

**Basi di Dati**  
**Architettura dei DBMS:**  
**gestione della concorrenza**

Corso B

# Esempi di transazioni

$T_1$
<b>read(A)</b>
$A := A - 50$
<b>write(A)</b>
<b>read(B)</b>
$B := B + 50$
<b>write(B)</b>

$T_2$
<b>read(A)</b>
$temp := 0,1 \cdot A$
$A := A - temp$
<b>write(A)</b>
<b>read(B)</b>
$B := B + temp$
<b>write(B)</b>

**Interpretazione:** operazioni bancarie che aggiornano saldi A e B

# Proprietà delle transazioni

I DBMS gestiscono le transazioni garantendo, per ogni transazione  $T_i$ , il soddisfacimento delle proprietà ACID

- Atomicità
- Consistenza
- Isolamento
- Durabilità

oltre al rispetto dei vincoli

# Proprietà di $T_1$ e $T_2$

Immaginiamo che tutti i vincoli della base di dati siano rispettati

Si vede immediatamente che le due transazioni non modificano la somma dei valori di A e B

Supponiamo ora che le due transazioni entrino in parallelo nel sistema e vengano eseguite in successione e che all'inizio A contenga il valore 1000 e B il valore 2000

# Esecuzione di $T_1T_2$

La prima transazione toglie da A il valore 50 e lo aggiunge a B, quindi alla fine avremo

- $A = 950$ ,  $B = 2050$  e  $A + B = 3000$

La seconda transazione toglie il 10% da A e lo aggiunge a B

- $A = 855$ ,  $B = 2145$  con  $A + B = 3000$

# Esecuzione di $T_2T_1$

La seconda transazione toglie il 10% da A e lo aggiunge a B

- $A = 900$ ,  $B = 2100$  con  $A + B = 3000$

La prima transazione toglie da A il valore 50 e lo aggiunge a B, quindi alla fine avremo

- $A = 850$ ,  $B = 2150$  con  $A + B = 3000$

# DBMS e isolamento

- Supponiamo che il DBMS riceva contemporaneamente le transazioni  $T_1$  e  $T_2$
- Le transazioni  $T_1$  e  $T_2$  devono godere della proprietà di isolamento
- Dal punto di vista del DBMS, l'esecuzione completa di  $T_1$  (isolata) seguita dall'esecuzione completa di  $T_2$  (sempre isolata) ha la proprietà di lasciare la base di dati in una situazione di consistenza
- Se le riceve contemporaneamente, dal punto di vista del DBMS la scelta di eseguire la sequenza  $T_1T_2$  oppure  $T_2T_1$  è **irrilevante**

# Sistema informativo e DBMS

- Se per il sistema informativo è importante l'ordine di esecuzione delle due transazioni, è suo compito mandarle al DBMS nell'ordine corretto
- Se però il sistema informativo manda le due transazioni in parallelo al DBMS, allora il DBMS può eseguirle nell'ordine che vuole



# Esecuzione in parallelo

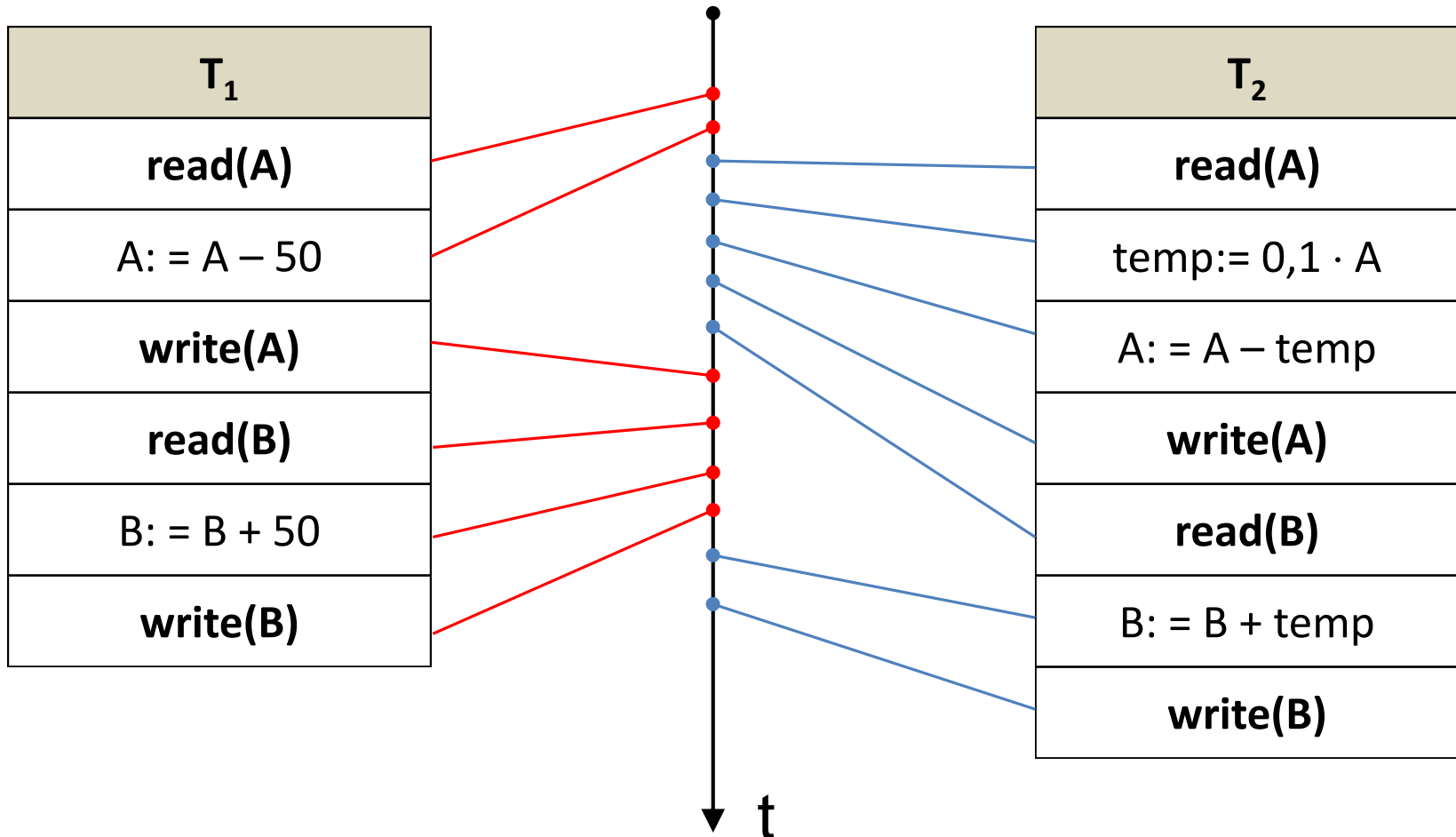
- L'esecuzione in sequenza delle transazioni è **inefficiente**
- Come sappiamo, gli accessi alla periferica sono molto costosi in termini di tempo
- Dal punto di vista del DBMS una esecuzione in sequenza lascia tempi morti di elaborazione molto lunghi per via dei frequenti accessi alle periferiche (read e write)
- Tutti i DBMS sono stati sviluppati in modo da mandare avanti in parallelo operazioni provenienti da transazioni diverse

# Problema

- L'esecuzione in parallelo di due transazioni non è scontata
- L'esecuzione in parallelo delle due transazioni, infatti, richiede di **interfogliare** le attività delle transazioni
- **Interfogliamento**: traduzione italiana del termine più noto **interleaving**

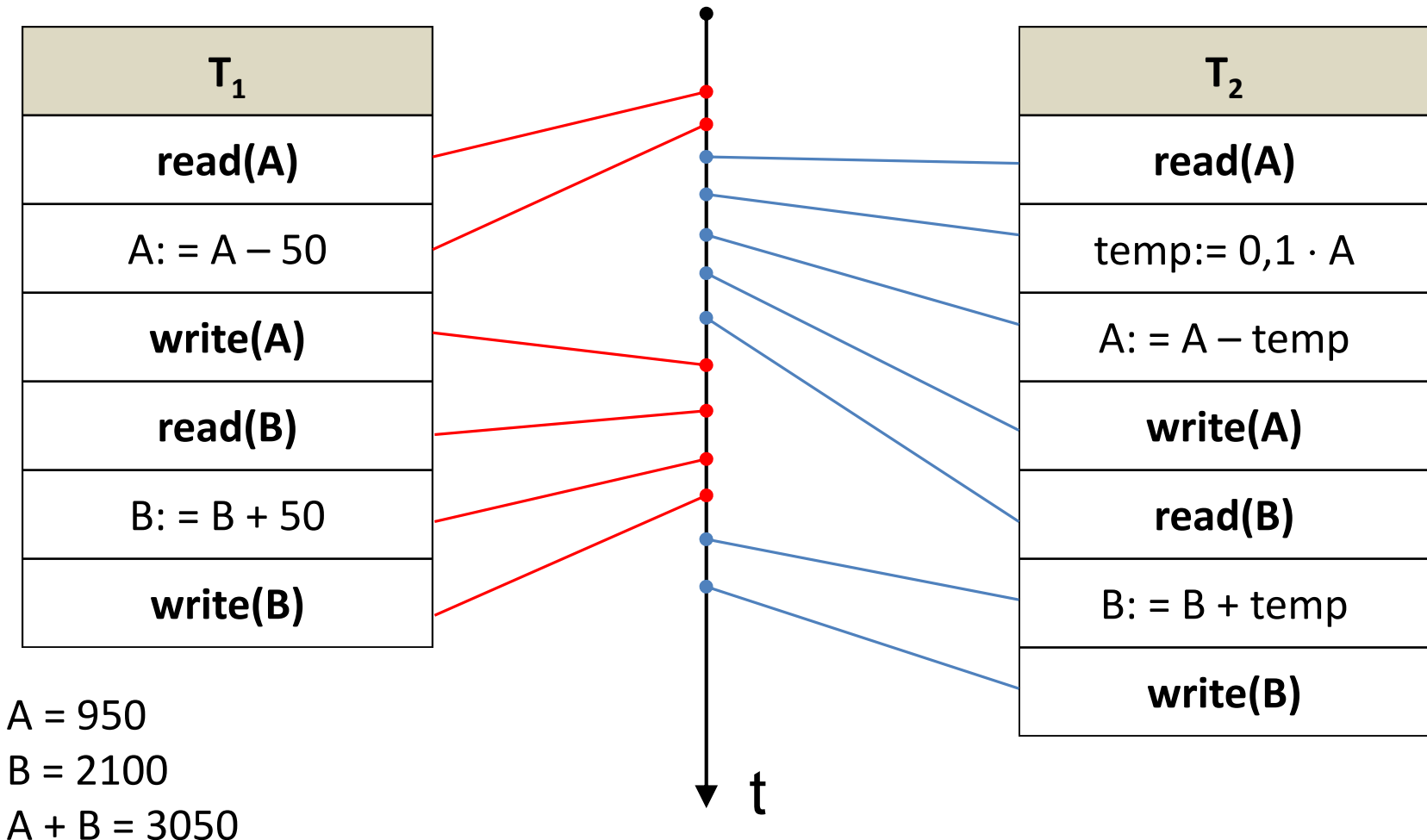
# Esempio

Supponiamo che il DBMS esegua il seguente interfogliamento



# Esempio

Supponiamo che il DBMS esegua il seguente interfogliamento



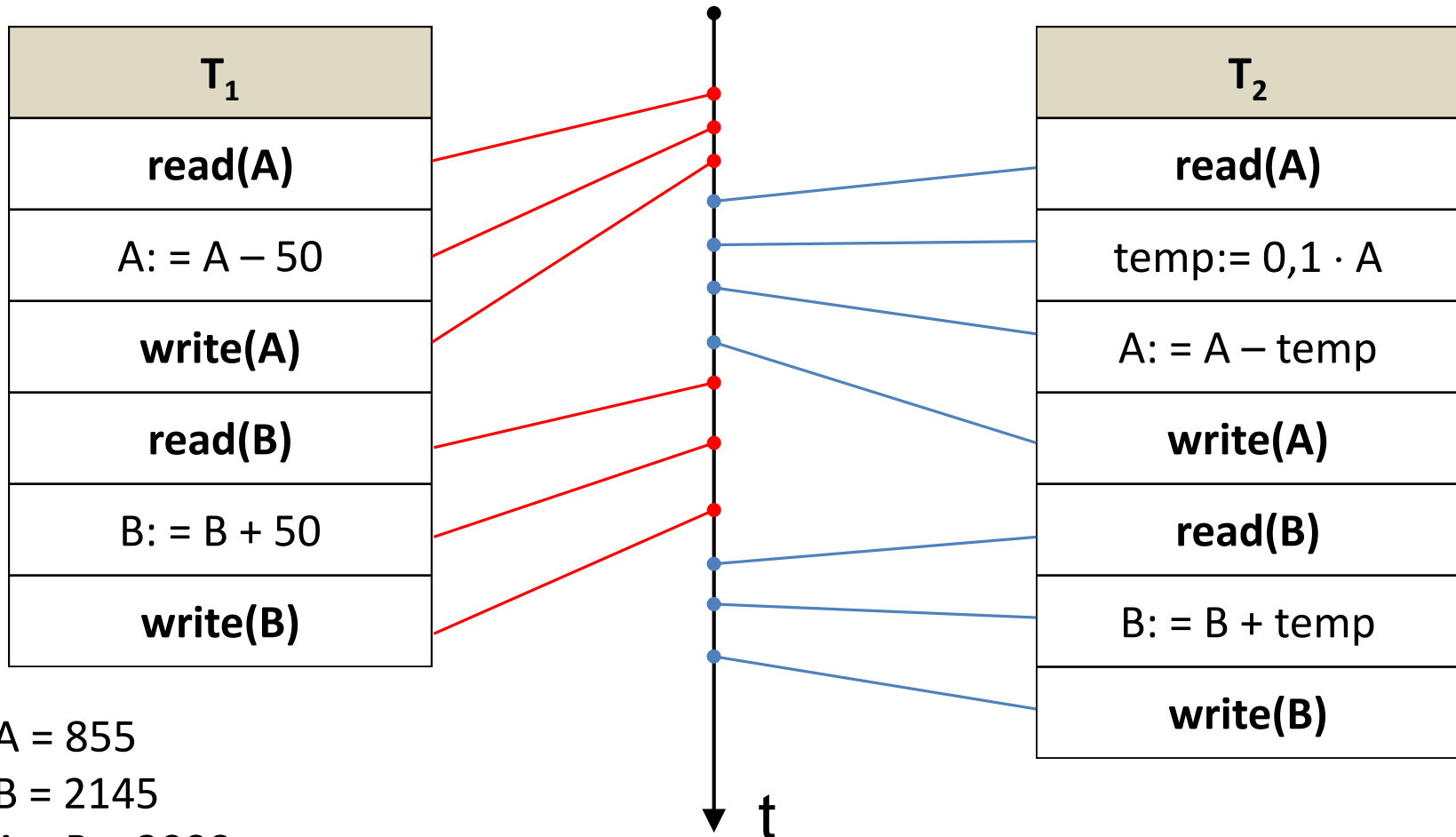
# Inconsistenza

L'inconsistenza è dovuta al fatto che l'interfogliamento ha interferito con le due transazioni

Non è detto che l'interfogliamento di due transazioni sia sempre nocivo, ma può lasciare la base di dati in uno stato consistente

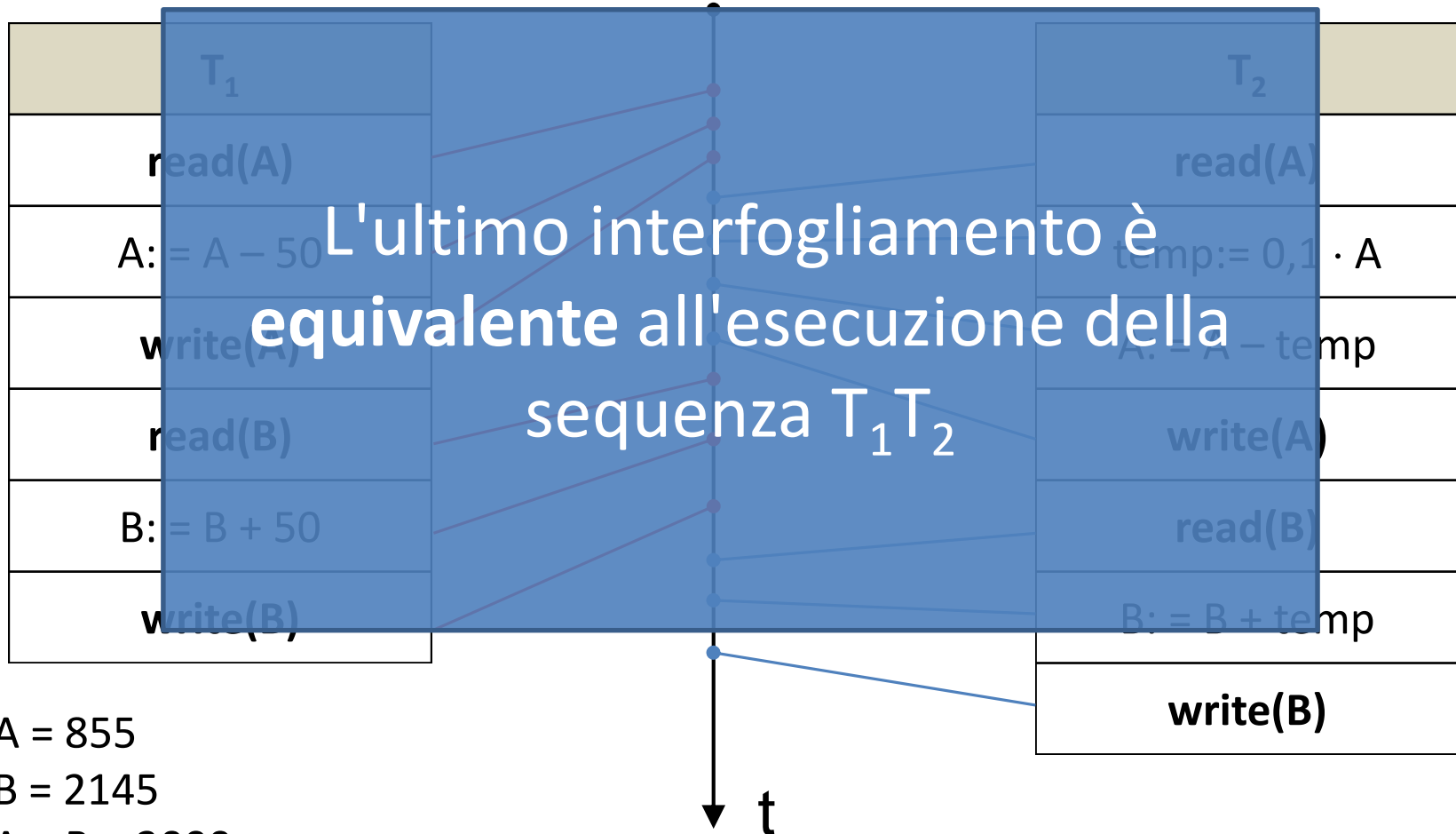
# Esempio

Supponiamo che il DBMS esegua il seguente interfogliamento



# Esempio

Supponiamo che il DBMS esegue il seguente interfogliamento



$A = 855$

$B = 2145$

$A + B = 3000$

# Inconsistenze

Sono state studiate le varie forme di inconsistenza che un interfogliamento può produrre in una base di dati

Chiamiamo:

- $r_i(X)$ : una read della variabile  $X$  da parte della transazione  $T_i$
- $w_j(Y)$ : una write della variabile  $Y$  da parte della transazione  $T_j$



# Lecture sporche

Immaginiamo il seguente interfogliamento di due transazioni  $T_1$  e  $T_2$

$w_1(X), r_2(X), \text{abort}(T_1), \text{update}_2(X), w_2(X)$

Dopo la write, la transazione  $T_1$  viene abortita, ma nel frattempo  $T_2$  aveva letto il valore di  $X$  modificato da  $T_1$ . Il valore letto da  $T_2$  è però inconsistente, in quanto prodotto da una transazione  $T_1$  abortita.

# Perdita di aggiornamento

Consideriamo queste due transazioni:

- $T_1: r_1(X), \text{update}_1(X), w_1(X)$
- $T_2: r_2(X), \text{update}_2(X), w_2(X)$

dove  $\text{update}_1$  e  $\text{update}_2$  eseguono entrambe  $X := X + 1$

Immaginiamo il seguente interfogliamento:

$r_1(X), r_2(X), \text{update}_2(X), w_2(X), \text{update}_1(X), w_1(X)$

Dopo l'esecuzione in sequenza di  $T_1T_2$  (o  $T_2T_1$ ),  $X$  conterrà  $X + 2$

Ma nell'esecuzione interfogliata si perde l'aggiornamento di  $T_2$

# Letture non ripetibili

Immaginiamo il seguente interfogliamento di due transazioni  $T_1$  e  $T_2$

$$r_1(X), w_2(X), r_1(X)$$

Nella seconda read, la transazione  $T_1$  suppone di leggere il medesimo valore letto durante la prima read, ma nel frattempo, tale valore è stato modificato dalla transazione  $T_2$ , la seconda lettura diventa quindi una lettura non ripetibile

# Aggiornamento fantasma

Immaginiamo un'invarianza  $X + Y = 1000$

<b>t</b>	<b>T<sub>1</sub></b>	<b>T<sub>2</sub></b>
1	<b>read(X)</b>	
2		<b>read(X)</b>
3		$X := X - 100$
4		<b>read(Y)</b>
5		$Y := Y + 100$
6		<b>write(X)</b>
7		<b>write(Y)</b>
8	<b>read(Y)</b>	
9	...	

# Aggiornamento fantasma

Le due transazioni hanno la proprietà di mantenere invariante la somma  $X + Y$

- $T_1$  non modifica né  $X$  né  $Y$
- $T_2$  sottrae 100 da  $X$  e lo aggiunge ad  $Y$

L'intefogliamento non rende inconsistente la base di dati (il vincolo  $X + Y = 1000$  è mantenuto)

Dal punto di vista della  $T_1$  però la base di dati non è consistente, infatti leggerà una somma  $X + Y = 1100$

Durante l'esecuzione di  $T_1$  avviene un **aggiornamento fantasma**

# Inserimento fantasma

L'inserimento fantasma si verifica sugli insiemi

- $T_1$  legge un insieme  $X_i$  di dati e calcola la cardinalità ( $\text{count}(*))$
- Durante l'esecuzione di  $T_1$ ,  $T_2$  inserisce un elemento in  $X_i$
- $T_1$  ricalcola la cardinalità dell'insieme  $X_i$

La cardinalità letta la seconda volta sarà diversa da quella letta la prima volta, in contrasto con la proprietà di isolamento

# Storie

L'esecuzione interfogliata delle transazione si chiama schedulazione o storia

Una **storia** è la sequenza di azioni eseguite dal DBMS a fronte delle richieste da parte delle transazioni

# Storie: esempi

Consideriamo le transazioni  $T_1$  e  $T_2$

$T_1$
<b>read(A)</b>
$A := A - 50$
<b>write(A)</b>
<b>read(B)</b>
$B := B + 50$
<b>write(B)</b>

$T_2$
<b>read(A)</b>
$\text{temp} := 0,1 \cdot A$
$A := A - \text{temp}$
<b>write(A)</b>
<b>read(B)</b>
$B := B + \text{temp}$
<b>write(B)</b>



# Storie: esempi

Il DBMS non vede gli aggiornamenti eseguiti all'interno delle transazioni, ma vede solo gli ordini di lettura e scrittura

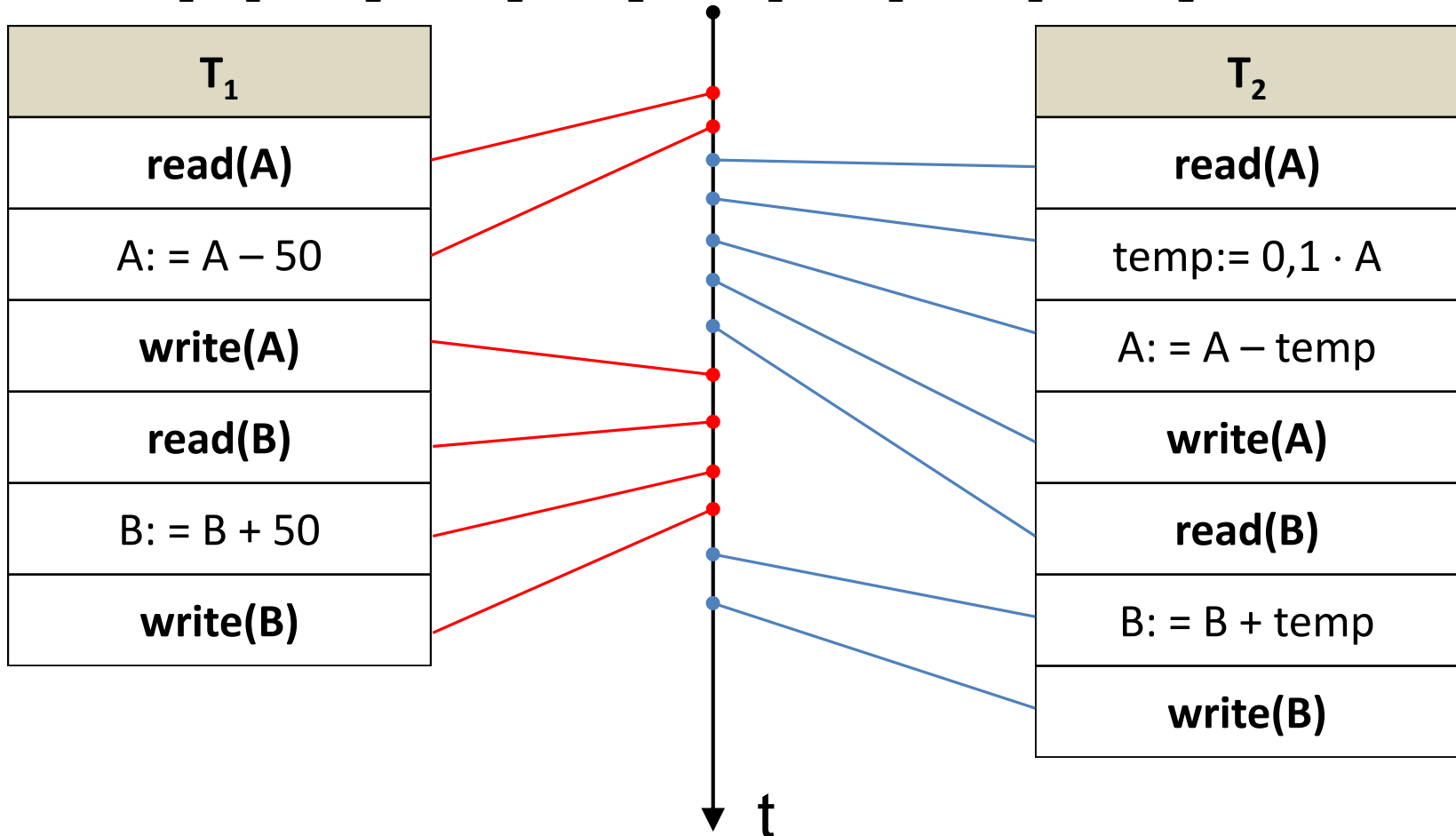
Descriveremo le storie  $T_1T_2$  e  $T_2T_1$  con questa notazione:

- Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$
- Storia( $T_2T_1$ ):  $r_2(A), w_2(A), r_2(B), w_2(B), r_1(A), w_1(A), r_1(B), w_1(B)$

Consideriamo ora le storie delle transazioni interfogliate

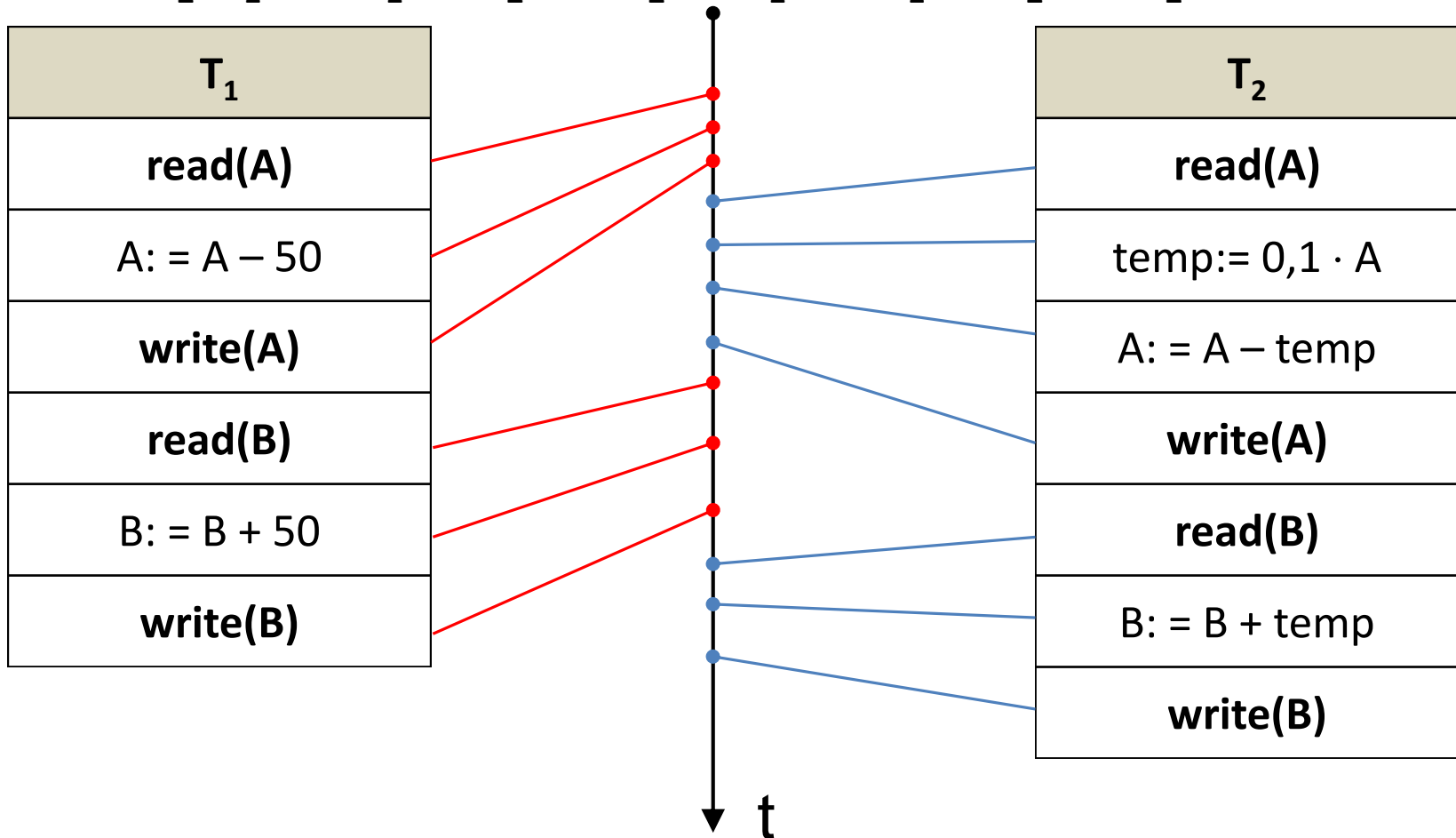
# Storie: esempi

Storia  $S_1$ :  $r_1(A), r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$



# Esempio

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$



# Gestione della concorrenza

La gestione della concorrenza è basata sull'analisi delle storie possibili che il DBMS può generare

# Semplificazioni

- Le transazioni leggono e scrivono un oggetto X solo una volta
  - Eviteremo quindi il problema delle riletture del medesimo dato escludendo dalla trattazione le letture non ripetibili e gli inserimenti fantasma
- Non si farà cenno alla possibilità che la transazione vada in abort
  - Considereremo sempre transazioni in commit

# Definizione di view-equivalenza

La storia  $S$  è **equivalente (view-equivalente)** alla storia  $S'$  ( $S \equiv S'$ ) se sono soddisfatte le seguenti condizioni:

1.  $S$  ed  $S'$  sono storie definite sullo stesso insieme di azioni (stesse azioni di read e write su stessi dati con gli stessi indici)
2. Condizioni sugli input
  - a. Se in  $S$  avviene una  $r_i(X)$  senza che  $X$  sia stato modificato, anche in  $S'$  deve avvenire lo stesso
  - b. Se in una storia  $S$  una  $T_j$  scrive l'oggetto  $X$  e successivamente  $T_i$  legge  $X$ , anche nella storia  $S'$  deve succedere la stessa cosa
    - $S: \dots w_j(X), \dots r_i(X), \dots$                        $S': \dots w_j(X), \dots r_i(X) \dots$
3. Condizione sullo stato: se in  $S$  una  $T_i$  scrive per ultima l'oggetto  $X$ , anche in  $S'$   $T_i$  deve scrivere per ultima l'oggetto  $X$

# Equivalenza tra storie

- La definizione di view-equivalenza ci dà un criterio rigoroso di confronto tra storie
- La transazione è vista come una funzione, ovvero, dati gli stessi input, la transazione deve generare gli stessi output
- Non è importante la posizione assoluta delle letture e scritture nelle storie, l'importante è che le letture delle transazioni avvengano sui medesimi "valori" (condizioni sull'input) e lo stato finale della base di dati prodotto da storie equivalenti sia il medesimo (condizione sullo stato della BD)

# Applicazione dell'equivalenza

- Sappiamo a priori che una storia di transazioni eseguita in successione è per definizione corretta perché la transazione gode della proprietà di consistenza, quindi se la transazione è applicata ad uno stato di BD consistente, la transazione lascerà la BD in uno stato consistente
- Se si esegue  $T_1$  su uno stato consistente e  $T_1$  lascia la BD in uno stato consistente,  $T_2$  opera su uno stato consistente e lascerà la BD in uno stato consistente (vale lo stesso per la sequenza  $T_2T_1$ )



# Criterio di serializzabilità

Se partiamo da storie corrette (le diverse esecuzioni seriali) e abbiamo una storia interfogliata, possiamo concludere che la storia interfogliata è corretta se è **view-equivalente** ad una qualsiasi storia seriale

Il **criterio di serializzabilità** dice quindi che una storia  $S$  è corretta se è **view-equivalente** ad una storia seriale **qualsiasi** delle transazioni coinvolte da  $S$

**N.B.:** date  $n$  transazioni esistono  $n!$  storie seriali

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

$S_2$  è view-equivalente alla storia  $T_1T_2$ :

- Le due storie condividono il medesimo insieme di azioni

# Esempio

Storia( $T_1T_2$ ):  $\mathbf{r_1(A)}, w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

Storia  $S_2$ :  $\mathbf{r_1(A)}, w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

$S_2$  è view-equivalente alla storia  $T_1T_2$ :

- Le due storie condividono il medesimo insieme di azioni
- $T_1$  legge A senza che A sia stato modificato prima

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), \mathbf{r_1(B)}, w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), \mathbf{r_1(B)}, w_1(B), r_2(B), w_2(B)$

$S_2$  è view-equivalente alla storia  $T_1T_2$ :

- Le due storie condividono il medesimo insieme di azioni
- $T_1$  legge A senza che A sia stato modificato prima
- $T_1$  legge B senza che B sia stato modificato prima

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), \mathbf{w_1(A)}, r_1(B), w_1(B), \mathbf{r_2(A)}, w_2(A), r_2(B), w_2(B)$

Storia  $S_2$ :  $r_1(A), \mathbf{w_1(A)}, \mathbf{r_2(A)}, w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

$S_2$  è view-equivalente alla storia  $T_1T_2$ :

- Le due storie condividono il medesimo insieme di azioni
- $T_1$  legge A senza che A sia stato modificato prima
- $T_1$  legge B senza che B sia stato modificato prima
- $T_2$  legge A dopo che  $T_1$  lo ha modificato

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), \mathbf{w_1(B)}, r_2(A), w_2(A), \mathbf{r_2(B)}, w_2(B)$

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), \mathbf{w_1(B)}, \mathbf{r_2(B)}, w_2(B)$

$S_2$  è view-equivalente alla storia  $T_1T_2$ :

- Le due storie condividono il medesimo insieme di azioni
- $T_1$  legge A senza che A sia stato modificato prima
- $T_1$  legge B senza che B sia stato modificato prima
- $T_2$  legge A dopo che  $T_1$  lo ha modificato
- $T_2$  legge B dopo che  $T_1$  lo ha modificato

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), \mathbf{w_2(A)}, r_2(B), w_2(B)$

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), \mathbf{w_2(A)}, r_1(B), w_1(B), r_2(B), w_2(B)$

$S_2$  è view-equivalente alla storia  $T_1T_2$ :

- Le due storie condividono il medesimo insieme di azioni
- $T_1$  legge A senza che A sia stato modificato prima
- $T_1$  legge B senza che B sia stato modificato prima
- $T_2$  legge A dopo che  $T_1$  lo ha modificato
- $T_2$  legge B dopo che  $T_1$  lo ha modificato
- $T_2$  è l'ultima a scrivere A

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), \mathbf{w_2(B)}$

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), \mathbf{w_2(B)}$

$S_2$  è view-equivalente alla storia  $T_1T_2$ :

- Le due storie condividono il medesimo insieme di azioni
- $T_1$  legge A senza che A sia stato modificato prima
- $T_1$  legge B senza che B sia stato modificato prima
- $T_2$  legge A dopo che  $T_1$  lo ha modificato
- $T_2$  legge B dopo che  $T_1$  lo ha modificato
- $T_2$  è l'ultima a scrivere A
- $T_2$  è l'ultima a scrivere B



# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

$S_2$  è view-equivalente alla storia  $T_1T_2$ :

- Le due storie condividono il medesimo insieme di azioni
- $T_1$  legge A senza che A sia stato modificato prima
- $T_1$  legge B senza che B sia stato modificato prima
- $T_2$  legge A dopo che  $T_1$  lo ha modificato
- $T_2$  legge B dopo che  $T_1$  lo ha modificato
- $T_2$  è l'ultima a scrivere A
- $T_2$  è l'ultima a scrivere B

La storia  $S_2$  è serializzabile

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

Storia( $T_2T_1$ ):  $r_2(A), w_2(A), r_2(B), w_2(B), r_1(A), w_1(A), r_1(B), w_1(B)$

Storia  $S_1$ :  $r_1(A), r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$

- Le due storie condividono il medesimo insieme di azioni

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), \mathbf{w_1(A)}, r_1(B), w_1(B), \mathbf{r_2(A)}, w_2(A), r_2(B), w_2(B)$

Storia( $T_2T_1$ ):  $r_2(A), w_2(A), r_2(B), w_2(B), r_1(A), w_1(A), r_1(B), w_1(B)$

Storia  $S_1$ :  $r_1(A), \mathbf{r_2(A)}, w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$

- Le due storie condividono il medesimo insieme di azioni
- $T_2$  legge A prima che  $T_1$  lo abbia modificato

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

Storia( $T_2T_1$ ):  $r_2(A), \mathbf{w_2(A)}, r_2(B), w_2(B), \mathbf{r_1(A)}, w_1(A), r_1(B), w_1(B)$

Storia  $S_1$ :  $\mathbf{r_1(A)}, r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$

- Le due storie condividono il medesimo insieme di azioni
- $T_2$  legge A prima che  $T_1$  lo abbia modificato
- $T_1$  legge A prima che  $T_2$  lo abbia modificato

$S_1$  non è equivalente né alla storia  $T_1T_2$ , né alla storia  $T_2T_1$

# Esempio

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

Storia( $T_2T_1$ ):  $r_2(A), w_2(A), r_2(B), w_2(B), r_1(A), w_1(A), r_1(B), w_1(B)$

Storia  $S_1$ :  $r_1(A), r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$

- Le due storie condividono il medesimo insieme di azioni
- $T_2$  legge A prima che  $T_1$  lo abbia modificato
- $T_1$  legge A prima che  $T_2$  lo abbia modificato

**La storia  $S_1$  non è serializzabile**

$S_1$  non è equivalente né alla storia  $T_1T_2$ , né alla storia  $T_2T_1$

# Complessità

- Mettere a confronto due storie ha complessità polinomiale
- L'utilizzo dell'equivalenza per verificare il criterio di serializzabilità richiede un numero  $n!$  di confronti

# Soluzione al problema

- Per poter risolvere il problema ci si è messi in un contesto meno generale della view-equivalenza
- In questo contesto sono importanti le cosiddette **azioni in conflitto** in una storia

# Azioni in conflitto in una storia

In una storia sono in conflitto le seguenti azioni

$S_1: \dots r_i(X), \dots, w_j(X), \dots$  (conflitto)

$S_2: \dots w_i(X), \dots, r_j(X), \dots$  (conflitto)

$S_3: \dots w_i(X), \dots, w_j(X), \dots$  (conflitto)

$S_4: \dots r_i(X), \dots, r_j(X), \dots$  (non c'è conflitto)

Le azioni in conflitto riguardano azioni provenienti da transazioni diverse sullo stesso oggetto



# Conflitto

Sono state chiamate azioni in conflitto perché cambiando l'ordine delle due operazioni può cambiare l'attività sulla base di dati o la risposta delle transazioni

$S_1: \dots r_i(X), \dots, w_j(X), \dots$

- se invertiamo,  $T_i$  legge il prodotto della write di  $T_j$  su  $X$

$S_2: \dots w_i(X), \dots, r_j(X), \dots$

- se invertiamo,  $T_j$  non legge più il prodotto della write di  $T_i$  su  $X$

$S_3: \dots w_i(X), \dots, w_j(X), \dots$

- se invertiamo, si può modificare lo stato finale della base di dati (se  $w_j(X)$  fosse stata l'ultima write, scambiando l'ordine diventa  $w_i(X)$  l'ultima write ad essere eseguita)

# Grafo dei conflitti

Data una storia  $S$ , il **grafo dei conflitti** di  $S$  è un grafo orientato che ha:

- **come nodi**: le transazioni  $T_i$  che compongono la storia
- **come archi** (di conflitto):
  1. considero tutte le letture  $r_i(X)$  della transazione sull'oggetto  $X$ , si ispeziona la storia da  $r_i(X)$  in avanti cercando una  $w_j(X)$ : se esiste, si traccia l'arco  $T_i \rightarrow T_j$
  2. Considero le  $w_i(X)$  e ispeziono la storia da  $w_i(X)$  in avanti: **ogni volta** che si trova una  $r_j(X)$  lungo il percorso, si traccia l'arco  $T_i \rightarrow T_j$ , fino a quando non si trova una  $w_h(X)$  e si costruisce l'arco  $T_i \rightarrow T_h$

**N.B.:** gli archi sono **unici** (non è un multigrafo)

# Esempio: storia $T_1T_2$

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

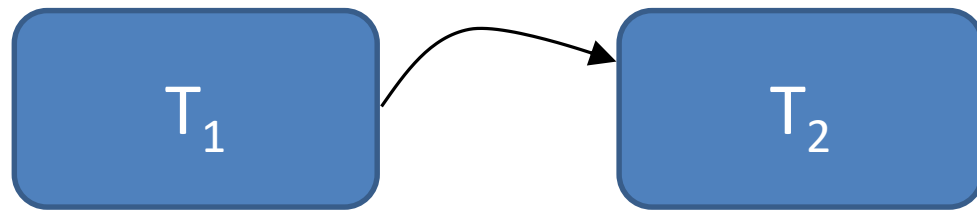


$T_1$

$T_2$

# Esempio: storia $T_1T_2$

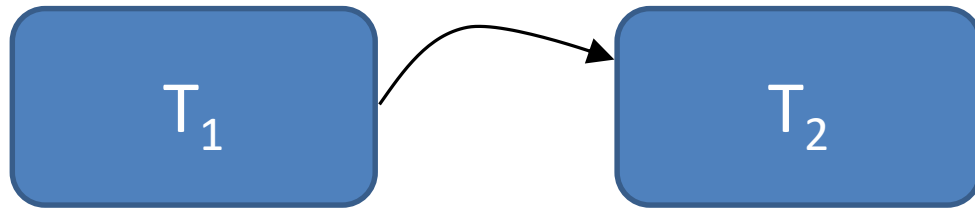
Storia( $T_1T_2$ ):  $r_1(\mathbf{A}), w_1(A), r_1(B), w_1(B), r_2(A), \mathbf{w_2(A)}, r_2(B), w_2(B)$



Arco di tipo 1

# Esempio: storia $T_1T_2$

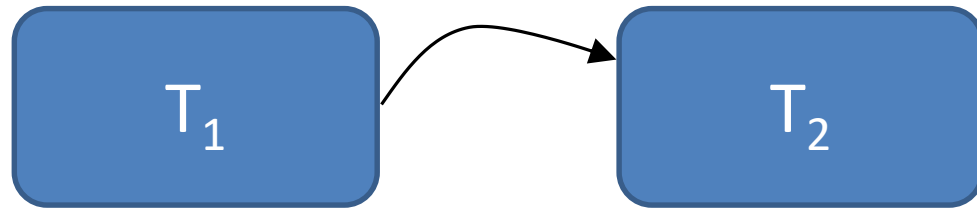
Storia( $T_1T_2$ ):  $r_1(A), \mathbf{w_1(A)}, r_1(B), w_1(B), \mathbf{r_2(A)}, w_2(A), r_2(B), w_2(B)$



Arco di tipo 2 (confermato)

# Esempio: storia $T_1T_2$

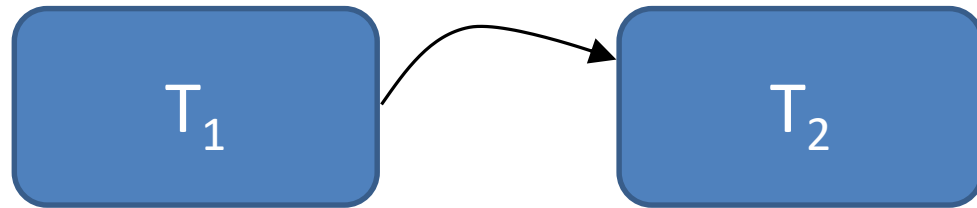
Storia( $T_1T_2$ ):  $r_1(A), \mathbf{w_1(A)}, r_1(B), w_1(B), r_2(\mathbf{A}), \mathbf{w_2(A)}, r_2(B), w_2(B)$



Arco di tipo 2 (confermato)

# Esempio: storia $T_1T_2$

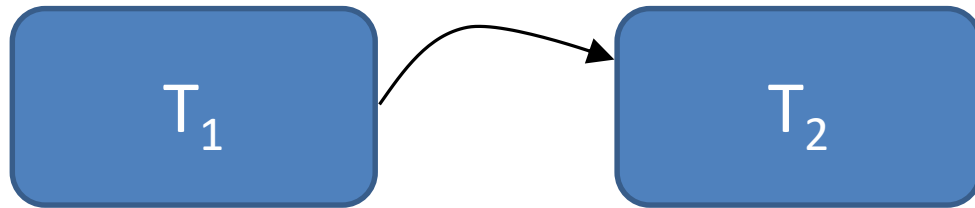
Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(\mathbf{B}), w_1(B), r_2(A), w_2(A), r_2(B), \mathbf{w_2(B)}$



Arco di tipo 1 (confermato)

# Esempio: storia $T_1T_2$

Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), \mathbf{w_1(B)}, r_2(A), w_2(A), r_2(\mathbf{B}), w_2(B)$

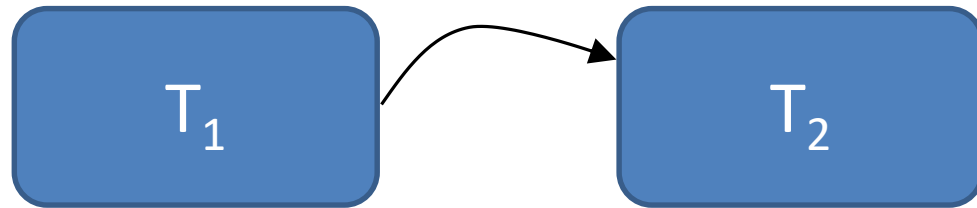


Arco di tipo 2 (confermato)



# Esempio: storia $T_1T_2$

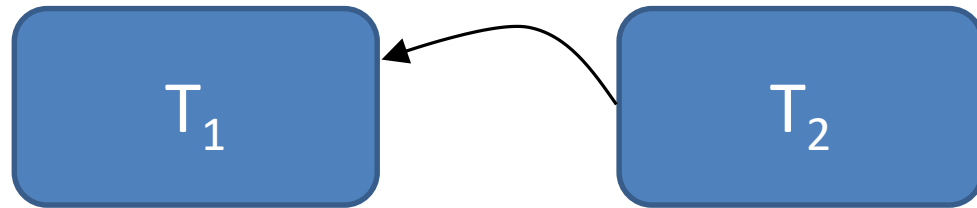
Storia( $T_1T_2$ ):  $r_1(A), w_1(A), r_1(B), \mathbf{w_1(B)}, r_2(A), w_2(A), r_2(B), \mathbf{w_2(B)}$



Arco di tipo 2 (confermato)

# Esempio: storia $T_2T_1$

Storia( $T_2T_1$ ):  $r_2(A), w_2(A), r_2(B), w_2(B), r_1(A), w_1(A), r_1(B), w_1(B)$



# Esempio: storia $S_1$

Storia  $S_1$ :  $r_1(A), r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$



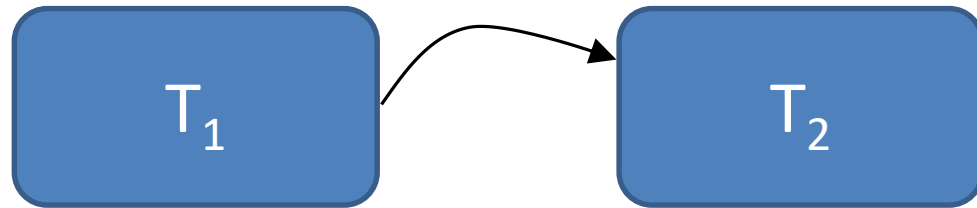
$T_1$



$T_2$

# Esempio: storia $S_1$

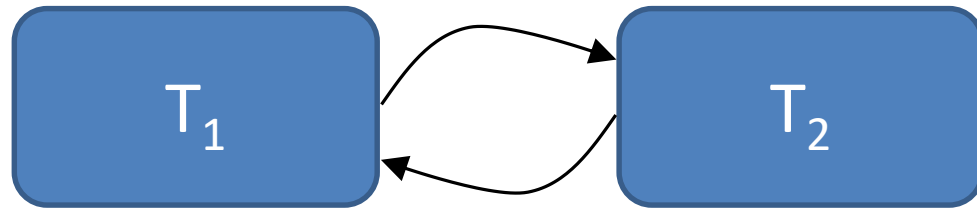
Storia  $S_1$ :  $r_1(\mathbf{A}), r_2(A), w_2(\mathbf{A}), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$



Arco di tipo 1

# Esempio: storia $S_1$

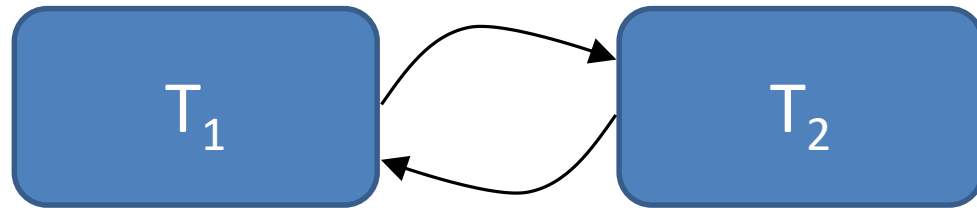
Storia  $S_1$ :  $r_1(A), r_2(\mathbf{A}), w_2(A), r_2(B), \mathbf{w}_1(\mathbf{A}), r_1(B), w_1(B), w_2(B)$



Arco di tipo 1

# Esempio: storia $S_1$

Storia  $S_1$ :  $r_1(A), r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$



Posso fermarmi, non esistono altri archi tra  $T_1$  e  $T_2$

# Esempio: storia $S_2$

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$



$T_1$

$T_2$

# Esempio: storia $S_2$

Storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

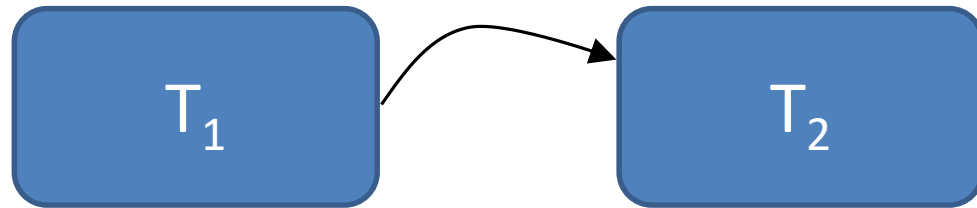


Grafico identico a quello della storia seriale  $T_1T_2$



# Esempio

$S_x: r_1(X), r_2(X), r_3(Z), w_3(Z), r_1(Y), r_3(X), w_3(X)$



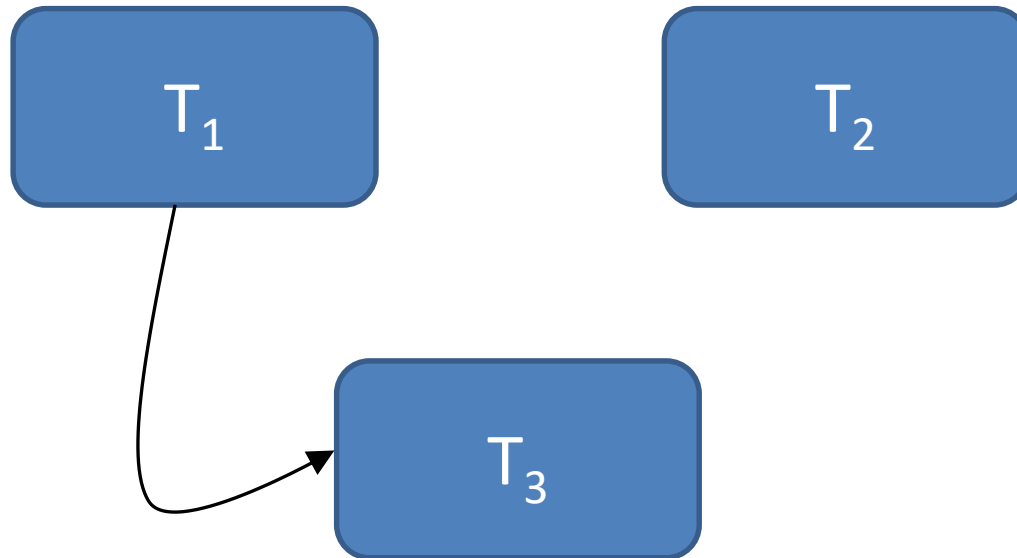
$T_1$

$T_2$

$T_3$

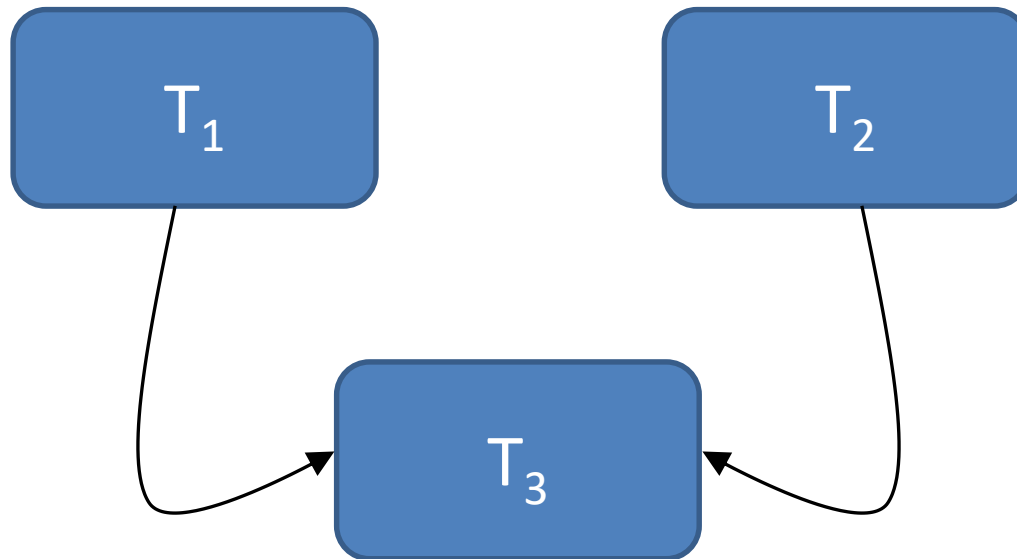
# Esempio

$S_x: \mathbf{r}_1(\mathbf{X}), r_2(X), r_3(Z), w_3(Z), r_1(Y), r_3(X), \mathbf{w}_3(\mathbf{X})$



# Esempio

$S_x: r_1(X), r_2(\mathbf{X}), r_3(Z), w_3(Z), r_1(Y), r_3(X), \mathbf{w}_3(\mathbf{X})$



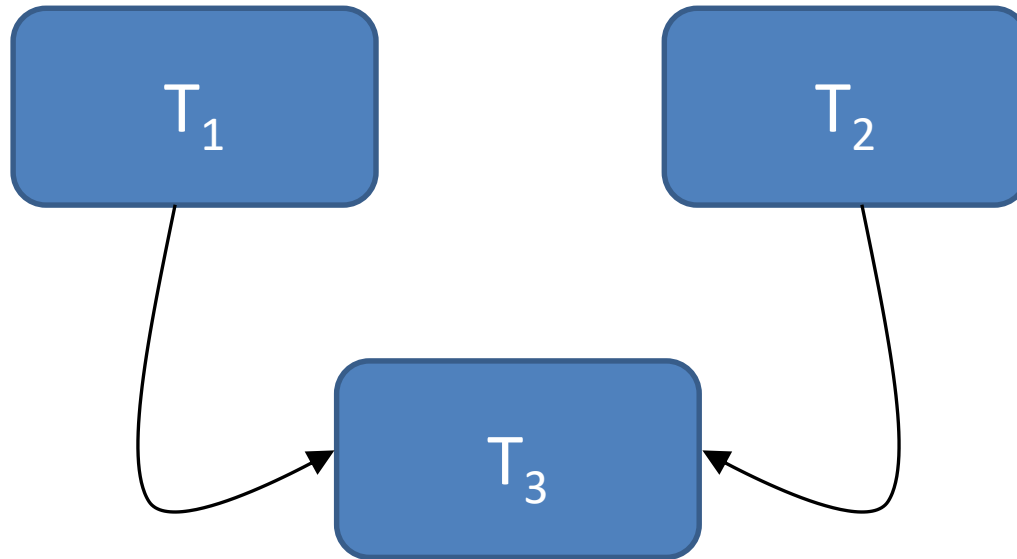
# Test di serializzabilità

Condizione **sufficiente** di serializzabilità di una storia  $S$  è che il grafo dei conflitti di  $S$  sia aciclico

(la condizione è solo sufficiente, esistono quindi storie serializzabili che presentano grafi con cicli)

# Esempio: storia $S_x$

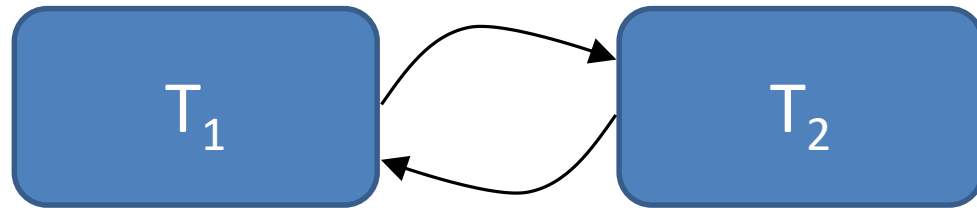
$S_x: r_1(X), r_2(X), r_3(Z), w_3(Z), r_1(Y), r_3(X), w_3(X)$



Il grafo è aciclico, quindi è equivalente ad una o più storie seriali di  $T_1$ ,  $T_2$  e  $T_3$

# Esempio: storia $S_1$

Storia  $S_1$ :  $r_1(A), r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$



La storia è **potenzialmente** non serializzabile (in realtà sappiamo che  $S_1$  non lo è per nulla)

# Dimostrazione

La dimostrazione si fa per induzione sul numero di transazioni

Se abbiamo una sola transazione  $T_1$ , il test di serializzabilità è banale, quindi una storia composta da azioni provenienti da una sola transazione è banalmente serializzata

Ipotizziamo che il test di serializzabilità sia vero per il grafo con  $n - 1$  transazioni, aggiungiamo una transazione e dimostriamo che il test è vero anche per il grafo con  $n$  transazioni

Dobbiamo quindi dimostrare il **passo di induzione**

# Passo di induzione

Per ipotesi il grafo della storia è aciclico

Una proprietà del grafo aciclico è che esso ammette almeno un nodo privo di archi entranti

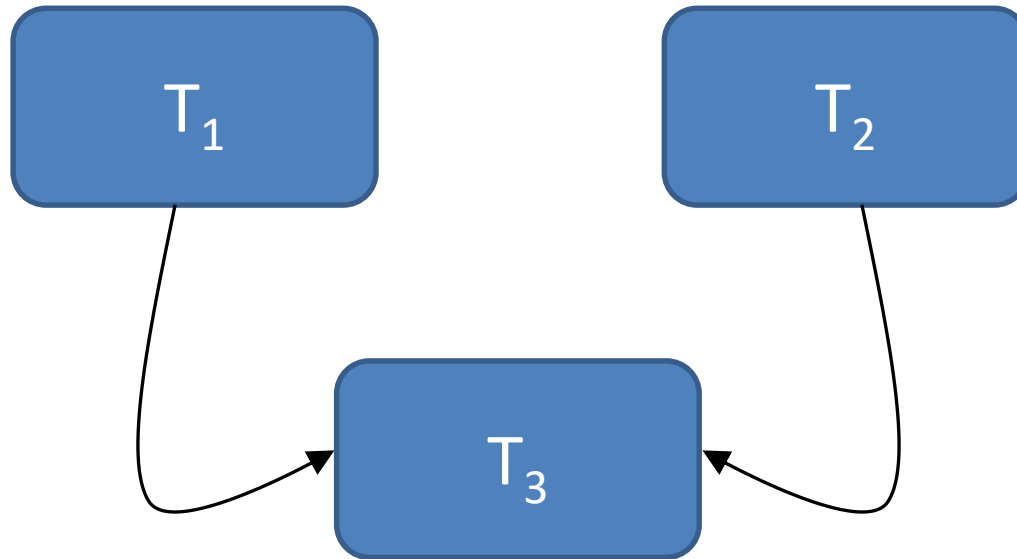
Prendiamo la storia  $S$ , scegliamo un nodo  $T_i$  privo di archi entranti e costruiamo la storia  $S^*$  in questo modo:

- Esploriamo la storia  $S$  nodo per nodo e trasferiamo in testa ad  $S^*$  le azioni che riguardano il nodo prescelto, nell'ordine in cui le troviamo in  $S$
- Rimane una parte di storia  $S'$  che è la storia originale senza le azioni del nodo prescelto



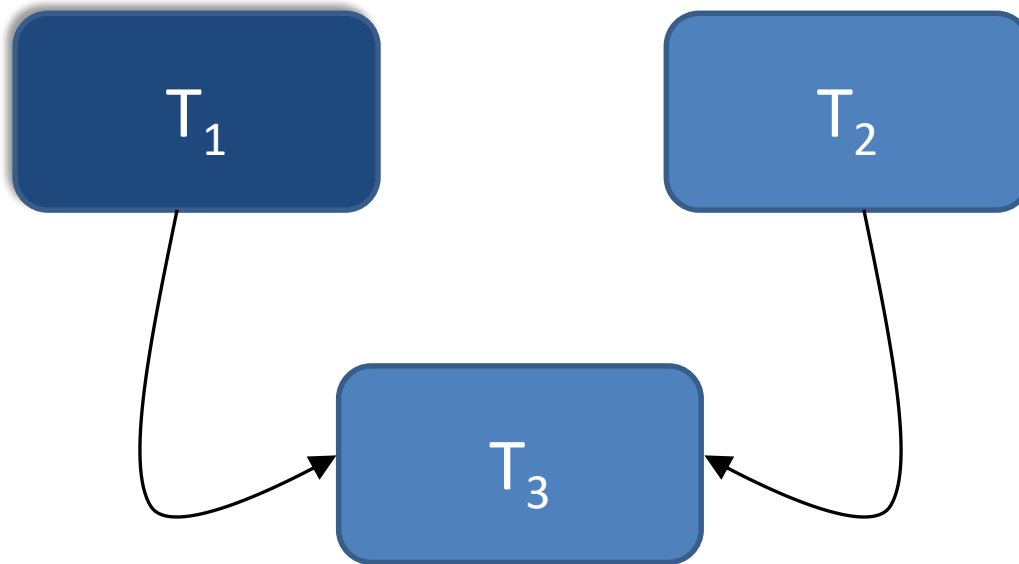
# Esempio: storia $S_x$

$S_x: r_1(X), r_2(X), r_3(Z), w_3(Z), r_1(Y), r_3(X), w_3(X)$



# Esempio: storia $S_x$

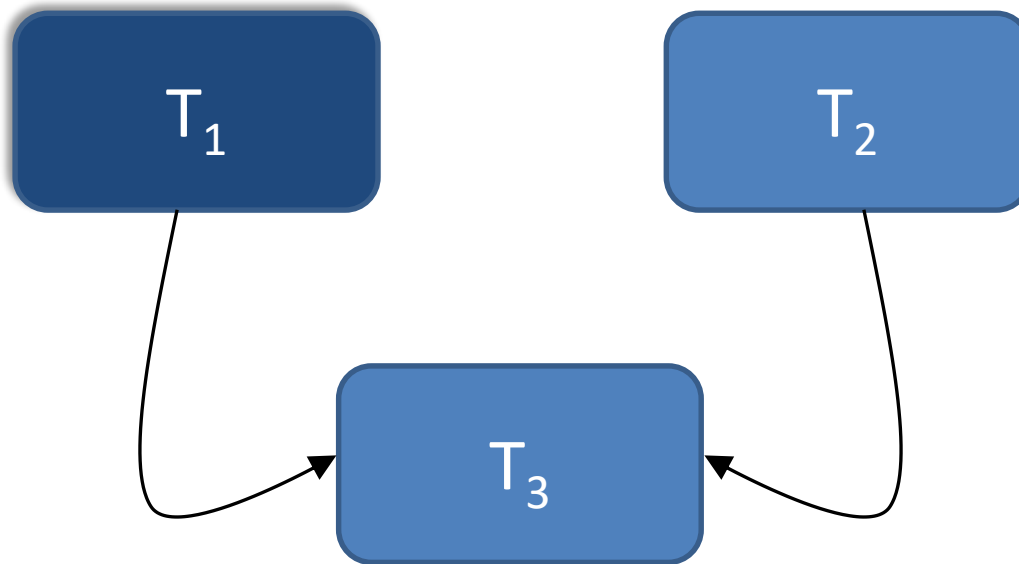
$S_x: r_1(X), r_2(X), r_3(Z), w_3(Z), r_1(Y), r_3(X), w_3(X)$



Scegliamo  $T_1$

# Esempio: storia $S_x$

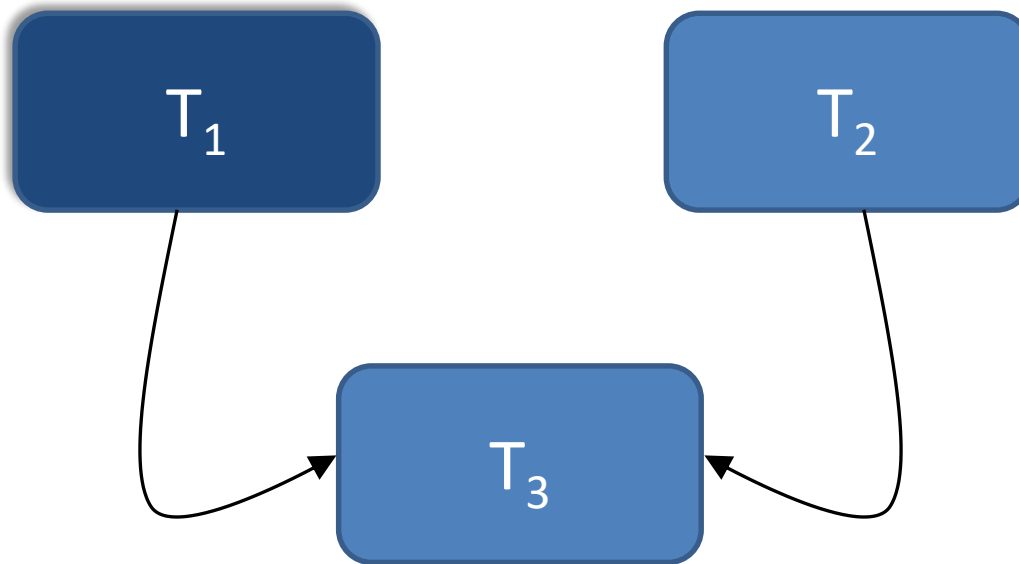
$S_x: \mathbf{r_1(X)}, r_2(X), r_3(Z), w_3(Z), \mathbf{r_1(Y)}, r_3(X), w_3(X)$



Costruiamo  $S^*: r_1(X), r_1(Y)$

# Esempio: storia $S_x$

$S_x: r_1(\mathbf{X}), r_2(X), r_3(Z), w_3(Z), r_1(\mathbf{Y}), r_3(X), w_3(X)$



Costruiamo  $S^*: r_1(X), r_1(Y)S'$

con  $S' = r_2(X), r_3(Z), w_3(Z), r_3(X), w_3(X)$

# View-equivalenza di $S^*$ ed $S$

La storia  $S^*$  è view-equivalente alla storia  $S$  originaria

- Spostando in testa una  $r_i(X)$  potrebbe esserci il rischio di un'inversione rispetto ad un'ipotetica  $w_j(X)$ 
  - se fosse così, ci sarebbe stato un arco  $T_j \rightarrow T_i$ , ma per ipotesi,  $T_i$  è privo di archi entranti
- Spostando in testa una  $w_i(X)$ , potrebbe esserci, come possibile violazione, una  $r_j(X)$  che precede la  $w_i(X)$ 
  - in tal caso ci sarebbe stato un arco  $T_j \rightarrow T_i$ , ma per ipotesi  $T_i$  è privo di archi entranti
- Spostando in testa una  $w_i(X)$ , potrebbe esserci una  $w_j(X)$ , che precedeva  $w_i(X)$ , che si trova ad essere l'ultima scrittura su  $X$ 
  - In tal caso ci sarebbe stato un arco  $T_j \rightarrow T_i$ , ma per ipotesi  $T_i$  è privo di archi entranti

# Passo di induzione

I tre casi visti in precedenza sono i soli casi possibili, ma non possono verificarsi, quindi il trasferimento delle azioni della transazione  $T_i$  in testa non altera la view-equivalenza della storia rispetto alla storia originaria

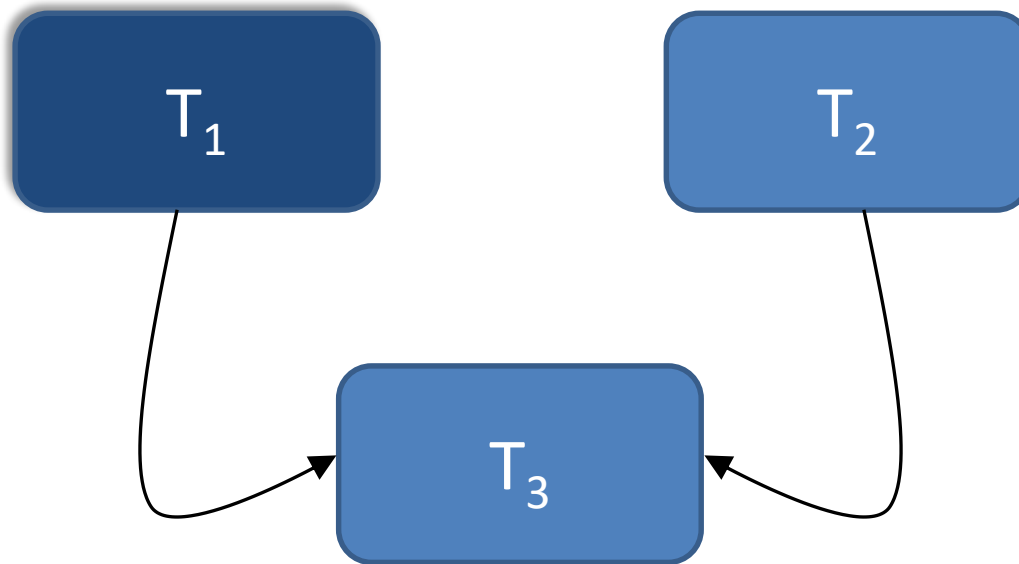
# Passo di induzione

Un'altra proprietà dei grafi aciclici è la seguente:  
se in un grafo aciclico cancello un nodo, il grafo continua ad essere aciclico

- Nella storia  $S'$  trovo  $n - 1$  transazioni (tutte le transazioni di  $S$  – la transazione  $T_i$ ) ed il grafo dei conflitti di  $S'$  è aciclico per la proprietà citata
- Procedendo in maniera iterativa arriverò in una situazione in cui ho una storia  $S^*$  con le transazioni in sequenza

# Esempio: storia $S_x$

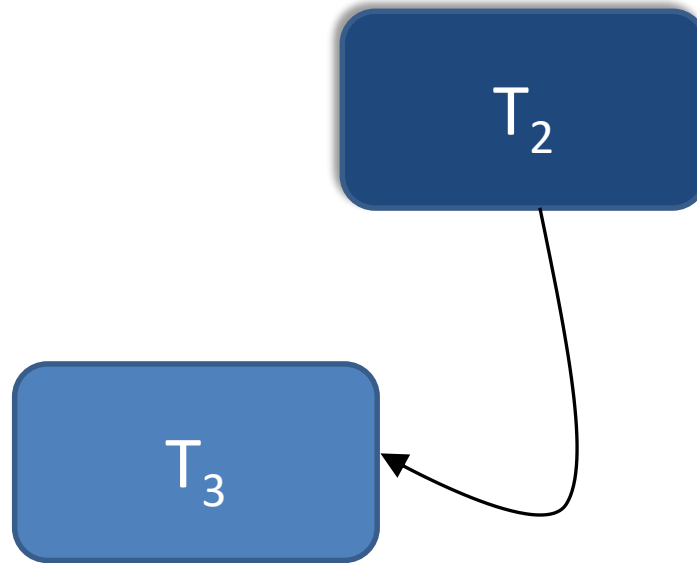
$S^*: r_1(X), r_1(Y), r_2(X), r_3(Z), w_3(Z), r_3(X), w_3(X)$





# Esempio: storia $S_x$

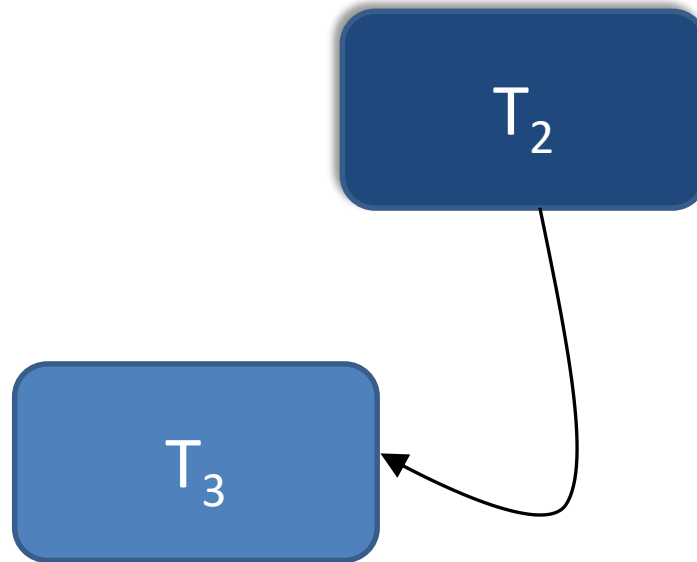
$S': r_2(X), r_3(Z), w_3(Z), r_3(X), w_3(X)$



Scegliamo  $T_2$

# Esempio: storia $S_x$

$S': r_2(X), r_3(Z), w_3(Z), r_3(X), w_3(X)$

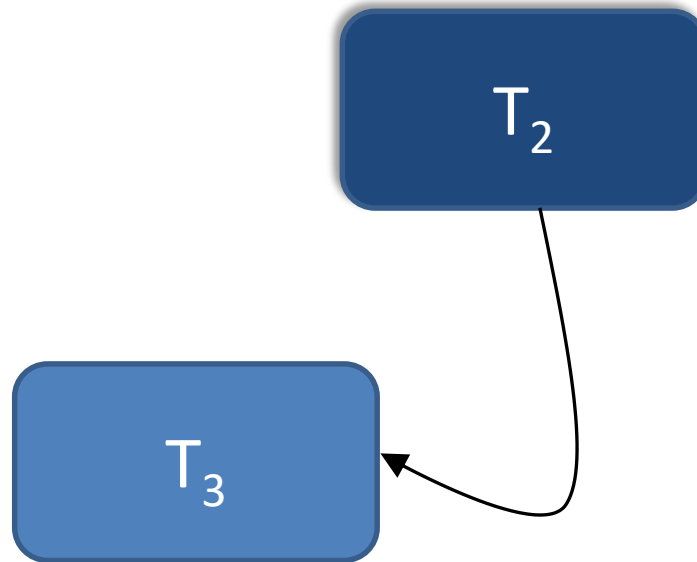


Costruiamo  $S^*: r_2(X)$

con  $S' = r_3(Z), w_3(Z), r_3(X), w_3(X)$

# Esempio: storia $S_x$

$S^*: r_2(X), r_3(Z), w_3(Z), r_3(X), w_3(X)$



# Esempio: storia $S_x$

$S': r_3(Z), w_3(Z), r_3(X), w_3(X)$



$T_3$

Ho una sola transazione che è banalmente aciclica

# Storia $S^*$

Alla fine ho generato una storia

$$S^*: r_1(X), r_1(Y), r_2(X), r_3(Z), w_3(Z), r_3(X), w_3(X)$$

che è view-equivalente rispetto alla storia

$$S_x: r_1(X), r_2(X), r_3(Z), w_3(Z), r_1(Y), r_3(X), w_3(X)$$

e in più è una serializzazione di  $S_x$

**N.B.:** non è l'unica serializzazione (basta partire da  $T_2$ )

# Conclusione

Qualsiasi **ordinamento topologico** del grafo dei conflitti aciclico è una storia seriale equivalente alla storia originale

Quindi, per verificare che un **interfogliamento** sia equivalente ad una storia seriale qualsiasi è sufficiente costruire il grafo dei conflitti e verificare che sia aciclico

# DBMS e concorrenza

Esistono diversi metodi di gestione della concorrenza, ma si possono tutti dividere in due grosse categorie:

- Metodi ottimistici
- Metodi pessimistici

# Metodi ottimistici

- Sono metodi che lanciano le azioni delle transazioni a mano a mano che le azioni vengono richieste senza particolari controlli
- Quando la transazione giunge al **commit** il DBMS fa un controllo per verificare che la storia è serializzabile, se non lo è forza un **rollback**
- Sono detti ottimistici perché partono dall'assunto che le transazioni non generano conflitti nelle storie

Non verranno trattati in questo corso



# Metodi pessimistici

- Partono dall'assunto che ogni transazione può produrre delle storie non serializzabili
- I metodi non ottimistici prevengono la non serializzabilità della storia
- Presenteremo il **metodo basato sui lock** perché è il più diffuso nei DBMS

# Lock

Il DBMS mette a disposizione delle transazioni i seguenti comandi che permettono di chiedere delle autorizzazioni a compiere azioni su oggetti:

- $LS(X)$ : **lock shared** sull'oggetto X
- $LX(X)$ : **lock exclusive** sull'oggetto X
- $UN(X)$ : **unlock** sull'oggetto X

Possiamo assimilare l'oggetto X ad una tupla

# Lock shared

Quando una transazione  $T_i$  vuole eseguire un'azione  $r_i(X)$ , prima di effettuare la lettura deve aver acquisito almeno il lock shared sull'oggetto  $X$ , ovvero il permesso per leggere l'oggetto  $X$

Il **lock shared** (condiviso) si chiama così perché transazioni diverse possono acquisire il lock shared sul medesimo oggetto

Il lock shared può essere quindi concesso dal DBMS a più transazioni

**Esempio:** la transazione  $T_j$  può chiedere e ottenere il LS sull'oggetto  $X$  già ottenuto dalla transazione  $T_i$

# Lock exclusive

Il **lock exclusive** richiede un accesso esclusivo all'oggetto  $X$  da parte di una transazione  $T_i$

La richiesta di lock exclusive è di norma fatta da una transazione per modificare un oggetto  $X$ : questa autorizzazione deve sempre precedere la una  $w_i(X)$

Quando  $T_i$  ha acquisito l'autorizzazione esclusiva a scrivere l'oggetto  $X$ , nessun'altra transazione può acquisire lock sullo stesso oggetto (né LS né LX) cioè, la transazione  $T_i$  diventa proprietaria esclusiva di  $X$

# Gestione del lock

Transazioni diverse possono lanciare attività parallele su medesimi oggetti

Come viene amministrata dal DBMS la compresenza di più lock?

- Il DBMS si avvale della **tabella di compatibilità**

possesso \ richiesta	LS	LX
LS	concede	nega
LX	nega	nega

Ad esempio se  $T_i$  possiede il lock sull'oggetto X e  $T_j$  ne fa richiesta, il DBMS concede o nega il lock a seconda dei casi elencati nella tabella di compatibilità

# Gestione del lock

- Quando il DBMS non può concedere il lock, la transazione richiedente va in **wait** (stato di attesa)
- Quando una transazione non ha più bisogno di leggere/scrivere l'oggetto X, può rilasciare il lock (shared o exclusive) attraverso **l'unlock**
- Per governare queste situazioni, il DBMS mantiene una tabella dei lock

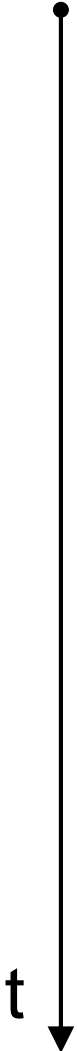
# Gestione del lock

T	Q1	Q2
stato iniziale	libera	libera

t



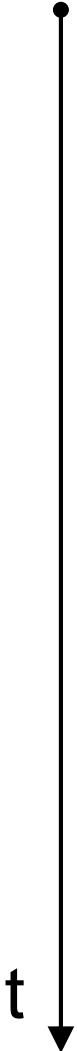
# Gestione del lock



T	Q1	Q2
stato iniziale	libera	libera
T1: LS(Q1)	LS a T1	libera

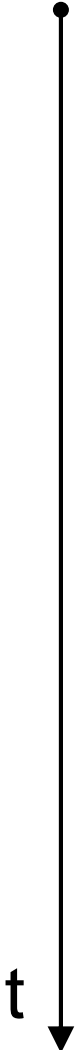


# Gestione del lock



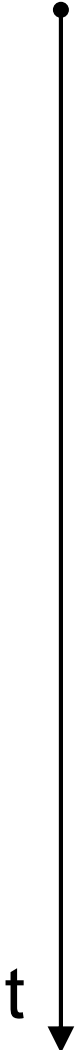
T	Q1	Q2
stato iniziale	libera	libera
T1: LS(Q1)	LS a T1	libera
T1: LX(Q2)	LS a T1	LX a T1

# Gestione del lock



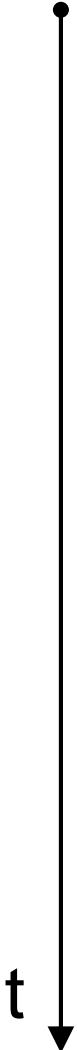
T	Q1	Q2
stato iniziale	libera	libera
T1: LS(Q1)	LS a T1	libera
T1: LX(Q2)	LS a T1	LX a T1
T2: LX(Q1)	LS a T1 WAIT(LX) a T2	LX a T1

# Gestione del lock



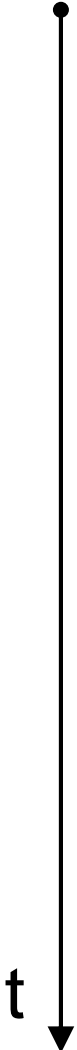
T	Q1	Q2
stato iniziale	libera	libera
T1: LS(Q1)	LS a T1	libera
T1: LX(Q2)	LS a T1	LX a T1
T2: LX(Q1)	LS a T1 WAIT(LX) a T2	LX a T1
T3: LS(Q1)	LS a T1 e T3 WAIT(LX) a T2	LX a T1

# Gestione del lock



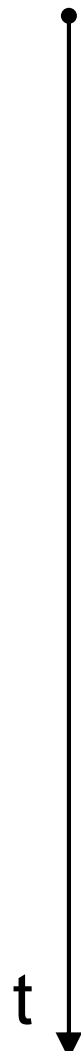
T	Q1	Q2
stato iniziale	libera	libera
T1: LS(Q1)	LS a T1	libera
T1: LX(Q2)	LS a T1	LX a T1
T2: LX(Q1)	LS a T1 WAIT(LX) a T2	LX a T1
T3: LS(Q1)	LS a T1 e T3 WAIT(LX) a T2	LX a T1
T3: LS(Q2)	LS a T1 e T3 WAIT(LX) a T2	LX a T1 WAIT(LS) a T3

# Gestione del lock



T	Q1	Q2
stato iniziale	libera	libera
T1: LS(Q1)	LS a T1	libera
T1: LX(Q2)	LS a T1	LX a T1
T2: LX(Q1)	LS a T1 WAIT(LX) a T2	LX a T1
T3: LS(Q1)	LS a T1 e T3 WAIT(LX) a T2	LX a T1
T3: LS(Q2)	LS a T1 e T3 WAIT(LX) a T2	LX a T1 WAIT(LS) a T3
T4: LS(Q1)	LS a T1, T3 e T4 WAIT(LX) a T2	LX a T1 WAIT(LS) a T3

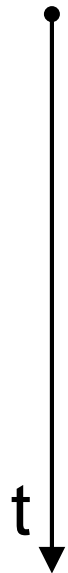
# Gestione del lock



T	Q1	Q2
stato iniziale	libera	libera
T1: LS(Q1)	LS a T1	libera
T1: LX(Q2)	LS a T1	LX a T1
T2: LX(Q1)	LS a T1 WAIT(LX) a T2	LX a T1
T3: LS(Q1)	LS a T1 e T3 WAIT(LX) a T2	LX a T1
T3: LS(Q2)	LS a T1 e T3 WAIT(LX) a T2	LX a T1 WAIT(LS) a T3
T4: LS(Q1)	LS a T1, T3 e T4 WAIT(LX) a T2	LX a T1 WAIT(LS) a T3
T4: LX(Q2)	LS a T1, T3 e T4 WAIT(LX) a T2	LX a T1 WAIT(LS) a T3 WAIT(LX) a T4

# Gestione dei lock

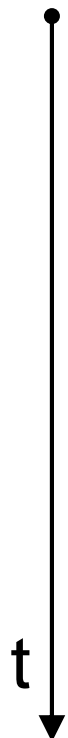
La situazione si sblocca quando la transazione T1 termina o rilascia l'oggetto (unlock)



T	Q1	Q2
situazione precedente	LS a <b>T1</b> , T3 e T4 WAIT(LX) a T2	LX a <b>T1</b> WAIT(LS) a T3 WAIT(LX) a T4
T1: UN(Q1,Q2)	LS a <del>T1</del> , T3 e T4 WAIT(LX) a T2	<del>LX a T1</del> WAIT(LS) a T3 WAIT(LX) a T4

# Gestione dei lock

La situazione si sblocca quando la transazione T1 termina o rilascia l'oggetto (unlock)

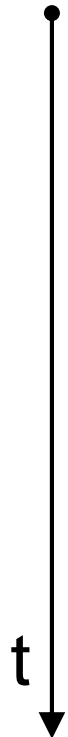


T	Q1	Q2
situazione precedente	LS a <b>T1</b> , T3 e T4 WAIT(LX) a T2	LX a <b>T1</b> WAIT(LS) a T3 WAIT(LX) a T4
T1: UN(Q1,Q2)	LS a <del>T1</del> , T3 e T4 WAIT(LX) a T2	<del>LX a T1</del> WAIT(LS) a T3 WAIT(LX) a T4
...	LS a T3 e T4 WAIT(LX) a T2	LS a T3 WAIT(LX) a T4



# Gestione dei lock


Le code delle transazioni in attesa vengono gestite con politiche FIFO



T	Q1	Q2
situazione precedente	LS a <b>T1</b> , T3 e T4 WAIT(LX) a T2	LX a <b>T1</b> WAIT(LS) a T3 WAIT(LX) a T4
T1: UN(Q1,Q2)	LS a <del>T1</del> , T3 e T4 WAIT(LX) a T2	<del>LX a T1</del> WAIT(LS) a T3 WAIT(LX) a T4
...	LS a T3 e T4 WAIT(LX) a T2	LS a T3 WAIT(LX) a T4
T3: UN(Q1,Q2)	LS a T4 WAIT(LX) a T2	LX a T4

# Gestione dei lock

Le code delle transazioni in attesa vengono gestite con politiche FIFO



T	Q1	Q2
situazione precedente	LS a <b>T1</b> , T3 e T4 WAIT(LX) a T2	LX a <b>T1</b> WAIT(LS) a T3 WAIT(LX) a T4
T1: UN(Q1,Q2)	LS a <del>T1</del> , T3 e T4 WAIT(LX) a T2	<del>LX a T1</del> WAIT(LS) a T3 WAIT(LX) a T4
...	LS a T3 e T4 WAIT(LX) a T2	LS a T3 WAIT(LX) a T4
T3: UN(Q1,Q2)	LS a T4 WAIT(LX) a T2	LX a T4
T4: UN(Q1,Q2)	LX a T2	libera

# Gestione del lock

Transazioni diverse possono lanciare attività parallele su medesimi oggetti

Come viene amministrata dal DBMS la compresenza di più lock?

- Il DBMS si avvale della **tabella di compatibilità**

possesso \ richiesta	LS	LX
LS	concede	nega
LX	nega	nega

**ATTENZIONE:** E' possibile **aggiornare** il lock da LS ad LX se una stessa transazione ne fa richiesta ed è l'unica che possiede il LS!

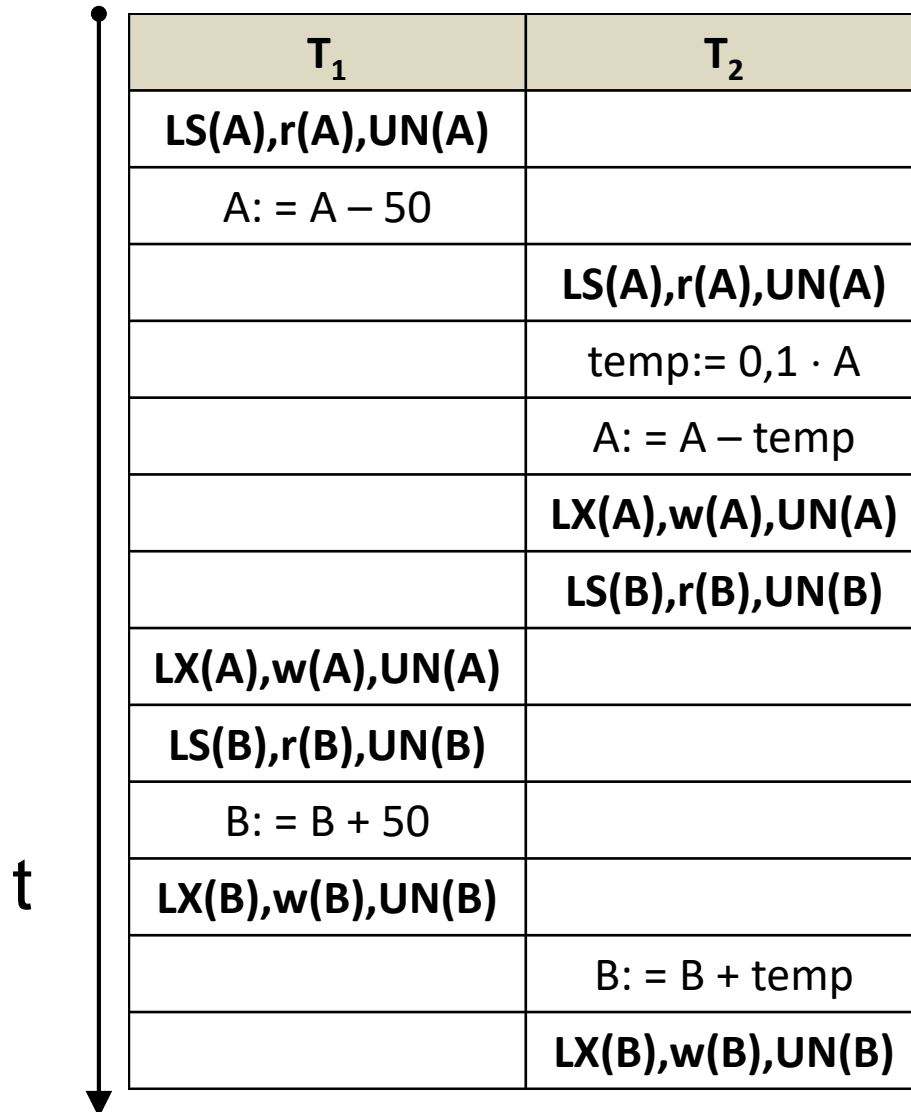
# Lock e concorrenza

Il sistema dei lock è stato introdotto per gestire la concorrenza

Ci chiediamo ora se questo sistema sia in grado di costruire delle storie serializzabili

La risposta è **no** e si può vedere sull'esempio  $S_1$

# Esempio: storia $S_1$



$T_1$	$T_2$
LS(A),r(A),UN(A)	
A: = A - 50	
	LS(A),r(A),UN(A)
	temp:= 0,1 · A
	A: = A - temp
	LX(A),w(A),UN(A)
	LS(B),r(B),UN(B)
LX(A),w(A),UN(A)	
LS(B),r(B),UN(B)	
B: = B + 50	
LX(B),w(B),UN(B)	
	B: = B + temp
	LX(B),w(B),UN(B)

L'interfogliamento inconsistente  $S_1$  si può riottenere con la politica dei lock in modo coerente con la tabella di compatibilità (ogni richiesta di lock su una risorsa viene soddisfatta perché la risorsa è già stata liberata)

**Qual è il beneficio del lock?**

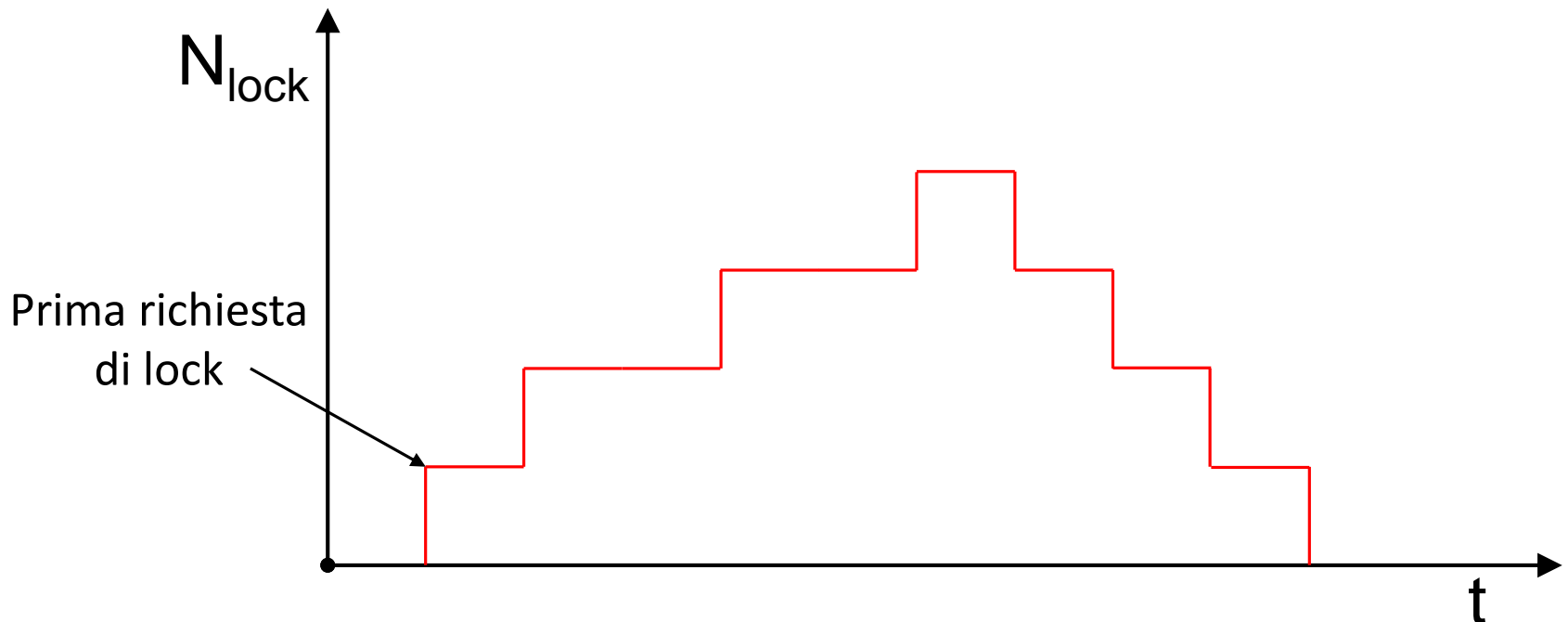
# Politica del locking a due fasi

Se si impone alle transazioni un vincolo nell'utilizzo degli **unlock**, allora la politica dei lock diventa la soluzione nella gestione della concorrenza

Il vincolo è stabilito dalla **politica di lock delle transazioni a due fasi** detta anche two-phase lock (2PL)

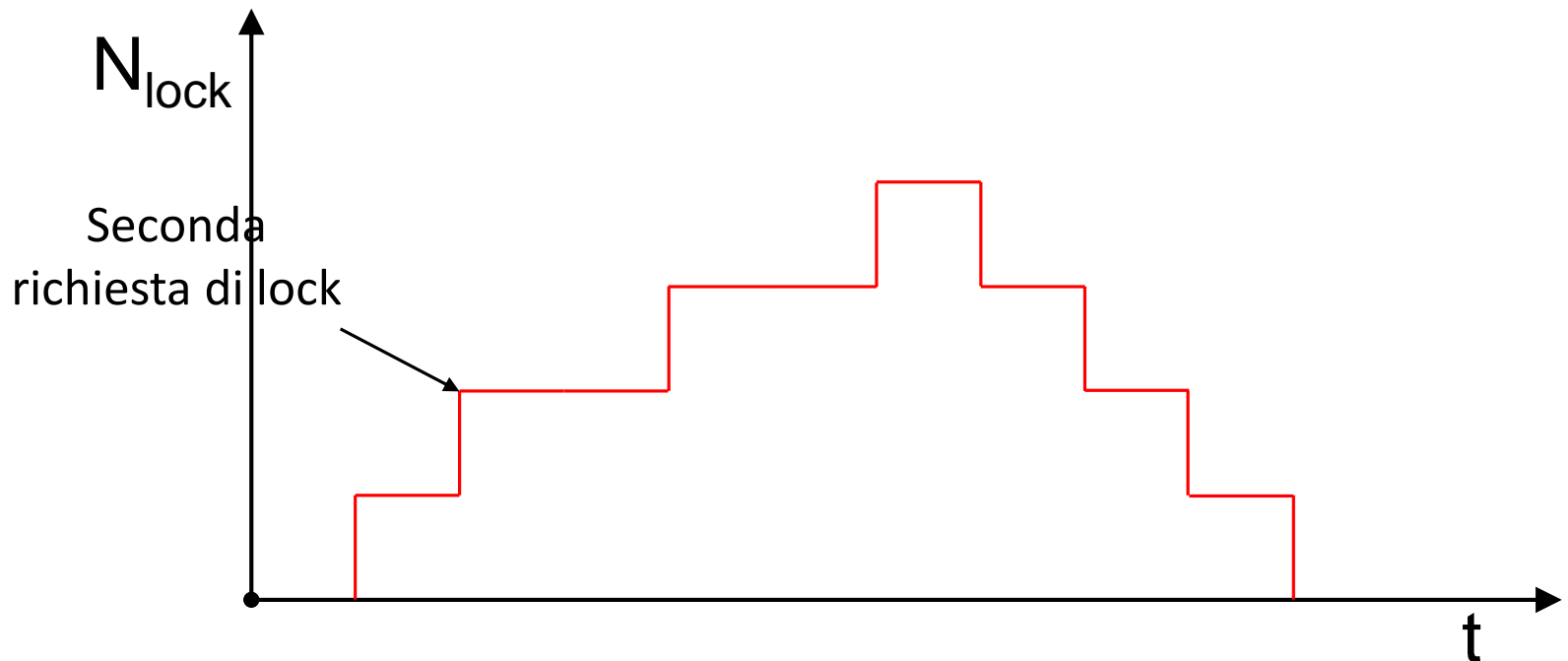
# Lock a due fasi

Possiamo vedere il funzionamento in un grafico (X,Y) in cui nell'asse delle X abbiamo il tempo e nell'asse delle Y abbiamo il numero di lock concessi dal DBMS ad una transazione  $T_i$



# Lock a due fasi

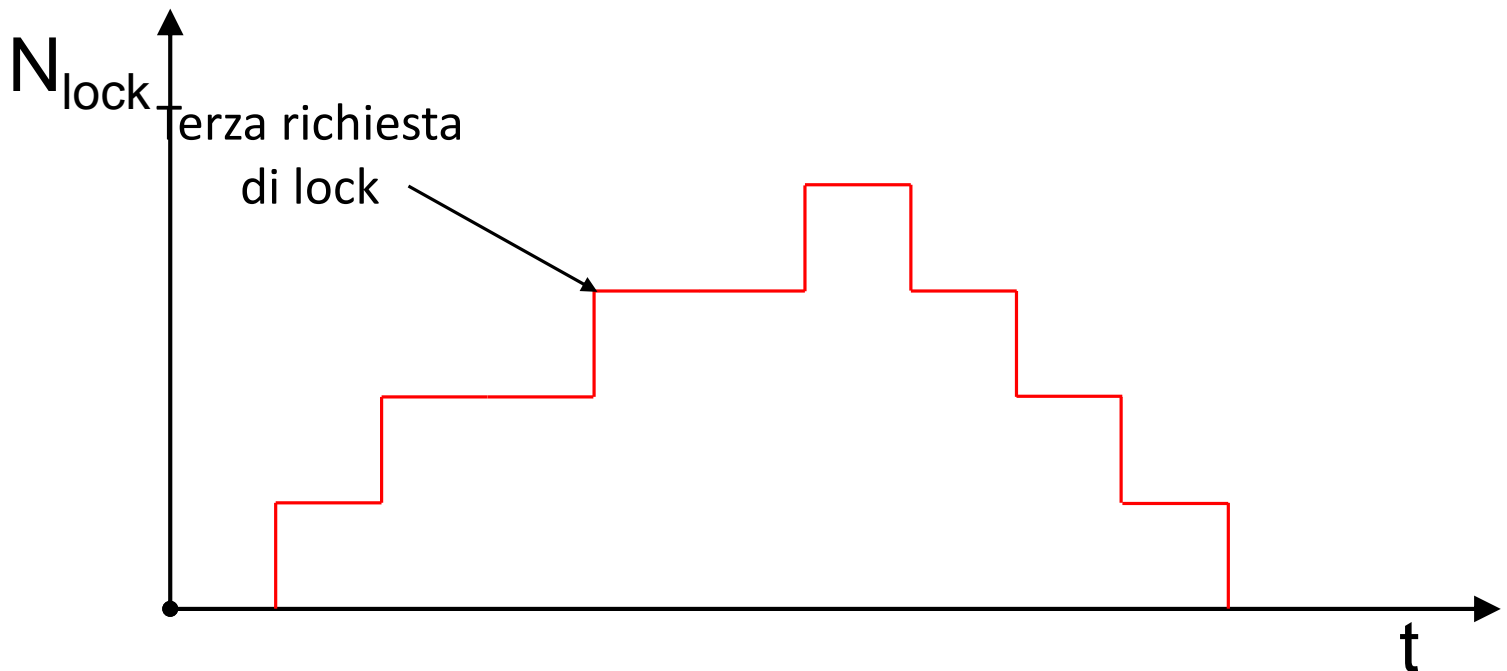
Possiamo vedere il funzionamento in un grafico (X,Y) in cui nell'asse delle X abbiamo il tempo e nell'asse delle Y abbiamo il numero di lock concessi dal DBMS ad una transazione  $T_i$





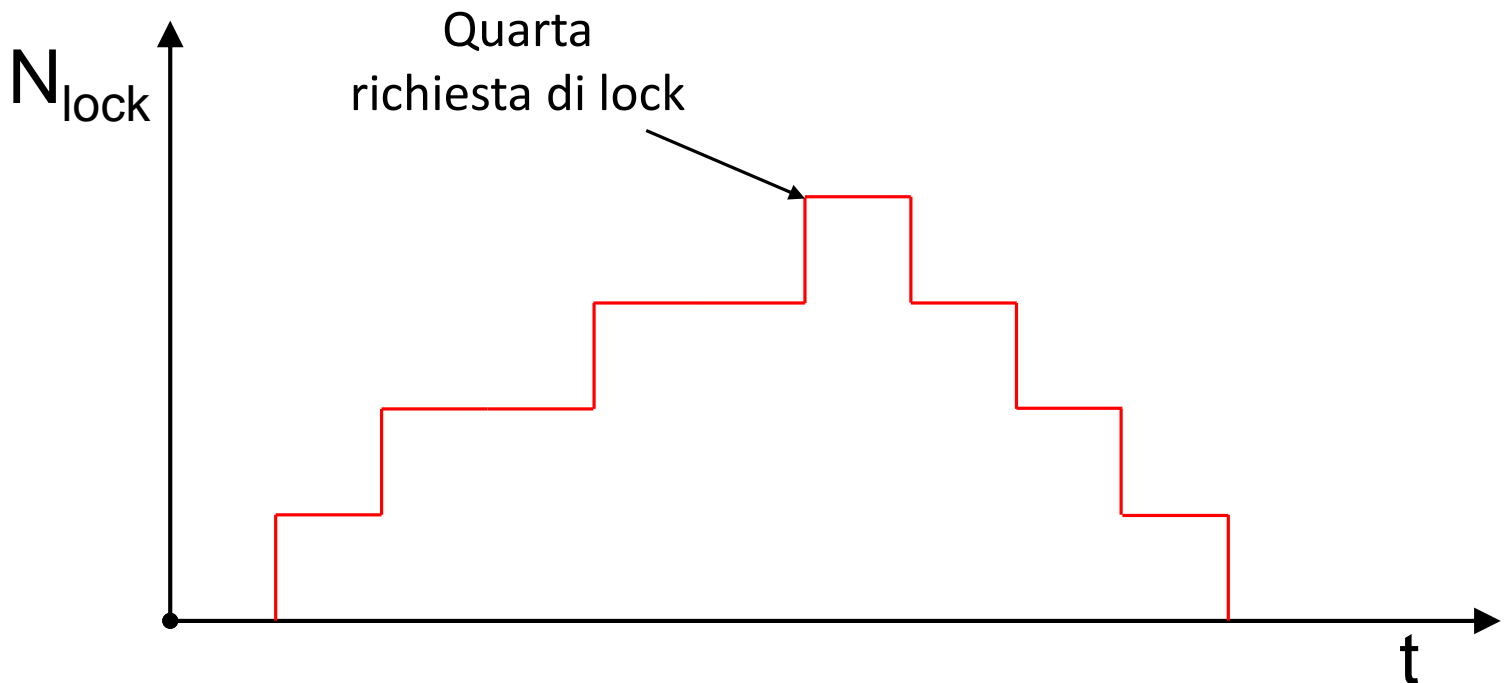
# Lock a due fasi

Possiamo vedere il funzionamento in un grafico (X,Y) in cui nell'asse delle X abbiamo il tempo e nell'asse delle Y abbiamo il numero di lock concessi dal DBMS ad una transazione  $T_i$



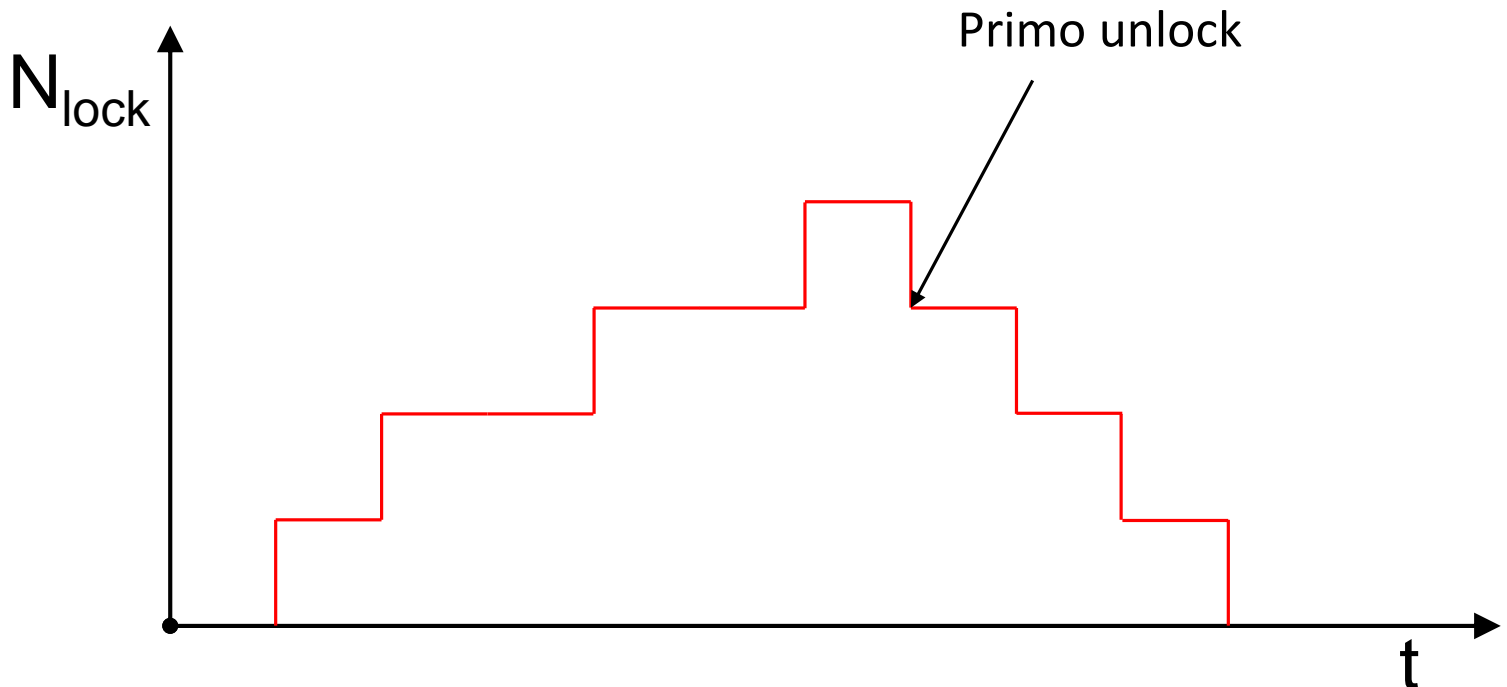
# Lock a due fasi

Possiamo vedere il funzionamento in un grafico (X,Y) in cui nell'asse delle X abbiamo il tempo e nell'asse delle Y abbiamo il numero di lock concessi dal DBMS ad una transazione  $T_i$



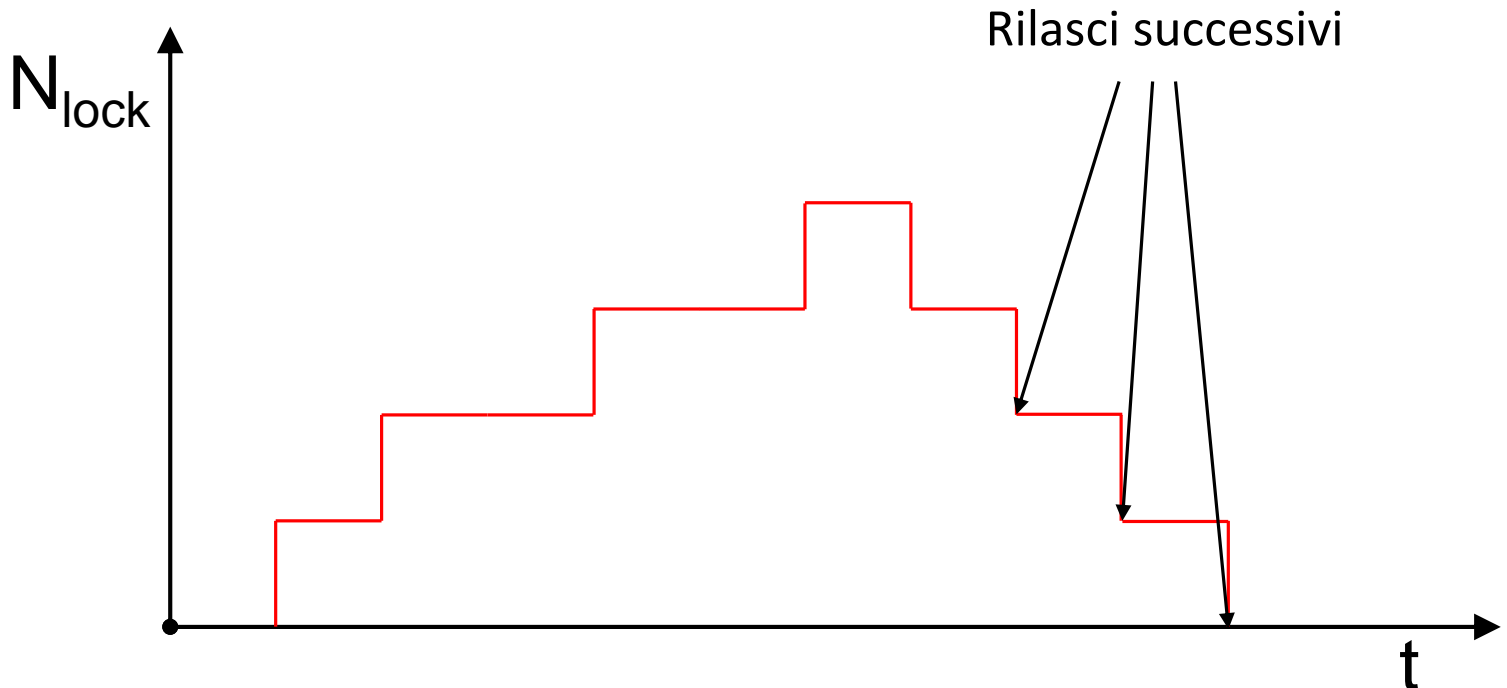
# Lock a due fasi

Ad un certo punto la transazione incomincia a rilasciare dei lock: quando una transazione inizia la fase di rilascio, da quel momento in poi non può più acquisirne



# Lock a due fasi

Ad un certo punto la transazione incomincia a rilasciare dei lock: quando una transazione inizia la fase di rilascio, da quel momento in poi non può più acquisirne



# Lock a due fasi

La politica si chiama "**a due fasi**" perché c'è una fase di **acquisizione** dei lock seguita da una fase di **rilascio** dei lock

Si può dimostrare che la politica del lock a due fasi garantisce la serializzabilità delle storie che genera

# Condizioni

- **Transazione ben formata:** una transazione è ben formata se ad ogni richiesta di lock (LS o LX) corrisponde un unlock
- **Storia legale:** la storia è una sequenza di azioni  $a_1, a_2, \dots, a_n$  dove  $a_i \in \{LS, LX, UN, read, write\}$ , considerata una posizione qualsiasi nella storia, se in questa posizione un certo oggetto  $X$  è in LX da parte di una transazione  $T_i$ , quell'oggetto  $X$  non può essere soggetto in quel punto della storia a lock LS o LX da parte di nessun'altra transazione  
**In altre parole** una storia è legale quando rispetta la tabella di compatibilità, in un determinato istante un oggetto  $X$  può essere soggetto al più ad un lock esclusiva

# Teorema di serializzabilità

Se le transazioni seguono il protocollo di lock a due fasi e sono ben formate, allora ogni storia  $S$  legale è serializzabile

# Dimostrazione

Consideriamo una storia e i tre casi di **azioni in conflitto**:

1. S: ... $r_i(X)$ , ...,  $w_j(X)$
2. S: ... $w_i(X)$ , ...,  $r_j(X)$
3. S: ... $w_i(X)$ , ...,  $w_j(X)$

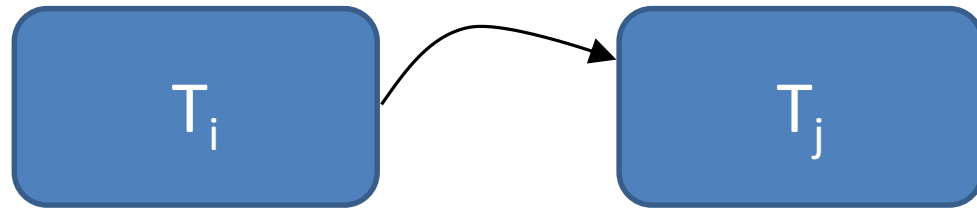
Affinché le operazioni possano essere eseguite è necessario richiamare le opportune azioni di lock

1. S: ... $r_i(X)$ , ...,  **$UN_i(X)$** , ...,  **$LX_j(X)$** ...,  $w_j(X)$
2. S: ... $w_i(X)$ , ...,  **$UN_i(X)$** , ...,  **$LS_j(X)$** ...,  $r_j(X)$
3. S: ... $w_i(X)$ , ...,  **$UN_i(X)$** , ...,  **$LX_j(X)$** ...,  $w_j(X)$



# Dimostrazione

Abbiamo quindi una transazione  $T_i$  e una transazione  $T_j$  legate da un arco di conflitto



# Dimostrazione

Dimostriamo ora che il grafo della storia  $S$  sotto la condizione del lock a due fasi è necessariamente aciclico

All'interno del grafo, abbinato a ciascuno nodo (transazione) immaginiamo una funzione chiamata **PrimoRilascio( $T_i$ )**:

- Esplorando la storia  $S$  dall'inizio, cerco il **primo punto** nella sequenza di azioni in cui appare un **unlock** della transazione  $T_i$  su un oggetto qualsiasi,
- Chiamiamo **k** la posizione di questo unlock
- Allora  **$k = \text{PrimoRilascio}(T_i)$**  è la prima posizione della storia in cui troviamo un unlock della transazione  $T_i$

# Dimostrazione

- Supponiamo che l'unlock della prima sequenza  
 $\dots r_i(X), \dots, \mathbf{UN}_i(\mathbf{X}), \dots, \mathbf{LX}_j(\mathbf{X}), \dots, w_j(X)$   
si trovi nella posizione  $k$ -esima
- Supponiamo che il lock di  $X$  da parte della  $T_j$  si trovi  
nella posizione  $h$ -esima

Sicuramente sarà  $k < h$  (l'unlock precede il lock)

Questo vale anche per gli altri due conflitti

# Dimostrazione

Fin qui non abbiamo usato l'ipotesi del lock a due fasi

- Chiamata  $k$  la posizione generica di un unlock di una qualsiasi delle tre sequenze, il primo rilascio della transazione  $T_i$  non può che essere in una posizione  $\text{PrimoRilascio}(T_i) \leq k$
- Ma  $k < h$ , dove  $h$  è la posizione generica in cui appare un lock della transazione  $T_j$

# Dimostrazione

Utilizziamo l'ipotesi del lock a due fasi

- Se nella posizione  $h$  della storia abbiamo un lock della transazione  $T_j$ , allora vuol dire che il primo unlock della transazione  $T_j$  non può che essere successivo ad  $h$ , quindi  $h < \text{PrimoRilascio}(T_j)$
- Mettiamo insieme le disequaglianze:

$$\text{PrimoRilascio}(T_i) < \text{PrimoRilascio}(T_j)$$

Se il grafo dei conflitti avesse un ciclo avremmo una situazione del genere:

$$\text{PrimoRilascio}(T_1) < \text{PrimoRilascio}(T_2) < \text{PrimoRilascio}(T_3) < \text{PrimoRilascio}(T_1)$$

che è un assurdo, quindi **il grafo è aciclico**

# Raffinamenti del 2PL

Ci siamo sempre riferiti a storie di transazioni terminate in commit, consideriamo ora il caso di transazioni che vanno in rollback

- Supponiamo che la transazione  $T_i$  faccia l'unlock dell'oggetto  $X$  dopo averlo scritto
- Nell'attività parallela può succedere che una transazione  $T_j$  acquisisca l'oggetto  $X$  prodotto dalla transazione  $T_i$
- Immaginiamo ora che  $T_i$  vada in **rollback** e quindi anche la modifica su  $X$  deve essere disfatta, ci troviamo quindi in un caso di lettura sporca da parte di  $T_j$

...w<sub>i</sub>(X),...,UN<sub>i</sub>(X),...,LS<sub>j</sub>(X),...,r<sub>j</sub>(X),...,rollback( $T_i$ )

# Raffinamenti del 2PL

Una tecnica per porre rimedio a questo problema è quella del **rollback in cascata**:

- Il DBMS riconosce il rollback della transazione  $T_i$ , riconosce le transazioni  $T_j$  che hanno letto oggetti modificati dalla transazione  $T_i$  e provoca il rollback di **tutte** le transazioni  $T_j$
- Il rollback in cascata è una soluzione drastica e particolarmente disastrosa per le transazioni che si vedono abortite dal DBMS senza averlo richiesto

# Protocollo a due fasi stretto

Tutti i DBMS adottano un protocollo a due fasi modificato:

## **protocollo a due fasi stretto**

- La transazione  $T_i$  acquisisce lock fin quando può, quando inizia a rilasciare i lock, può solo rilasciare i lock shared ( $LS_i(X)$ , dove  $X$  sono oggetti letti da  $T_i$  ma non modificati da  $T_i$ )
- La transazione rilascia i lock exclusive soltanto quando termina (con un commit o rollback)
- Non c'è più bisogno di effettuare il rollback in cascata in quanto le altre transazioni non hanno avuto accesso agli oggetti modificati da  $X$  prima del rollback di  $T_i$



# Protocollo a due fasi forte

Alcuni DBMS, per ragioni di efficienza, utilizzano il **protocollo a due fasi forte**

- La transazione  $T_i$  acquisisce lock fin quando può
- La transazione rilascia **tutti** i lock soltanto quando termina (con un commit o rollback)

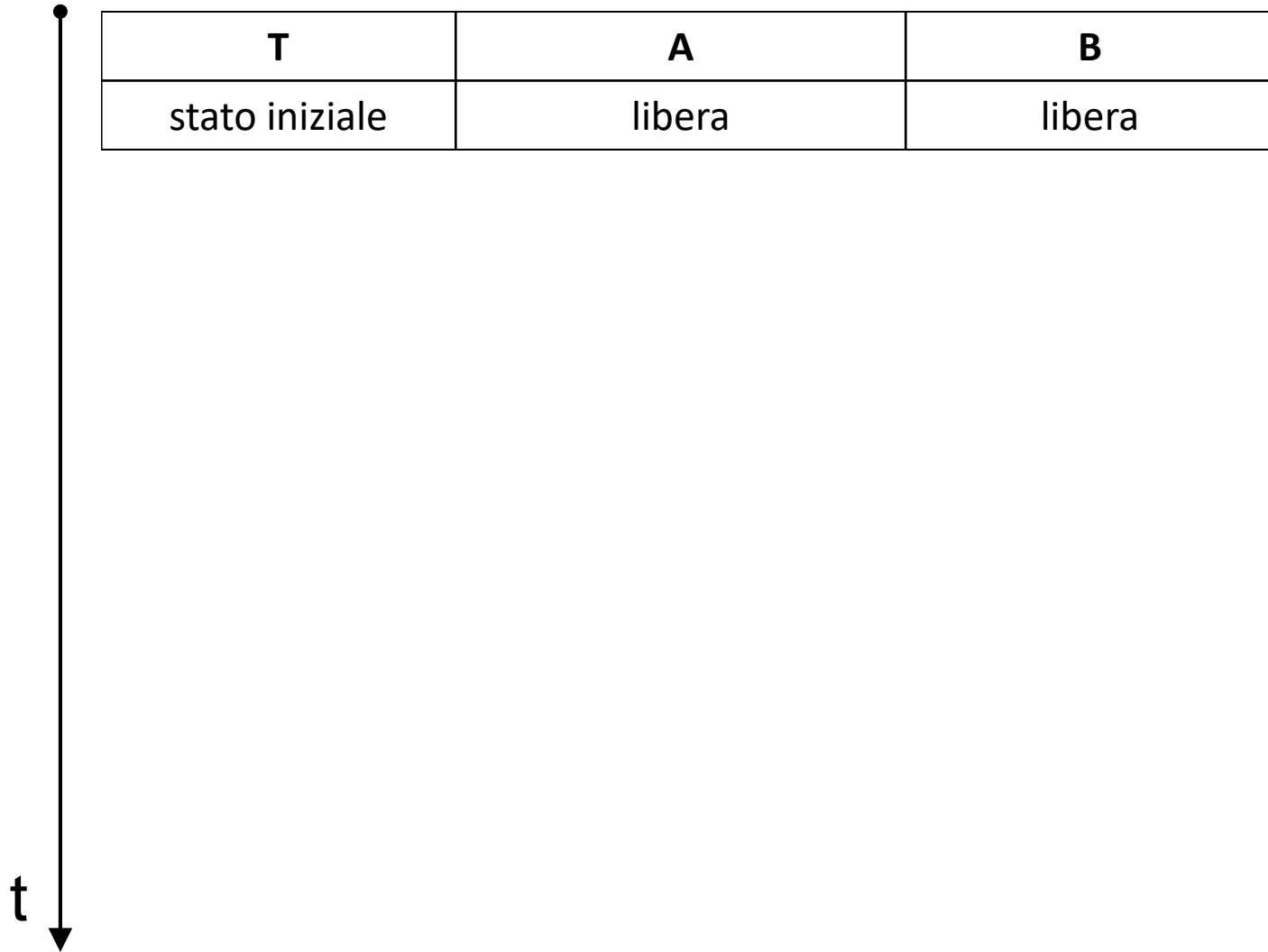
Questi ultimi due protocolli limitano il parallelismo del DBMS

# Osservazione

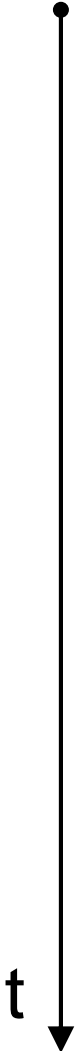
Il protocollo del lock a due fasi è piuttosto rigido

- Esaminando con attenzione l'esempio  $S_2$  (l'interfogliamento view-equivalente alla storia seriale  $T_1T_2$ ) si nota come la storia  $S_2$  non è possibile con il lock a due fasi
- Il lock a due fasi permette interfogliamenti consistenti nelle letture, ma i lock exclusive sono bloccanti
- Nello standard SQL sono previsti dei comandi per abbassare il livello di sicurezza sulla serializzabilità

# Gestione del deadlock

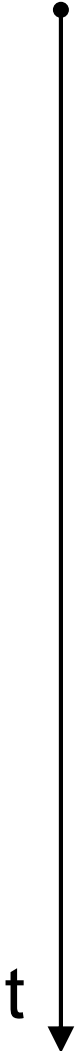


# Gestione del deadlock



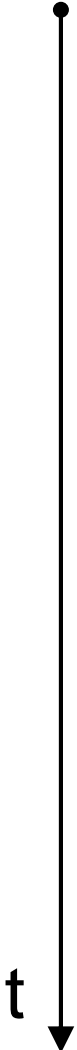
T	A	B
stato iniziale	libera	libera
T1: LX(A)	LX a T1	libera

# Gestione del deadlock



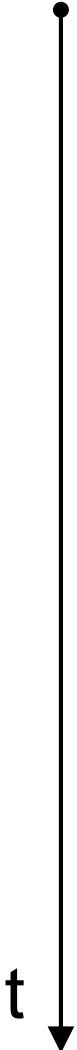
<b>T</b>	<b>A</b>	<b>B</b>
stato iniziale	libera	libera
T1: LX(A)	LX a T1	libera
T2: LX(B)	LX a T1	LX a T2

# Gestione del deadlock



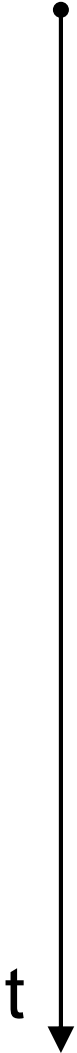
T	A	B
stato iniziale	libera	libera
T1: LX(A)	LX a T1	libera
T2: LX(B)	LX a T1	LX a T2
T1: LS(B)	LX a T1	LX a T2 WAIT(LS) a T1

# Gestione del deadlock



T	A	B
stato iniziale	libera	libera
T1: LX(A)	LX a T1	libera
T2: LX(B)	LX a T1	LX a T2
T1: LS(B)	LX a T1	LX a T2 WAIT(LS) a T1
T2: LS(A)	LX a T1 WAIT(LS) a T2	LX a T2 WAIT(LS) a T1

# Gestione del deadlock



T	A	B
stato iniziale	libera	libera
T1: LX(A)	LX a T1	libera
T2: LX(B)	LX a T1	LX a T2
T1: LS(B)	LX a T1	LX a T2 WAIT(LS) a T1
T2: LS(A)	LX a T1 <b>WAIT(LS) a T2</b>	LX a T2 <b>WAIT(LS) a T1</b>

**Deadlock:** le due transazioni sono bloccate



# Gestione del deadlock

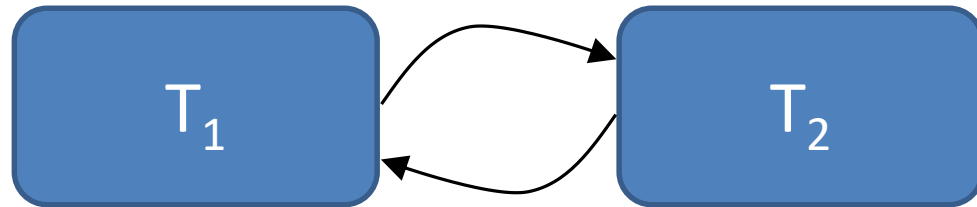
Presentiamo tre strategie per gestire o prevenire il **deadlock**:

1. Analisi del grafo di attesa
2. Timeout
3. Timestamp

Ce ne sono altre basate su euristiche molto più complesse che però non tratteremo

# Grafo di attesa

Il DBMS può fare un'analisi del cosiddetto grafo di attesa (da non confondere con il grafo dei conflitti)



- $T_1$  è in attesa di un rilascio da parte di  $T_2$  e  $T_2$  è in attesa di un rilascio da parte di  $T_1$
- Quando il grafo di attesa presenta un ciclo, c'è deadlock

# 1. Analisi del grafo di attesa

- I DBMS fanno un analisi del grafo per riconoscere un ciclo
- Devono poi scegliere una transazione di cui effettuare il rollback
- In questo modo, la transazione abortita libera i lock e l'altra transazione può procedere
- Nei grafi più complessi (con più transazioni), la scelta della transazione da abortire è fatta sulla base di euristiche che considerano la **starvation** (inedia)

# Starvation

C'è il rischio che una transazione abortita rientri nel sistema e venga nuovamente abortita

Le euristiche dei sistemi di gestione del deadlock pesano i rientri in modo da non far ricadere la scelta della transazione da abortire su quelle che sono state già abortite più volte

In ogni caso, queste euristiche sono costose

## 2. Timeout

- Una possibilità è di non costruire nessun grafo particolare, ma valutare da quanto tempo una transazione è in wait
- Se una transazione è in wait da molto tempo (è in **timeout**), il DBMS la abortisce, supponendo (non è detto che sia vero) che la transazione sia in deadlock
- In questo modo il DBMS spera di rompere il ciclo delle attese e rendere più fluido il processamento delle altre transazioni

# Problema

- Il problema dei metodi di gestione del deadlock con timeout è la scelta della soglia di timeout
- Se la soglia di timeout è troppo bassa, si rischia di far fare rollback a transazioni che sono in attesa standard, non in deadlock
- Se la soglia è troppo alta, prima che il DBMS si accorga del deadlock ci mette troppo tempo, comportando un degrado delle performance
- I DB administrator devono scegliere accuratamente la soglia di timeout

# 3. Timestamp

Esiste una tecnica di gestione delle transazioni che non richiede analisi di grafi d'attesa, ma **previene** i deadlock

La tecnica è basata sui **timestamp** (tempo di arrivo di una transazione nel sistema)

Sono possibili due situazioni

# Timestamp: prima situazione

- Immaginiamo una transazione  $T_j$  che inizia a lavorare ad un istante  $t_0$  e possiede, ad un certo punto, l'oggetto  $Q$
- In un tempo  $t$  successivo a  $t_0$  arriva una transazione  $T_i$  che inizia a lavorare
- La transazione  $T_i$  chiede un lock su  $Q$  incompatibile con il lock già in possesso di  $T_j$
- Il sistema mette in **wait** la transazione  $T_i$  e continua l'esecuzione di  $T_j$  ( $T_i$  è in attesa di  $T_j$ )



# Timestamp: seconda situazione

- Immaginiamo una transazione  $T_j$  che inizia a lavorare ad un istante  $t_0$
- In un tempo  $t$  successivo a  $t_0$  arriva una transazione  $T_i$  che inizia a lavorare e acquisisce l'oggetto  $Q$
- La transazione  $T_j$  chiede un lock su  $Q$  incompatibile con il lock già in possesso da  $T_i$
- Il sistema forza il **rollback** di  $T_i$ , libera  $Q$  e continua l'esecuzione di  $T_j$
- Il rollback **evita** il conflitto " $T_j$  in attesa di  $T_i$ "

# 3. Timestamp

Complessivamente in questo sistema non può mai formarsi il ciclo del grafo delle attese, perché gli archi di attesa possono andare solo da transazioni giovani verso transazioni vecchie

# Granularità del lock

Abbiamo parlato in generale di lock di oggetti X, ma i DBMS possono avere diverse scelte:

- X può essere una tupla (soluzione più diffusa)
- X può essere un attributo (lock complessi da gestire, ma aumentano il parallelismo)
- X può essere una pagina (usati nei sistemi distribuiti: i lock diminuiscono il parallelismo, ma sono molto più facili da gestire)
- X può essere un intero file (lock usati di solito nella gestione degli indici)