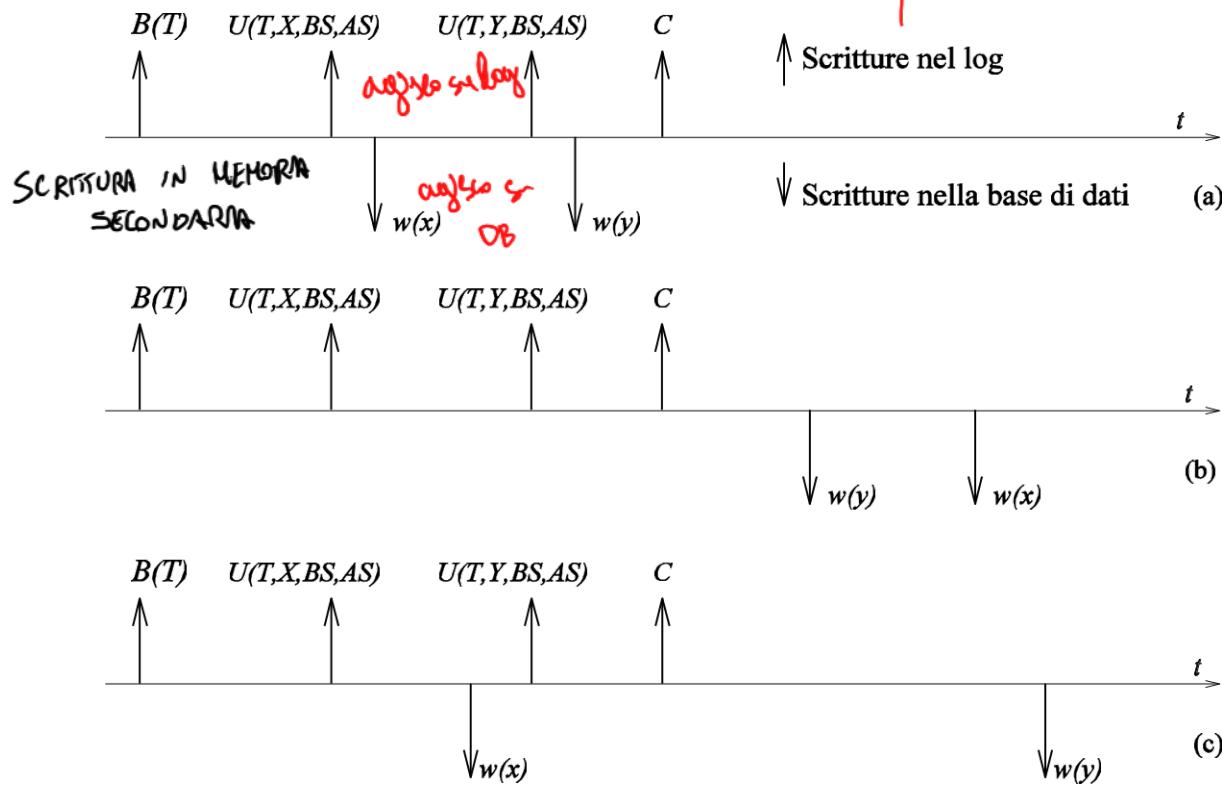
- Impone che la parte **after image** dei record di un log venga scritta nel log prima di effettuare il commit
  - consente di **rifare** le azioni poiché se le pagine modificate non sono state ancora trascritte dal buffer manager viene reso disponibile nel log il valore in esse registrato già decise da una transazione che ha effettuato un commit

**REDO PRIVILEGE**

Undo: tutte le transaz. ols cui non ho fatto commit prima del guasto.

Redo: dopo questo ripetere transazioni complete.

# Scrittura congiunta di log e base di dati: Protocollo



#### Modalità Immediata

Ogni scrittura sul log è 'immediatamente' seguita da una scrittura sulla base dati. Richiede **Undo** delle operazioni di transazioni uncommitted al momento di un guasto (evita operazioni di **Redo**).

#### Modalità Differita

Ogni scrittura di transazioni sulla base dati è 'differita' alla successiva memorizzazione del record commit nel file di log. Evita operazioni di **Undo** richiede **Redo** delle operazioni di transazioni committed in caso di guasto.

#### Modalità Mista

La scrittura può avvenire in modalità sia immediata che differita, causando incertezza su tutte le transazioni nella lista del CK. Consente l'ottimizzazione delle operazioni di flush. Richiede sia **Undo** che **Redo**.

1 entra redo e rallegra un po'.

2 effettua tutto alla fine dopo quello che doveva fare. Se fanno con A non faccio niente.

3 puo' agire in modo immediato per due i controlli

# Gestione dei guasti

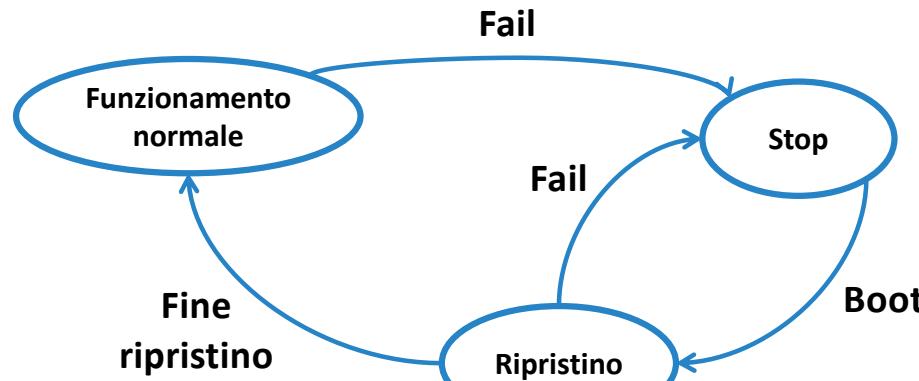
Effetto	Trucco
Armi 1	R1 R2 L1 R2 ←→↑↓←→↑↓←→↑
Armi 2	R1 R2 L1 R2 ←→↑↓←→↑↓←→↑↓←→↑
Armi 3	R1 R2 L1 R2 ←→↑↓←→↑↓←→↑↓←→↑
Munizioni infinite	L1 R1 □ R1 ←→ R2 R1 □ L1 L1 L1
Livello armi max	↓ □ × ←→ R1 R2 ←→ ↓ L1 L1 L1
Mirino sui veicoli	↑↑□ L2 →× R1 ↓ R2 ○
Paracadute	←→ L1 L2 R1 R2 R2 ↑↓ L1
Jetpack	←→ L1 L2 R1 R2 ↑↓ ←→
Mai ricercato	○→○→←□△↑
Sospetto +2 stelline	R1 R1 ○ R2 ←→↑↓←→↑↓
Ricercato con 6 stelle	○→○→←□×↓
Azzera il livello ricercato	R1 R1 ○ R2 ↑↓↑↓↑↓

I guasti, dal punto di vista di un DBMS, si dividono in due classi:

- **Guasti di sistema (soft):** errori di programma, crash di sistema, cali 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 di dispositivo (hard):** sui dispositivi di gestione della 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

- ☐ Quando il sistema individua uno dei due tipi di guasti **forza un arresto completo delle transazioni attive (fail-stop)**
- ☐ Successivamente viene **ripristinato il corretto funzionamento del sistema operativo (boot)**
- ☐ Viene **attivata una procedura di ripristino**, chiamata **ripresa a caldo (warm restart)** nel caso di guasto di sistema e **ripresa a freddo (cold restart)** nel caso di guasto di dispositivo.



# Processo di ripresa (restart)

- Con questo modello il **guasto** è un evento istantaneo.  
L'obiettivo della ripresa 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)
- Alcuni sistemi aggiungono al log **un record di end** per semplificare i protocolli di ripristino. Il modello **fail-stop** non utilizza questa tipologia di record.



# Ripresa a caldo

a seguito di un crash (soft)

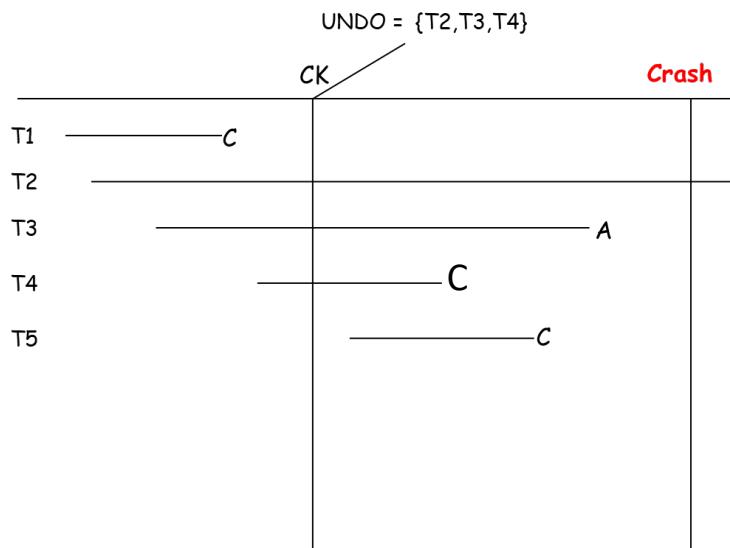
- La ripresa a caldo si articola in quattro fasi:

1. Si ripercorre all'indietro il log fino a trovare l'ultimo record di checkpoint
2. Si decidono le transazioni da rifare o disfare. Si costruiscono due insiemi **UNDO** (transazioni che hanno effettuato *l'abort*) e **REDO** (transazioni che hanno effettuato il *commit*). Inizializzazione di Undo con le **transazioni attive** al checkpoint, Redo si inizializza ad insieme vuoto.
3. Si ripercorre il log **all'indietro**, fino alla più vecchia azione delle transazioni in **UNDO** e **REDO**, disfacendo tutte le azioni delle transazioni in **UNDO**
4. Si ripercorre il log in **avanti**, rifacendo tutte le azioni delle transazioni in **REDO** (*si replica il comportamento delle transazioni originali*)

- Questo meccanismo garantisce atomicità e persistenza.

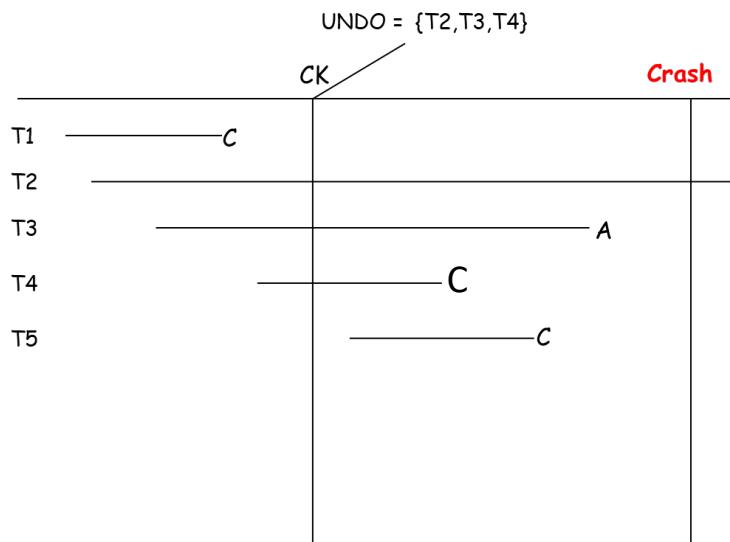
# Ripresa a caldo: Esempio

- 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)



# Ripresa a caldo: Esempio

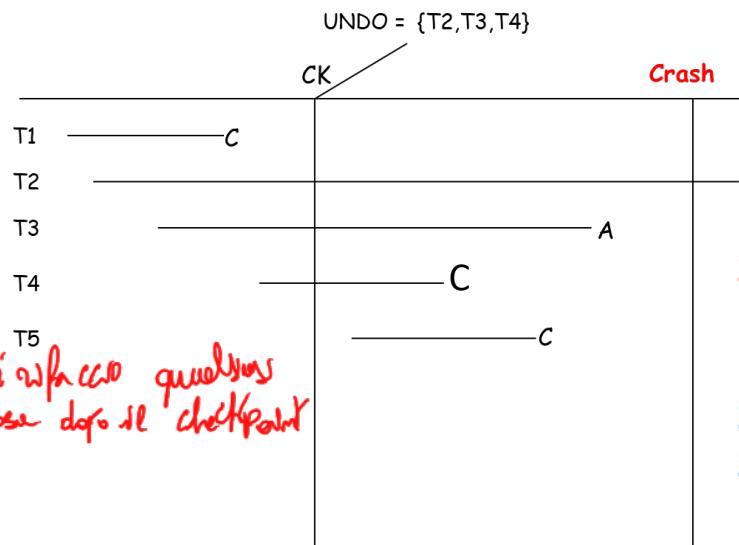
- B(T1)
- B(T2)
- U(T2, O1, B1, A1)
- I(T1, O2, A2)
- B(T3)
- C(T1)
- B(T4)
- U(T3,O2,B3,A3)
- U(T4,O3,B4,A4)
- CK(T2,T3,T4)
- C(T4)
- B(T5)
- U(T3,O3,B5,A5)
- U(T5,O4,B6,A6)
- D(T3,O5,B7)
- A(T3)
- C(T5)
- I(T2,O6,A8)



1. Ricerca dell'ultimo checkpoint e costruzione degli insieme UNDO e REDO
0. UNDO = {T2,T3,T4}; REDO = {}

# Ripresa a caldo: Esempio

- 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) *dov'è infarto, perché non faccio qualcosa*  
2. • B(T5) *cose dopo il checkpoint*
- 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)

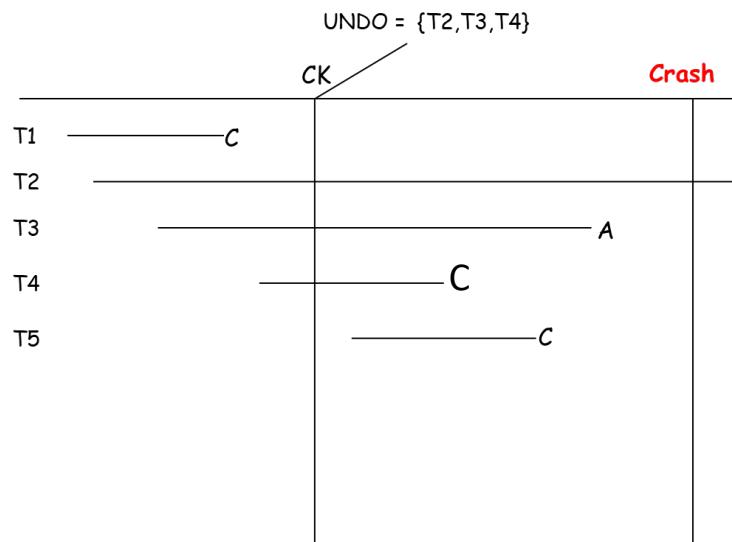


2. Si percorre in avanti il record di log e si aggiornano gli insiemi Undo e Redo
1. C(T4): UNDO = {T2,T3}; REDO = {T4}
  2. B(T5): UNDO = {T2,T3,T5}; REDO = {T4}
  3. C(T5): UNDO = {T2,T3}; REDO = {T4,T5}
- SETUP

# Ripresa a caldo: Esempio

UNDO:  $T_2$  e  $T_3$

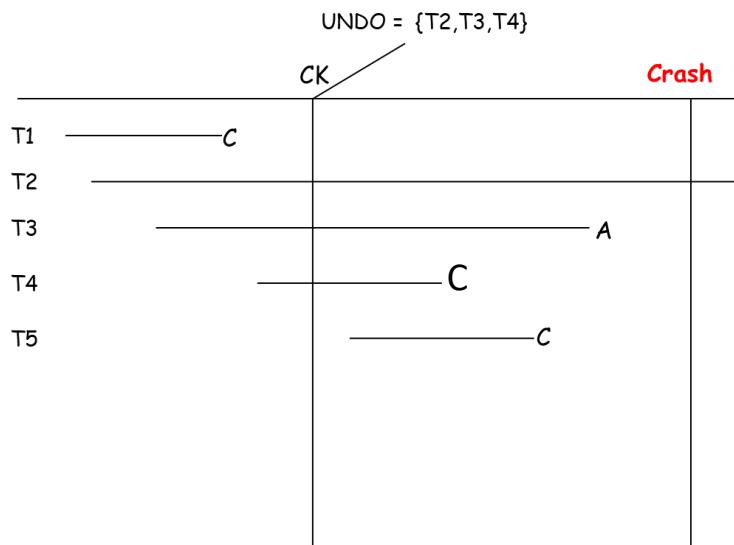
- B(T1)
- B(T2)
- 8. • U(T2, O1, B1, A1)
- I(T1, O2, A2)
- B(T3)
- C(T1)
- B(T4)
- 7. • U(T3,O2,B3,A3)
- 9. • U(T4,O3,B4,A4)
- CK(T2,T3,T4)
- 1. • C(T4)
- 2. • B(T5)
- 6. • U(T3,O3,B5,A5)
- 10. • U(T5,O4,B6,A6)
- 5. • D(T3,O5,B7)
- A(T3)
- 3. • C(T5)
- 4. • I(T2,O6,A8)



- fino all'azione più vecchia di quelle da effettuare
3. Si ripercorre indietro il log fino all'azione Undo:
  4. D(O6)
  5. O5 = B7 (Re-Insert)
  6. O3 = B5
  7. O2 = B3
  8. O1 = B1
- UNDO

# Ripresa a caldo: Esempio

- 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. Infine si svolgono le azioni di Redo:  
9. O3 = A4  
10. O4 = A6

REDO

# Ripresa a freddo

---

□ La ripresa a freddo risponde ad un guasto che provoca il deterioramento di una base di dati. Si articola in tre fasi:

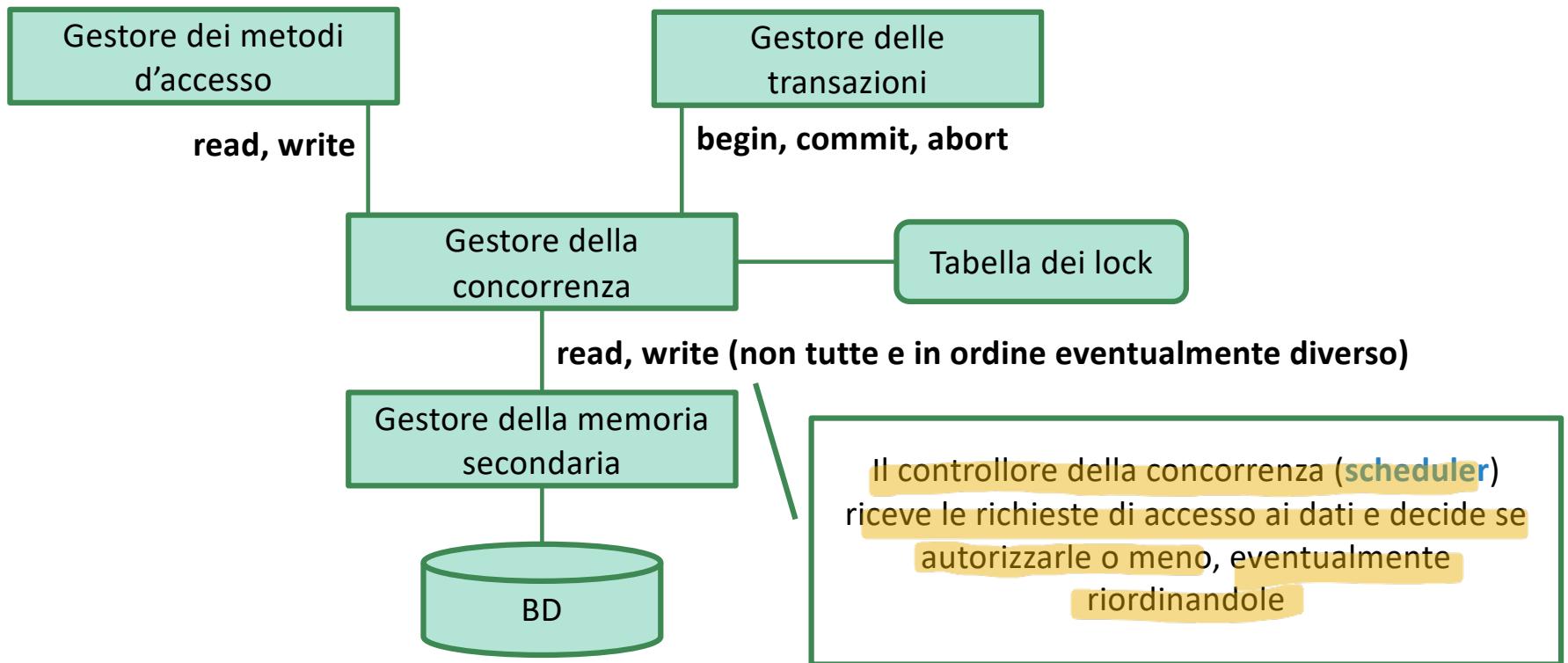
1. Si ripristinano i dati a partire dal backup accedendo al più recente record di dump del log.
2. Si ripercorre in avanti il log, applicando relativamente alla parte deteriorata, tutte le operazioni registrate sul log riportandosi all'istante precedente il guasto (abort e commit)
3. Infine si esegue una ripresa a caldo

# Controllo della concorrenza

---

- Un DBMS deve spesso servire diverse applicazioni, e rispondere alle richieste provenienti da diversi utenti. Per misurare il carico applicativo di un DBMS si usa il parametro tps (numero di transazioni al secondo)
  
- In genere, pensando ad un sistema informativo bancario o finanziario ci sono decine o centinaia di transazioni al secondo. È indispensabile quindi eseguire le transazioni concorrentemente, infatti è impensabile una loro esecuzione seriale.

# Architettura per il controllo di concorrenza



# Modello di riferimento

---

- Utilizzeremo una descrizione astratta della base di dati nei termini di oggetti  $x$ ,  $y$ ,  $z$  ed esemplificheremo operazioni in memoria centrale su di esse come se i loro valori fossero **numerici**
- La loro lettura e scrittura richiedono lettura e scrittura dell'intera pagina in cui risiedono i dati, si utilizzeranno le azioni **read( $x,y$ )** e **write( $x,y$ )** per denotare la **lettura** e la **scrittura** della pagina, dove  $x$  rappresenta un oggetto astratto della base di dati (es. **campi di tabella**) ed  $y$  il valore letto o da scrivere. Inoltre è presente l'istante temporale in cui avvengono
- L'esecuzione concorrente di diverse transazioni può causare alcuni problemi di correttezza (**anomalie**). Esaminiamo 5 casi tipici.

# Esempio di Modello di Transazione

t<sub>1</sub>: **bot**  
t<sub>2</sub>: read(qtaP<sub>x</sub>, A)  
t<sub>3</sub>: A = A - y  
t<sub>4</sub>: write(qtaP<sub>x</sub>, A)  
t<sub>5</sub>: B = z  
t<sub>6</sub>: write(clienteC<sub>xz</sub>, B)  
t<sub>7</sub>: C = x  
t<sub>8</sub>: write(prodC<sub>xz</sub>, C)  
t<sub>9</sub>: D = y  
t<sub>10</sub>: write(qtaC<sub>xz</sub>, D)  
t<sub>11</sub>: E ='13-Gen-2014 19:30:54'  
t<sub>12</sub>: write(dataC<sub>xz</sub>, E)  
t<sub>13</sub>: **commit**

Con  
qtaP<sub>x</sub>: quantità del prodotto x  
cliente C<sub>xz</sub>: codice del cliente z  
prodC<sub>xz</sub>: che acquista il prodotto x  
qtaC<sub>xz</sub>: quantità acquistata  
dataC<sub>xz</sub>: nella data specificata

# Perdita di aggiornamento (Lost update)

- Si verifica quando un'operazione di aggiornamento di una transazione si sovrappone a quella di un'altra che stava aggiornando lo stesso oggetto (prima che quest'ultima abbia effettuato il commit), annullandone nella pratica gli effetti.

Es: consideriamo due transazioni di acquisito dello stesso prodotto generate da due utenti diversi utenti ( $z_1$  e  $z_2$ ) e sovrapposte in parte temporalmente. In particolare, si suppone che la seconda transazione inizia la propria esecuzione e legge il valore della quantità del prodotto  $x$  disponibile in magazzino prima che la prima transazione abbia effettuato alcun aggiornamento ed il conseguente commit.

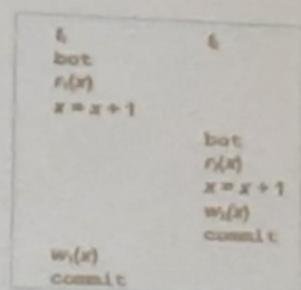
Time	$T_1$	$T_2$	$qtaP_x$
$t_1$	<b>bot</b>		100
$t_2$	read( $qtaP_x$ , A)	<b>bot</b>	100
$t_3$	$A = A - 50$	read( $qtaP_x$ , A)	100
$t_4$	write( $qtaP_x$ , A)	$A = A - 30$	50
$t_5$	write( $qtaC_{xz1}$ , 50)	write( $qtaP_x$ , A)	70
$t_6$	<b>commit</b>	write( $qtaC_{xz2}$ , 30)	70
$t_7$		<b>commit</b>	70

# Perdita di aggiornamento (Lost update)

- Supponiamo di avere due transazioni identiche che operano sullo stesso oggetto della BD:
  - $t_1 : r_1(x), x = x + 1, w_1(x)$
  - $t_2 : r_2(x), x = x + 1, w_2(x)$
- Inizialmente  $x=2$ . Dopo l'esecuzione seriale di  $t_1$  e  $t_2$ ,  $x=4$
- Vediamo l'esecuzione concorrente

Un aggiornamento viene perso, il valore finale di  $x=3$ .  
Si perdono gli effetti della transazione  $t_2$ .

$r(x)$  lettura del generico oggetto  $x$  da parte della transazione  $t_i$ , e  $w(x)$  la scrittura dello stesso oggetto.  
L'incremento dell'oggetto  $x$  denota ad esempio un update fatto da un programma applicativo.



# Lettura sporca (Dirty read)

- Si verifica quando una transazione, durante la sua esecuzione, legge un valore "sporco" di un oggetto, ovvero modificato da un'altra transazione che poi viene **abortita**.

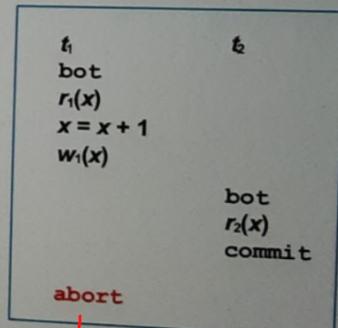
Es: due transazioni di acquisto dello stesso prodotto generate da utenti diversi si sovrappongono temporalmente. Questa volta si suppone che la prima, dopo avere aggiornato la quantità di prodotto disponibile ed il relativo record nel carrello, effettua in un istante di tempo successivo agli aggiornamenti un **abort spontaneo** (ad esempio l'utente desidera per qualche motivo annullare la transazione). La seconda transazione, che si ipotizza essere iniziata dopo le operazioni di aggiornamento della prima, opererà su un valore non corretto della quantità di prodotto disponibile che non è 50 ma in realtà 100.

Time	T <sub>1</sub>	T <sub>2</sub>	qtaP <sub>x</sub>
t <sub>1</sub>	<b>bot</b>		100
t <sub>2</sub>	read(qtaP <sub>x</sub> , A)		100
t <sub>3</sub>	A = A - 50		100
t <sub>4</sub>	write(qtaP <sub>x</sub> , A)		50
t <sub>5</sub>	write(qtaC <sub>XZ1</sub> , 50)	<b>bot</b>	50
t <sub>6</sub>	...	read(qtaP <sub>x</sub> , A)	50
t <sub>7</sub>	...	A = A - 30	50
t <sub>8</sub>	...	write(qtaP <sub>x</sub> , A)	20
t <sub>9</sub>	<b>rollback</b>	write(qtaC <sub>XZ2</sub> , 30)	70
t <sub>10</sub>		<b>commit</b>	70

# Lettura sporca (Dirty read)

- Supponiamo di avere due transazioni, una che incrementa un oggetto X di 1, ma va in abort dopo aver scritto e una che legge tale valore:
  - $t_1 : r_1(x), x = x + 1, w_1(x)$
  - $t_2 : r_2(x)$
- Inizialmente  $x=2$ .
- Vediamo l'esecuzione concorrente

Il valore finale di x al termine dell'esecuzione è 2, ma  $t_2$  ha letto il valore 1 (che rappresenta uno stato intermedio, ma non definitivo, in quanto  $t_1$  non ha salvato il valore (abort))



# Letture inconsistenti (Inconsistent read)

- La lettura inconsistente (inconsistent Read) si verifica quando una transazione legge in due diversi istanti della sua esecuzione due valori differenti di un oggetto che viene, intanto, modificato da un'altra transazione.

Supponiamo una transazione di acquisto a cui si sovrappone temporalmente l'esecuzione di una transazione che effettua due letture in istanti successivi della quantità di un prodotto disponibile in magazzino: una prima e uno dopo il commit di una transazione.

Non va bene, una transazione che accede due volte alla BD deve trovare esattamente lo stesso valore per ogni dato letto (non deve risentire di altre transazioni).

Time	T <sub>1</sub>	T <sub>2</sub>	qtaP <sub>x</sub>
t <sub>1</sub>	<b>bot</b>		20
t <sub>2</sub>	read(qtaP <sub>x</sub> , A)		20
t <sub>3</sub>	...		20
t <sub>4</sub>	...		20
t <sub>5</sub>	...	<b>bot</b>	20
t <sub>6</sub>	...	read(qtaP <sub>x</sub> , A)	20
t <sub>7</sub>	...	A = A - 20	20
t <sub>8</sub>	...	write(qtaP <sub>x</sub> , A)	0
t <sub>9</sub>		write(qtaC <sub>XZ2</sub> , 30)	0
t <sub>10</sub>	...	<b>commit</b>	0
t <sub>11</sub>	read(qtaP <sub>x</sub> , A)		0
t <sub>12</sub>	...		0
t <sub>13</sub>	<b>commit</b>		0

# Letture inconsistenti (Inconsistent read)

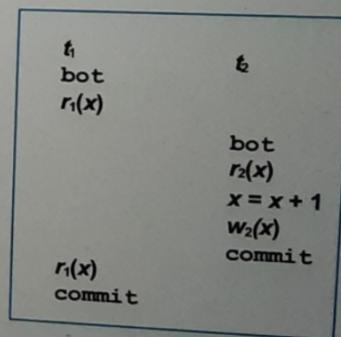
- Supponiamo che t1 svolga due operazioni di lettura in istanti successivi, e di avere t2 che incrementa un oggetto X di 1:

- t<sub>1</sub> : r<sub>1</sub>(x), r<sub>1</sub>(x)
- t<sub>2</sub> : r<sub>2</sub>(x), x = x + 1, w<sub>2</sub>(x)

- Inizialmente x=2.

- Vediamo l'esecuzione concorrente

In questo caso t1 legge il valore 2 dopo la prima operazione di lettura e il valore 3 dopo la seconda operazione di lettura. Non va bene, una transazione che accede due volte alla BD deve trovare esattamente lo stesso valore per ogni dato letto (non deve risentire di altre transazioni)



# Aggiornamento fantasma(ghost update)

---

- L'aggiornamento fantasma (Ghost Update) si verifica quando una o più operazioni di aggiornamento di una transazione che riguardano due o più oggetti della base di dati il cui valore è correlato (ad esempio, causa presenza di un vincolo di integrità) non vengono "viste" correttamente da altre transazioni temporalmente sovrapposte
- Per queste ultime, è come se alcuni degli aggiornamenti non siamo mai stati effettuati sulla base di dati e per questo si definiscono "fantasmi".

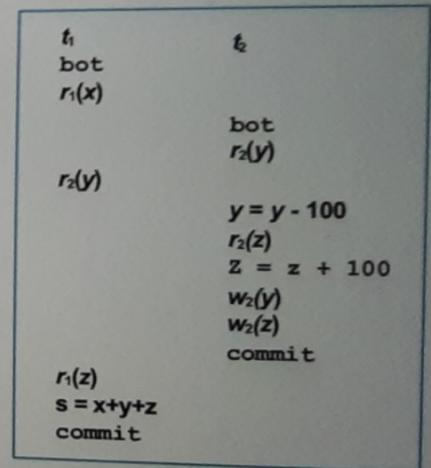
# I Aggiornamento fantasma(ghost update)

- Supponiamo di avere 3 oggetti x, y e z, che soddisfano il seguente vincolo di integrità:  $x+y+z = 1000$

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

- Vediamo l'esecuzione concorrente
- Es: X=500, Y=400, z=100.

T2 non altera la somma dei valori, non viola il vincolo di integrità. La variabile s della transazione t1, al termine dell'esecuzione contiene il valore 1100. T1 osserva solo in parte gli effetti di t2, osserva uno stato che non soddisfa i vincoli di integrità.



# Aggiornamento fantasma(ghost update)

Supponiamo due transazioni sovrapposte temporalmente, una relativa all'acquisto di un prodotto costituito da due componenti ( $x_1$  e  $x_2$  anch'esse prodotti) che vanno ordinati in coppia (ad esempio un kit con cuffie e microfono) e per cui esiste il vincolo che le quantità in magazzino delle due componenti sia uguale ( $qtaP_{x_1} = qtaP_{x_2}$ ).

La transazione T1 legge periodicamente le quantità disponibili per controllare, ad esempio, che coincidano i valori dei relativi oggetti della base di dati. Se T1 legge il valore della quantità disponibile di una delle due componenti prima dell'esecuzione della transazione d'acquisto e il valore dell'altra componente quando invece la transazione d'acquisto ha modificato il valore della prima componente ma non della seconda, allora il vincolo di integrità per la transazione di controllo risulterà violato (quando in realtà non lo è) ed è come se il secondo aggiornamento non fosse mai avvenuto.

Time	T <sub>1</sub>	T <sub>2</sub>	qtaP <sub>x<sub>1</sub></sub>	qtaP <sub>x<sub>2</sub></sub>
t <sub>1</sub>	<b>bot</b>		100	100
t <sub>2</sub>	read(qtaP <sub>x<sub>1</sub></sub> , A)	<b>bot</b>	100	100
t <sub>3</sub>	...	read(qtaP <sub>x<sub>1</sub></sub> , A)	100	100
t <sub>4</sub>	...	A = A - 30	100	100
t <sub>5</sub>	...	write(qtaP <sub>x<sub>1</sub></sub> , A)	70	100
t <sub>6</sub>	...	write(qtaC <sub>x<sub>1</sub>Z</sub> , 30)	70	100
t <sub>7</sub>	...	read(qtaP <sub>x<sub>2</sub></sub> , B)	70	100
t <sub>8</sub>	...	B = B - 30	70	100
t <sub>9</sub>		write(qtaP <sub>x<sub>2</sub></sub> , B)	70	70
t <sub>10</sub>	...	write(qtaC <sub>x<sub>2</sub>Z</sub> , 30)	70	70
t <sub>11</sub>	...	<b>commit</b>	70	70
t <sub>12</sub>	read(qtaP <sub>x<sub>2</sub></sub> , B)		70	70
t <sub>13</sub>	qtaP <sub>x<sub>1</sub></sub> ≠ qtaP <sub>x<sub>2</sub></sub>		70	70
t <sub>14</sub>	<b>commit</b>		70	70

# Inserimento fantasma(phantom)

- Supponiamo di avere una transazione che calcola due volte la media degli stipendi degli impiegati di un Dipartimento A, e di avere una seconda transazione che inserisce un nuovo impiegato nel Dipartimento A. Supponiamo che le due medie vengono calcolate una prima dell'inserimento e una dopo

Questa anomalia non può essere evitata facendo riferimento ai soli dati già presenti nella BD, è necessario tenere presente che vi è una nuova tupla che compare come uno **spettro**.



# Sintesi anomalie

- Perdita di aggiornamento: W-W *2 scrittura sequenziali*
- Lettura sporca: R-W (oppure W-W) con abort *\**  
*↳ sembra commit ma abort (come nell'esempio)*
- Letture inconsistenti: R-W
- Aggiornamento fantasma: R-W
- Inserimento fantasma: R-W su dato «nuovo»

*\* Devo evitare rollback a W-W*

# Gestione della concorrenza in SQL e in JDBC

- In SQL e in JDBC è possibile indicare il livello di isolamento per ciascuna transazione, scegliendo tra quattro possibili alternative:
- Read uncommitted**: permette le anomalie di lettura sporca, lettura inconsistente, aggiornamento fantasma, inserimento fantasma.
- Read committed**: evita la lettura sporca, ma non la lettura inconsistente, l'aggiornamento fantasma e l'inserimento fantasma
- Repeatable read**: evita tutte le anomalie, escluso l'inserimento fantasma
- Serializable**: evita tutte le anomalie

# Gestione della concorrenza in SQL e in JDBC

- In SQL e in JDBC è possibile indicare il livello di isolamento per ciascuna transazione, scegliendo tra quattro possibili alternative:
- **Read uncommitted:** la transazione accetta di leggere dati modificati da una transazione che non ha ancora fatto il commit (permette le anomalie di lettura sporca, lettura inconsistente, aggiornamento fantasma, inserimento fantasma).
- **Read committed:** la transazione accetta di leggere dati modificati da una transazione solo se questa ha fatto il commit. Se un dato viene letto due volte può trovare valori diversi (evita la lettura sporca, ma non la lettura inconsistente, l'aggiornamento fantasma e l'inserimento fantasma)

Permette però la modifica

AA 2020-2021, SISTEMI WEB E BASI DI DATI, PROF. DIOMAIUTA CRESCENZO

Permette l'aggiornamento?

# Gestione della concorrenza in SQL e in JDBC

- **Repeatable read:** la transazione accetta di leggere dati modificati da una transazione solo se questa ha fatto il commit, inoltre se un dato è letto due volte si avrà lo stesso risultato (evita tutte le anomalie, escluso l'inserimento fantasma)
- **Serializable:** produce schedule serializzabili (evita tutte le anomalie)

AA 2020-2021, SISTEMI WEB E BASI DI DATI, PROF. DIOMAIUTA CRESCENZO

Se ho anomalie, mando le transazioni, che devono essere ripetute.

# Teoria del controllo di concorrenza

- In questo ambito una transazione è definita **come una sequenza temporale di azioni di lettura (read) e scrittura (write) su oggetti della base dati**
- Una transazione è un oggetto sintattico di cui si conoscono soltanto le operazioni di ingresso/uscita
- $T_1: r_1(X) w_1(X) r_1(Y) r_1(Z) w_1(Z)$
- $T_2: r_2(X) r_2(Y) r_2(Z) w_2(Y)$
- Le transazioni sono eseguite in modo concorrente per cui le operazioni sono **"MISCHIATE"** (senza alterare l'ordine interno alla generica transazione)
- Uno **schedule** è **una sequenza di operazioni di lettura/scrittura generate da un insieme di transazioni concorrenti che preserva l'ordine temporale di esecuzione delle operazioni**

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

(il pedice indica il numero di transazione)

AA 2020-2021, SISTEMI WEB E BASI DI DATI, PROF. DIOMAIUTA CRESCENZO

# Teoria del controllo di concorrenza

- Il **controllore della concorrenza (scheduler)** è un **sistema** che accetta o rifiuta o riordina le operazioni richieste dalle transazioni (quegli schedule "pericolosi" che possono dare luogo ad anomalie)
- Esempi di possibili Schedule per le transazioni  $T_1$  e  $T_2$  sono:

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

Esempio di Schedule da slide Perdita di Aggiornamento:  
 $r1(x) r2(x) w2(x) w1(x)$

# Teoria del controllo di concorrenza

- Il controllore della concorrenza (**scheduler**) ha la funzione di accettare alcuni schedule e rifiutarne altri (quegli schedule "pericolosi" che possono dare luogo ad anomalie)
- La teoria del controllo di concorrenza si basa su tre ipotesi:
  - Nessuna transazione legga o scriva lo stesso dato più volte (eliminiamo lettura inconsistente)
  - **commit-proiezione** (ignoriamo le transazioni che vanno in **abort**, rimuovendo tutte le loro azioni dallo schedule, lasciando solo quelle con **commit**) (non so cosa succede non posso ripristinare che vanno tutte in **commit**) (elimina lettura sporca).
  - Sappiamo ogni transazione cosa farà

Sarà ovviamente necessario rimuovere queste ipotesi, in quanto lo scheduler deve decidere se accettare o meno, senza conoscere l'esito della transazione

# Teoria del controllo di concorrenza

- Uno schedule si definisce **seriale**, se le azioni di ciascuna transazione compaiono in sequenza, senza essere inframmezzate da istruzioni di altre transazioni

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

- Ovviamente, un schedule seriale non presenta anomalie
- Un schedule concorrente si dice **serializzabile** se produce lo stesso risultato di uno schedule seriale sulle stesse transazioni (richiede una nozione di equivalenza fra schedule)

Per come ragiona il DBMS non ragiona a livello seriale perché ruolano i processi.

# Equivalenza di scheduling

- Il **concetto di equivalenza è un concetto complesso**
- Equivalenza rispetto al risultato**
  - Due **schedule sono equivalenti se producono lo stesso risultato qualunque sia lo stato della base di dati.**
- Producono **lo stesso risultato se e sole se il valore iniziale di X è pari a 10, quindi non sono equivalenti**

T1	T2
Read(x)	Read(X)
X= X + 5	X=X*1.5
Write(X)	Write(X)

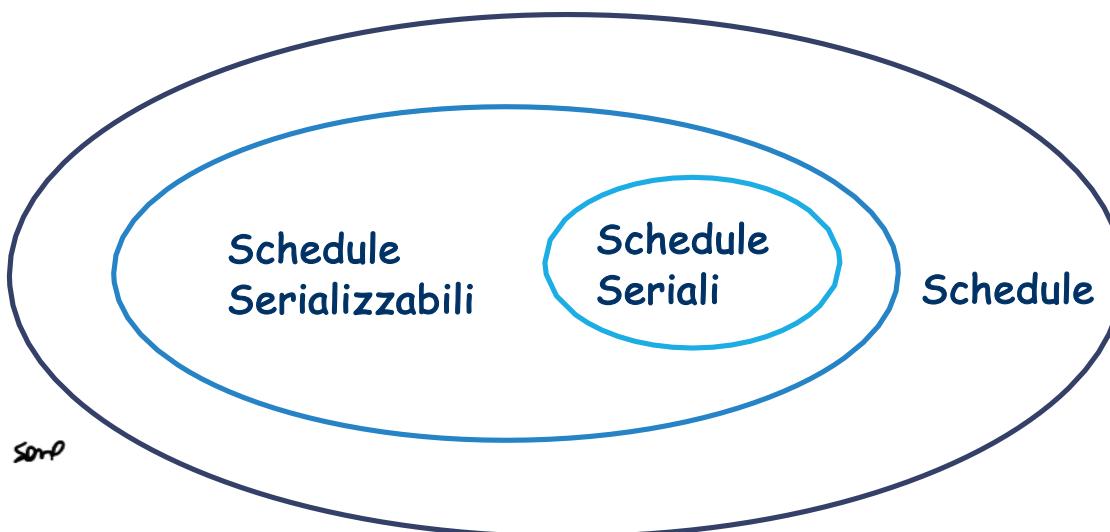
T1	T2
Read(x)	Read(X)
X= X + 5	X=X*1.5
Write(X)	Write(X)

# Idea base

- Individuare classi di schedule serializzabili
- Equivalenti a schedule seriali e per forza di cose privi di anomalie
- la proprietà di **serializzabilità** deve essere verificabile a costo basso

Voglio una schedule  
equivalente a quello  
serial, quindi si  
serializzabile.

Combinando tra loro  
le transazioni, solo alcuni sono  
serializzabili.



# View-equivalenza

- Relazione che lega coppie di operazioni di lettura e scrittura in uno schedule S:
  - $r_i(x)$  **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  $S$  ( $i \neq j$ )
  - $w_l(x)$  in uno schedule  $S$  è **scrittura finale** se è l'ultima scrittura dell'oggetto  $x$  in  $S$

## □ Esempio:

□ S:  $W_0(x)$   $R_2(x)$   $R_1(x)$   $W_2(x)$   $W_2(z)$

$R_2(x)$  legge da  $W_0(x)$   
 $R_1(x)$  legge da  $W_0(x)$   
 $W_2(x)$  scrittura finale  
 $W_2(z)$  scrittura finale

# View-equivalenza

---

- Due schedule  $S_1$  ed  $S_2$  sono view equivalenti ( $S_1 \approx_v S_2$ ) se e solo se:
  - Includono le stesse operazioni
  - Hanno le stesse relazioni legge-da
  - Hanno le stesse scritture finali
- Se  $S_1 \approx_v S_2$  allora si ha che  $S_1$  ed  $S_2$  producono lo stesso risultato indipendentemente dallo stato della BD
- Uno schedule  $S$  si dice **view-serializzabile** se è view-equivalente ad un qualche schedule seriale
- Indichiamo con **VSR** l'insieme degli schedule view-serializzabili

→ Non ci sono anomalie,

## View serializzabilità: Esempi

$$\begin{array}{l} S_3: w_0(x) \ r_2(x) \ r_1(x) \ w_2(x) \ w_2(z) \\ S_4: \mathbf{w_0(x) \ r_1(x) \ r_2(x) \ w_2(x) \ w_2(z)} \\ S_5: w_0(x) \ r_1(x) \ w_1(x) \ r_2(x) \ w_1(z) \\ S_6: \mathbf{w_0(x) \ r_1(x) \ w_1(x) \ w_1(z) \ r_2(x)} \end{array} \quad \left. \begin{array}{c} \\ \\ \end{array} \right\}$$

- $S_3$  è view-equivalente allo schedule seriale  $S_4$  (e quindi è view-serializzabile)
- $S_5$  non è view-equivalente ad  $S_4$ , ma è view-equivalente allo schedule seriale  $S_6$ , e quindi è view-serializzabile.  
*Due Schedule view-serializzabili non sono necessariamente view-equivalenti.*
- Notiamo che i seguenti schedule:
  - $S_7 : r_1(x) \ r_2(x) \ w_1(x) \ w_2(x)$  Perdita di aggiornamento
  - $S_8 : r_1(x) \ r_2(x) \ w_2(x) \ r_1(x)$  Letture inconsistenti
  - $S_9 : r_1(y) \ r_2(y) \ r_2(z) \ w_2(y) \ w_2(z) \ r_1(z)$  Aggiornamento fantasma
- $S_7, S_8, S_9$  non view-serializzabili

# View serializzabilità: In pratica ...

---

- Determinare la view-equivalenza di due schedule è un problema con complessità lineare (polinomiale).
- Decidere sulla view serializzabilità di uno schedule è un problema NP-completo (possono esistere un numero esponenziale di schedule seriali, considerando tutte le permutazioni delle transazioni)
- Non è utilizzabile in pratica      *Sempre hp ds mo abort*

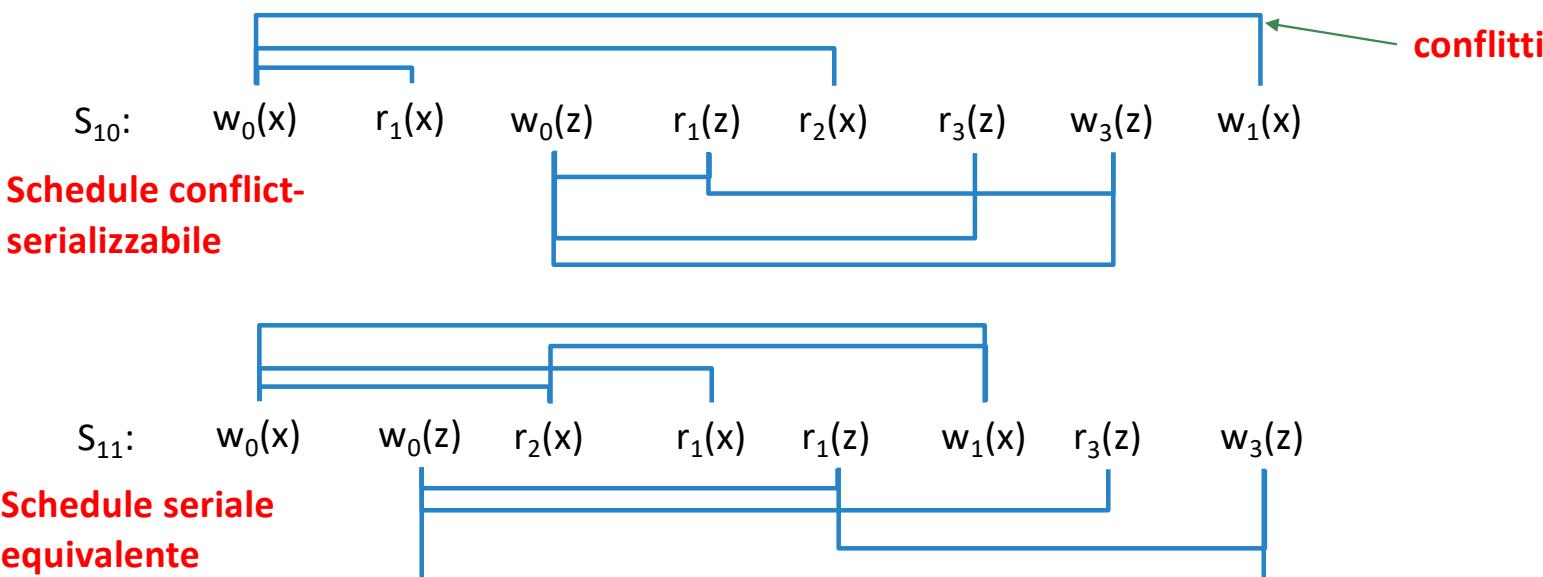
# Conflict-equivalenza

---

- Una nozione di equivalenza meno restrittiva si basa sulla definizione di **conflitto**:
- In uno **schedule S** due azioni  $a_i$  e  $a_j$  sono in **conflitto** se e solo se
  - $i \neq j$  (appartengono a transazioni diverse)
  - Operano sullo stesso oggetto (riguardano la stessa risorsa)
  - Almeno una delle due è una scrittura (lettura-scrittura, scrittura-lettura, scrittura-scrittura)
  - Esistono due casi:
    - Conflitto **read-write** (rw o wr):  $(r_1(x), w_2(x)) \neq (w_2(x), r_1(x))$
    - Conflitto **write-write** (ww):  $(w_1(x), w_2(x)) \neq (w_2(x), w_1(x))$
- Due schedule si dicono **conflict-equivalenti** ( $S_i \approx_C S_j$ ) se e solo includono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi.

# Conflict-equivalenza

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



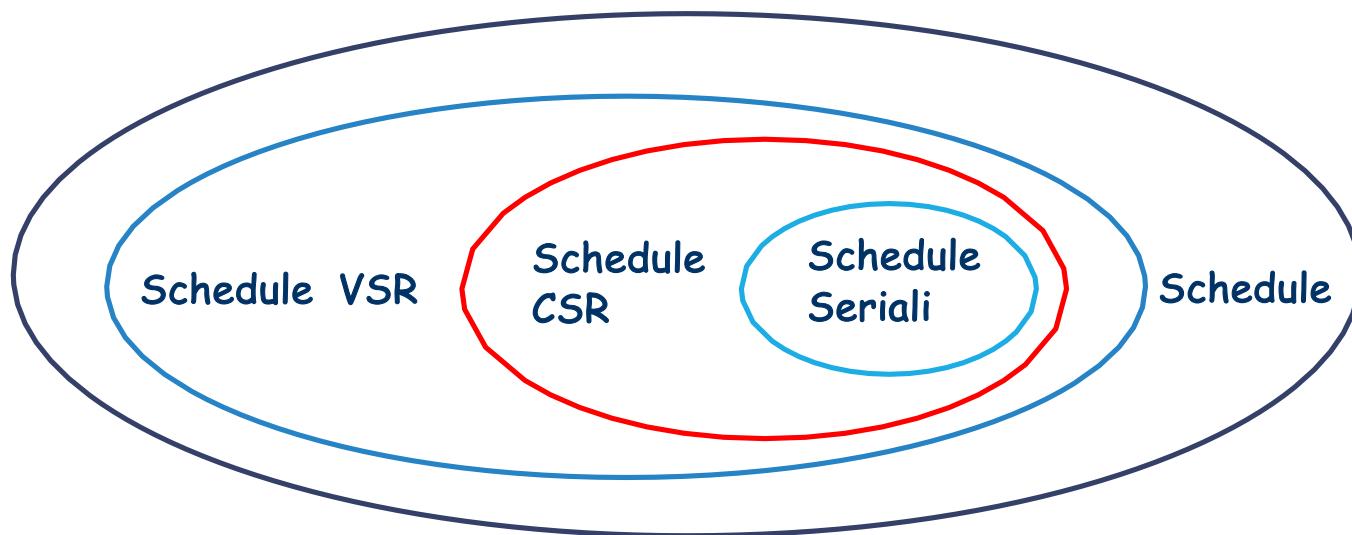
# Teorema CSR implica VSR

---

- Si può dimostrare che la classe degli schedule CSR è strettamente inclusa in quella degli schedule VSR, quindi la conflict-serializzabilità è condizione **sufficiente ma non necessaria** per la view-serializzabilità
- 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  $\approx_v$

# CSR e VSR

---



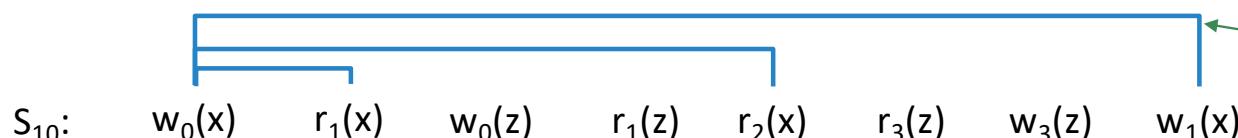
# Verifica di conflict - serializzabilità

- È possibile determinare se uno schedule è conflict-serializzabile tramite il **grafo dei conflitti**:

- un nodo per ogni transazione  $t_i$ ,

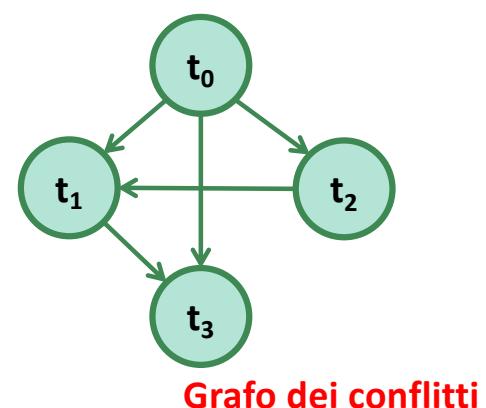
- un arco (orientato) da  $t_i$  a  $t_j$  se esiste almeno un conflitto tra un'azione  $a_i$  e un'azione  $a_j$  e si ha che  $a_i$  precede  $a_j$ .

Se ho conflitti tra  $t_0$  e  $t_3$ .



Schedule conflict-serializzabile

↑ arco orientato



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

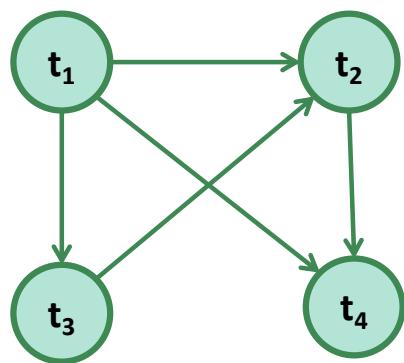
# Teorema

---

- Uno schedule è in CSR se e solo se il suo grafo dei conflitti è aciclico
- Se uno schedule  $S$  è CSR allora è  $\sim_C$  (conflict-equivalente) ad uno schedule seriale  $S_0$ . Supponiamo le transazioni nello schedule seriale **ordinate** secondo  $t_1, t_2, \dots, t_n$ . Poiché lo schedule seriale  $S_0$  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 in ogni conflitto  $(i,j)$  si ha sempre  $i < j$ .

# Esempio

$S = w1(x) r2(x) w1(z) r2(z) r3(x) r4(z) w4(z) w2(x)$



Il grafo è aciclico , quindi  $S$  è conflict-serializzabile

Pertanto lo schedule seriale  $w1(x) w1(z) r3(x) r2(x) r2(z) w2(x) r4(z) w4(z)$  è conflict-equivalente a  $S$ .

# Controllo della concorrenza in pratica

---

- Verificare l'aciclicità di un grafo 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

---

- I meccanismi di controllo della concorrenza utilizzati in partica, superano le limitazioni discusse finora. Tra questi il più utilizzato dai DBMS in commercio si basa sul **locking**.
- Dato
  - Un oggetto della base dati (l'intera base dati, una tabella, una riga, un attributo)
- Lock
  - Una variabile associata ad un dato che ne descrive lo stato con riferimento alle varie operazioni che ad esso possono essere applicate.
- Lock binari
  - Possiede solo due stati: locked/unlocked
- Restrittivo

Lock sono esclusivi.

Scrittura è mutualmente esclusiva

## Shared/Exclusive lock

- In particolare la tecnica realmente utilizzata sfrutta il seguente 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 upgrade)
- Il lock manager riceve queste richieste dalle transazioni e le accoglie o rifiuta, sulla base della tavola dei conflitti

Altre letture sono ammesse  
Poco sentire nra in determinato  
ntr da solo

} Transazione  
ben formata

# Gestione dei lock

- La gestione dei lock si basa sulla tavola dei conflitti

che  
può arrivare

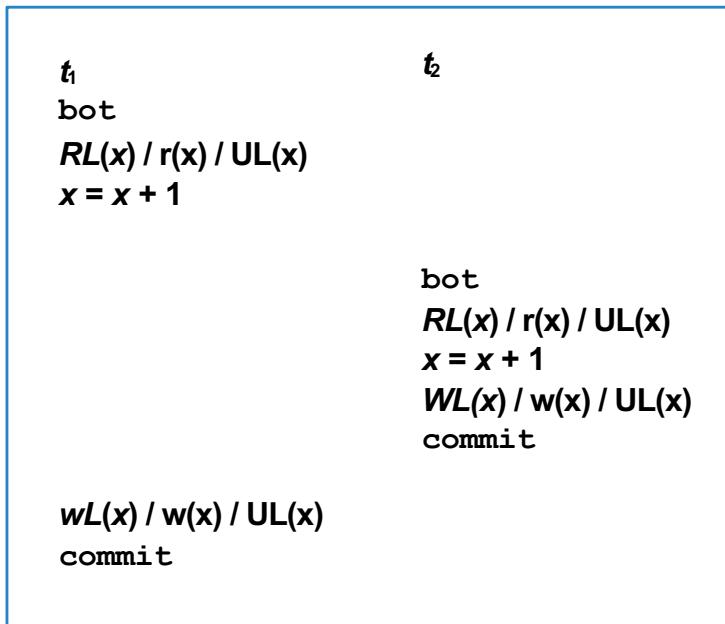
Richiesta	Stato risorsa		
	Libero	r_locked	w_locked
r_lock	OK / r_locked	OK / r_locked	NO / w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / dipende (se ci sono più lettori)	OK / libero

OK / r\_locked  
↑  
Stato assunto dalla  
risorsa dopo la  
primitiva  
  
Esito della richiesta

- 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

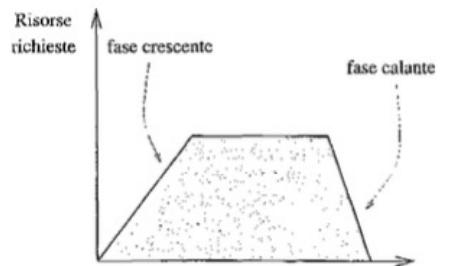
# Esempio: Perdita di aggiornamento

---



# Locking a due fasi (2PL)

- Lo scheduler diventa un lock manager che riceve richieste di lock e decide il da farsi.
- 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
- In una transazione esiste
  - La **fase crescente**: si acquisiscono i lock per le risorse cui si deve accedere
  - La **fase calante**: i lock acquisiti vengono rilasciati.
- È concesso l'incremento del livello di lock sulla risorsa (passaggio da `r_lock` a `w_lock`)
  - ↳ qui vale btp che non posso fare 2 lettine dello stesso oggetto



↳ acquisisci tutti i lock sulla variabile prima.  
Non posso rilasciare

# La classe 2PL

- ❑ In un sistema in cui le transazioni sono **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 dei conflitti
  - ❑ 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 ⊂ CSR: dimostrazione~~

- Ipotizziamo per assurdo che esista un schedule S, tale che  $S \in 2PL$  e  $S \notin CSR$
- $S \notin CSR$  **implica** che nel grafo dei conflitti esiste un ciclo  $T_1, T_2, \dots, T_N$
- Se esiste un conflitto tra  $T_1$  e  $T_2$  vuol dire che esiste una risorsa x su cui operano entrambe le transazioni in modo conflittuale
- Affinché  $T_2$  possa procedere è necessario che  $T_1$  rilascia il lock su x
- Se osserviamo il conflitto tra  $T_n$  e  $T_1$ , vuol dire che esiste una risorsa y su cui operano entrambe le transazioni in modo conflittuale
- Affinchè  $T_1$  possa procedere è necessario che  $T_n$  rilascia il lock che viene acquisito da  $T_1$
- Quindi  $T_1$  non può essere a due fasi: essa rilascia la risorsa x prima di acquisire la risorsa y**

# ~~2PL $\neq$ CSR~~

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

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

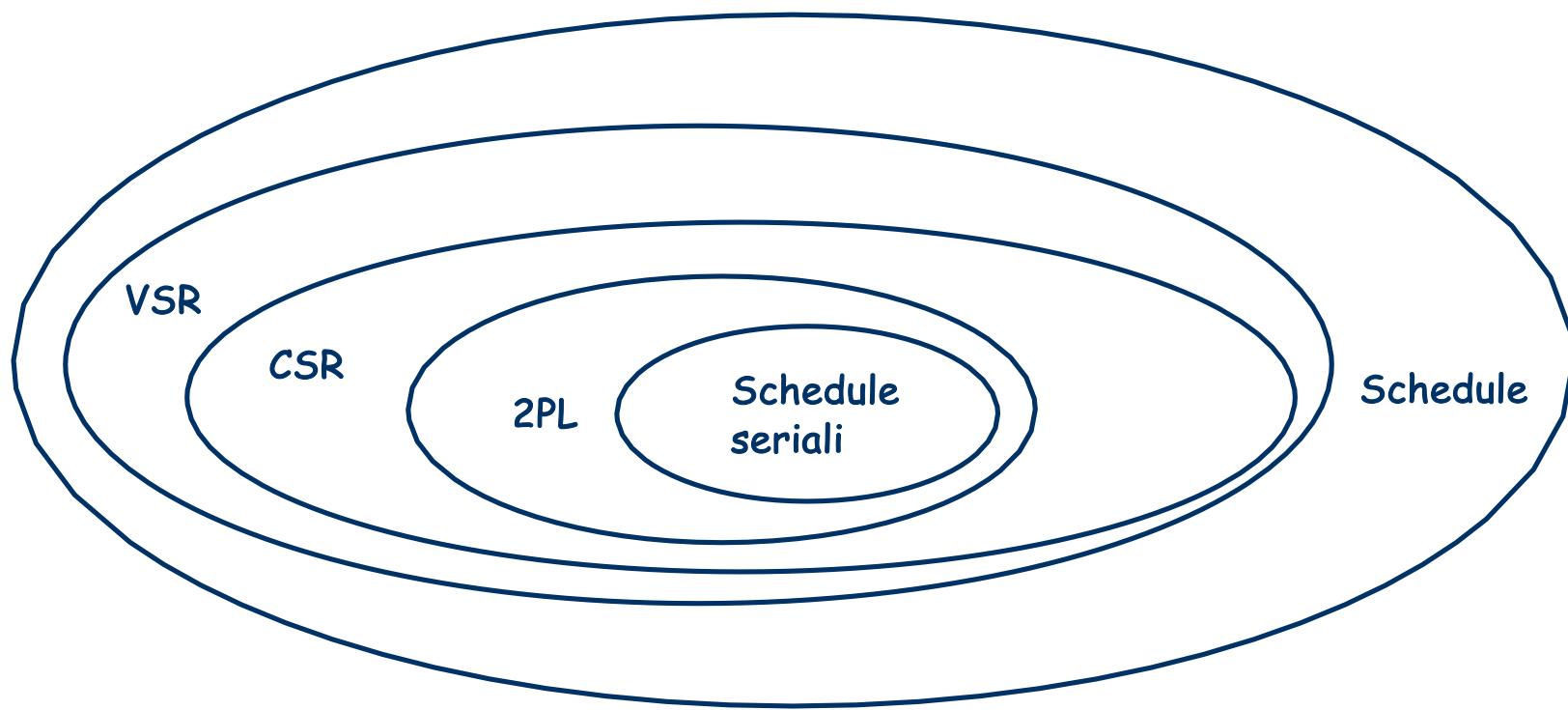
$t_1 \downarrow$   
 $t_2 \uparrow$   
 $t_3$

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

- Viola 2PL: non è a due fasi poiché  $t_1$  deve cedere un w-lock sulla risorsa x e poi richiederne un altro sulla risorsa y
- Conflict-serializzabile rispetto alla sequenza  $t_3, t_1, t_2$

# CSR, VSR e 2PL

---



# Prevenzione di un aggiornamento fantasma tramite il locking a due fasi

T1	T2	X	Y	z
		Free	Free	Free
R_lock1(x)			1:read	
R1(x)				
	W_lock2(y)		2:write	
	R2(y)			
R_lock1(y)			1:wait	
	Y = y-100			
	W_lock2(z)			2:write
	R2(z)			
	Z=z+100			
	W2(y)			
	W2(z)			
	Commit			
	Unlock2(y)		1:read	
R1(y)				
R_lock1(z)				1:wait
	Unlock2(z)			1:read
R1(z)				
S=x+y+z				
Commit				
Unlock1(x)		free		
Unlock1(y)			free	
Unlock1(z)				free

→ qui so che ho devo sconfiggere. Così risolve l'aggiornamento fantasma

→ solo post commit  
faccio unlock

STESO E SERCIZIO PER  
LE ALTRE ANOMALIE

# Effetto domino e dirty read

---

- lettura sporche:**
  - una transazione non può andare in commit finché non sono andate in commit tutte le transazioni da cui ha letto; *stanno solo dopo commit. Se prima T2 va in abort, T1 era niente.*
  - 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

# 2PL stretto

---

- Il problema è legato all'ipotesi di Commit Proiezione che è alla base della teoria della concorrenza.
- Nei sistemi reali tale ipotesi deve, per forza di cose, essere abbandonata.
- Le transazioni possono fallire:
  - rollback a cascata ("effetto domino"):
    - se Ti ha letto un dato scritto da Tk e Tk fallisce, allora anche Ti deve fallire.
  - letture sporche:
    - se Ti ha letto un dato scritto da Tk e Tk fallisce, ma nel frattempo Ti è andata in commit, allora abbiamo l'anomalia
- 2PL stretto:**
  - al 2PL si aggiunge l'ulteriore condizione che le risorse (i lock) possono essere rilasciate solo dopo il commit o abort.

# Controllo di concorrenza basato su timestamp

- Vediamo infine un **altro metodo di controllo della concorrenza** (alternativo a 2PL) che utilizza **i timestamp**
- Un **timestamp**: è un' **identificatore associato ad ogni evento temporale che definisce un ordinamento totale sugli eventi di un sistema** (*whilst in our world*)
  - Identificatore numerico assegnato dal DBMS ad ogni transazione:  $TS(T1) < TS(T2) \Rightarrow T1$  è iniziata prima di  $T2$
  - 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 con timestamp superiore.**

**Una transazione non può scrivere un oggetto letto o scritto da una transazione con timestamp superiore**

# Algoritmo TS monoversione

- Ad ogni oggetto  $X$  del DB vengono associati due contatori  $RTM(x)$  e  $WTM(x)$  che sono i timestamp della transazione che ha eseguito l'ultima scrittura e della transazione con  $t$  più grande che ha letto  $x$
- Lo scheduler riceve richieste di letture  $r_t(x)$  e scritture  $w_t(x)$  (dove  $t$  rappresenta il timestamp della transazione che esegue la lettura o la scrittura):
- $r_t(x)$ :
  - se  $t < WTM(x)$  allora la richiesta è respinta e la transazione viene uccisa;
  - altrimenti, la richiesta viene accolta e  $RTM(x)$  è posto pari al massimo fra  $RTM(x)$  e  $t$
- $w_t(x)$ :
  - se  $t < WTM(x)$  o  $t < RTM(x)$  allora la richiesta è respinta e la transazione viene uccisa,
  - altrimenti, la richiesta è accettata e  $WTM(x)$  è posto uguale a  $t$
- Vengono uccise molte transazioni
- Funziona sotto l' ipotesi di commit-proiezione.**

Transazione deve leggere solo corrette al suo timestamp; quindi se cambio valore dopo deve comunque leggere valore vecchio

# Algoritmo TS monoversione stretto

---

- Per eliminare il vincolo della commit-proiezione occorre:
  - “bufferizzare” le scritture effettuandole in memoria;
  - trascriverle solo dopo il commit
- Inoltre:
  - le letture di dati bufferizzati da parte di altre transazioni mettono anche queste ultime in “attesa” del commit della transazione scrivente.
- Vengono in tal modo introdotti meccanismi di “wait” simili a quelli del 2PL

# Algoritmo TS monoversione: Esempio

Richiesta	Risposta	VALORI INIZIALI:	RTM(x) = 7	WTM(x) = 5
		NUOVI VALORI		
$r_6(x)$	ok			
$r_7(x)$	ok			
$r_9(x)$	ok			RTM(x) = 9
$w_8(x)$	no			$T_8$ uccisa
$w_{11}(x)$	Ok			WTM(x) = 11
$r_{10}(x)$	no			$T_{10}$ uccisa

Una transazione non può scrivere un oggetto letto o scritto da una transazione più giovane, cioè con timestamp superiore.  
Una transazione non può leggere un oggetto scritto da una transazione più giovane, cioè con timestamp superiore.

# Algoritmo TS multiversione

- Per ogni transazione che modifica la base dati vengono mantenute diverse copie degli oggetti modificati.
- Ogni volta che una transazione scrive un oggetto  $x$ , la vecchia copia non viene persa, e viene creata una nuova copia  $WTM_N(x)$
- La copia  $RTM(x)$  rimane unica e globale, *sempre dall'ultima transazione che ha letto*
- Le copie sono rilasciate quando non vi sono più transazioni che devono leggere il loro valore.
- Le nuove regole:
  - $r_t(x)$ 
    - la lettura è sempre accettata, si legge un  $x_k$  siffatto: se  $t > WTM_N(x)$ , allora  $k=N$ , altrimenti si prende  $i$  in modo che sia  $WTM_i(x) < t < WTM_{i+1}(x)$ ;
  - $w_t(x)$  *mo check WTM?*
    - Se  $t < RTM(x)$  si rifiuta la richiesta altrimenti si aggiunge una nuova versione dell'oggetto  $x$  ( $N$  cresce di uno) con  $WTM_N(x) = t$

L'utente: Siamo ho copie che rappresenta vero valore che andrò a leggere.

# Algoritmo TS multiversione: Esempio

---

Richiesta	Risposta	VALORI INIZIALI: RTM(x) = 7 WTM1(x) = 4 NUOVI VALORI
$r_6(x)$	ok	
$r_8(x)$	Ok	RTM(x) = 8
$r_9(x)$	ok	RTM(x) = 9
$w_8(x)$	no, $t_8$ uccisa	
$w_{11}(x)$	ok	WTM2(x) = 11
$r_{10}(x)$	ok	RTM(x)=10 legge da x1